CS2040 Lecture Note #2: **Abstract Data Type**

Walls

Lecture Note #2: **ADT**

- **Objectives:**
  - Able to understand the need of data abstraction
  - Able to define ADT with Java Interface

    (Java Interface: A group of related methods with empty bodies)
  - Able to implement data structure given a Java interface

1 Software Engineering Issues

Motivation

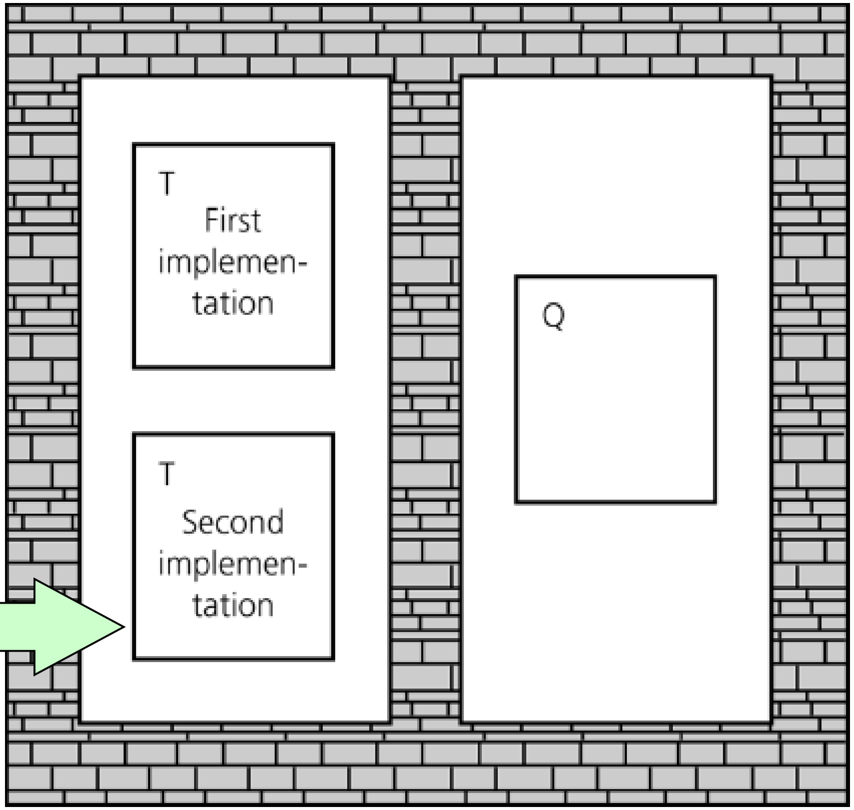
1. Software Engineering Issues (1/4)

- Program Design Principles
  - **Abstraction**
    - **Concentrate on what it can be done and not how it can be done**
    - **Use of Java Interface**
  - **Coupling**
    - **Restrict interdependent relationship among classes to the minimum**
  - **Coherent**
    - **A class should be about a single entity only**
    - **There should be a clear logical grouping of all functionalities**
  - **Information Hiding**
    - **Only expose necessary information to outside**

1. Software Engineering Issues (2/4)

- Information Hiding
  - Information hiding is like walls building around the various classes of a program.
  - The wall around each class *T* prevents the other classes from seeing how *T* works.
  - Thus, if class *Q* uses *T*, and if the approach for performing *T* changes, class *Q* will not be affected.

Makes it easy to substitute new, improved versions of how to do a task later



1. Software Engineering Issues (3/4)

- Information Hiding is not complete isolation of the classes
  - Q does not know how *T* does the work, but needs to know how to invoke *T* and what *T* produces
  - What goes in and comes out is governed by the terms of the method's specifications
    - If you use this method in this way, this is exactly what it will do for you.

