Inheritance

Lecture 4
CS2110 – Fall 2011

Gregor Johann Mendel (1822–1884)
Announcements

• A1 due Wednesday 11:59pm
• For A1 files, please use the default package (which is no package at all). I.e., your files should not contain any package declarations
• Please make sure your method headers match the specifications in the writeup exactly
• Please activate your Piazza account if you have not already
What is Inheritance?

• a mechanism for **Extensibility** in OO programming

• **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
  ▪ promotes code reuse

• **Encapsulation**: permits code to be used without knowing implementation details
  ▪ classes, objects
  ▪ visibility declarations such as *private*, *protected*

• **OO-programming** = Encapsulation + Extensibility
Running Example: 8-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 row, int col) {...}

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

1 3
2 4
5 6
7 8
Representation of State

• One possibility: model puzzle state as an integer between 123456789 and 987654321
  ▪ 9 represents the empty square
• To convert integer s into a grid representation:
  ▪ Remainder s%10: tile in bottom right position: 8
  ▪ Quotient s/10: encoding of remaining tiles: 13942675
  ▪ Repeat remainder and quotient to extract remaining tiles
• A similar encoding is used for multidimensional arrays
• We declared state private, so only the Puzzle class knows we are using this representation -- Encapsulation
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` should increment the counter
  ▪ method `scramble` should reset the 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 **Puzzle**

• Tell Java that **EPuzzle** is just like **Puzzle**, except:
  ▪ it has a new instance variable **numMoves**
  ▪ it has a new instance method **printNumMoves**
  ▪ it has modified versions of **scramble** and **move**
Picture

Puzzle

state
scramble()
tile()
move()

EPuzzle

state
scramble()
tile()
move()
numMoves
printNumMoves()
• Class **EPuzzle** is a subclass of class **Puzzle**
• Class **Puzzle** is a superclass of class **EPuzzle**
• An **EPuzzle** object
  ▪ *has its own* instance field **numMoves** and instance method **printNumMoves**
  ▪ *overrides* methods **scramble** and **move** of **Puzzle**
  ▪ *inherits* method **tile** of **Puzzle**

```java
class EPuzzle extends Puzzle {
    private int numMoves = 0;
    public void scramble() {...}
    public boolean move(char d) {...}
    public void printNumMoves() {...}
}
```
Every class (except **Object**) has a unique immediate superclass, called its **parent**.
Overriding

- A method in a subclass *overrides* a method in superclass if:
  1. both methods have the same name,
  2. both methods have the same signature (number and type of parameters and return type), and
  3. both are static methods or both are instance methods.
Single Inheritance

- Every class is implicitly a subclass of `Object`
- A class can have exactly one parent
  - `class Puzzle {...}
    - implicitly extends `Object`
  - `class EPuzzle extends Puzzle {...}
    - explicitly extends `Puzzle`, and implicitly extends `Object` since `Puzzle` is a subclass of `Object`

- Class hierarchy in Java is a tree
  - subclasses = descendants, superclasses = ancestors
- In C++, a class can have more than one superclass (multiple inheritance)
  - class hierarchy 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);
    }

    public void scramble() {...}
    public boolean move(char d) {...}
}
**scramble and move**

