Inheritance

What is inheritance?

- OO-programming = Encapsulation + Extensibility
- Encapsulation: permits code to be used without knowing implementation details
- Extensibility: permit the behavior of classes to be extended incrementally w/o involving class implementor
  - (eg) to upgrade radio in car, we do not send it back to the manufacturer
- Mechanism for extensibility in OO-programming: inheritance
- Inheritance promotes code reuse
  - permits you to change the behavior of a class without having to rewrite the code of the class

Running Example: puzzle

```
interface IPuzzle {
      void scramble();
      int tile(int r, int c);
      boolean move(char c);
}

class Puzzle implements IPuzzle{
     private int state;
     public void scramble() {…}
     public int tile(int r, int c){…}
     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.
  - move method invocation increments counter.
  - scramble method invocation resets counter.
  - New method: printNumMoves for printing value of counter.

New Specification

We want the code to implement a new interface:

```
interface IEPuzzle extends IPuzzle{
   void printNumMoves();
}
```

Implementing the new interface

- Three approaches:
  - Call supplier, apologize profusely, and send him new interface. Expensive.
  - Throw away the supplier's code and write it yourself. Expensive.
  - Use inheritance to define a new class that extend the behavior of the supplier's class. Right!

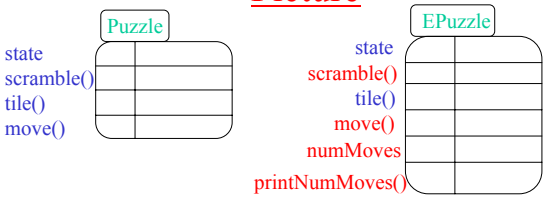
Goal: to define a class EPuzzle

that implements the interface IEPuzzle

by extending the class Puzzle

that implemented the interface IPuzzle

Picture



- Can we tell Java that class EPuzzle is just like Puzzle except that
  - 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?

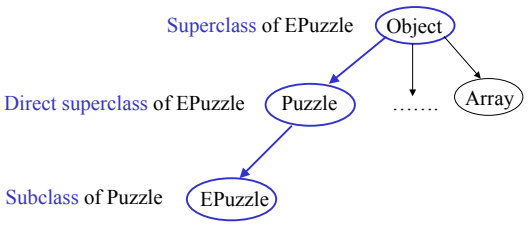
```
class EPuzzle extends Puzzle{
  private int numMoves = 0;
  public void scramble() {…}
  public boolean move(char d){…}
  public void printNumMoves(){…}
}
```

- Class EPuzzle is a sub-class of class Puzzle.
- Class Puzzle is a super-class 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 instance variable state and method tile from class Puzzle

Note on overriding

- A method declaration m in sub-class B can override a method m in super-class A only 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

Class Hierarchy



Every class other than Object has a unique direct superclass that is called the parent class of that class.

Single inheritance

- In Java, every class is implicitly a sub-class 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 sub-class of Object.
- Class hierarchy in Java is a tree.
- C++: a class can be a direct sub-class of more than one super-class.
  - Class hierarchy is a directed acyclic graph.

Writing EPuzzle Class

First, let us implement the new members of EPuzzle.

```
class EPuzzle extends Puzzle implements IEPuzzle{
   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.

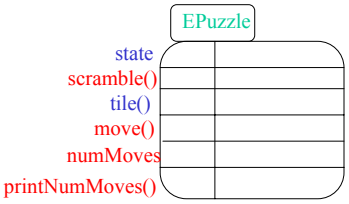
```
Class EPuzzle extends Puzzle implements IEPuzzle{
    private int numMoves = 0;
    ….
    public void scramble() {
      state = "978654321";
      numMoves = 0;
    }
}
```

- We can write the move method the same way.
- Problem: state was declared to be a private variable 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 it is inherited from the super-class.
- Scramble method defined in class Epuzzle does not have access to state.
- Similarly, private methods in super-class are not accessible to methods in sub-class.

Interesting point

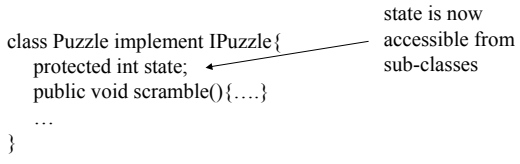


- EPuzzle objects have an instance variable for state because EPuzzle extends Puzzle.
- However, state is accessible only to methods inherited from Puzzle (such as tile()) and not to methods written in EPuzzle class (such as scramble()) because state was declared to be private.

One solution: 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 sub-class of S.
- A protected method in class S can be invoked from an instance method defined either in class S or in a sub-class of S.

