



Gregor Johann Mendel (1822–1884)

Inheritance

Lecture 8
CS211 – Spring 2007

1

What is Inheritance?

- OO-programming = Encapsulation + Extensibility
- **Encapsulation**: permits code to be used without knowing implementation details
 - classes, objects
 - visibility declarations such as **private**, **protected**
- **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
- Mechanism for extensibility in OO-programming: **Inheritance**

2

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	
4	2	6
7	5	8

3

Representation of State

1	3	
4	2	6
7	5	8



139426758

- 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**

4

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

5

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. 😊

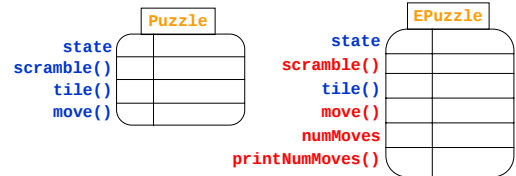
6

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**

7

Picture



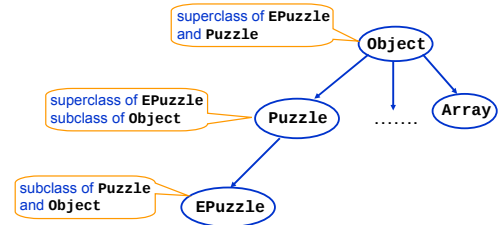
8

```
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**
 - overrides methods **scramble** and **move** of **Puzzle**
 - inherits method **tile** of **Puzzle**

9

Class Hierarchy



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

10

Overriding

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

11

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)

12

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) {...}
}
```

13

scramble and move

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

```
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**!

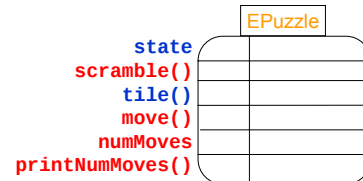
14

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

15

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()**)

16

Protected Access

- Access specifier: **protected**
- A protected instance variable in class **S** can be accessed by instance methods defined in **S** or in 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 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?

17

Proper Code for Puzzle Class

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

says state is accessible from subclasses

18

Code for EPuzzle

```
class EPuzzle extends Puzzle {
    ...

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

    //similar code for move
}
```

19

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?

20

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

`super.m()`

- In particular, can invoke the overridden method in the overriding method!
- Caveat: cannot compose super more than once as in `super.super.m()`

21

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++; //legal move?
        return p;
    }
}
```

Do not need **protected** access to **state** after all!

22

Subtypes

- Inheritance gives a mechanism for creating *subtypes*
 - (*Interfaces* are another such mechanism)
- If class **B** extends class **A** then **B** is a subtype of **A**
- Examples:
 - `Puzzle p = new EPuzzle();` //up-casting
 - `EPuzzle e = (EPuzzle)p;` //down-casting

23

Unexpected Consequence

An overriding method cannot have more restricted access than the method it overrides

```
class A {
    public int m() {...}
}

class B extends A {
    private int m() {...} //illegal!
}

A supR = new B(); //upcasting
supR.m(); //would invoke private method in
class B at runtime!
```

24

Shadowing Variables

- Like overriding, but for fields instead of methods
 - Superclass: variable **v** of some type
 - Subclass: variable **v** perhaps of some other type
 - Method in subclass can access shadowed variable using **super.v**
- Variable references are resolved using *static binding* (i.e., at compile-time), not *dynamic binding* (i.e., not at runtime)
 - Variable reference **r.v** uses the *static type* (declared type) of the variable **r**, not the *runtime type* of the object referred to by **r**
- Shadowing variables is bad medicine and should be avoided

25

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

26

Abstract Classes

- An **abstract class** cannot be instantiated
- May have methods without bodies that *must be* overridden by a (non-abstract) subclass

```
abstract class Puzzle {
    protected int state;
    public void scramble() {
        state = 978654321;
    }

    //abstract methods (no code)
    abstract public int tile(int r, int c);
    abstract public void move(char d);
}
```

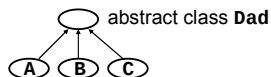
27

Abstract Classes

- An abstract class is an incomplete specification
 - Cannot be instantiated directly
 - Not all methods in abstract class need to be abstract — allows code sharing
 - Abstract classes are part of the class hierarchy and the usual subtyping rules apply

28

Use of Abstract Classes



- Variables/methods common to a bunch of related subclasses can be declared once in Dad and inherited by all subclasses
- If subclass **C** wants to do something differently, it can override **Dad's** methods as needed

29

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

30