1. Software Engineering Issues (4/4)

- Information Hiding CAN apply to data
  - Data abstraction asks that you think in terms of what you can do to a collection of data independently of how you do it
  - Data structure is a construct that can be defined within a programming language to store a collection of data
  - **Abstract data type (ADT)** is a collection of data & a specification on the set of operations on that data
    - Typical operations on data are: *add*, *remove*, and *query* (in general, management of data)
    - Specification indicates what ADT operations do, but not how to implement them

2 Abstract Data Type

Collection of data + set of operations on the data

2. Data Structure

- Data structure is a construct that can be defined within a programming language to store a collection of data
  - Arrays, which are built into Java, are data structure
  - We can create other data structures. For example, we want a data structure (a collection of data) to store both the names and salaries of a collection of employees

```
static final int MAX_NUMBER = 500;   // static final means “constant”
String [] names = new String [MAX_NUMBER];
double [] salaries = new double [MAX_NUMBER];
// employee names[i] has a salary of salaries[i]
```

or

(better choice)

```
class Employee {
   static final int MAX_NUMBER = 500;
   private String names;
   private double salaries;
}
…..
Employee[ ] workers = new Employee[Employee.MAX_NUMBER];
```

2. Abstract Data Type (ADT) (1/4)

- **ADT** is a collection of data together with a specification of a set of operations on that data
  - Specifications indicate what ADT operations do, but not how to implement them
  - Data structures are part of an ADT's implementation

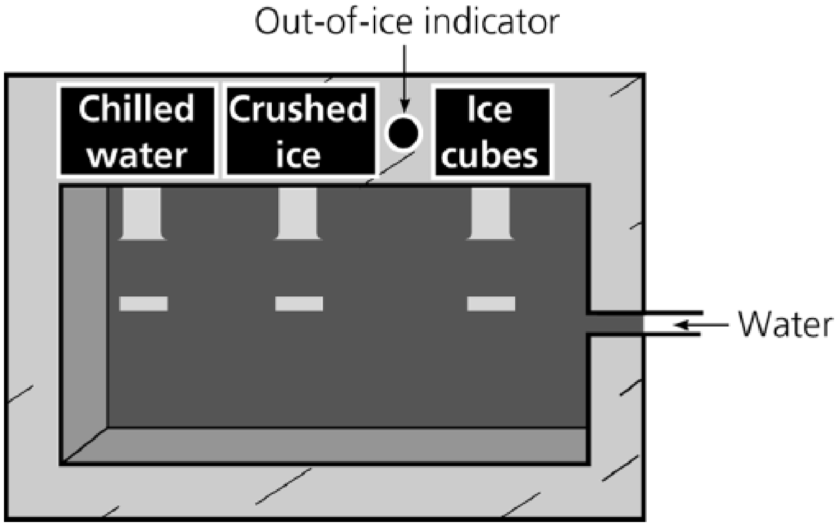
- **When a program needs data operations that are not directly supported by a language, you need to create your own ADT**

- You should first design the ADT by carefully specifying the operations before implementation

2. Abstract Data Type (ADT) (2/4)

- Example: A water dispenser as an ADT
  - Data: water
  - Operations: *chill*, *crush*, *cube*, and *isEmpty*
  - Data structure: the internal structure of the dispenser
  - Walls: made of steel

    The only slits in the walls:
    - **Input: water**
    - **Output: chilled water, crushed ice, or ice cubes.**

