



CS1102: Data Structures and Algorithms

Part 2

Data Abstraction

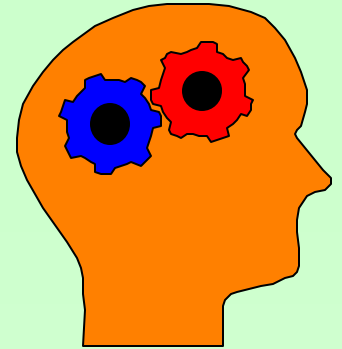
Zoltan KATO

S16 06-12

Adopted from Chin Wei Ngan's cs1102 lecture notes



Data Abstraction



- Importance of Abstraction
- Types of Abstraction
 - Procedural Abstraction
 - Data Abstraction
- Primitive ADT
- Complex ADT
 - Cartesian and Polar Implementation
 - Functions vs Mutators
 - Interface (Java Digression)
- Appointment Book ADT
- Ordered List ADT
 - Detailed Specification
 - Generic Objects (Java Digression)
- Sorted List ADT



Importance of Abstraction

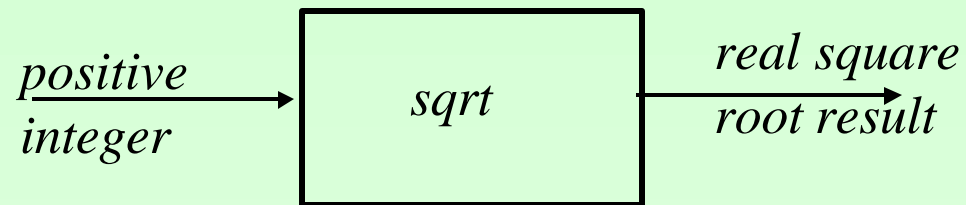
- structure large systems into layered components
- allow program codes to be more generic/reusable
- allow focus of "what" (specification) to be separated from "how" (implementation)
- use modularity to support localised changes

Procedural Abstraction

Achieved by:

- *treat function as black box*
- *give concise specification of its purpose*

Example: sqrt function



Do not need to know the detailed algorithm.

Textual Specification

Procedural Abstraction

**Alternatively we could use textual specification,
focusing on what the function does.**

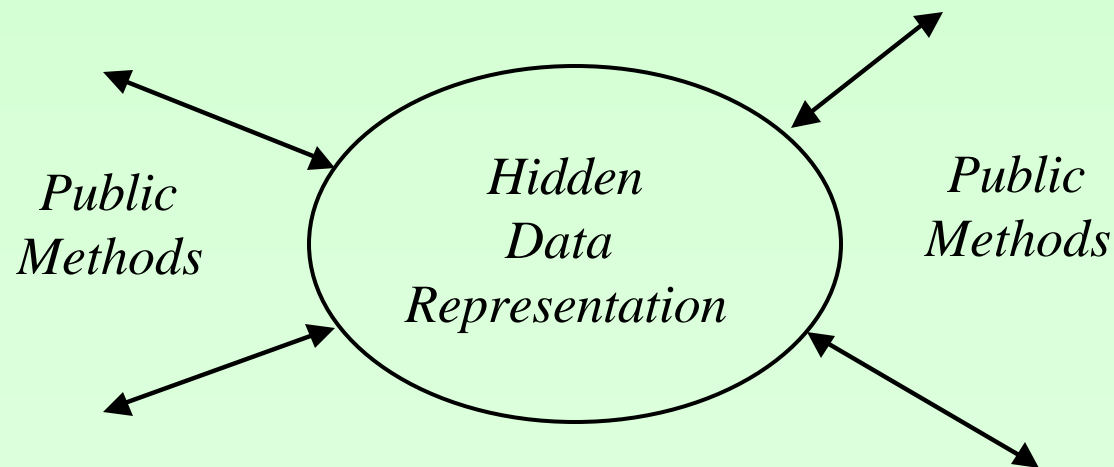
Example: sqrt function

```
real sqrt (int x)
  // pre : x be positive
  // post : returns square root of x as answer
  //       or returns r such that  $r^2 = x$ 
```

Data Abstraction

Commonly known technique - Abstract Data Type (ADT).

