Lecture 3

Abstract Data Types

Java Examples of ADT

Java Generics

Readings

- Chapter 4: ADTs, Classes and Interfaces,
  Pages 171-206

  Chapter 9: Inheritance, Dynamic Binding

  Pages 422-438

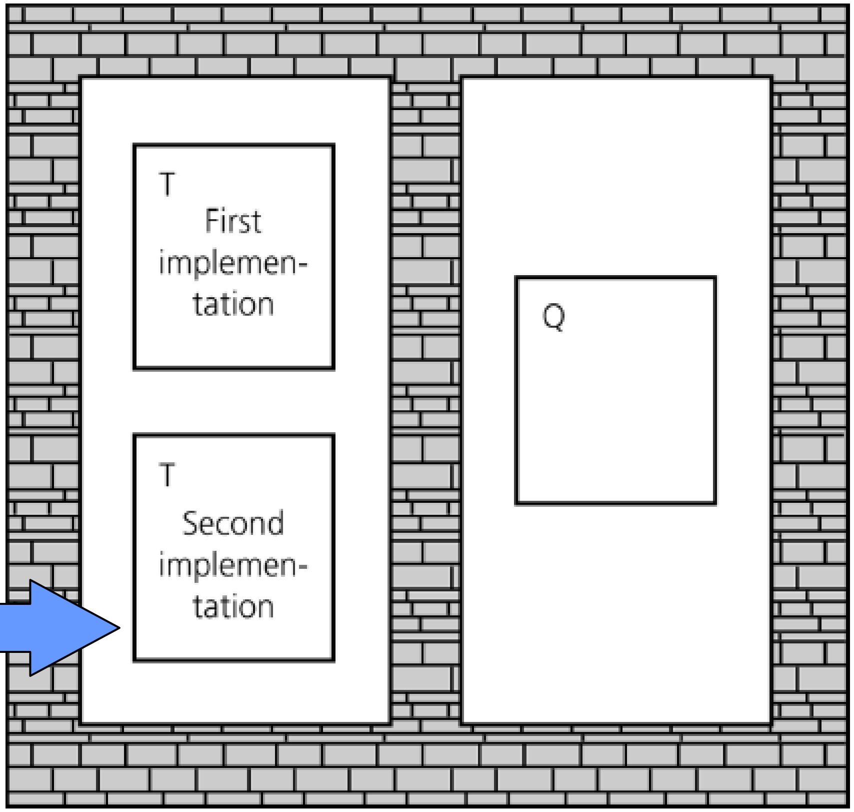
Recap: Principles of Software Engineering

- A modular program is easier to write, read and modify.
- Write specifications for each module before implementing it.
- Isolate the implementation details of a module from other modules.

Recap: Loose coupling

- Isolated tasks: the implementation of task T does not affect task Q
- Q does not know how task T is performed.
- Q must know what task T is and how to initiate it.

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



Data Abstraction

- When we talk about a program doing something, proper abstraction means we should decide what is done and not how
- We can do the same thing about the data used in the program
- Data Abstraction: What operations a data collection supports and not how it supports it.

Definition of a Data Structure

- is a construct that can be defined within a programming language to store a collection of data.
  - For example, arrays, which are built into Java, are data structures.
  - You can also invent other data structures. For example, you want a data structure to store both the names and salaries of a group of employees

Example: Personnel Data Structure

In Java, you can define

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

Example: Personnel Data Structure

Even better, we can use a class to describe employee:

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

Typical Operations on Data

- Add data to a data collection
- Remove data from a data collection
- Ask questions about the data in a data collection

The details of the operation, vary from application to application, but the overall theme is the **management of data**

Abstract Data Type (ADT)

- A collection of data together with 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.