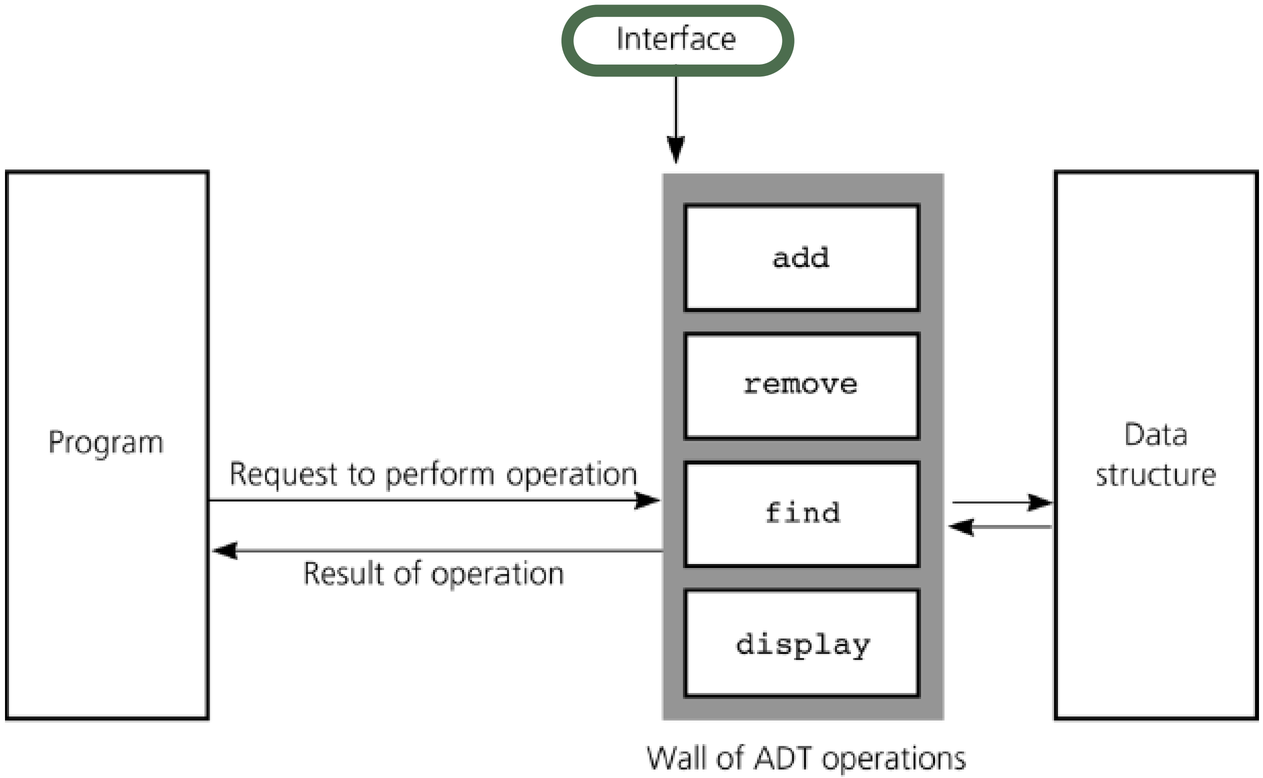


Crushed ice can be made in many different ways.
We don't care how it was made

  - Using an ADT is like using a vending machine.

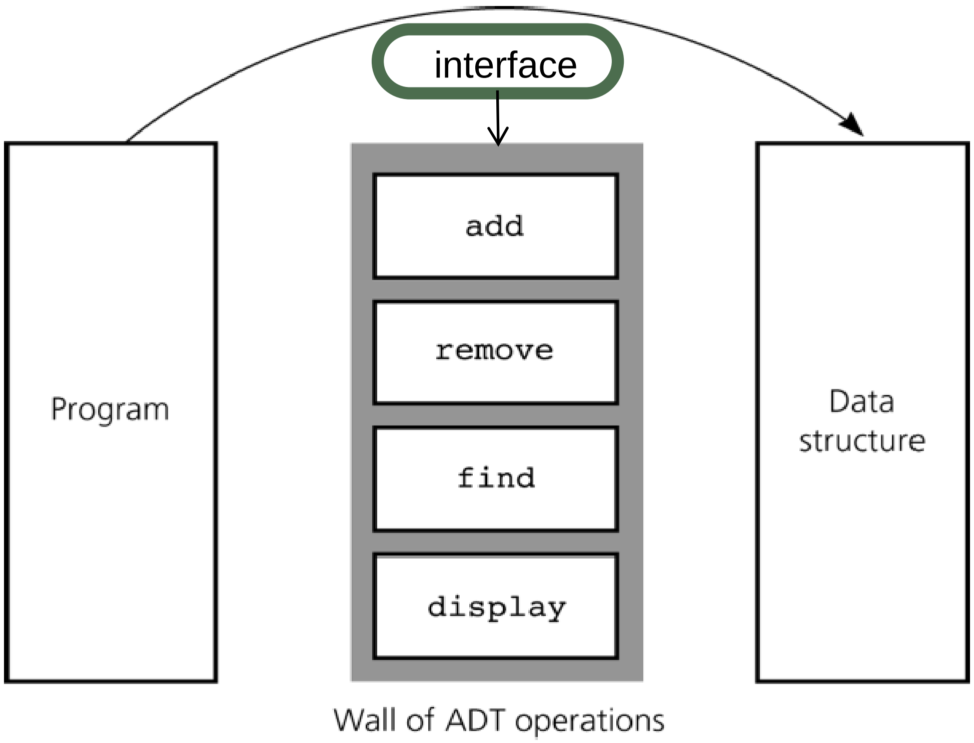
2. Abstract Data Type (ADT) (3/4)

