Gregor Johann Mendel (1822 - 1884)

Inheritance & OOP

Lecture 8
CS211 – Fall 2006

Announcements

- On A1, we had some programs that were similar, but not to the level of an Academic Integrity (AI) violation
- For A2, be sure to "form your group" on CMS!
  - It does not happen automatically
- ACSU Announcements
  - Microsoft Info Session
    - Tuesday, Sept 19 at 7pm in Hollister B14
    - Raffling off an Xbox 360
  - Amazon.com Info Session
    - Wednesday, Sept 20 at 5pm in Olin Hall 155
    - Raffling off a Nintendo DS Lite
  - Infusion Info Session
    - Thursday, Sept 21 at 6pm in Olin Hall 255
    - Talk on next generation UI platform for games and business applications
    - Raffling off an Xbox 360 and ipods
  - Free food will be served at all events!

Some AI guidelines

- You are welcome to discuss your assignments with others, but you should not be sharing code
  - Your discussion is too detailed if each line of your "discussion" corresponds to one line of code
  - Of course, you can share code with your official partner
- If you re-use code from another source (e.g., a previous CS course or our text or lecture or section) then you should indicate your source

Object-Oriented Programming

- What do we mean by object-oriented?
- Why use it?
  - Modularity (implementation hiding)
  - Code reuse
  - Type safety
  - Basically, it's used because it seems to help (i.e., it makes it easier for humans)
- Implementation
  - Heap allocation of objects
  - References to objects

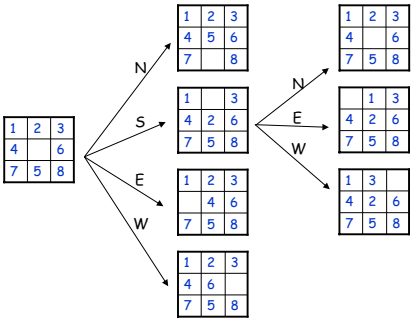
Some Context

- Programming "in the large"
  - Big applications require many programmers
- General approach
  - Break problem into smaller subproblems
  - Assign responsibility for each subproblem to somebody
  - Keep the interfaces small!
- Each subproblem must have a specification
  - Functionality: What services must code provide?
  - Interface: What input conditions does the code expect? What output conditions does it guarantee?
- Job of the programmer: provide an implementation (code) that meets the specification

The Message

- Separate the specification from the implementation
  - Called data abstraction in the literature
  - More modular, easier to maintain
  - Implementation is hidden from the client, can be changed without changing the interface
    - Thus, the client's code does not break
- Object-oriented languages
  - Encourage data abstraction
  - Encourage more modular code

The 8-Puzzle



Program Organization

- class Puzzle
  - An implementation of the game (written by "you")
  - Functionality:
    - init – put puzzle in the initial state
    - move – move a tile N, S, E, or W to get a new state
    - tile – report which tile is in a given position

- class TestPuzzle
  - A client class (written by someone else)
  - Will communicate with Puzzle (your code) to play the game

Implementation

- Two subtasks
  - How do we represent a state (puzzle configuration)?
  - Given this representation, how do we implement init, move, and tile?
- Suppose we don't use objects...

Representation of State

- Model puzzle state as an integer between 123456789 and 987654321
  - 9 represents the empty square
- To convert integer s into a grid representation
  - Remainder when s is divided by 10: tile in bottom right position
    Java expression: s %10
  - Quotient after dividing by 10 gives encoding of remaining tiles
    Java expression: s / 10
  - Repeat remainder/quotient operations to extract remaining tiles
- A somewhat similar encoding is used for storing multidimensional arrays in memory

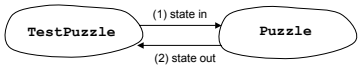
Implementing Operations

- init: put into initial configuration
  s = 123456879;
- tile: what tile is in position (row,col)?
  return s/((int)Math.pow(10,8-(3*row+col)))%10;
- move: left to the reader

