PURDUE UNIVERSITY

CS18000: Problem Solving And Object-Oriented Programming

Types as Data Abstraction

7 February 2011

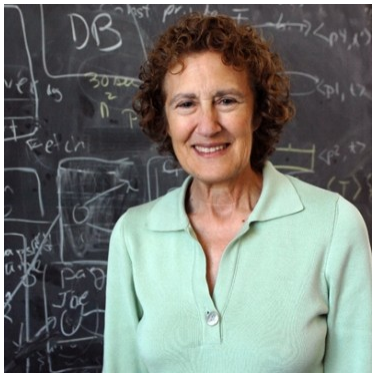
Prof. Chris Clifton

COMPUTER SCIENCES

Abstraction: How Programming Scales

- Functional Abstraction
  - Clear specification of what a function does
  - Don't sweat the details of *How* something is done
- Data Abstraction
  - Clear specification of what something *is*
  - Don't worry about how it is represented

Know who this is?



- Barbara Liskov
  - 2008 ACM Turing Award winner
  - "For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing."

Data Abstraction: Types

- char ch = 'a'
- How is ch *really* stored in the computer?
  1. a
  2. 97
  3. 01100001
  4. None of the above
- The char data type is a data abstraction

What makes a data type?

- Semantics: *What* is being represented
  - not how
- What we can do with it
  - Functional abstractions
- Constraints
  - Can all operations be applied to any object of the type?
  - Can an object be modified?
- Special values
  - constants

Primitive Types as Data Abstractions

Primitive Type

- Integers
  - byte, short, int, long
- Rationals
  - float, double
- Boolean
  - boolean
- Character
  - char

Class

- Number
  - Byte, Short, Integer, Long
  - BigInteger
- Number
  - Float, Double
- Boolean
- Character

Primitive Type vs. Class

- Primitive types are “special”
  - literals (constant values)
  - infix operators (+, -)
  - …
- Classes used to generate objects
  - double 3. vs. object of type Double containing the value 3.
- *For now, big issue is equality test*
  - a.equals(b) vs. a == b
  - == : variables a and b refer to the same object
  - *Two different objects can have same value!*

Type: (semi)Formal Definition

- Domain: Set of possible values
  - Infinite?
- Operations
  - Constructor
  - Field Access
  - Methods – Accessor, Mutator
- Invariants
  - Must be true at completion of every operation

Type: Implementation as Class

- Representation
  - Must represent all possible domain values
- Operations
  - Define Constructor(s)
  - Define methods
- Invariants
  - Ensure invariants hold after execution of Constructor(s), methods

Field Access vs. Accessor Methods

- Point p = new point(3.0,4.0);
- We want the horizontal axis:
  - x = p.x;
  - x = p.getx();
- Which is better? Why?
- What if we wanted:
  - r = p.radial;
  - theta = p.angle;

Class example: Point

```
public class Point {
    private double x; \\ Horizontal
      axis
    private double y; \\ Vertical axis
    public Point ( double px; double
      py ) {
        x = px;
        y = py;
    }
    public double getX() {
        return x;
    }
    public void setX(double x) {
      this.x = x;
    }
                                    …
                                   }
```

```
public double getR() {
      return Math.sqrt(x*x+y*y);
}
public void setTheta(Angle
   theta) {
      double r = getR();
      x = r*theta.cos();
      y = r*theta.sin();
}

public Point (double r, Angle
   theta) {
      this(r *theta.cos(), r*theta.sin());
}
```


Class example: Point

```
public class Point {
    private double x; \\ Horizontal
      axis
    private double y; \\ Vertical axis
    public Point ( double px; double
      py ) {
        x = px;
        y = py;
    }
    public double getX() {
        return x;
    }
    public void setX(double x) {
      this.x = x;
    }
                                    …
                                   }
```

```
public double getR() {
      return Math.sqrt(x*x+y*y);
}
public void setTheta(Angle
   theta) {
      double r = getR();
      x = r*theta.cos();
      y = r*theta.sin();
}

public Point (double r, Angle
   theta) {
      this(r *theta.cos(), r*theta.sin());
}
```

(name) Overloading

- Two methods named “Point”
  - Which is which?
- Signature: Method identified by
  - type (class),
  - name, and
  - *type/number of arguments*
- Can do for any method
  - Use sparingly, can be confusing

Referring to Self: this

- this – object of class method is in
  - Specifically, the instance that was invoked
  - In getX(), return x;
    - x is shorthand for this.x;
- Allows parameters, instance variables to have same name
  - Makes interface (abstraction) cleaner