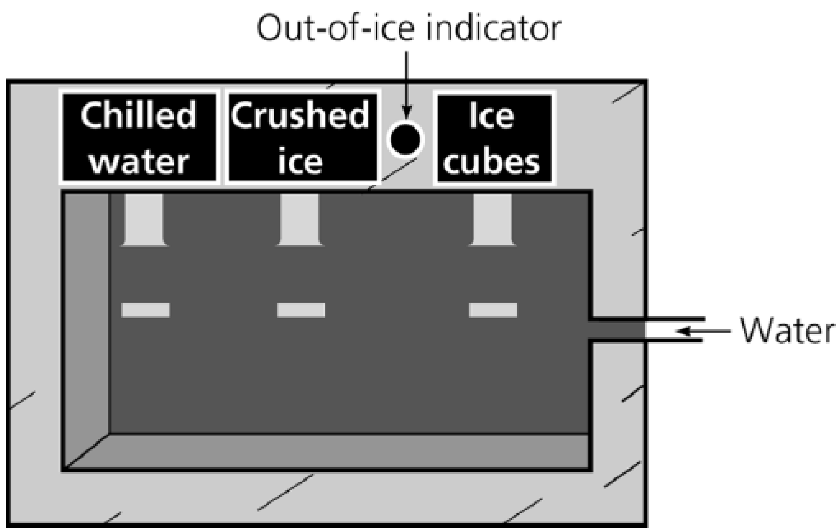
  - When a program needs data operations that are not directly supported by a language, you need an ADT.
  - You should first design the ADT by carefully specifying the operations before implementation.

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.

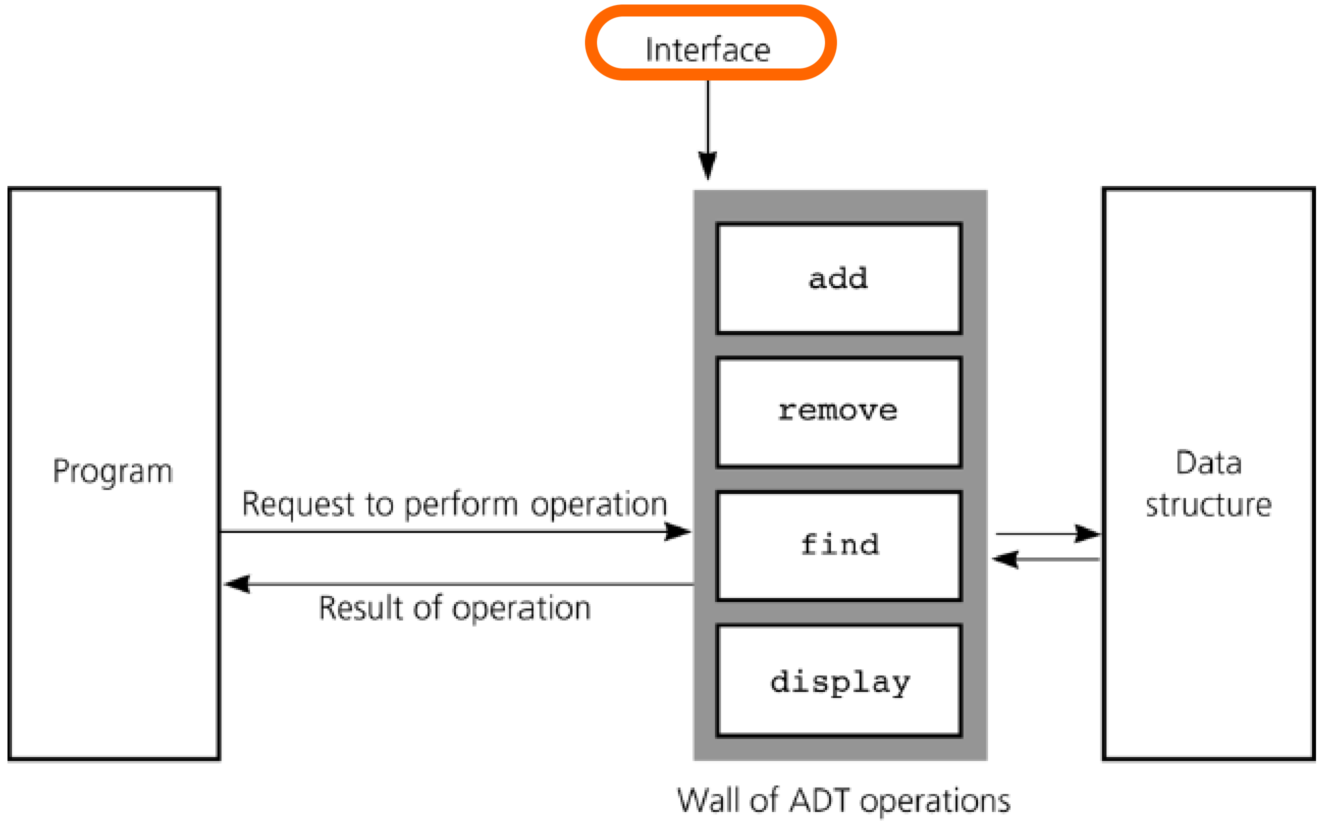


Crushed ice can be made in many different ways.
We don't care how it was made

- Using an ADT is like using a vending machine.