A Key Question

- Where do we keep the state?
  - Is it a method-parameter/local-variable?
    - Client keeps track of it
    - Passed to Puzzle methods on each call
    - Allocated on stack
  - Is it a class variable of Puzzle class?
    - Client does not see it
    - Allocated in static area
  - Is it an instance variable of Puzzle class?
    - Client does not see it
    - Allocated on stack
- These implementation choices affect the interface of the Puzzle class

Interface L(ocal)



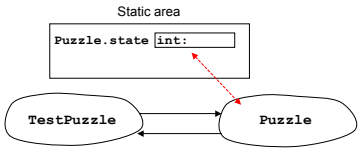
- State is implemented as local variable in class TestPuzzle
  - Passed-to/returned-from methods in Puzzle class
- Interface of Puzzle class:

```
//return encoding of initial state
int init();
//return number of tile at grid (r,c)
int tile(int s, int r, int c);
//move to a new state, return new encoding
int move(int s, char d);
```

Critique of Interface L

- No data abstraction!
  - Puzzle class implementer chose to implement state as an int
  - This representation is exposed in the interface, so the client code is aware of it
  - Client's code may depend on this encoding
  - If Puzzle class implementer decides to change the implementation (say, to represent state as a long), client code breaks

Interface S(tatic)



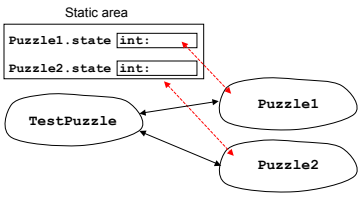
- State is implemented as class variable in class Puzzle
  - State does not have to be passed back and forth
  - Representation is hidden from client
- Interface of Puzzle class:

```
void init();          //initialize the state
int tile(int r, int c);   //return tile in position (r,c)
void move(char d);    //move in direction d
```

Critique of Interface S

- Data abstraction: yes!
  - Puzzle class implementer chose to implement state as int
  - State representation is not visible outside of Puzzle class
  - If Puzzle class implementer decides to change implementation of state to long, client code does not have to change
- Problem: only one client and one puzzle at a time
  - State is a private static variable in class Puzzle
  - Mechanism we have used (class variable) gives right of puzzle creation to implementer of class rather than the client of the class

A Sneaky Solution



- Make copies of `Puzzle` class and rename them
- If client wants *n* puzzles, make *n* copies

Critique

- Data abstraction: yes
- Creation on demand: yes, but at cost of duplication of code
- Must know number of instances at compile time
- Naming issues

The Case for Objects

- Copying and renaming gives us
  - A unique name for each instance of the puzzle
  - A separate variable (state) to store the state of each instance
  - Allows multiple simultaneous instances of the puzzle
- But all the instances are identical!
- Can we design language mechanisms to support the creation of separate instances?

Solution: Ask Gutenberg!

- Algorithm for making a copy of a book in the middle ages
  - Hire a monk
  - Give monk paper and quill
  - Ask monk to copy text of book
- Algorithm for making *n* copies of a book
  - Hire a monk
  - Give monk lots of paper and quills
  - Ask monk to copy text of book *n* times
- Modern algorithm (Gutenberg, Strasbourg ca.1450 AD)
  - First make a template using movable type
  - Stamp out as many copies of book as needed
- Copying class code is like medieval approach to copying books!
- How do we exploit Gutenberg's insight in our context?
  - What is the template for puzzles?
  - How do we stamp out new puzzle instances from the template?
  - How do we name different puzzle instances?

