

CS 3

Introduction to Software Engineering

4: Data Abstraction

Questions?

- Problem set due tonight/tomorrow, 4 a.m.
- Next one will be available by tomorrow.

- (I will explain about grading)

History

- First programming language (FORTRAN, mid-1950s) had no abstraction at all.
 - FORTRAN II (1958) had subroutines. (Procedural abstraction)
- 1960s: Procedural abstraction seen as important.
 - ALGOL; Pascal in 1970; C in 1972
 - Programs built out of procedures (which could have sub-procedures...)
- Mid-late 1970s: Programs built out of **modules**.
 - Module = type definition(s) + procedure definition(s).
 - Types are central; procedures in supporting roles.
 - Object-oriented paradigm incorporated this.
 - Ada, Modula-2, C++, ML, Java, ...

Subroutines : Procedural Abstraction ::
Modules (classes) : **Data Abstraction**

Data Abstraction

- A **data abstraction** consists of:
 - A **type**.
 - Some **operations** on that type.(And that's just what a Java class is!)
- Also known as an **abstract data type** or **ADT**.
- What's "abstract"?
 - Separation of code that **implements** the abstraction from code that **uses** the abstraction.
 - Only the implementing code needs to understand the representation.
 - Using code ("client" code) treats abstraction as a **black box**. Only needs to understand specifications of operations.

Classic Example

```
public final class Complex {
    private double a;
    private double b;

    public Complex(double a,double b) { this.a = a; this.b = b; }

    public double real() { return a; }
    public double imag() { return b; }
    public double mag() { return Math.sqrt(a*a+b*b); }
    public double ang() { return Math.atan2(a,b); }

    public Complex add(Complex other) {
        return new Complex(a+other.a,b+other.b);
    }

    public Complex mul(Complex other) {
        return new Complex(a*other.a-b*other.b,a*other.b+b*other.a);
    }

    ...
}
```

Note: doc comments omitted to fit class on slide.

Abstraction Function

- OK, so a Complex is a complex number—**which one?**
- Mapping from concrete objects to the things they represent is **abstraction function**.
- Every data abstraction has one; it should be documented.

- In the example:

```
/* The abstraction function is:
```

```
 *  $AF(c) = c.a + c.b*i$ 
```

```
*/
```

Different Representations

```
public final class Complex {
    private double a;
    private double b;

    public Complex(double a, double b) {
        this.a = a; this.b = b;
    }

    public double real() { return a; }
    public double imag() { return b; }
    public double mag() { return Math.sqrt(a*a+b*b); }
    public double ang() { return Math.atan2(a,b); }

    public Complex add(Complex other) {
        return new Complex(a+other.a,
                           b+other.b);
    }

    public Complex mul(Complex other) {
        return new Complex(
            a*other.a-b*other.b,
            a*other.b+b*other.a);
    }

    ...
}
```

```
public final class Complex {
    private double r;
    private double th;

    public Complex(double r, double th) {
        this.r = r; this.th = th;
    }

    public double real() { return r*Math.cos(th); }
    public double imag() { return r*Math.sin(th); }
    public double mag() { return r; }
    public double ang() { return th; }

    public Complex add(Complex other) {
        double a = real()+other.real();
        double b = imag()+other.imag();
        return new Complex(Math.sqrt(a*a+b*b),
                           Math.atan2(a,b));
    }

    public Complex mul(Complex other) {
        return new Complex(r*other.r, th+other.th);
    }
}
```

Different Representations

- This *also* represents complex numbers as pairs of doubles.
- But it's not the same!
- Correspondence between concrete rep and abstract concept is different.
- In other words, different abstraction function:
/* The abstraction function is:
 $AF(c) = c.r * \exp(i * c.th)$
*/

```
public final class Complex {
    private double r;
    private double th;

    public Complex(double r, double th) {
        this.r = r; this.th = th;
    }

    public double real() { return r * Math.cos(th); }
    public double imag() { return r * Math.sin(th); }
    public double mag() { return r; }
    public double ang() { return th; }

    public Complex add(Complex other) {
        double a = real() + other.real();
        double b = imag() + other.imag();
        return new Complex(Math.sqrt(a * a + b * b),
            Math.atan2(a, b));
    }

    public Complex mul(Complex other) {
        return new Complex(r * other.r, th + other.th);
    }
}
```

Different Representations

- Does *every* pair (r,th) represent a complex number?
- Might want always to have $r \geq 0$.
- That's a **representation invariant**.
/* The rep invariant is:
 c.r \geq 0.0
*/
- Need to make sure the invariant actually holds!
Check constructor argument.

```
public final class Complex {
    private double r;
    private double th;

    public Complex(double r, double th) {
        if (r < 0) { throw new IllegalArgumentException(); }
        this.r = r; this.th = th;
    }

    public double real() { return r*Math.cos(th); }
    public double imag(){ return r*Math.sin(th); }
    public double mag(){ return r; }
    public double ang(){ return th; }

    public Complex add(Complex other) {
        double a = real()+other.real();
        double b = imag()+other.imag();
        return new Complex(Math.sqrt(a*a+b*b),
            Math.atan2(a,b));
    }

    public Complex mul(Complex other) {
        return new Complex(r*other.r,th+other.th);
    }
}
```