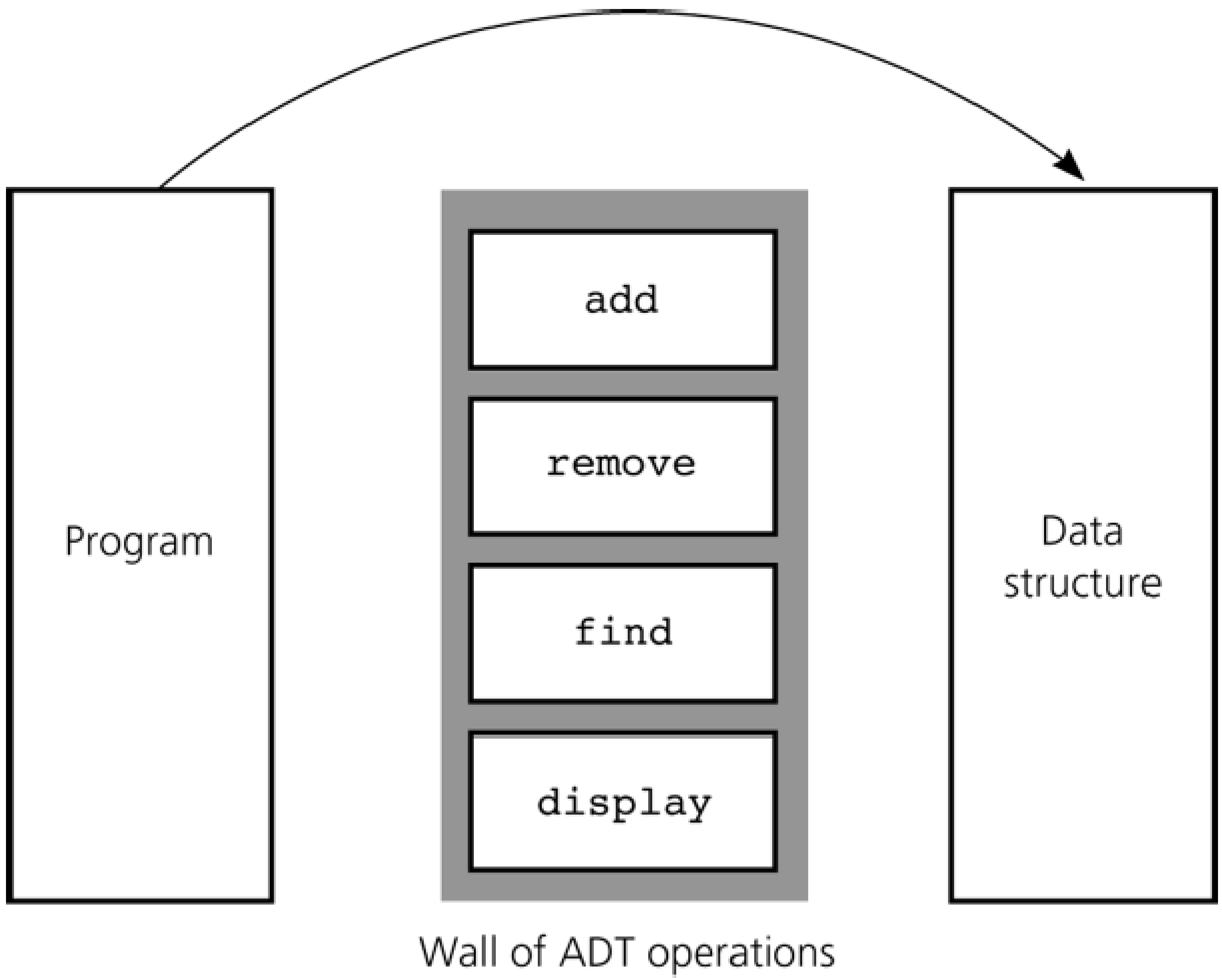
ADT operations provide access

- A wall of ADT operations isolates a data structure from the program that uses it



.02

Violating the wall of ADT operations



ADTs illustrated: Java examples

1. Primitive Types as ADTs
2. Complex Number ADT
3. Abstract Classes
4. Interfaces
5. Sphere’s ADT
6. Generic Objects
7. Case Study:
   Compare the implementation of List ADT using array and ArrayList <E>.