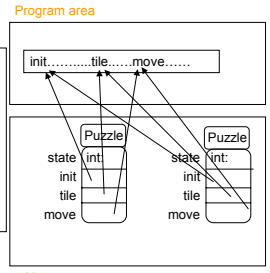
Gutenberg Bible

Object-Oriented Languages

- The class definition is the template
- Instances of the class are called objects
- Objects are stamped out (created) in an area of memory called the heap
- Instance variables: when different instances are stamped out, they will each have their *own copies* of all instance variables (e.g. state)
- Instance methods: code is shared among all instances of the same class, but references to instance variables in the code access those belonging to the correct object!
- Constructor: a special method associated with a class invoked to create new instances of that class

Heap Allocation

```
class  Puzzle {
  private int state;

  public void init() {
    state = 123456879;
  }

  public int tile(int r, int c) {
    return state/((int)Math.pow(10,8-(3*r+c)))%10;
  }

  public void move(char d) {
    ...
  }
}
```



- Heap shows two instances of class **Puzzle**
- Class name is used as type of object
- Each object has its own instance variables
- Instance variables are declared private, so not accessible to client
- Compiled instance methods are stored in Program area
- All objects of type **Puzzle** share code for instance methods as shown

Critique

- Data abstraction: yes
- Creation on demand: yes
- Duplicate class code: no
- Duplicate client code: no

Garbage Collection

- Intuitively, an object is live at time t if that object is still in use and can be accessed by the program
- Formally (recursive definition), an object O is live if:
  - The runtime stack contains a reference to O
  - There is a live object O' that contains a reference to O
- Everything else is garbage
- Periodically, system detects garbage and reclaims it
  - Start with the stack, trace all references, mark all objects seen
  - Anything not marked is garbage
- In C, C++:
  - Pointer arithmetic makes it hard to determine what is a reference
  - Storage reclamation must be done explicitly by programmer (malloc, mfree)
  - Highly error-prone

What is Inheritance?

- OOP provides Encapsulation + Extensibility
  - Encapsulation: permits code to be used without knowing implementation details
  - Extensibility: permits behavior of classes to be changed or extended without having to rewrite the code of the class
    - No need to involve the class implementer
- Mechanism for extensibility in OO-programming
  - Inheritance

Running Example: Puzzle

```
class Puzzle {

  //representation of a puzzle state
  private int state;

  //create a new random instance
  public void scramble() {...}

  //say which tile occupies a given position
  public int tile(int r, int c) {...}

  //move a tile
  public boolean move(char c) {...}
}
```

New Requirement

- Suppose you are the client. After receiving puzzle code, you decide you want the code to keep track of the number of moves made since the last scramble operation.
- Implementation is simple:
  - Keep a counter numMoves, initialized to 0
  - Method move increments counter
  - Method scramble resets counter to 0
  - New method printNumMoves for printing value of counter

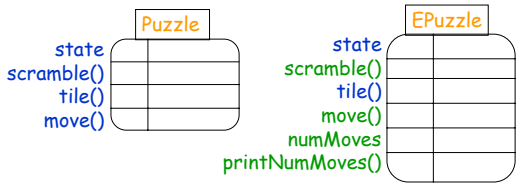
Implementation

- Three approaches:
  - Call supplier, apologize profusely, and send them a new specification. They implement it and charge you an extra $5K. ☹
  - Rewrite the supplier's code yourself. Three months later, you still haven't figured it out. ☹
  - Use inheritance to define a new class that extends the behavior of the supplier's class. ☺

Goal

- Define a new class EPuzzle that extends the class Puzzle
- Tell Java that EPuzzle is just like Puzzle, *except*
  - It has a new integer instance variable named numMoves
  - It has a new instance method called printNumMoves
  - It has modified versions of scramble and move methods

Goal in Picture Form



The EPuzzle Class

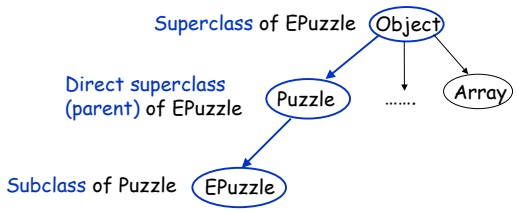
```
class EPuzzle extends Puzzle  {
   private int numMoves = 0;
   public void scramble() {...}
   public boolean move(char d) {...}
   public void printNumMoves() {...}
}
```