Calling methods in methods

- Methods in a class can be called from other methods
  - But be careful
  - Don't call from constructor
    - Except other constructor
    - Or static method
  - *static methods can be called anytime*
    - But can't access instance variables
- Suggestion: Think Invariants
  - Method (may) assume valid instance when called
  - If you are modifying instance, does this hold?
    - If not, may not be safe

Classes: Some Confusing Keywords

Access

- public
  - Part of "abstraction"
- private
  - Only accessible inside the class
- (default)
  - The same *package*
- protected
  - package as well as inheriting classes (we'll cover this later)

Modifiers

- static
  - Method: can be invoked without a valid instance
  - Variable: Class variable
    - Same variable shared among class instances
- final
  - Must be initialized
  - *Typically doesn't change*
- transient
  - Not saved if instance "stored"
- volatile
  - Used with threads to enforce consistency

Goal: Separate Abstraction and Implementation

- Using an Abstract Data Type
  - Know *what* it is / *what* it does
  - *How* isn't important
- Once defined, can use
  - Doesn't need to be implemented
  - *(okay, we can't run the program – but we can write a program using it)*

Abstract Data Type: (semi)Formal Definition

- Domain: Set of possible values
- Operations
  - Constructor
  - Field Access
  - Methods – Accessor, Mutator
- Invariants
  - Must be true at completion of every operation

Abstract Data Type: Definition

- Challenge: How do we know we are using the type correctly?
  - May not even be written yet
  - Can we compile our program?
  - Or do we get obscure run-time errors?
- Solution: Define "unimplemented" class
  - **interface** for class

Java interface

- Defines an (abstract) data type
  - Javaspeak: *reference* type
- Declares
  - Constants (static final variables)
  - Method type, name, and parameters
  - Assumed to be public
- No constructor(s)
  - Cannot construct an instance of an interface

Interface Example

```
interface Queue {
    // Constants
    Student EMPTYRESULT =
       null;
    // Methods
    void enqueue ( Person s );
    Person dequeue();
    boolean isEmpty();
}
```

- Constants implicitly public static final
  - Could say "public static final Person EMPTYRESULT", but considered poor style
- Methods implicitly public abstract
  - Again, poor style to say it
  - *Cannot* be static
    - Why?

Using an interface

- Can define variables of type
  - Queue q;
  - Cannot create an instance, though
  - *In the above, q has the value null*
- Use as parameters

```
void bankTeller(Queue q) {
    while ( ! q.isEmpty() ) {
        manageCustomer(q.dequeue)
    }
```

Implementing an interface

- Implement *all* methods defined in the interface
  - May use constants, methods, and even the interface itself
- Gives a normal class
  - Can create instances
- Instance of implementation can be given where interface used
  - variables, parameters

Implementation Example

```
class StupidQueue implements
    Queue {
    // Instance variables
    Person q[ ];
    int head, tail;
    // Constructor
    public StupidQueue() {
        head = 0;
        tail = 0;
        q = new Person[50];
        q[head] = EMPTYRESULT;
        }
    // Methods
    public void enqueue ( Person item ) {
        q[tail] = item;
        tail += 1;
        q[tail] = EMPTYRESULT;
    }
```

- Define representation of data
  - instance variables
- Declare and implement all methods in interface
  - Must be public

Implementation Example: Complete

```
class StupidQueue implements
    Queue {
    // Instance variables
    Person q[ ];
    int head = 0, tail = 0;
    // Constructor
    public StupidQueue() {
        q = new Person[50];
        q[head] = EMPTYRESULT;
        }
    // Methods
    public void enqueue ( Person item
        ) {
        q[tail] = item;
        q[++tail] = EMPTYRESULT;
        }
    Person dequeue() {
        return q[head++];
        }
    boolean isEmpty() {
        return q[head] ==
        EMPTYRESULT;
        }
}
```


Implementation Example: Additional Methods

```
class StupidQueue implements
    Queue {
    // Additional constants
    static final int MAXSIZE = 50;
    // Instance variables
    String q[] = new Person[MAXSIZE];
    int head = 0, tail = 0;
    // Constructor
    public StupidQueue() {
        q[head] = EMPTYRESULT;
        }
    // Methods
    public void enqueue ( Person item ) {
        if (! isFull()) {
            q[tail] = item;
            q[++tail] = EMPTYRESULT;
            }
        }
    public Person dequeue() {
        if ( isEmpty() )
            return EMPTYRESULT;
        else
            return q[head++];
        }

    public boolean isEmpty() {
        return head >= tail;
        }

    public boolean isFull() {
        return tail >= MAXSIZE;
        }
}
```