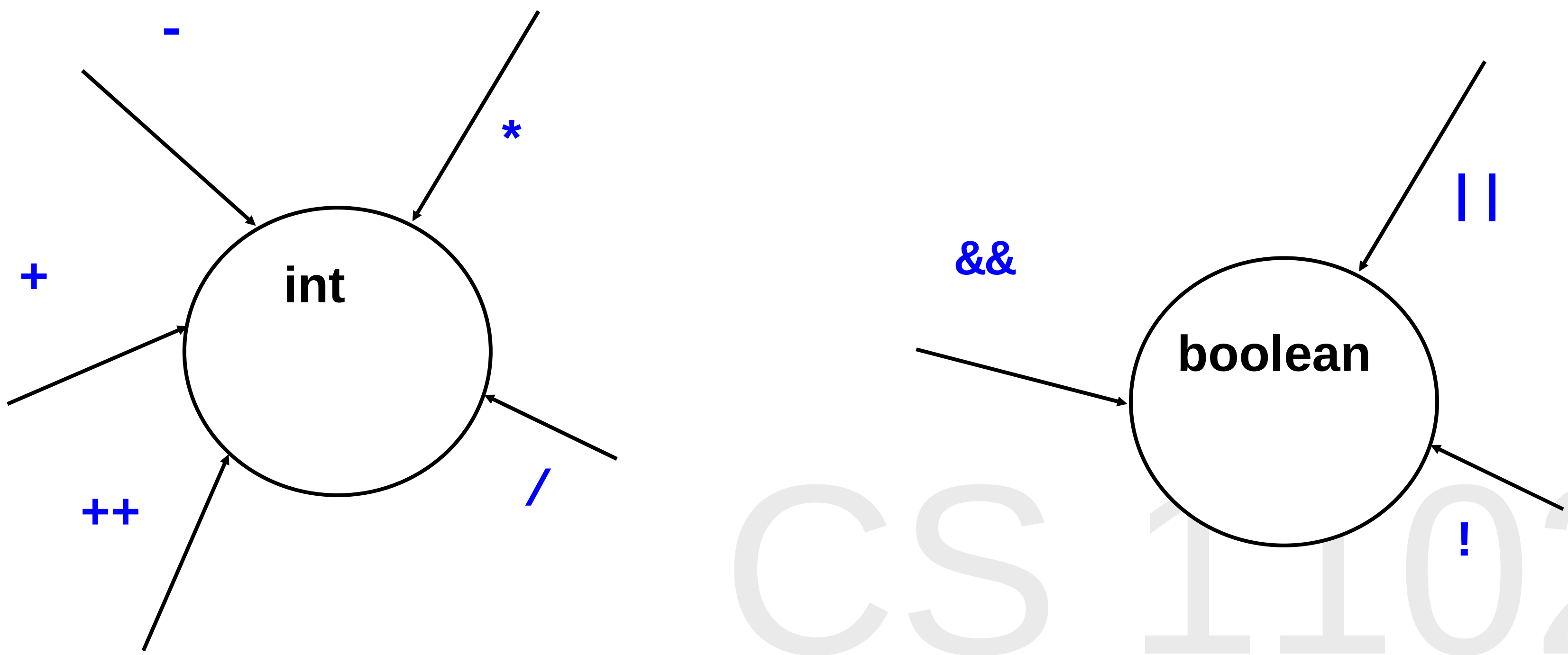
Object-oriented languages, such as Java, provide a way to enforce the wall of an ADT.

Encapsulation – one of OOP’s three fundamental principles – enables us to enforce the walls of an ADT.