Proper code for Puzzle class



Code for EPuzzle

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

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

Protected access

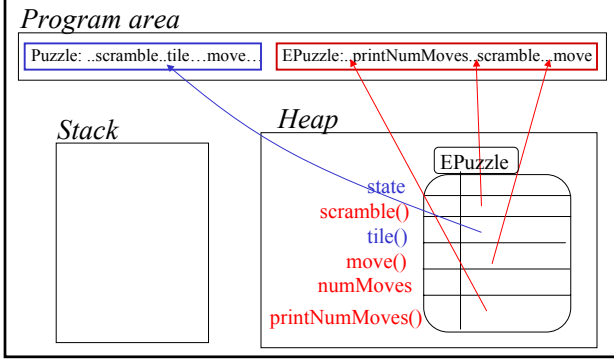
- Should all instance variables and methods be declared protected?
- Need to think about extensibility: if you believe that sub-classes will want access to a member, it should be declared protected.
- Analogy:
  - Which components of a car might a user want to upgrade?
  - What wires/sub-systems need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

Another solution

- Suppose sub-class S overrides a method m in its super-class.
- Methods in sub-class S can invoke overridden method of super-class as

  super.m()
- Caveats:
  - cannot compose super many times as in super.super.m()
  - static binding: super.m is resolved at compile-time, so no object look-up at runtime

Static binding:Compiler resolves method in invocation super.scramble() in EPuzzle method scramble to scramble method in Puzzle class.



Another definition of EPuzzle

```
class EPuzzle extends Puzzle implements IEPuzzle{
    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;
    }
}
```

For this solution, you do not need protected access to state.

Sub-typing

- Inheritance gives another mechanism in Java for creating sub-types.
  - other mechanism: implementing interfaces.
- If class B extends class A, B is a sub-type of A.
- Examples:
  - Puzzle p = new EPuzzle();//up-casting
  - EPuzzle e = (EPuzzle)p; //down-casting
    - legal if type of reference p is Object, Puzzle, or EPuzzle and if type of object referenced by p is EPuzzle.

Unexpected consequence

- Sub-class method m that overrides a super-class method cannot have more restricted access than the super-class method.

```
class A {
  public int m(){…}
}
class B {
  private int m(){…}
}
….
B subR = new B();
subR.m();//should be illegal
A supR = subR;//upcasting
supR.m();// protection is OK, and will invoke method in class B at
                    //runtime!
```

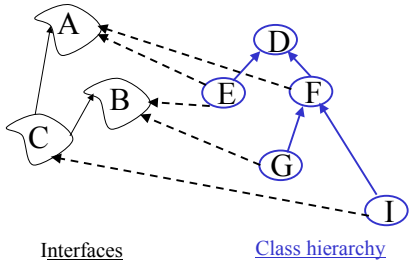
Java restriction

- If method m in sub-class B overrides a method m in super-class A,
  - method m in sub-class B must have the same or less restricted access than method M declared in super-class A

Interfaces and inheritance

- A class can
  - implement many interfaces, but
  - it can extend only one class.

Example



Interfaces

Class hierarchy

```
interface C extends A,B{
  …
}
```

```
class F extends D implements A{
…..}
class E extends D implements A,B{
….}
```

Shadowing variables

- Like overriding but for variables rather methods
  - Super-class: variable v of some type
  - Sub-class: variable v perhaps of some other type
  - Method in sub-class can access shadowed variable by using super.v
- Variable references are resolved using static binding, not dynamic binding.
  - Variable reference r.v: type of r and not of the object referred to by r determines which variable is accessed.
- Shadowing variables is usually bad practice and we will not worry about it.

Constructors

- No overriding of constructors: each class has its own constructor.
- Super-class constructor can be invoked explicitly by sub-class constructor by invoking super() with parameters as needed.
- Object initialization in the presence of inheritance can be quite complex: see Java manual.

Abstract class

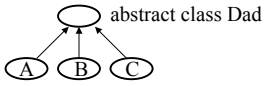
- Abstract class has one or more methods that must be overridden by a sub-class that can be instantiated.

```
abstract class Puzzle{
  protected int state;
  public void scramble(){state = 978654321;}
  abstract public int tile(int r, int c);//no code
  abstract public void move(char d);//no code
}
```

Abstract classes (contd)

- Abstract class is an incomplete spec.
  - cannot be instantiated directly
  - not all methods in abstract class need to be abstract
  - somewhere between interfaces and concrete classes
  - abstract classes are part of the class hierarchy and usual sub-typing rules apply

Use of abstract class



- Variables/methods common to a bunch of related sub-classes can be declared once in Dad and inherited by all sub-classes.
- If sub-class C wants to do something differently, it can override methods as needed.

OO-programming

- OO-programming:
  - Encapsulation: classes and access control
  - Inheritance: extending the behavior of classes without rewriting them from scratch
- Key intellectual concepts:
  - Dynamic storage allocation
  - Access control: public/private/protected
  - Sub-typing
- Procedural languages: C/Pascal/….
  - Dynamic storage allocation is available (malloc)
  - You can fake access control with proper discipline.
  - Sub-typing: function pointers are unsafe way of faking it.