Object-Oriented Programming

Abstract Data Type (The Walls)

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy, and Dr. Low Kok Lim for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Course website address is changed to http://sakai.it.tdt.edu.vn
- Course codes cs1010, cs1020, cs2010 are placed by 501042, 501043, 502043 respectively.

Objectives

Understanding data abstraction

Defining ADT with Java Interface

Implementing data structure given a Java Interface

References

Book

- Chapter 4, pages 221 to 258

IT-TDT Sakai → 501043 website → Lessons

- http://sakai.it.tdt.edu.vn

Outline

1. Software Engineering Issues (Motivation)
   - 1.1 Loose coupling
   - 1.2 Data abstraction
2. Abstract Data Type
   - 2.1 Data Structure
   - 2.2 Understanding ADT
3. Java Interface
   - 3.1 Using Java interface to define ADT
   - 3.2 Complex Number Interface
   - 3.3 Complex ADT: Cartesian Implementation
   - 3.4 Complex ADT: Polar Implementation
4. Practice Exercises: Fraction as ADT

1 Software Engineering Issues

Motivation

1. Software Engineering Issues (1/5)

Program Design Principles

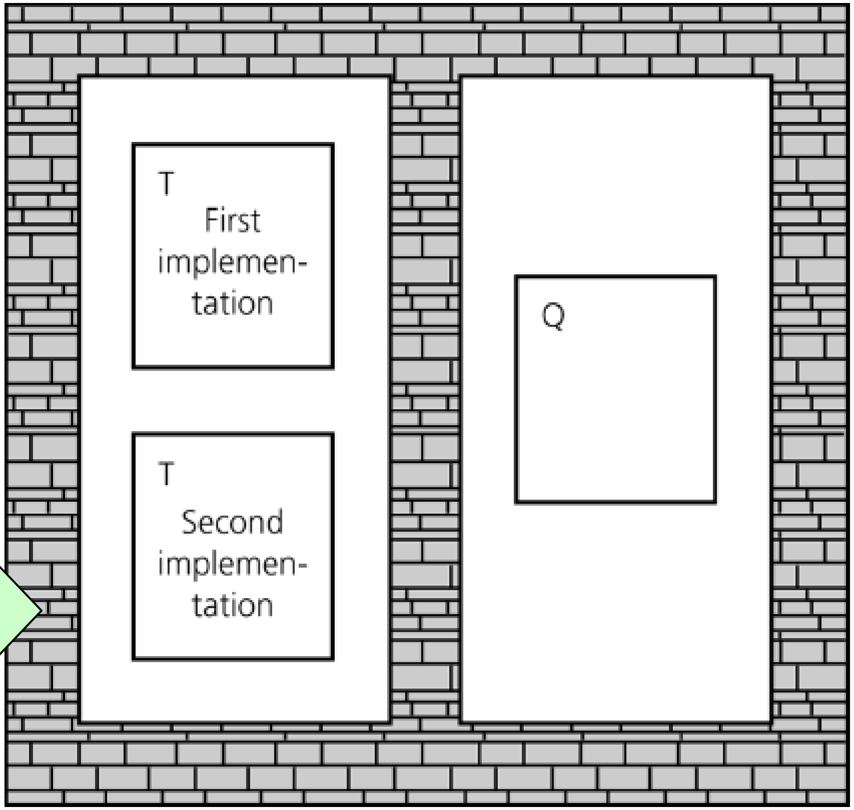
- **Abstraction**
  - Concentrate on what it can do and not how it does it
  - Eg: Use of Java Interface
- **Coupling**
  - Restrict interdependent relationship among classes to the minimum
- **Cohesion**
  - A class should be about a single entity only
  - There should be a clear logical grouping of all functionalities
- **Information Hiding**
  - Expose only necessary information to outside

1. Software Engineering Issues (2/5)

- 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 (depends on) *T*, and if the approach for performing *T* changes, class *Q* will not be affected.

Our textbook is called the "Walls & Mirrors". What are the walls?