Focus on “*what*” the data structure can do, rather than “*how*” it can be represented.



Data Abstraction

- Why?
- *Hide the unnecessary details.*
 - *Help manage software complexity.*
 - *Easier Software Maintenance.*
 - *Functionalities are less likely to change.*
 - *Localised rather than Global Changes.*

Key Techniques

Data Abstraction

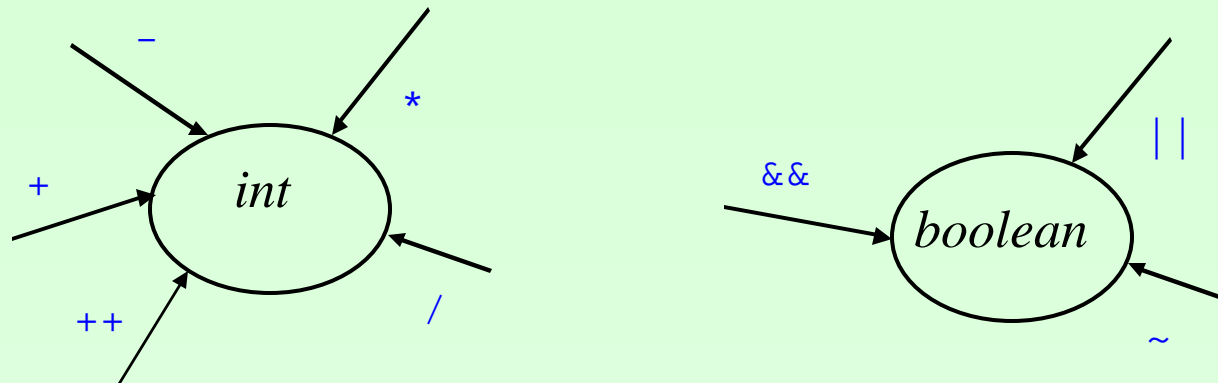
- ***Data Encapsulation*** - *keeping data and operations together in one location*
- ***Information Hiding*** - *restricting visibility of implementation details to where needed.*

Primitive Types as ADTs

Predefined data types of Java are ADTs!

Representation details are hidden which this aids *portability* too.

Examples : int, boolean, Strings, float.



Types of Operations

Primitive Types as ADTs

Broadly classified as:

- constructors* (to create objects)
- mutators* (to update objects)
- accessors* (to query about state of objects)
- destroyers* (to terminate an object)

In the case of array ADT type, we have:

constructor

```
int[] x = {2,4,6,8}
```

mutator

```
x[3]=10
```

accessor

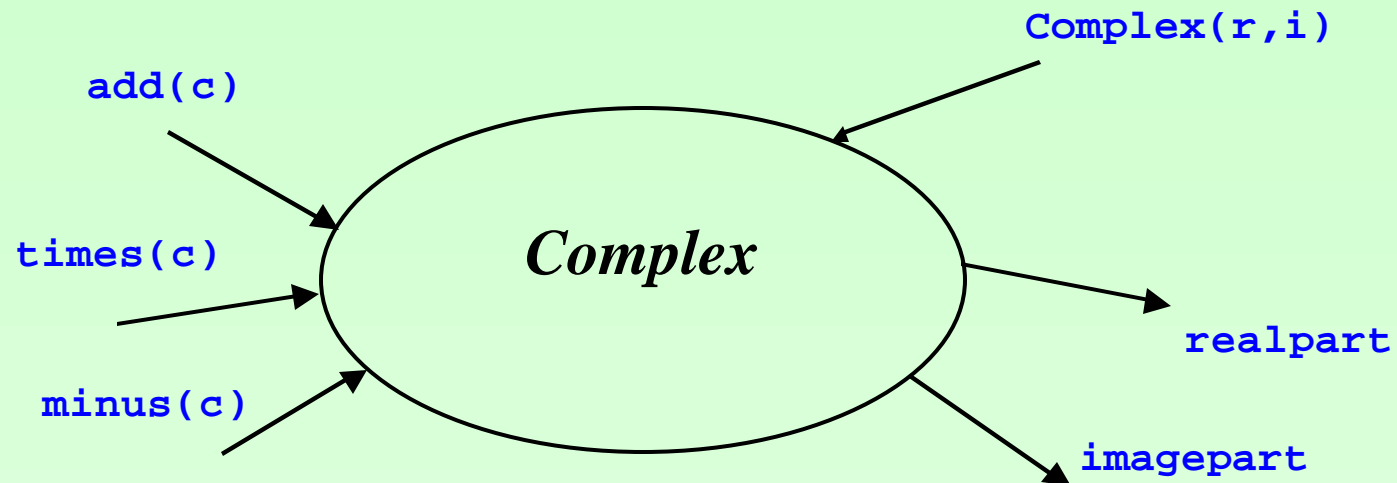
```
int y = ...x[3]...
```

destroyer

garbage collector

Complex ADT

User-defined data types can also be organised as ADTs.



Complex Number ADT

An ADT interface for Complex Numbers:

```
class Complex
{ private ...           // declaration of hidden fields
  public Complex(float r, float i) { ... } // create a new object
  public void add (Complex c) { ... }      // this = this + c
  public void minus (Complex c) { ... }    // this = this - c
  public void times (Complex c) { ... }    // this = this * c
  public float realpart () { ... }         // returns this.real
  public float imagepart () { ... }        // returns this.image
}
```

constructor

mutators

accessors

Example

Complex Number ADT

Can create, add, minus, subtract, times, as follows:

```
Complex c = new Complex(1,2);      // c = (1,2)
```

```
Complex d = new Complex(3,5);      // d = (3,5)
```

```
c.add(d);                          // c = c+d
```

```
d.minus(new Complex(1,1));         // d = d-(1,1)
```

```
c.times(d);                        // c = c*d
```

Cartesian Implementation

Complex ADT

```
class Complex
{ private float real;
  private float image;

  public Complex(float r, float i)  // create a new complex } constructor
    { real = r; image = i; }

  public float realpart ()          // returns this.real }
    { return real ; }
  public float imagepart ()        // returns this.image } accessors
    { return image; }

  public void add (Complex c)        // this = this + c
    { real = real+c.real; image = image+c.image; }
  public void minus (Complex c)     // this = this - c
    { real = real-c.real; image = image-c.image; }
  public void times (Complex c)     // this = this * c
    { real = real*c.real - image*c.image;
      image = real*c.image + image*c.real; }
}
```

To allow more efficient multiplication, we may use:

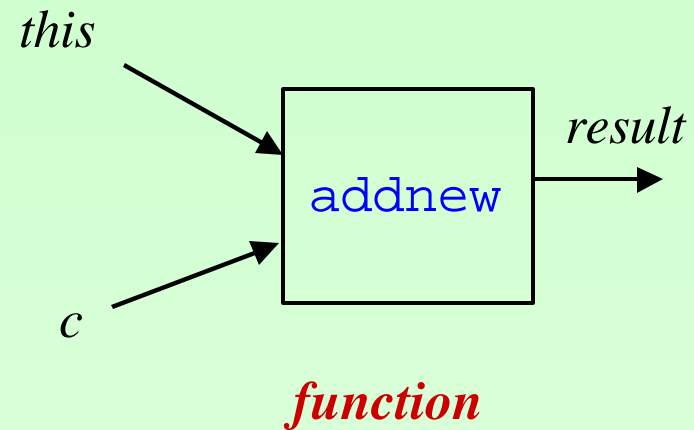
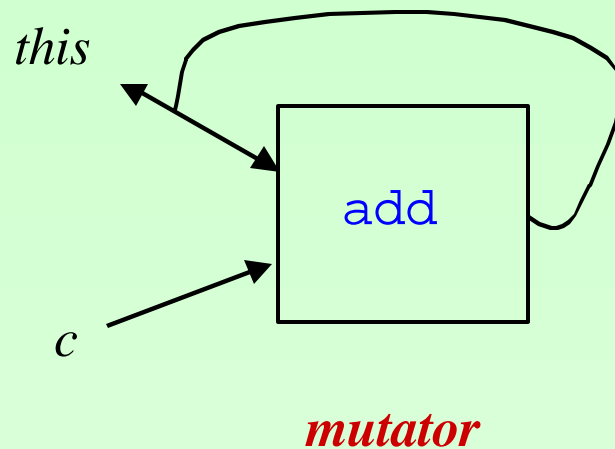
```
class Complex
{ private float radian;
  private float magnitude;

  :
  public times (Complex c)    // this = this * c
    { radian = radian+c.radian;
      magnitude = magnitude*c.magnitude; }
  :
}
```

Functions vs Mutators

Complex ADT

Instead of updating in-place, we can change the mutators into functions that return Complex results:



Example Code:

```
Complex e = c.addnew(d);           // e = c+d
```

```
Complex f = c.minusnew(d);        // f = c-d
```


Code for the new functions:

```
public Complex addnew (Complex c) // returns (this + c)
    { return new Complex(this.real+c.real, this.image+c.image); }

public Complex minusnew (Complex c) // returns (this - c)
    { return new Complex(this.real-c.real, this.image-c.image); }
```

Interface Mechanism

Java Digression

Java interface mechanism allows further abstraction/generalisation.

Example:

```
interface Storable {  
    final int READ = 0;  
    final int WRITE = 1;  
    byte[] get();  
    put(byte[] data);  
    int lastOp(); // returns READ or WRITE;  
}
```

Constants

Abstract methods

Later, we can declare:

```
class disk implements Storable {  
    ...  
}  
class tapes implements Storable {  
    ...  
}
```

Java Interface can be used for ADT too.

Use abstract methods in the interface

```
interface Complex
{
    void add (Complex c)           // this = this + c
    void minus (Complex c)         // this = this - c
    void times (Complex c)         // this = this * c
    float realpart ()              // returns this.real
    float imagepart ()             // returns this.image
    float radian ()                // returns this.redian
    float magnitude()              // returns this.magnitude
}
```

Can provide different implementation as classes.

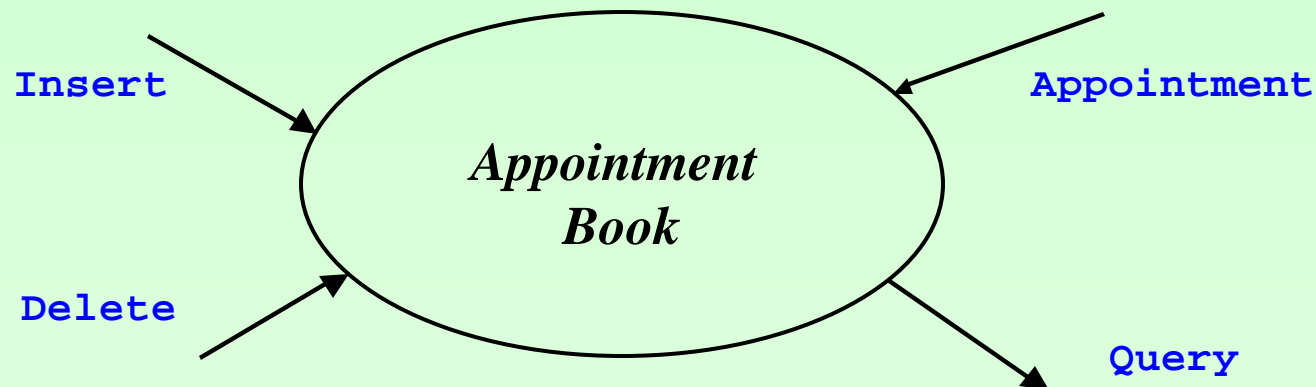
```
class ComplexCart implements Complex {  
    static Complex makeComplex(float r, float i) // recreate a complex object  
  
    ....  
}  
  
class ComplexPolar implements Complex {  
    static Complex makeComplex(float r, float i) // recreate a complex object  
  
    ....  
}
```