- A WALL of ADT operations isolates a data structure from the program that uses it
- An **interface** is what a program/module/class should understand and use the ADT



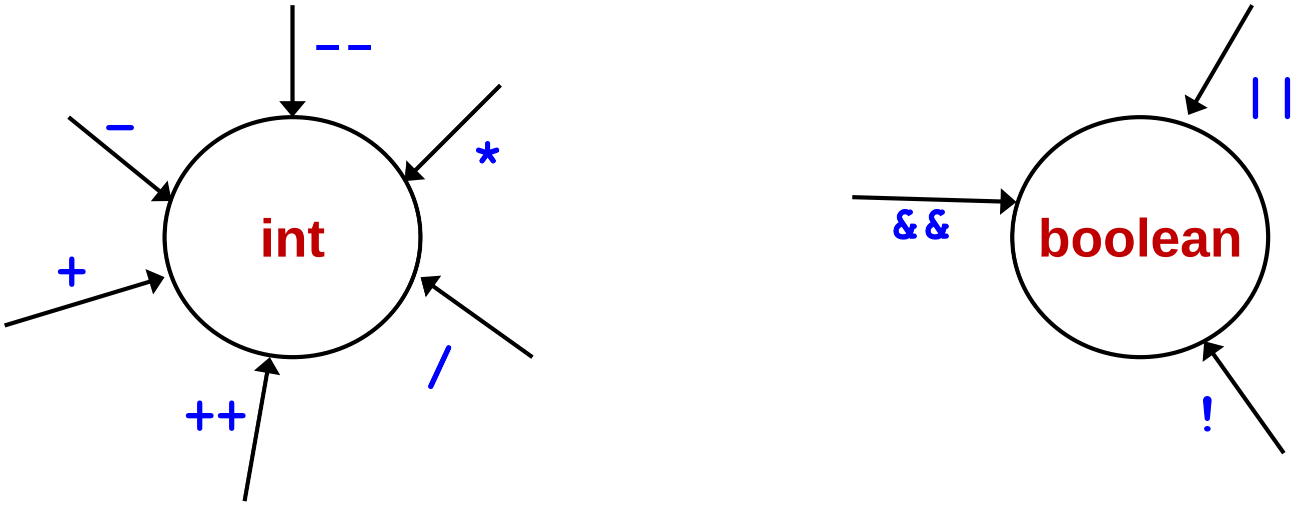
2. Abstract Data Type (ADT) (4/4)

- An **interface** is what a program/module/class should understand and use the ADT
- The following bypasses the interface to access the data structure. This violates the wall of ADT operations.



2. Example: Primitive Types as ADTs (1/2)

- Java's predefined data types are ADTs
- Representation details are hidden which aids portability as well
- Examples: int, boolean, String, double