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



1. Software Engineering Issues (3/5)

- Information Hiding is not complete isolation of the classes
  - Information released is on a **need-to-know** basis
  - Class *Q* does not know how class *T* does the work, but it needs to know how to invoke *T* and what *T* produces
    - E.g: The designers of the methods of **Math** and **Scanner** classes have hidden the details of the implementations of the methods from you, but provide enough information (the method headers and explanation) to allow you to use their methods
  - 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 (pre- and post-conditions)

1. Software Engineering Issues (4/5)

- Pre- and post-conditions (for documentation)
  - Pre-conditions
    - Conditions that must be true before a method is called
    - “This is what I expect from you”
    - The programmer is responsible for making sure that the pre-conditions are satisfied when calling the method
  - Post-conditions
    - Conditions that must be true after the method is completed
    - “This is what I promise to do for you”
  - Example

```
// Pre-cond: x >= 0
// Post-cond: Return the square root of x
public static double squareRoot(double x) {
  . . .
}
```

1. Software Engineering Issues (5/5)

- Information Hiding CAN also 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/methods 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

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 structures
  - 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; // defining a 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];
```

Abstract Data Type (ADT) (1/4)

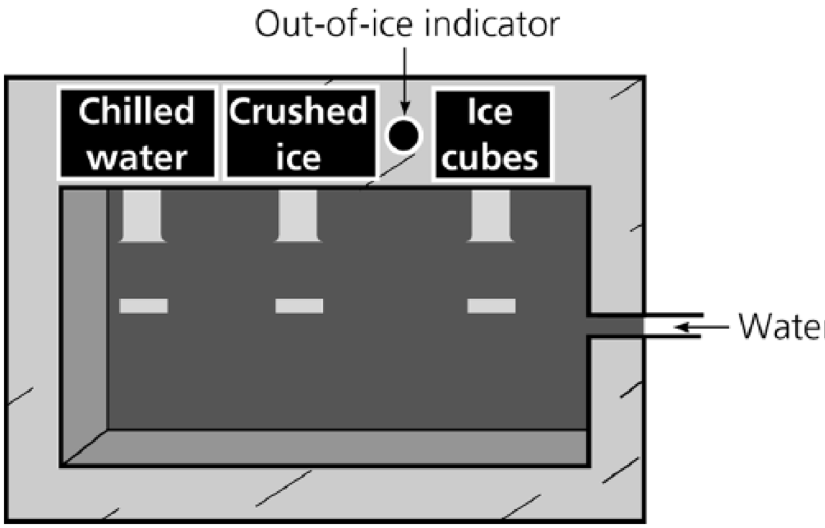
- An **ADT** is a collection of data together with a specification of a set of operations on the data
  - Specifications indicate **what** ADT operations do, ***not*** **how** to implement them
  - **Data structures** are part of an ADT's implementation

- 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

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.**

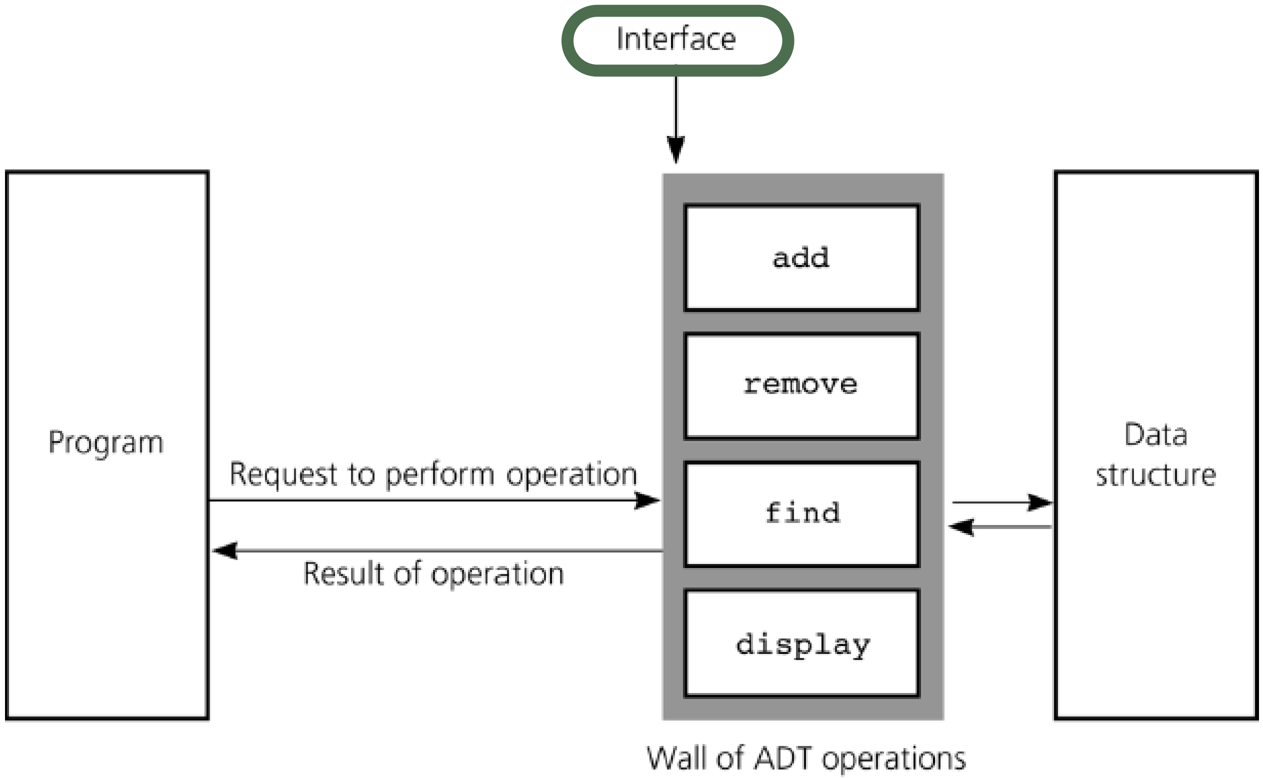


Crushed ice can be made in many ways.
We don't care how it was made

  - Using an ADT is like using a vending machine.

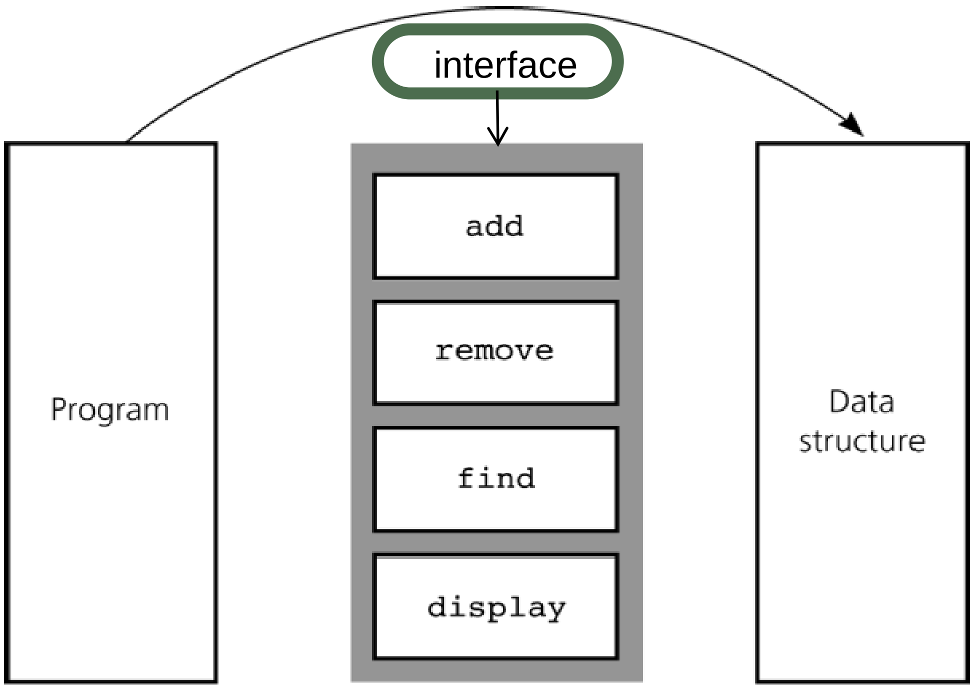
Abstract Data Type (ADT) (3/4)

- 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 on using the ADT



Abstract Data Type (ADT) (4/4)

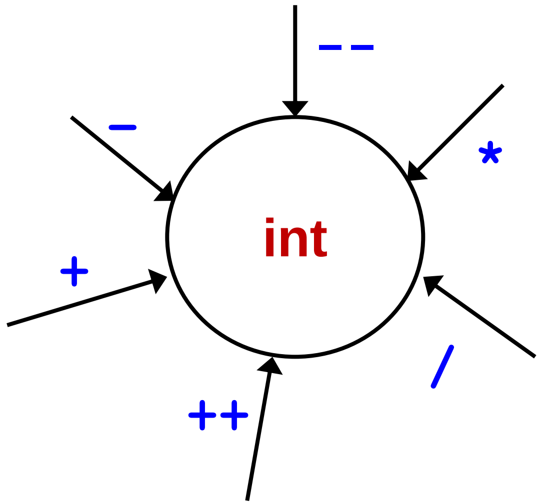
- An interface is what a program/module/class should understand on using the ADT