Let’s see some examples

1. Primitive Types as ADTs

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

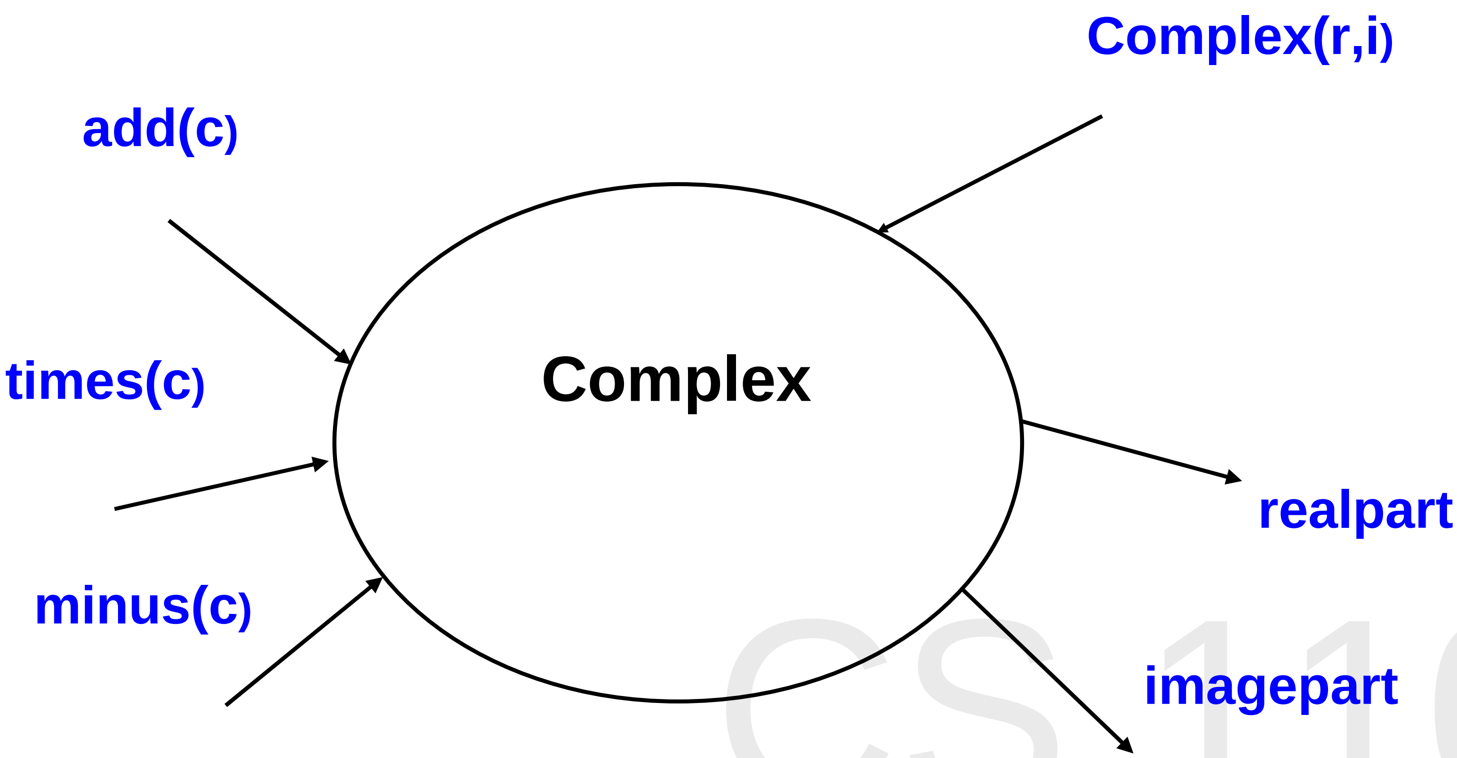


Primitive data types: Operations

- Broadly classified as:
  (here we use the array ADT in the example):
  - 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]...
  - Destroyers (to remove, terminate an object)
    - Java's very own garbage collector

2. Complex Number ADT

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



A possible Complex ADT Class

```
public 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
}
```

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
```

Cartesian Implementation of ADT

