Inheritance

Slogan from first lecture:

Object-oriented programming = Encapsulation + Extensibility

Encapsulation: permits code to be used without knowing details of implementation

Motto: Build appliances, not tool-kits !!

Extensibility: permit the behavior of classes to be extended incrementally w/o involving the class implementor.

Example: upgrading the radio in a car
If car is designed properly, radio can be upgraded without returning car to manufacturer.

Mechanism for extensibility in OO-programming: inheritance

Inheritance is a key feature that promotes code reuse.

Example: Puzzle

Interface provided to supplier:

```
interface PuzzleInterface {
 void Scramble();
 int Tile (int r, int c);
 boolean Move(char c);}
```

Implementation by supplier:

```
class Puzzle implements PuzzleInterface{
 private int state; //represents configuration
 private final int Base = 10;
 public void Scramble() {
  state = ....; }
 public int Tile(int r, int c) {
  return state/CoolPower(Base,3*r+c)%Base; }
 public void Move(char c) {
  ...}}
```


Imagine you are client.
Suppose you decide that you want to keep track of number of moves made since last `Scramble` operation.

Implementation of this idea:

- Keep a counter `NumMoves` initialized to 0.
- Each `Move` operation increments `NumMoves`.
- `NumMoves` is reset to 0 by `Scramble` operation.
- New operation: `PrintNumMoves` for printing value.

Let us specify this more formally:

```
interface CoolPuzzleInterface {
 void Scramble();
 int Tile (int r, int c);
 boolean Move(char c);
 void PrintNumMoves(); //new method
 }
```

We can also say

```
interface CoolPuzzleInterface extends PuzzleInterface{//keyword:extends
 void PrintNumMoves(); //new method
 }
```


Terminology: PuzzleInterface is super-interface of CoolPuzzleInterface

Use of `extends` permits us to say how interface is different from super-interface, rather than specify from scratch.

Class that implements interface must implement all the methods specified in the superinterface(s) as well.

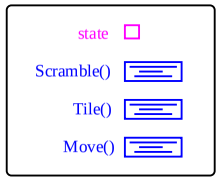
No way to "remove" methods from a super-interface.

Question: How should we implement the new interface?

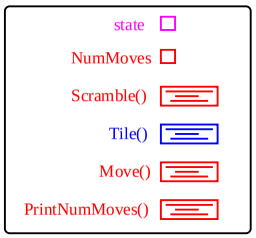
Three approaches:

1. Call supplier, apologize profusely and ask him to implement the new interface. Expensive.
2. Throw away supplier code and start all over on your own. Expensive.
3. Use inheritance to define a new class that *extends the behavior of the old class incrementally*. Right!!

Goal: to define a class `CoolPuzzle`
that implements interface `CoolPuzzleInterface`
by building on class `Puzzle`
that implements interface `PuzzleInterface`.



Instance of class Puzzle



Instance of class CoolPuzzle

How do we tell Java that class CoolPuzzle is just like class Puzzle
except that (i) it has a new integer named NumMoves
(ii) it has a new method PrintNumMoves
(iii) it has slightly modified versions of Scramble and Move?


```
class CoolPuzzle extends  Puzzle {
        private int NumMoves = 0;
        public void Scramble() {
          ......}
        public void Move(char c) {
          ......}
        public void PrintNumMoves() {
          .......}
}
```

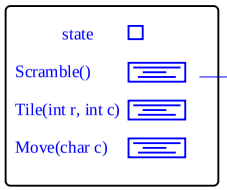
Class CoolPuzzle is a SUBCLASS of class Puzzle.
Class Puzzle is the SUPERCLASS of class CoolPuzzle.

CoolPuzzle has its own instance variables (NumMoves)
and instance methods (PrintNumMoves).

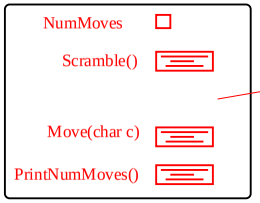
CoolPuzzle inherits instance variable state and method Tile
from superclass.

CoolPuzzle overrides the methods Scramble and Move in the superclass.