One advantage : *different implementations can co-exist side-by-side.*

Appointment Book ADT

Consider an appointment book

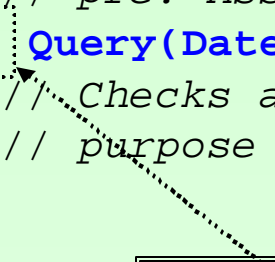
- for over 1 year period
- at half hour interval appointments
- brief description for each appointment



Java Declaration

Appointment Book ADT

```
class Appointment {  
    ..  
    public Appointment () {...} ;  
        // create a new empty appointment book  
    public void Insert(Date d, Time t, String p) {...};  
        // Makes an appointment.  
        // pre: Assumes there is no current appointment  
    public void Delete(Date d, Time t) {...};  
        // Deletes an appointment.  
        // pre: Assumes there exist an appointment to delete  
    public SPair Query(Date d, Time t) {...};  
        // Checks an appointment, and return True and the  
        // purpose if one exist; otherwise return false  
}
```



```
class SPair {  
    boolean flag  
    String purpose  
}
```

New Operations

Appointment Book ADT

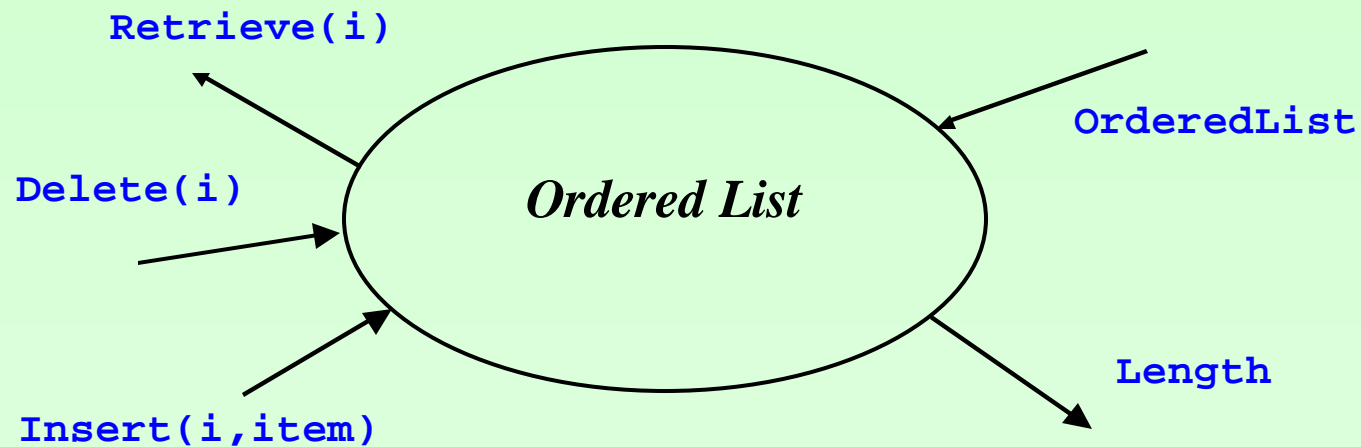
More complex operations can be defined in terms of basic ADT operations.
Example: **Change** can be defined in term of **Query**, **Insert**, **Delete**.

```
public SPair Change(Date olddate, Time oldtime,
                    Date newdate, Time newtime);
{ String status;
  Purpose p,q;
  if (this.Query(olddate,oldtime,p)) {
    if (this.Query(newdate,newtime,q)) {
      {status = "There is an existing appointment in the new time";
       return new SPair(false,status);}
    else
      {status = "Ok";
       this.Insert(newdate,newtime,p);
       this.Delete(olddate,oldtime);
       return new SPair(true,status);}
    } else
      {status = "You do not have such an appointment";
       return new SPair(false,status);}
    }
```

Ordered List ADT

A sequence of items where positional order matters.