- The following <u>bypasses</u> the interface to access the data structure. This violates the wall of ADT operations.



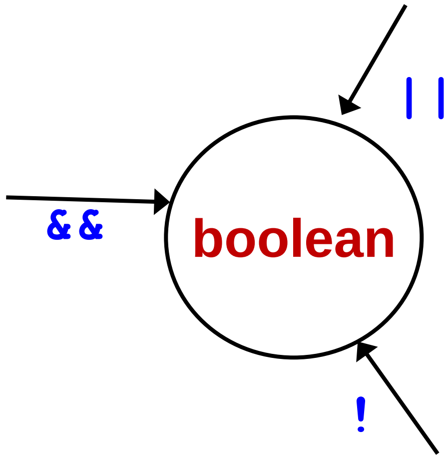
Wall of ADT operations

Eg: 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, double



int type with the operations (e.g.: --, /) defined on it.



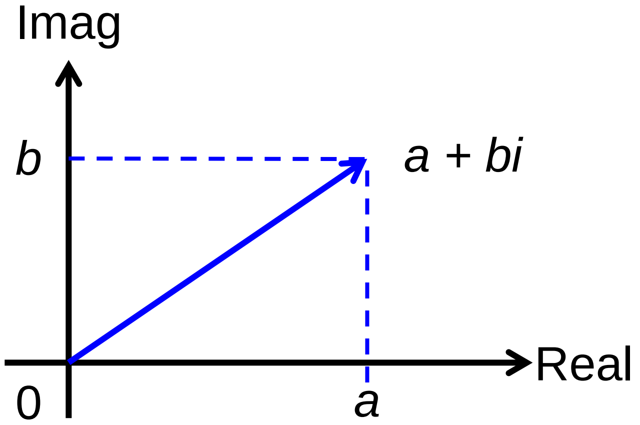
boolean type with the operations (e.g.: &&) defined on it.

Eg: Primitive Types as ADTs (2/2)

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

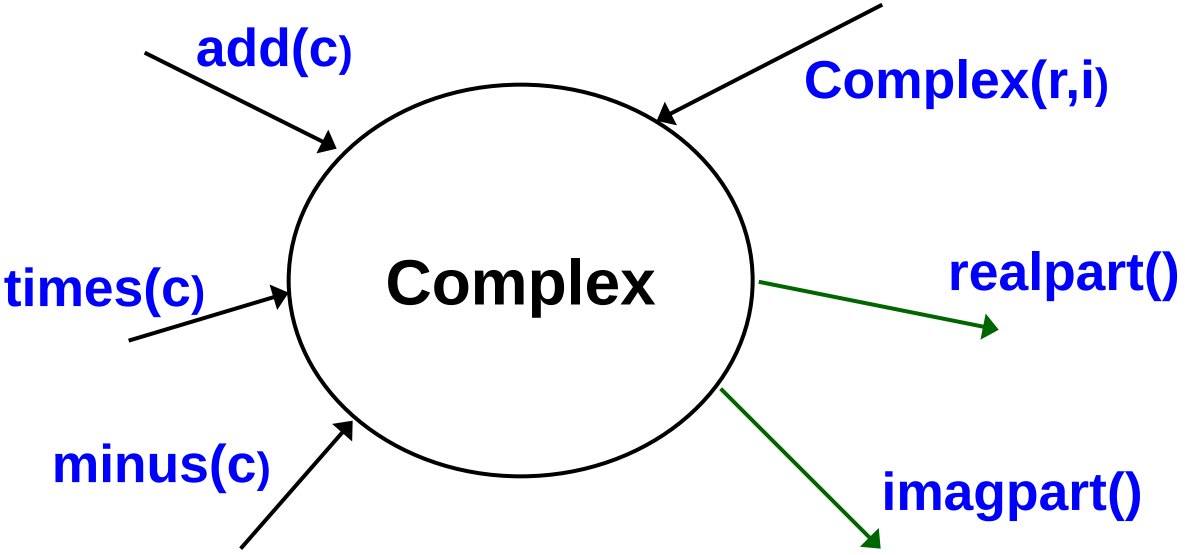
Eg: Complex Number as ADT (1/6)

- A **complex number** comprises a real part *a* and an imaginary part *b*, and is written as $a + bi$
- i is a value such that $i^2 = -1$.
- Examples: $12 + 3i$, $15 – 9i$, $-5 + 4i$, -23, $18i$
- A complex number can be visually represented as a pair of numbers (a, b) representing a vector on the two-dimensional complex plane (horizontal axis for real part, vertical axis for imaginary part)



Eg: Complex Number as ADT (2/6)

- User-defined data types can also be organized as ADTs
- Let's create a "Complex" ADT for complex numbers



Note: add(c) means to add complex number object c to "this" object. Likewise for times(c) and minus(c).

Eg: Complex Number as ADT (3/6)

- 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
```