- Class EPuzzle is a subclass of class Puzzle
- Class Puzzle is a superclass of class EPuzzle
- An EPuzzle object has
  - Its own instance variable numMoves and instance method printNumMoves
  - It overrides methods scramble and move in class Puzzle
  - It inherits method tile from class Puzzle

Overriding

- A method declaration m in subclass B overrides a method m in superclass A if both methods have
  - The same name,
  - Both are class methods or both are instance methods, and
  - Both have the same number and type of parameters and same return type

Class Hierarchy



Every class (except Object) has a unique direct superclass, called its parent class

Single Inheritance

- Every class is implicitly a subclass of Object
- A class can extend exactly one other class
  - class Puzzle {...}
    - This class implicitly extends Object
  - class EPuzzle extends Puzzle {...}
    - This class explicitly extends Puzzle, and implicitly extends Object since Puzzle is a subclass of Object
- Class hierarchy in Java is a tree
- In C++, a class can have more than one superclass (multiple inheritance)
  - Class hierarchy in C++ is a directed acyclic graph (dag)

Writing the EPuzzle Class

```
class EPuzzle extends Puzzle {
  private int numMoves = 0;

  public void printNumMoves() {
     System.out.println("Number of moves = " + numMoves);
  }

  //other method definitions
  ...
}
```

scramble and move

- How should we write these methods?
  - One option: write them from scratch.

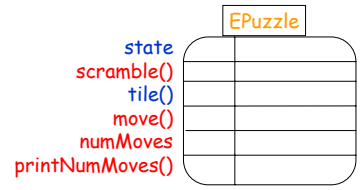
```
Class EPuzzle extends Puzzle {
  private int numMoves = 0;

  public void scramble() {
    state = "978654321";
    numMoves = 0;
  }
}
```

  - Problem: State was declared to be a private field in class Puzzle, so it is *not accessible* to methods in class EPuzzle

Difficulty with Private Variables

- Variable state is declared *private*, so it is only accessible to instance methods in class Puzzle
- In an instance of class EPuzzle, the tile method can access this variable because method tile is *inherited* from the superclass
- Method scramble defined in class Epuzzle does *not* have access to state
- Similarly, any *private* methods in a superclass are not accessible to methods in subclass

Reiterating



- EPuzzle objects have an instance variable for state because EPuzzle extends Puzzle
- However, state is accessible only to methods inherited from Puzzle (such as tile()) and not to methods written in EPuzzle class (such as scramble()) because state was declared to be private

Protected Access

- New access specifier: protected
- A protected instance variable in class S can be accessed by instance methods defined either in class S *or in a subclass of S*
- A protected method in class S can be invoked from an instance method defined in class S *or in a subclass of S*
- Access checks are done by compiler at compile time
  - For an invocation r.m():
    - Determine type of reference r
    - Does the corresponding class/interface have a method named m with appropriate arguments?
    - Are the access specifiers of that method appropriate?

Revised Code for *Puzzle Class*

says state is accessible from subclasses

```
class Puzzle {
  protected int state;
  public void scramble(){…}
  …
}
```

Code for EPuzzle

```
class EPuzzle  extends Puzzle {
  protected int numMoves = 0;

  public void printNumMoves(){
    System.out.println("Number of moves = " + numMoves);
  }
  public void scramble() {
    state = "978654321"; //OK since state is inherited
    numMoves = 0;
  }
  //similar code for move
}
```

Protected Access

- Should all instance variables and methods be declared protected?
- Need to think about extensibility
  - If you believe that subclasses will want access to a member, it should be declared protected
- Analogy
  - Which components of a car might a user want to upgrade?
  - What wires/sub-systems need to be exposed to make the upgrade easy?
- Extending a class requires *more knowledge of the class* than is needed just to use it