Inheritance

What is Inheritance?

- OO-programming = 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
- Inheritance promotes code reuse

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
- move method increments counter
- scramble method resets counter to 0
- New method printNumMoves for printing value of counter

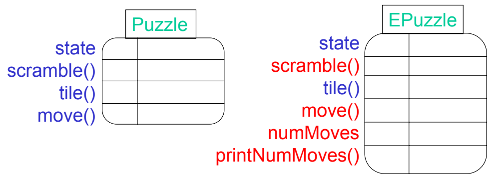
Implementation

- 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

Picture



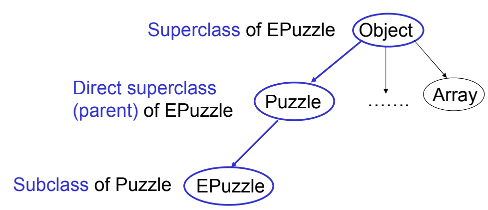

```java
class EPuzzle extends Puzzle  {
   private int numMoves = 0;
   public void scramble() {...}
   public boolean move(char d) {...}
   public void printNumMoves() {...}
}
```

- 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 the parent class of that 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 is a directed acyclic graph

Writing EPuzzle Class

```java
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.

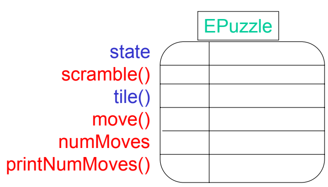
```
Class EPuzzle extends Puzzle {
   private int numMoves = 0;

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

- Problem: state was declared to be a private 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 superclass
- Scramble method defined in class Epuzzle does not have access to state
- Similarly, private methods in superclass are not accessible to methods in subclass

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

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 either 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?

Proper 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

Another Solution

- Suppose subclass S overrides a method m in its superclass.
- Methods in subclass S can invoke overridden method of superclass 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

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!

Subtypes

- Inheritance gives a mechanism in Java for creating subtypes
  - another other mechanism: interfaces
- If class B extends class A, B is a subtype of A
- Examples:
  - `Puzzle p = new EPuzzle(); //up-casting`
  - `EPuzzle e = (EPuzzle)p; //down-casting`

Unexpected Consequence

A method that overrides a superclass method cannot have more restricted access than the superclass method

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

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

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

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 by using super.v
- Variable references are resolved using static binding, not dynamic binding
  - Variable reference r.v: static type of the variable r, not runtime type of the object referred to by r, determines which variable is accessed
- Shadowing variables is bad medicine and should be avoided

Constructors

- No overriding of constructors: each class has its own constructor
- Superclass constructor can be invoked explicitly by subclass constructor by invoking 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

Abstract Classes

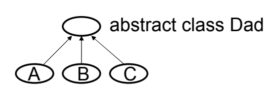
- An abstract class cannot be instantiated
- May have methods without bodies that must be overridden by a (non-abstract) subclass

```
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);
}
```

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

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

Conclusion

- Key features of OO-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