Eg: Complex Number as ADT (4/6)

- One possible 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();
  }
}
```

(a + b*i*) + (c + d*i*)
= (a + c) + (b + d)*i*

(a + b*i*) – (c + d*i*)
= (a – c) + (b – d)*i*

(a + b*i*) × (c + d*i*)
= (ac – bd) + (ad + bc)*i*

Eg: Complex Number as ADT (5/6)

- Another possible implementation: Polar

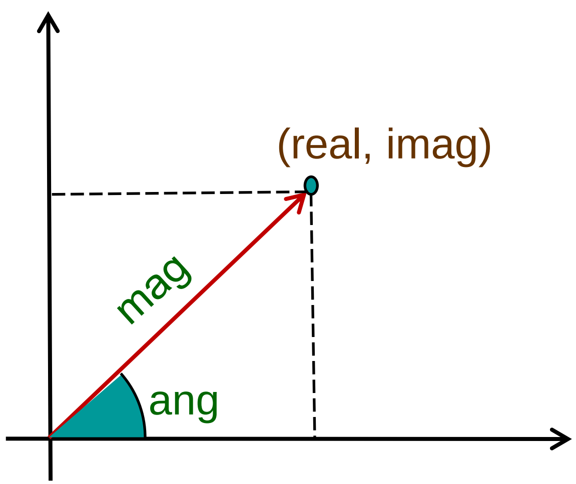
```
class Complex {
  private double ang;   // the angle of the vector
  private double mag;   // the magnitude of the vector
      :
      :
  public times(Complex c)  { // this = this * c
      ang += c.angle();
      mag *= c.mag();
  }
      :
      :
}
```

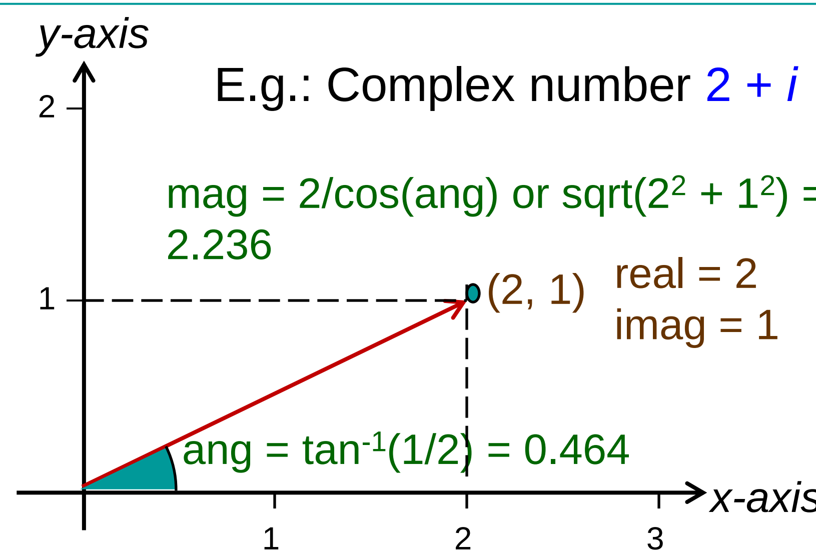
Eg: Complex Number as ADT (6/6)

- "Relationship" between Cartesian and Polar representations

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);
or mag = sqrt(real^2 + imag^2);





3 Java Interface

Specifying related methods

Java Interface

- Java interfaces provide a way to specify common behaviour for a set of (possibly 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
    - A Java interface is a group of related methods with empty bodies
  - It can have constant definitions (which are implicitly public static final)
- A class is said to implement the interface if it provides implementations for **ALL** the methods in the interface

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;
  }
}
```