```
class Complex {
  private float real; private float image; // image here stands for imaginary
  // CONSTRUCTORS
  public Complex (float r, float i) { real = r; image = i; }
  // ACCESSORS
  public float realpart () { return real ; }            // returns this.real
  public float imagepart () { return image; }           // returns this.image
  // MUTATORS
  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;
  }
}
```

There's a problem here. What is it?
Hint: the formulas are correct!

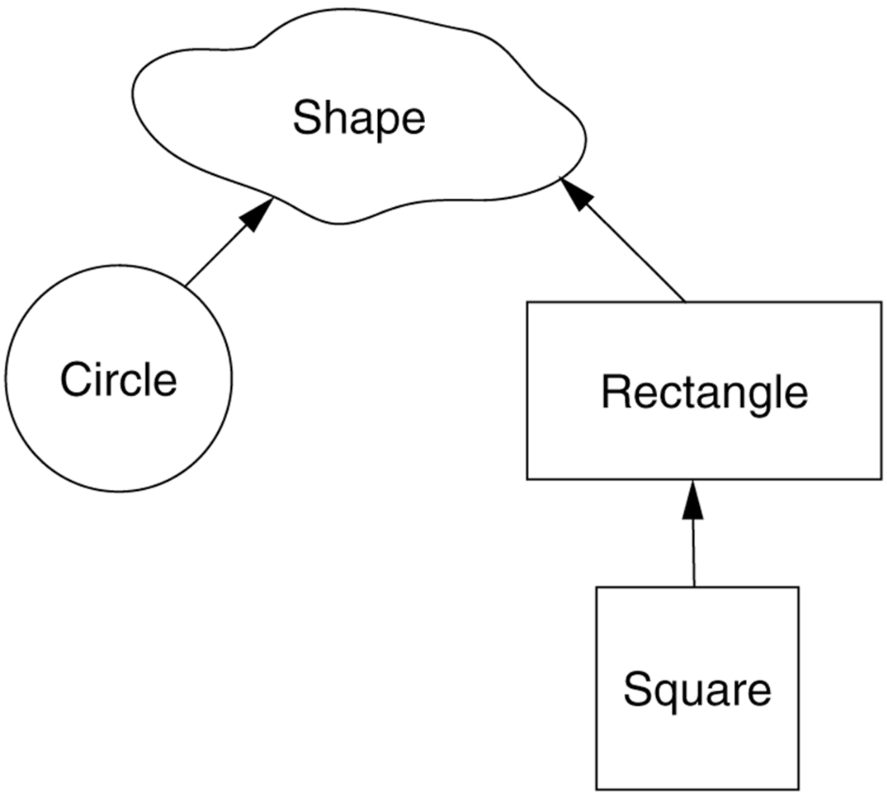
Polar Implementation

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

3. Abstract Classes

- An abstract class has no instances
- An abstract class is used only as the basis of subclasses
  - defines a minimum set of methods and data fields for its subclasses
- It can be used for ADT
  - allows further abstraction/ generalisation

Example: Shape Abstract Class



Example: Shape Abstract Class

- The Shape class exists simply as a common superclass for others
  - The Shape class and its area and perimeter methods are placeholders
  - These methods are not intended to be called directly

```
public class Shape {
  public double area() {
    return -1;
  }
  public double perimeter() {
    return -1;
  }
}
```

The Use of Abstract Methods and Classes

- An abstract method is a method that declares functionality that all derived class objects must eventually implement

```
public abstract class Shape {
  public abstract double area();
  public abstract double perimeter();

  public double semiperimeter() {
     return perimeter()/ 2;
  }
}
```

- An abstract class is a class that has at least one abstract method

Circle extending Shape

```
public class Circle extends Shape {
   public static final float PI =3.1415927;
   private double radius;

   public double area() {
    return PI * radius * radius;
   }

  public double perimeter() {
    return 2.0 * PI * radius;
  }
  // other methods
  …
}
```

4. Interfaces

- Java interfaces provide another mechanism for specifying common behavior for a set of (perhaps unrelated) classes
- It can be used for ADT
  - allows further abstraction/generalisation
  - use abstract methods in the interface

Interfaces (cont)

- The interface in Java is the ultimate abstract class
  - It consists only of public abstract methods and constants
    - which are implicitly public static final.

- A class is said to implement the interface if it provides definitions for all of the abstract methods in the interface

Interfaces (cont)

- It uses the keyword interface, not class
- It consists of a list of methods that are not implemented
- The methods defined in the interface must be implemented by the subclasses

```
// package in java.lang;

public interface Comparable <T> {
  int compareTo (T other);
}
```

Generics, coming up next

Implementing an interface

- Declaring that it implements the interface
- Defining implementations for all the interface methods

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

Interfaces

- 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, as the result of casting, etc.

Implementing several interfaces

- Sometimes it is necessary to derive a subclass from several classes, thus inheriting their data and methods.
  Java, however, does not allow multiple inheritance.

- The extends keyword allows only one parent class. With interfaces, we can achieve the effect close to that of multiple inheritance by implementing several interfaces.