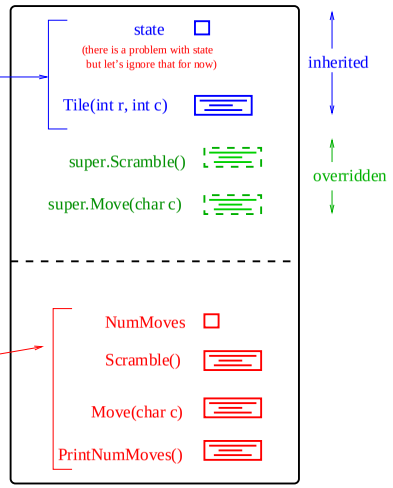
Class Puzzle



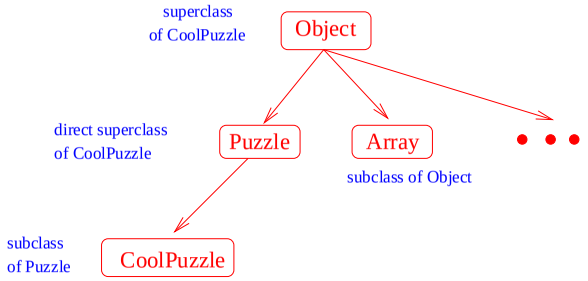
Class CoolPuzzle



Instance of class CoolPuzzle



Methods in superclass with same name
and type signature as methods in subclass
are overridden.
However, they can still be accessed as super.method()



Class Hierarchy: a Tree rooted at Object

In Java, every class other than object has a unique parent class.
In C++/Dylan etc, a class may have many parent classes (multiple inheritance).

Details of CoolPuzzle Definition

First, let us implement the new members of CoolPuzzle.

```
class CoolPuzzle extends Puzzle { //keyword: extends
       private int NumMoves = 0;

       public void PrintNumMoves() {
         System.out.println("Number of moves = " + NumMoves);
       }
       ......//other method definitions
}
```


How should we write *Scramble* and *Move*?

One option: write them from "scratch"

```
class CoolPuzzle extends Puzzle {
 private int NumMoves = 0;
 .......
 public void Scramble() {
   state = 78654321;
   NumMoves = 0;
 }
}
```

We can write the new Move method similarly.

Problem:`state` was declared to be a `private` variable.

Difficulty with `state` as `private` variable

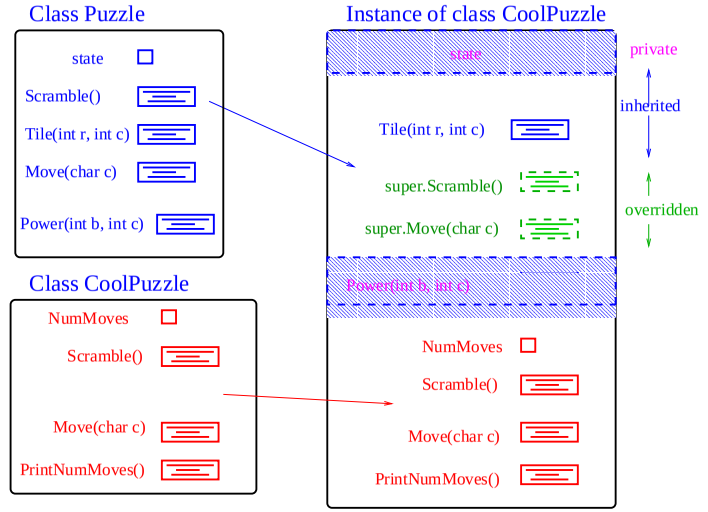
`state` is accessible only from instance methods defined in `Puzzle` class.

Therefore, in instance of class `CoolPuzzle`, the following methods can access `state`:
super.Move(), super.Scramble() and Tile().

So the `Scramble` method defined in subclass `CoolPuzzle` does not have access to `state`.

This is true for `private` methods of a class as well: they are not visible to methods in subclasses.

~~Private methods/variables in superclass are not visible to methods in subclasses.~~



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 any subclass of class S.

A `protected` method in class S can be invoked in an instance method defined either in class S or in any subclass of class S.

New definition of Puzzle

```
class Puzzle implements PuzzleInterface{
 protected int state; //accessible from subclasses as well
 protected final int Base = 10;
 public void Scramble() {
  state = ....; }
 public int Tile(int r, int c) {
  return state/CoolPower(Base,3*r+c)%Base; }
 public void Move(char c) {
  ...}
}
```



```
class CoolPuzzle extends Puzzle { //keyword: extends
       private int NumMoves = 0;

       public void PrintNumMoves() {
         System.out.println("Number of moves = " + NumMoves);
       }
       public void Scramble() {
         state = 78654321; //since state is PROTECTED, we have access
         NumMoves = 0;
       }
       ....
}
```

Question: Should `NumMoves` be `protected` rather than `private`?

Should all variables/methods be declared to be protected?

Need to think about extensibility - if you believe that subclasses will want access to them, they should be declared to be `protected`.

Analogy:

- Which components in car might the user want to upgrade?
- What internal wires/subsystems etc need to be exposed so new components can be plugged in easily?

Note: In general, extending a class requires a more detailed knowledge of class implementation than is required just to use it.