Equality

- For many types, need to ask:
Is it possible for two separate objects to represent “the same” value?
- Answer for Complex: Of course.
Every call to `new Complex(0,0)` produces a separate object, all representing “0”.
- Abstraction not complete until we give programmers a working **equals** method:

```
/** Equality test for Cartesian complex numbers. */  
public boolean equals(Object o) {  
    if (!(o instanceof Complex)) { return false; }  
    Complex z = (Complex) o;  
    return a == z.a && b == z.b;  
}
```

Equality

- What should it look like in the polar version?

Equality

- What should it look like in the polar version?

$$re^{i\theta} = se^{i\varphi} \text{ iff } r = s = 0$$

or $r = s$ and $\varphi - \theta = 2k\pi$
for an integer k .

- First case is easy; what about the part with θ and φ ?

Equality

- What should it look like in the polar version?

$$re^{i\theta} = se^{i\varphi} \text{ iff } r = s = 0$$

or $r = s$ and $\varphi - \theta = 2k\pi$
for an integer k .

- First case is easy; what about the part with θ and φ ?
- Idea: if we require $0 \leq \theta < 2\pi$, can just check $\theta = \varphi$.

/* The rep invariant is:

```
c.r >= 0 && 0 <= c.th < 2*Math.PI
```

```
*/
```

Polar Equality

```
/** Equality test for polar complex numbers.  
    Note that it relies on the strong rep invariant for  
    its correctness.  
*/  
public boolean equals(Object o) {  
    if (!(o instanceof Complex)) { return false; }  
    Complex z = (Complex) o;  
    if (r == z.r) {  
        if (r == 0) { return true; }  
        return (th == z.th);  
    }  
    return false;  
}
```

Methods All Objects Have

- `boolean equals(Object o)`
 - Returns true if o is “the same as” this.
 - Steps:
 - Check (`o instanceof ThisClass`), return false if not.
 - Cast o to `ThisClass`.
 - Compare significant fields.
If any don’t match, return false.
 - Otherwise, return true.
 - When is an object “the same as” another?
 - Immutable objects: iff all “significant” fields match.
 - Mutable objects: if they are “==” to each other, otherwise almost never. (Default behavior.)

Methods All Objects Have

- `int hashCode()`
 - Returns an integer. Used by Java library Hashtable classes.
 - Should not change for life of object.
 - Rule: if `x.equals(y)`, then `x.hashCode()==y.hashCode()`.
 - Therefore: if you override `equals`, override `hashCode`!

 - Should be fast to compute.
 - Un-“`equals()`” objects should be unlikely to have the same `hashCode`.

Methods All Objects Have

- `int hashCode()`
 - Returns an integer. Used by Java library Hashtable classes.
 - Should not change for life of object.
 - Rule: if `x.equals(y)`, then `x.hashCode()==y.hashCode()`.
 - Therefore: if you override `equals`, override `hashCode`!

 - Should be fast to compute.
 - Un-“`equals()`” objects should be unlikely to have the same `hashCode`.

- For Complex:*

```
public int hashCode() {  
    int result = 17;  
    long abits = Double.doubleToLongBits(a);  
    result = 37*result + (abits ^ (abits>>>32));  
    long bbits = Double.doubleToLongBits(b);  
    result = 37*result + (bbits ^ (bbits>>>32));  
    return result;  
}
```

*Following a recipe from Joshua Bloch, *Effective Java* (Addison-Wesley, 2001).

Methods All Objects Have

- String toString()
 - Returns a String representing/describing this.
 - Used when you System.out.print() an object, or concatenate an object with a String.
 - Should contain all interesting information about the object.
 - But should not be the *only* way to get at this information!

```
public String toString() {  
    if (b < 0) {  
        return "" + a + "-" + (-b) + "i";  
    }  
    return "" + a + "+" + b + "i";  
}
```

Modularity

- Data abstractions should provide *locality* and *modifiability*.
 - Locality: Implement abstraction independently of client code.
 - Modifiability: Can *change* implementation without affecting client code.
- Many (most?) abstractions have more than one possible implementation.
 - Example: Stack as array or linked list;
Dictionary as list / sorted list / tree / hash table;
Complex numbers in rectangular or polar coordinates.
- Clients should not need to know implementation details.
 - Clients should not *be able* to know implementation details.
- Can change to a different representation and/or re-implement operations without affecting existing clients.
- Complex Demo.