Implementation of compareTo()

Example #2: Interface for Complex

E.g. Complex ADT interface

- anticipate both Cartesian and Polar implementations

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 Java 7 and earlier, methods in an interface only have **signatures** (headers) but <u>no implementation</u>
- However, Java 8 introduces "default methods" to interfaces. They provide default implementations which can be overridden by the implementing class.

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));
      else return Math.atan(imag/real);
    }
    else if (imag == 0) return 0;
    else if (imag > 0) return Math.PI/2;
    else return -Math.PI/2;
  }
```

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);
```

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;
  }
```

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;
}
```

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");
  }
}
```

Example #2: TestComplex (1/3)

- Testing Complex class (Part 1 of 3)

TestComplex.java

```
public 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
```

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 2.83276944823992E-14+100.00000000000007i
```

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 -292.1281292110204-39.18584287042089i
d=d*a is 84924.59488697552+25620.40696350589i
```

Java Interface

- Each interface is compiled into a separate bytecode file, just like a regular class
  - We cannot create an instance of an interface, but we can use an interface as a data type for a variable, or as a result of casting

```
public boolean equals (Object cl) {
  if (cl instanceof Complex) {
     Complex temp = (Complex) cl; // result of casting
     return (Math.abs(realpart() - temp.realpart()) < EPSILON
          && Math.abs(imagpart() - temp.imagpart()) < EPSILON);
   }
  return false;
}
```

Note: EPSILON is a very small value (actual value up to programmer), defined as a constant at the beginning of the class, e.g.:

```
public static final double EPSILON = 0.0000001;
```

4 Fraction as ADT

Practice Exercises

Fraction as ADT (1/3)

- We are going to view **Fraction** as an ADT, before we proceed to provide two implementations of Fraction
- Qn: What are the data members (attributes) of a fraction object (without going into its implementation)?
- Qn: What are the behaviours (methods) you want to provide for this class (without going into its implementation)?

Data members
Numerator
Denominator

Behaviors
Add
Minus
Times
Simplify

We will leave out divide for the moment

Fraction as ADT (2/3)

- How do we write an **Interface** for Fraction? Let's call it FractionI
  - You may refer to interface Complex for idea
  - But this time, we wants add(), minus(), times() and simplify() to return a fraction object

FractionI.java

```
public interface FractionI {
   public int getNumer();   //returns numerator part
   public int getDenom();   //returns denominator part
   public void setNumer(int numer);  //sets new numerator
   public void setDenom(int denom);  //sets new denominator

   public FractionI add(FractionI f);    //returns this + f
   public FractionI minus(FractionI f);  //returns this - f
   public FractionI times(FractionI f);  //returns this * f
   public FractionI simplify(); //returns this simplified
}
```

Fraction as ADT (3/3)

- Now, to implement this Fraction ADT, we can try 2 approaches
  - **Fraction**: Use 2 integer data members for numerator and denominator (you have done this in Practice Exercise #11)
    - We will do this in Practice Exercise #26
  - **FractionArr**: Use a 2-element integer array for numerator and denominator
    - We will do this in Practice Exercise #27
  - We want to add a toString() method and an equals() method as well

PracEx#26: TestFraction (1/2)

- To write Fraction.java to implementation the FractionI interface.
- The client program TestFraction.java is given

TestFraction.java