$\langle a_1, a_2, \dots, a_{n-1}, a_n \rangle$



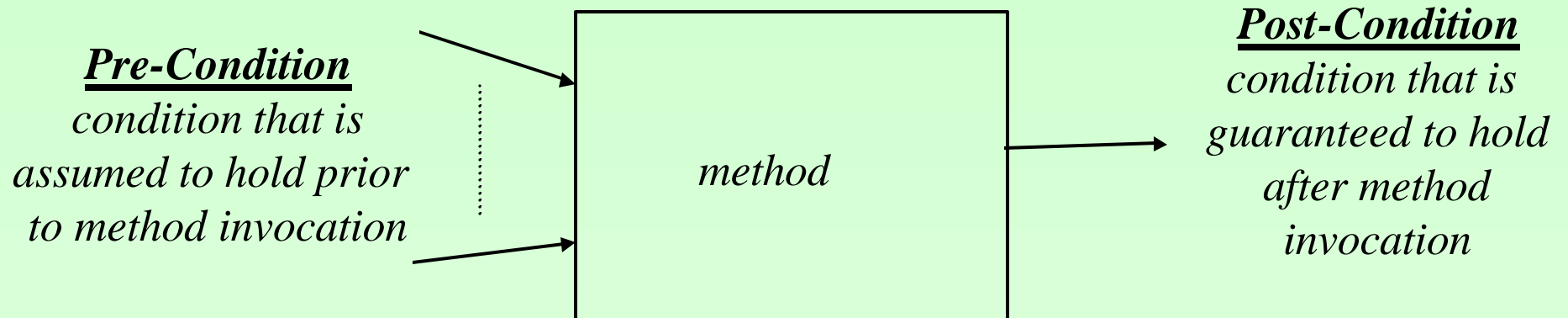
Ordered List ADT

Java Class Declaration

```
class OrderedList {  
    ...  
    public OrderedList() {...}  
        // to create an initial empty List  
    public int Length() {...}  
        // returns the number of items in this List  
    public void Insert(int i, Object item) {...}  
        // inserts a new item at position i.  
    public void Delete(int i) {...}  
        // deletes an existing item at position i  
    public Object Retrieve(i) {...}  
        // returns the item at position i  
}
```

Detailed Specification

Ordered List ADT



Detailed Specification

Ordered-List ADT

```
class OrderedList {  
    ...  
    public OrderedList() {...}  
        // pre : true  
        // post : this = <>  
  
    public int Length() {...}  
        // pre : this = <a1, ..., an> & n ≥ 0  
        // post : returns n  
  
    public void Insert(int i, Object item) {...}  
        // pre : this = <a1, ..., an> & 1 ≤ i ≤ n+1  
        // post : this = <a1, ..., ai-1, item, ai, ..., an>  
  
    public void Delete(int i) {...}  
        // pre : this = <a1, ..., an> and 1 ≤ i ≤ n  
        // post : this = <a1, ..., ai-1, ai+1, ..., an>  
  
    public Object Retrieve(i) {...}  
        // pre : this = <a1, ..., an> and 1 ≤ i ≤ n  
        // post : returns ai  
}
```

Here is a pair of Integers:

```
class PairInt {  
    Integer x; Integer y;  
    PairInt (Integer a,b) {x = a; y = b };  
    void swap(); { Integer temp;  
                    temp = x; x = y;  
                    y = temp } ;  
}
```

What about Pair of Strings? Pair of Arrays? Etc?

How can we make this code more generic?