```
public class Vector<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable,
Serializable
```

Complex number interface

```
public interface Complex {

   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
   public float radian ();                  // returns this.radian
   public float mag();                      // returns this.mag
   public void addPolar (Complex c);        // focus on this in the lecture

}
```

Complex ADT: Cartesian Implementation

```
public class ComplexCart implements Complex {
  private float real;
  private float image;

  // CONSTRUCTORS – create a new object
  public ComplexCart (float r, float i) { real = r; image = i; }

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

  public float rad () {
    if (real != 0) return (float) Math.atan (imag / real);
    else if (image > 0) return 3.14159/2; else return -3.14159/2;
  }

  public float mag () { return real * real + imag * imag; }
```

Complex ADT: Cartesian Implementation

```
// MUTATORS
public void add (Complex c)  {
  ComplexCart a = (ComplexCart) c;
  this.real = this.real + a.real;
  this.image = this.image + a.image;
}

public void minus (Complex c)   {
  ComplexCart a = (ComplexCart) c;
  this.real = this.real - a.real;
  this.image = this.image - a.image;
}

public void times (Complex c)    {
  ComplexCart a = (ComplexCart) c;
  this.real = this.real * a.real – this.image * a.image;
  this.image = this.real * a.image + this.image * a.real;
}
```

Complex ADT: Cartesian Implementation

```
  // MUTATORS
  public void addPolar (Complex c)  {
    ComplexPolar a = (ComplexPolar) c;
    float r = a.mag () * (float) Math.cos (a.rad ( ));
    float i = a.mag () * (float) math.sin (a.rad ( ));
    real += r; image += i;
  }
}
```

Complex ADT: Polar Implementation

```
class ComplexPolar implements Complex {
  private float mag;                    // magnitude
  private float rad;                    // angle

  // CONSTRUCTORS
  ComplexPolar (float m, float r) { mag = m; rad = r; }

  // ACCESSORS
  public float realpart () { return mag * (float)Math.cos(rad); }
  public float imagepart () { return mag * (float)Math.sin(rad); }
  public float rad () { return rad; }
  public float mag () { return mag; }
```

Complex ADT: Polar Implementation

```
// MUTATORS
public void add (Complex c)   {            // this = this + c
  ComplexPolar a = (ComplexPolar) c;
  float real =     mag * (float) Math.cos(rad) +
                   a.mag * (float) Math.cos(a.rad);
  float image =  mag * (float) Math.sin(rad) +
                   a.mag * (float) Math.sin(a.rad);
  mag = real*real + image*image;

  if (real != 0) rad = (float) Math.atan(image/real);
  else radian = 0;
}

//similar code for minus
```

Complex ADT: Polar Implementation

```
  public void times (Complex c)  {   // this = this * c
    ComplexPolar a = (ComplexPolar) c;
    mag *= a.mag;
    rad += a.rad;
  }

  public void addPolar (Complex c) {
    …
  }
}
```

Testing the Complex ADT

```
public class TestComplex {
  public static void main(String [] args) {
    Complex a = new ComplexCart((float)10.0, (float)12.0);
    Complex b = new ComplexCart((float)1.0, (float)2.0);
    a.add(b);
    System.out.println(a.realpart()+"+"+a.imagepart()+"i");

    Complex c = new ComplexPolar((float)10.0,(float)(Math.PI/6.0));
    Complex d = new ComplexPolar((float)10.0,(float)(Math.PI/3.0));
    c.times(d);
    System.out.println("magnitude="+c.mag()+",radian="+c.rad());

    a.addPolar(d);
    System.out.println(a.realpart()+"+"+a.imagepart()+"i");
  }
}
```

Interfaces vs. Abstract Classes

- Data
  - In an interface, all data are constants (keyword final is omitted)
  - An abstract class can have non-constant data fields.
- Methods
  - In an interface, all methods are not implemented
  - An abstract class can have concrete methods.
- Keyword abstract
  - In an interface, the keyword abstract in the method signature can be omitted
  - In an abstract class, it is needed for an abstract method.
- Inheritance
  - A class can implement multiple interfaces
  - A class can inherit only from one (abstract) class