How should we write `scramble` and `move`?
One option: write them from scratch.

```java
class EPuzzle extends Puzzle {
    ...

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

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

- Problem: `state` was declared `private` in the `Puzzle` class
- it is not accessible to `EPuzzle`!
Difficulty with Private Variables

- Variable `state` is declared `private`, so it is only accessible to methods in class `Puzzle`.
- In an instance of class `EPuzzle`, the `tile` method can access this variable because the `tile` method 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.
Interesting Point

- EPuzzle objects have an instance variable `state` because EPuzzle extends Puzzle.
- However, they cannot access it directly, because it is private!
- `state` is accessible to public methods inherited from Puzzle (such as `tile()`) but not to methods written in the EPuzzle class (such as `scramble()`).
Protected Access

- Access specifier: `protected`
- A protected instance field in class `S` can be accessed by instance methods defined in `S` or any subclass of `S`
- A protected method in class `S` can be invoked from an instance method defined in `S` or any subclass of `S`
- Access checks are done by compiler at compile time:
  - For an invocation `r.m()`:
    - Determine the static (compile-time) type of `r`
    - Does the corresponding class/interface have a method named `m` with appropriate arguments?
    - Are the access specifiers of that method appropriate?
Proper Code for Puzzle Class

class Puzzle {
    protected int state;
    public void scramble(){...}
    ...
}
class EPuzzle extends Puzzle {
    ...

    public void scramble() {
        state = "978654321"; //OK since state inherited
        numMoves = 0;
    }

    //similar code for move
}
Protected Access

• When should variables and methods be declared `protected` instead of `private`?
• Think about extensibility: if 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/subsystems need to be exposed to make the upgrade easy?
Another Solution

• Suppose a class $S$ overrides a method $m$ in its parent
• Methods in $S$ can invoke the overridden method in the parent as
  $$\text{super.m()}$$
• In particular, can invoke the overridden method in the overriding method!

• Caveat: cannot compose super more than once as in
  $$\text{super.super.m()}$$
Another Definition of EPuzzle

class EPuzzle extends Puzzle {
    protected int numMoves = 0;
    ...
    public void scramble() {
        super.scramble();
        numMoves = 0;
    }
    public boolean move(char d) {
        boolean p = super.move(d);
        if (p) numMoves++;
        return p;
    }
}

Do not need protected access to state after all!
Constructors

• Each class has its own constructor
• No overriding of constructors
• Superclass constructor can be invoked explicitly within subclass constructor using `super()` with parameters as needed
• Can invoke other constructors of the same class using `this()`
• Call to `super()` or `this()` must occur first in the constructor
A Digression – Exceptions in Java

• **Exceptions** = a mechanism for handling uncommon or exceptional situations
  - unusual, but may be expected from time to time during normal operation
  - example: user enters date in incorrect format

• **Errors** are typically more serious
  - program failure, usually symptom of a bug
A Digression – Exceptions in Java

Object

 Throwable

 Exception

 RuntimeException

 ClassCastException

 NullPointerException

 IllegalArgumentException

 NumberFormatException

 Error
A Digression – Exceptions in Java

• You can create your own

```java
class MyException extends Exception {
    MyException(String message) {
        super(message);
    }
}
...

MyException m = new MyException(“Bad karma”);
System.out.println(m.getMessage());
...
```
Throwing and Catching

• Certain operations may throw exceptions...

```java
int[] a = new int[3];
a[3] = -1; // throws ArrayIndexOutOfBoundsException

Object o = null;
String s = o.toString(); // throws NullPointerException

Object e = new Object();
Integer i = (Integer)e; // throws ClassCastException
```

• ... or you can do it explicitly with `throw`

```java
throw new NullPointerException();
throw new MyException("Bad karma");
```
try {
    ...code that might throw an exception...
} catch (NumberFormatException e) {
    ...exception handler for number format exceptions...
} catch (ClassCastException e) {
    ...exception handler for class cast exceptions...
} catch (Exception e) {
    ...default handler for all other exceptions...
}
Throwing and Catching

• What happens when an exception is thrown?
  ▪ Execution of the method is immediately interrupted
  ▪ Control passes to an enabled exception handler if there is one
    • the throw must have occurred in the try clause of a try/catch statement
    • the runtime type of the exception must be a subtype of the type specified in the catch clause
    • the first enabled handler is executed
  ▪ If there is no enabled exception handler, control passes to the calling method and the exception is rethrown at the call point
Throwing and Catching

• Example 1

```java
try {
    Object o = new Object();
    Integer i = (Integer)o;
} catch (NumberFormatException e) {
    ...
} catch (ClassCastException e) {
    ...  //this handler will be executed
} catch (Exception e) {
    ...
}
```
Throwing and Catching

• Example 2

```java
try {
    int[] a = new int[3];
    a[3] = -1;
} catch (NumberFormatException e) {
    ...
} catch (ClassCastException e) {
    ...
} catch (Exception e) {
    ... //this handler will be executed
}
```
Throwing and Catching

• Example 3

```java
try {
    //this will not be caught
    throw new Throwable();
} catch (NumberFormatException e) {
    ...
} catch (ClassCastException e) {
    ...
} catch (Exception e) {
    ...
}
```
Throwing and Catching

• Example 4

```java
try {
    int i = foo(s);
} catch (MyException e) {
    System.out.println(e.getMessage());
}
...

int foo(String s) throws MyException {
    if (s.equals("")) {
        throw new MyException("Empty argument");
    }
}
```

All potentially uncaught exceptions (except RuntimeException) must be declared
Conclusion

• Key features of object-oriented programming
  ▪ Encapsulation: classes and access control
  ▪ Inheritance: extending or changing the behavior of classes without rewriting them from scratch
  ▪ Dynamic storage allocation & garbage collection
  ▪ Access control: public/private/protected
  ▪ Subtyping