Why? Generic codes facilitate reuse.

A very general class is **Object**!

```
class Pair {  
    Object x; Object y;  
    Pair (Object a,b) {x = a; y = b }  
    void swap(); { Object temp;  
                    temp = x; x = y;  
                    y = temp; }  
}
```

Above code is more generic. Can now use it for other objects too.
Just need to cast it to Object type.

Polymorphic Pair Example

Java Digression

Same code can swap both Integers & Strings:

```
p = new Pair (new Integer(3), new Integer(5));  
  
p.swap();  
  
q = new Pair ( "This", "That");  
  
q.swap();
```

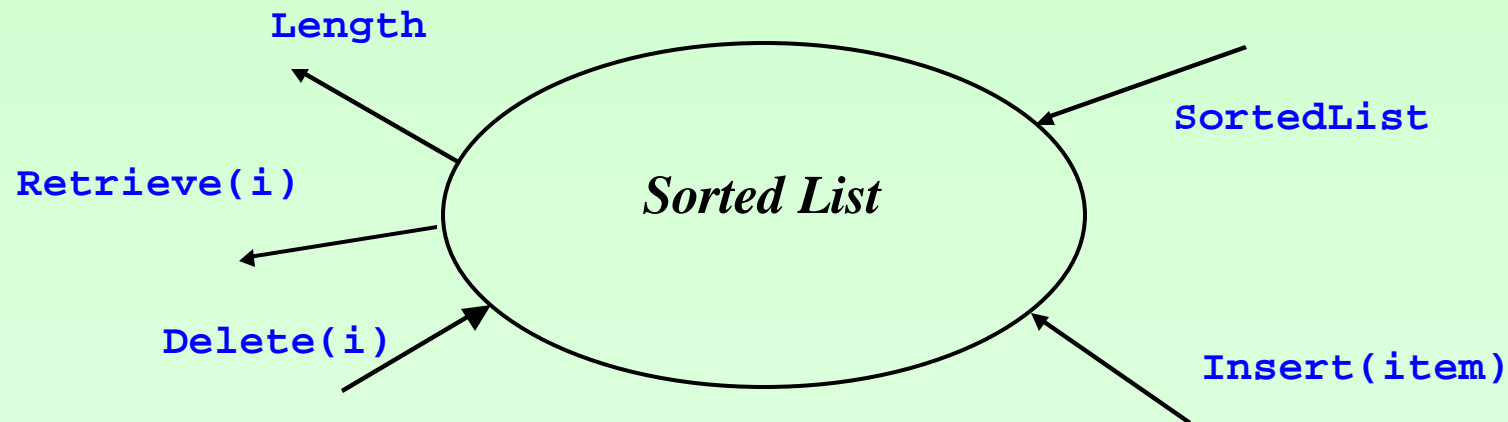
Looks good. But do be careful, since can also have:

```
r = new Pair (new Integer(3), "CS1102");  
  
r.swap();
```

Sorted List ADT

May like to maintain list $\langle a_1, \dots, a_n \rangle$ in a sorted order, such that:

$$a_1 < a_2 < \dots < a_n.$$



Sorted List ADT

Sorted list contains elements that implement the Comparable interface (i.e. can be compared/ordered)

```
class SortedList {  
    ...  
    public SortedList() {...}  
        // to create an initial empty List  
    public int Length() {...}  
        // returns the number of items in this List  
    public void Insert(Comparable item) {...}  
        // inserts a new item according to the sorted sequence.  
    public void Delete(int i) {...}  
        // deletes an existing item at position i  
    public Comparable Retrieve(int i) {...}  
        // returns the item at position i  
}
```


Comparable Interface

SortedList ADT

Many types belong to the `Comparable` interface.

Such type support comparison, e.g. Integer, Float, String, etc

```
interface Comparable
{
    int compareTo(Object rhs)
        // returns -1 (if this<rhs),
        // 0 (if this==rhs),
        // 1 (if this>rhs)
}
```