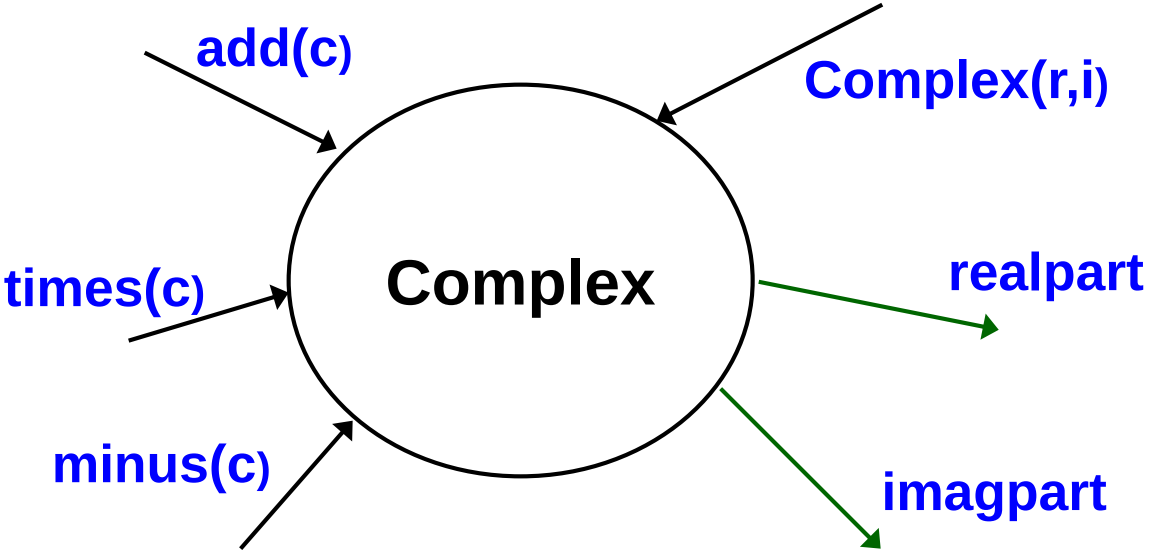


2. Example: Primitive Types as ADTs (2/2)

- Broadly classified as:
  (here the example uses the array ADT)
  - Constructors (to add, create objects)
    - int[ ] x = { 2,4,6,8 };
  - Mutators (to update objects)
    - x[3] = 10;
  - Accessors (to query about state of objects)
    - int y = x[3] + x[2];

2. Example: Complex Number ADT (1/5)

- User-defined data types can also be organized as ADTs



Note: add(c) means to add c to itself, etc.

2. Example: Complex Number ADT (2/5)

- A possible Complex ADT class:

```
class Complex {
   private ...                        // data members
   public Complex(double r, double 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 double realpart() { ... }            // returns this.real
   public double imagpart() { ... }            // returns this.imag
 }
```

- Using the Complex ADT:

```
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
```

2. Example: Complex Number ADT (3/5)

- 1st implementation: Cartesian

```
class Complex {
  private double real;
  private double imag;
  // CONSTRUCTOR
  public Complex(double r, double i) { real = r; imag = i; }
  // ACCESSORS
  public double realpart() { return real; }
  public double imagpart() { return imag; }
  // MUTATORS
  public void add (Complex c) {   // this = this + c
    real += c.realpart();
    imag += c.imagpart();
  }
  public void minus(Complex c) {  // this = this - c
    real -= c.realpart();
    imag -= c.imagpart();
  }
  public void times(Complex c) {  // this = this * c
    real = real*c.realpart() - imag*c.imagpart();
    imag = real*c.imagpart() + imag*c.realpart();
  }
}
```

2. Example: Complex Number ADT (4/5)

- 2nd implementation: Polar

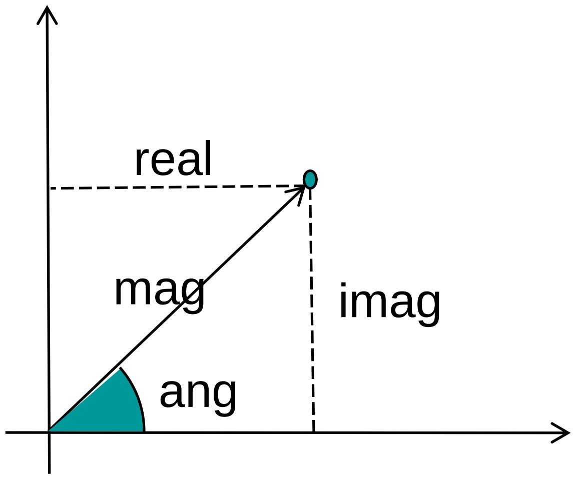
```
class Complex {
   private double ang;    // the angle of the vector
   private double mag;    // the magnitude of the vector
         :
         :
   public times(Complex c)  { // this = this * c
         ang += c.angle();
         mag *= c.mag();
   }
         :
         :
}
```