Another solution: use the fact that the "old" `Scramble` and `Move` methods are still accessible in CoolPuzzle instances by invoking `super.Scramble` and `super.Move`.

```
public void Scramble(){
  super.Scramble();
    //invoke Scramble in direct superclass
    //this is just like a regular method invocation
  NumMoves = 0;
}
public void Move() {
  super.Move();
  NumMoves = NumMoves + 1;
}
```

For this solution, you do not need `protected` access to `state`. However, in general, this kind of "clean separation" may not be possible. So if you think extensibility is required, make variables/methods `protected` rather than `private`.

Subtyping

Inheritance gives another mechanism in Java to creates subtypes (interfaces were the other mechinism).

If class B extends A, B is a subtype of A.

eg: `CoolPuzzle` *is-a* subtype of `Puzzle`.

This means that the following assignment is valid:

```
Puzzle P = new CoolPuzzle();
```

since a subtype object can be assigned to a supertype reference.

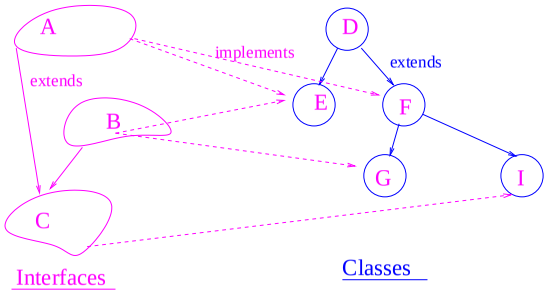
Difference between *inheritance* and *interface*:

If class B `implements` interface A, objects of type B do not inherit any *implementation* from A. In contrast, if B `extends` A, objects of type B inherit all members of A that are not overridden in B and that are not private.

Interfaces vs. inheritance

A class can

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



interface C extends A,B {
...}

class F extends D implements A{
...}

class E extends D implements A,B{
...}

Note: G implicitly implements A as well !

Dynamic Method Binding: Key Feature of OO

```
Puzzle MyPuzzle = new CoolPuzzle(); //type of object = CoolPuzzle
MyPuzzle.Move('N');                 //type of reference = Puzzle
```

Reference type (`Puzzle`) is supertype of object type (`CoolPuzzle`).

Both `Puzzle` and `CoolPuzzle` have methods named `Move`.

Which one should be invoked?

Answer:

```
Puzzle MyPuzzle = new CoolPuzzle();
MyPuzzle.Move('N'); //object is subtype of reference type
```

Binding Rule for Methods:

- Method that is invoked is the instance method in *object* class (`CoolPuzzle`); definition of method in class of *reference* is irrelevant (intuitively, we go to object and press button labeled 'Move').
- However, compiler checks that class of reference (Puzzle) has a method with that name (this ensures that whatever the class of the object, object is guaranteed to have a 'Move' method since class of object must be a subtype of `Puzzle`).

Note: This is consistent with how subtyping worked in *interfaces*. The method invoked was the method in the object. A little more intuitive in that context since methods in interface have no implementation!

Why is this called dynamic method binding?

```
....
 public void bar (Puzzle p) {
   p.Move('N'); //Which Move method is called?
 }
....
```

If `Puzzle` has n subtypes each of which have their own `Move` methods, the method invocation can result in execution of one of $n+1$ different pieces of code.

Which code is actually executed depends on the type of object named by `p`: determined at runtime.

Closest analogy in C/PASCAL: procedure/function parameters.

Dynamic method binding is quite subtle.

```
class Puzzle {
 private int state;
 public int Tile(int r, int c) {
  return state/Power(10, 3*r+c)%10;
 }
 public int Power(int b, int p) {
    ....}
 ....
}
class ColdPuzzle extends Puzzle{
 public int Power(int b, int p) {
    ....//body of coolpower}
}
.....
 Puzzle p = new ColdPuzzle();
 p.Tile(1,1); //which Power method will this invoke?
```


To see answer, let us write long version of method invocation:

```
 ....
public int Tile(int r, int c) {
 return state/this.Power(10, 3*r+c)%10; //long name of method
}
 ....
```