```
// To test out Fraction class
import java.util.*;
public class TestFraction {

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      System.out.print("Enter 1st fraction: ");
      int a = sc.nextInt();
      int b = sc.nextInt();
      FractionI f1 = new Fraction(a, b);

      System.out.print("Enter 2nd fraction: ");
      a = sc.nextInt();
      b = sc.nextInt();
      FractionI f2 = new Fraction(a, b);

      System.out.println("1st fraction is " + f1);
      System.out.println("2nd fraction is " + f2);
```

PracEx#26: TestFraction (2/2)

- To write Fraction.java, an implementation of FractionI interface.
- The client program TestFraction.java is given

TestFraction.java

```
    if (f1.equals(f2))
      System.out.println("The fractions are the same.");
    else
      System.out.println("The fractions are not the same.");

    FractionI sum = f1.add(f2);
    System.out.println("Sum is " + sum);

    FractionI diff = f1
    System.out.println(

    FractionI prod = f1
    System.out.println(
  }
}
```

```
Enter 1st fraction: 2 4
Enter 2nd fraction: 2 3
1st fraction is 2/4
2nd fraction is 2/3
The fractions are not the same.
Sum is 7/6
Difference is -1/6
Product is 1/3
```

PracEx#26: Fraction (1/2)

- Skeleton program for Fraction.java

Fraction.java

```
class Fraction implements FractionI {
  // Data members
  private int numer;
  private int denom;

  // Constructors
  public Fraction() { this(1,1); }

  public Fraction(int numer, int denom) {
    setNumer(numer);
    setDenom(denom);
  }

  // Accessors
  public int getNumer() { // fill in the code }
  public int getDenom() { // fill in the code }

  // Mutators
  public void setNumer(int numer) { // fill in the code }
  public void setDenom(int denom) { // fill in the code }
```

PracEx#26: Fraction (2/2)

Fraction.java

```
  // Returns greatest common divisor of a and b
  // private method as this is not accessible to clients
  private static int gcd(int a, int b) {
     int rem;
     while (b > 0) {
        rem = a%b;
        a = b;
        b = rem;
     }
     return a;
  }

  // Fill in the code for all the methods below
  public FractionI simplify() { // fill in the code }
  public FractionI add(FractionI f) { // fill in the code }
  public FractionI minus(FractionI f) { // fill in the code }
  public FractionI times(FractionI f) { // fill in the code }

  // Overriding methods toString() and equals()
  public String toString() { // fill in the code }
  public boolean equals() { // fill in the code }
}
```

PracEx#27: TestFractionArr

- To write FractionArr.java to implementation the FractionI interface.
- The client program TestFractionArr.java is given

TestFractionArr.java

```
// To test out FractionArr class
import java.util.*;
public class TestFractionArr {

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter 1st fraction: ");
    int a = sc.nextInt();
    int b = sc.nextInt();
    FractionI f1 = new FractionArr(a, b);

    System.out.print("Enter 2nd fraction: ");
    a = sc.nextInt();
    b = sc.nextInt();
    FractionI f2 = new FractionArr(a, b);

    // The rest of the code is the same as TestFraction.java
}
```

4. Pract. Ex.

PracEx#27: FractionArr

- Skeleton program for FractionArr.java

FractionArr.java

```
class FractionArr implements FractionI {
  // Data members
  private int[] members;

  // Constructors
  public FractionArr() { this(1,1); }

  public FractionArr(int numer, int denom) {
    members = new int[2];
    setNumer(numer);
    setDenom(denom);
  }

  // Accessors
  public int getNumer() { // fill in the code }
  public int getDenom() { // fill in the code }

  // Mutators
  public void setNumer(int numer) { // fill in the code }
  public void setDenom(int denom) { // fill in the code }

  // The rest are omitted here
}
```

Summary

- We learn about the need of data abstraction
- We learn about using Java Interface to define an ADT
- With this, we will learn and define various kinds of ADTs/data structures in subsequent lectures

End of file