2. Example: Complex Number ADT (5/5)

- "Relationship" between Cartesian and Polar

From Polar to Cartesian: real = mag * cos(ang);
imag = mag * sin(ang);

From Cartesian to Polar: ang = $\tan^{-1}$(imag/real);
mag = real / cos(ang);



3 Java Interface

Related methods

3 Java Interface

- Java interfaces provide a way to specify common behaviour for a set of (perhaps unrelated) classes
- Java interface can be used for ADT
  - It allows further abstraction / generalization
  - It uses the keyword **interface**, rather than **class**
  - It specifies methods to be implemented by subclasses
    - **A Java interface is a group of related methods with empty bodies**
  - It can have constant definition (which are implicitly **public static final**)
- A class is said to implement the interface if it provides implementations for all of the methods in the interface

3 Example #1

```
// package in java.lang;
public interface Comparable <T> {
    int compareTo (T other);
}
```

```
class Shape implements Comparable <Shape> {
  static final double PI = 3.14;
  double area() {...};
  double circumference() { ... };
  int compareTo(Shape x) {
    if (this.area() == x.area())
      return 0;
    else if (this.area() > x.area())
      return 1;
    else
      return -1;
  }
}
```

3 Example #2: Interface for Complex

E.g. Complex ADT interface

- anticipate both Cartesian and Polar approach

Complex.java

```
public interface Complex {
  public double realpart();  // returns this.real
  public double imagpart();  // returns this.imag
  public double angle();     // returns this.ang
  public double mag();       // returns this.mag
  public void add(Complex c);   // this = this + c
  public void minus(Complex c); // this = this - c
  public void times(Complex c); // this = this * c
}
```

- In interface, methods have signatures but no implementation

3 Example #2: ComplexCart (1/2)

- Cartesian Implementation (Part 1 of 2)

ComplexCart.java

```
class ComplexCart implements Complex {
  private double real;
  private double imag;

  // CONSTRUCTOR
  public ComplexCart(double r, double i) { real = r; imag = i; }

  // ACCESSORS
  public double realpart() { return this.real; }
  public double imagpart() { return this.imag; }
  public double mag() { return Math.sqrt(real*real + imag*imag); }
  public double angle() {
    if (real != 0) {
      if (real < 0) return (Math.PI + Math.atan(imag/real));
      return Math.atan(imag/real);
    }
    else if (imag == 0) return 0;
    else if (imag > 0) return Math.PI/2;
    else return -Math.PI/2;
  }
```

3 Example #2: ComplexCart (2/2)

- Cartesian Implementation (Part 2 of 2)

ComplexCart.java

```
  // MUTATORS
  public void add(Complex c) {
    this.real += c.realpart();
    this.imag += c.imagpart();
  }

  public void minus(Complex c) {
    this.real -= c.realpart();
    this.imag -= c.imagpart();
  }

  public void times(Complex c) {
    double tempReal = real * c.realpart() - imag * c.imagpart();
    imag = real * c.imagpart() + imag * c.realpart();
    real = tempReal;
  }

 public String toString() {
    if (imag == 0) return (real + "");
    else if (imag < 0) return (real + "" + imag + "i");
    else return (real + "+" + imag + "i");
  }
}
```

Why can't we write the following?

```
if (imag == 0) return (real);
```

3 Example #2 : ComplexPolar (1/3)

- Polar Implementation (Part 1 of 3)

ComplexPolar.java

```
class ComplexPolar implements Complex {
  private double mag;  // magnitude
  private double ang;  // angle
  // CONSTRUCTOR
  public ComplexPolar(double m, double a) { mag = m; ang = a; }
  // ACCESSORS
  public double realpart() { return mag * Math.cos(ang); }
  public double imagpart() { return mag * Math.sin(ang); }
  public double mag() { return mag; }
  public double angle() { return ang; }
  // MUTATORS
  public void add(Complex c) {   // this = this + c
    double real = this.realpart() + c.realpart();
    double imag = this.imagpart() + c.imagpart();
    mag = Math.sqrt(real*real + imag*imag);
    if (real != 0) {
      if (real < 0) ang = (Math.PI + Math.atan(imag/real));
      else ang = Math.atan(imag/real);
    }
    else if (imag == 0) ang = 0;
    else if (imag > 0) ang = Math.PI/2;
    else ang = -Math.PI/2;
  }
```

