PURDUE UNIVERSITY

CS18000: Problem Solving And Object-Oriented Programming

Data Abstraction: Inheritance

7 March 2011

Prof. Chris Clifton

COMPUTER SCIENCES

Data Abstraction Continued

- Abstract data type provides
  - Well-defined interface
  - Separation of specification and representation
  - Ability to use (*what*) without worrying about *how*
- But what if an existing Abstract Data Type isn't enough?
  - *Do we have to start from scratch?*

Solution 1: Extend the Abstraction

- We saw this last Wednesday
  - Abstraction didn't have isFull()
  - Implementation did
- Abstraction specifies what is necessary
  - *not everything possible*
- But what if we like the existing implementation?
  - It just doesn't go far enough

Solution 2: Use the abstraction

- We can use one abstraction when creating an instance of another
  - *Even if both implement the same abstraction*
- Nothing special about this
- Example: Paired Queue
  - StupidQueue runs out
  - Solution: When full, use another StupidQueue

PairedQueue

```
class PairedQueue implements Queue {

  StupidQueue first, second;

  public PairedQueue () {
    first = new StupidQueue();
    second = null;
  }

  public void enqueue ( String item ) {
    if ( first.isFull() ) {
      if ( second == null )
        second = new StupidQueue();
      second.enqueue(item);
    } else {
      first.enqueue(item);
    }
  }

  public String dequeue () {
    String result = first.dequeue();
    if (first.isEmpty() && second != null) {
      first = second;
      second = null;
    }
    return result;
  }

  public boolean isEmpty() {
    return first.isEmpty();
  }
}
```

PairedQueue must implement Queue to use a StupidQueue!

A. True
B. False

Why does enqueue() check if second is null?

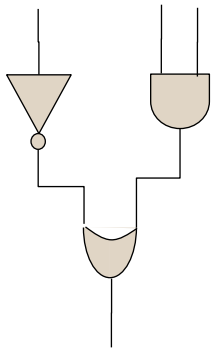
A. Can't create a new StupidQueue if second already holds one
B. If not null, we've already created the second one
C. There can't be more than one instance of the class StupidQueue at a time

Solution 3: *Extend* the implementation

- Inheritance: Define a new class that *includes* an existing class
  1. Same interface (just like using an Interface)
  2. Same methods
  3. Same representation
- Three separate concepts
  – But unfortunately not independent

Running Example: Boolean Circuits

```
public class Gate {
    String name;
    public Gate ( String name )
      {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public boolean getValue() {
        return false;
    }
}
```



Extending the Interface

```
Class UnaryOperator extends Gate {
    public UnaryOperator(String name) {
        }
    public void setInput(Gate input) {
        }
    public Gate getInput() { }
    }
```

Extending the Interface

- Child class must support all features of parent
  - Just like **implement**ing an interface
  - *Key difference: Some features already implemented*
- Can be used anywhere parent class can be used
  - *Just like an interface*
- Can define new fields and methods
  - But can't be used if the type is the parent class
  - *Just like an interface*

Extending the Methods

```
Class Not extends UnaryOperator{
    public Not() { super("Not"); }
    public boolean getValue() {
        return !(getInput().getValue());
        }
    }
```

Extending the Methods

- Overriding methods
  - If instance of child, use child (even if you think you have parent)

    Gate g = new Not();

    g.getValue() will use method from Not class
- Caveat: Class methods (static) not overridden
  - The one you get depends on the *type* of object

Using methods from Parent

- Non-overridden methods easy
  - Just use – example in Not
- Overridden methods accessed using **super** keyword
  - kind of like **this**
- Constructor of parent *must* be called in constructor
  - Exception: Default parent constructor (no arguments) called automatically before child constructor if you don't

Extending the Representation

- Child class gets all fields of parent class
  - Can access those not declared private or protected
  - Best to access through parent methods
- Can add new fields
  - Also hide existing by using same name
    - Bad idea – works like class methods: *Which you get depends on type*

Extending the Interface

```
Class UnaryOperator extends Gate {
    private Gate input;
    public UnaryOperator(String name) {
        super (name);
        }
    public void setInput(Gate input) {
        this.input = input;
        }
    public Gate getInput() { return input; }
    }
```

PURDUE UNIVERSITY

CS18000: Problem Solving And Object-Oriented Programming

Data Abstraction: Generics

9 March 2011

Prof. Chris Clifton

COMPUTER SCIENCES

Announcements:

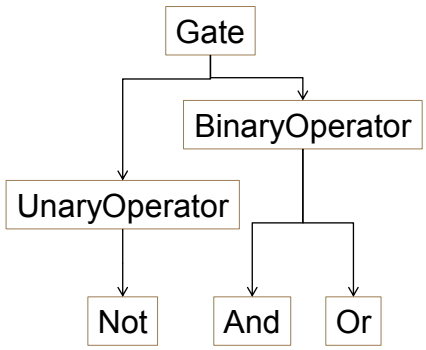
- Exam 1 scores and solution set out
  - I'll send more on how to interpret what you see once Project 2 scores are posted
  - Don't think too much about your score until you see this email
- No more labs this week
- Project 3 do soon

Putting it all together: Inheritance

- Three concepts:
  1. Inherit the interface
  2. Inherit the methods
  3. Inherit the representation
- Use inheritance (extends) when you want all three
  - If you just want interface, use interface/implements
  - If you just want methods, create as an instance variable

Getting it Right: ISA

- UnaryOperator ISA Gate
- Not ISA UnaryOperator
- GroceryLine ISA Queue



Putting it all together: Inheritance

- Three concepts:
  1. Inherit the interface
  2. Inherit the methods
  3. Inherit the representation
- Today: Mechanics and Caveats
  - How to use inheritance
  - How to *mis*use inheritance

Idea 1: Overriding

- Two implementations

```
power(BinaryOperator g) { /* high power */ }
power(UnaryOperator g) { /* low power */ }
Gate testg = new UnaryOperator();
power(testg);
```

- Will this work?
  - Unfortunately, no
  - Call based on (declared) type of argument
  - *There is no power(Gate g)*

A: Yes
B: No
C: Can't Tell

Idea 2: Cast

- Two implementations

```
power(BinaryOperator g) { /* high power */ }
power(UnaryOperator g) { /* low power */ }
Gate g = new UnaryOperator();
power((UnaryOperator) g);
```

- Will this work?
  - Close, but not quite
  - What if g was a BinaryOperator?

Solution: Ask an object it's Class

- Class getClass() defined for all objects
  - Every class (automatically) inherits from Class object
- Class has several useful methods
  - String getName();
  - Class getSuperclass();
  - boolean isInstance(Object obj);

```
if(g.getClass().getName().equals("UnaryOperator"))
    power( (UnaryOperator) g );
```

Caveats

- getClass() can be used as a crutch for bad design
  - It should be used only when you can't find a better way
- Try to compare Class objects, not names
  - In a large system, same name can be used multiple places (class defined inside a class)

Solution: Generics

- What if we want to support many types?
  - But only one at a time
- Generics: Parameter attached to a type
  - Queue<Integer>, Queue<String>
  - instead of IntegerQueue, StringQueue
  - One class
- (Messy) alternative: ObjectQueue
  - use getClass() to enforce single type…

Example: Typed Set

- Set of objects
  - All must be of the same type
  - But implement once
- Use: Set<Integer> s;
  - Will only take integers
- Code example:
  - interface Set<E>
  - class SearchTree<E> extends Set