3 Example #2 : ComplexPolar (2/3)

- Polar Implementation (Part 2 of 3)

ComplexPolar.java

```
public void minus(Complex c) {    // this = this - c
  double real = mag * Math.cos(ang) - c.realpart();
  double imag = mag * Math.sin(ang) - c.imagpart();
  mag = Math.sqrt(real*real + imag*imag);
  if (real != 0) {
    if (real < 0) ang = (Math.PI + Math.atan(imag/real));
    else ang = Math.atan(imag/real);
  }
  else if (imag == 0) ang = 0;
  else if (imag > 0) ang = Math.PI/2;
  else ang = -Math.PI/2;
}
```

3 Example #2 : ComplexPolar (3/3)

- Polar Implementation (Part 3 of 3)

ComplexPolar.java

```
  public void times(Complex c) {   // this = this * c
    mag *= c.mag();
    ang += c.angle();
  }

  public String toString() {
    if (imagpart() == 0)
      return (realpart() + "");
    else if (imagpart() < 0)
      return (realpart() + "" + imagpart() + "i");
    else
      return (realpart() + "+" + imagpart() + "i");
  }
}
```

3 Example #2 : TestComplex (1/3)

- Testing Complex class (Part 1 of 3)

TestComplex.java

```
class TestComplex {

  public static void main(String[] args) {
    // Testing ComplexCart
    Complex a = new ComplexCart(10.0, 12.0);
    Complex b = new ComplexCart(1.0, 2.0);

    System.out.println("Testing ComplexCart:");
    a.add(b);
    System.out.println("a=a+b is " + a);
    a.minus(b);
    System.out.println("a-b (which is the original a) is " + a);
    System.out.println("Angle of a is " + a.angle());
    a.times(b);
    System.out.println("a=a*b is " + a);
```

```
Testing ComplexCart:
a=a+b is 11.0+14.0i
a-b (which is the original a) is 10.0+12.0i
Angle of a is 0.8760580505981934
a=a*b is -14.0+32.0i
```

3 Example #2 : TestComplex (2/3)

- Testing Complex class (Part 2 of 3)

TestComplex.java

```
// Testing ComplexPolar
Complex c = new ComplexPolar(10.0, Math.PI/6.0);
Complex d = new ComplexPolar(1.0, Math.PI/3.0);

System.out.println("\nTesting ComplexPolar:");
System.out.println("c is " + c);
System.out.println("d is " + d);
c.add(d);
System.out.println("c=c+d is " + c);
c.minus(d);
System.out.println("c-d (which is the original c) is " + c);
c.times(d);
System.out.println("c=c*d is " + c);
```

```
Testing ComplexPolar:
c is 8.660254037844387+4.999999999999999i
d is 5.000000000000001+8.660254037844386i
c=c+d is 13.660254037844393+13.660254037844387i
c-d (which is ... c) is 8.660254037844393+5.000000000000002i
c=c*d is 2.8327694482399 2E-14+100.00000000000007i
```

3 Example #2 : TestComplex (3/3)

- Testing Complex class (Part 3 of 3)

TestComplex.java

```
    // Testing Combined
    System.out.println("\nTesting Combined:");
    System.out.println("a is " + a);
    System.out.println("d is " + d);
    a.minus(d);
    System.out.println("a=a-d is " + a);
    a.times(d);
    System.out.println("a=a*d is " + a);
    d.add(a);
    System.out.println("d=d+a is " + d);
    d.times(a);
    System.out.println("d=d*a is " + d);
  }
}
```

```
Testing Combined:
a is -14.0+32.0i
d is 5.000000000000001+8.660254037844386i
a=a-d is -19.0+23.339745962155614i
a=a*d is -297.1281292110204-47.84609690826524i
d=d+a is -292.12812921102045-39.18584287042089i
d=d*a is 84924.59488697552+25620.40696350589i
```