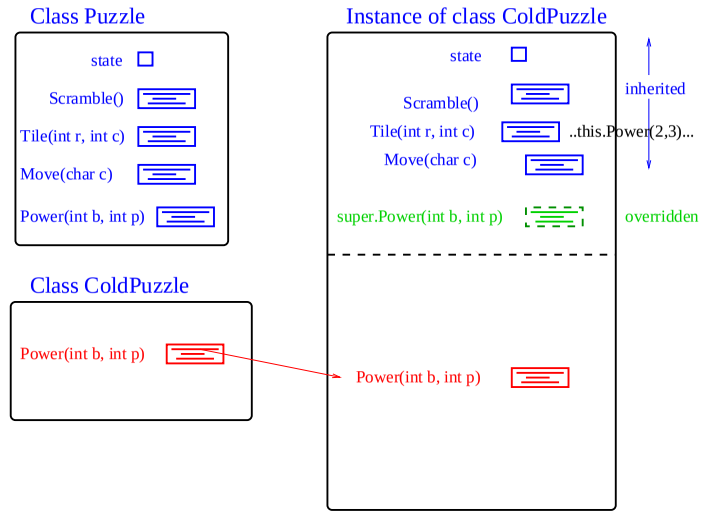
Type of reference `this` in class `Puzzle`: Puzzle

Type of object referred to by `this` in invocation `p.Tile`: `ColdPuzzle`

Use binding rule to determine which method is called =>
Method invoked is method in object (ie. method defined in `ColdPuzzle`)

Another way to see this: draw picture of object `p`.

Think: `Puzzle` class is written by supplier, but its behavior is modified by client code w/o altering code!



Unexpected consequence: subclass method that overrides a superclass method cannot have more restricted access than the superclass method

```
 class Puzzle {
    public int power (int b, int p) {
       .....}}
class ColdPuzzle extends Puzzle {
    private int power (int b, int p) {//illegal
       .... }}
....
ColdPuzzle Ob = new ColdPuzzle();
Ob.power(5); //should be illegal!
Puzzle bar = Ob;  //make superclass reference
bar.method2(5); //invokes method2 in class baz & is legal to boot!!
```

Java restriction: Overriding method in subclass must have same or less restricted access than overridden method in superclass — otherwise, it is an error.

Can we assign superclass reference to subclass reference?
Yes, but must use explicit cast.
Also, operation may give an error if cast is not meaningful.

```
Puzzle p = ...;
CoolPuzzle MC = (CoolPuzzle)p; //may or may not make sense
```

If object named by `p` is of type `CoolPuzzle` or one of `CoolPuzzle`'s subtypes, the cast is legal.

Otherwise, a runtime error (called an `exception`) occurs.

Summary of Dynamic Method Binding

```
class Puzzle {
 ...
  return state/this.Power(10, 3*r+c)%10;
}
```

The method actually invoked depends on type of object referred to by `this`.

Advantage: elegant way to modify behavior of class w/o modifying its code

Disadvantage: can be less efficient than procedure calls in C/Pascal

Overriding variables

We have seen that a subclass can override a superclass's methods by defining a method with the same name (eg. delete in class Creditcards).

Superclass variables can also be overridden the same way.
Example: if we define an integer instance variable NUM in class Creditcards, it would override NUM in class dBase.

Java: superclass variable can still be accessed in immediate subclass by using qualifier 'super' (as in super.NUM). Qualifier 'super' cannot be repeated (super.super.MUM is never legal).

It is usually bad practice to override variables.
We will not worry about it.

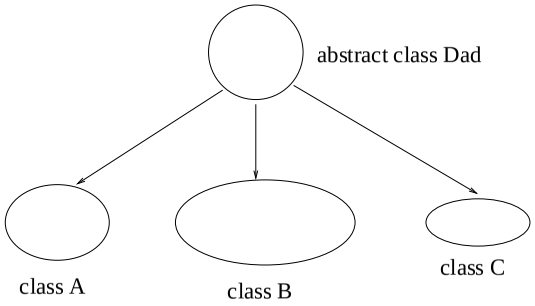
Abstract Classes

Abstract class has one or more methods that must be overridden by all subclasses.

Abstract method: only method header, no body.

```
abstract class Puzzle{
   protected int state;

   public void Scramble() {
     state = 876543210;
   }

   abstract public int Tile(int r, int c); //no implementation

   abstract public void Move(char c); //no implementation

  }
```

One use of abstract classes: repository for variables/methods common to a number of classes



Variables/methods that are common to classes A, B and C can be declared once in class Dad, and inherited by all subclasses.

If a subclass D wants to implement method differently, it can easily override superclass method.

Notes:

(1) Abstract class cannot be instantiated directly, since it has 'missing pieces'. Intuition: 'food' is an abstract class, 'cheeseburger' is a non-abstract (concrete) class.

(2) Not all methods in an abstract class need to be abstract.

(3) It is legal to assign a subclass object to an abstract superclass reference.

(4) Abstract classes are somewhere between interfaces and concrete classes: like interfaces, they cannot be instatiated, but like concrete classes, they have implementations that can be inherited by subclasses.

Cheat Sheet

Constructors: no inheritance of constructors.
However, superclass constructor can be invoked explicitly by invoking super() (with parameters if needed).

Overriding: superclass and subclass contain methods with same name and same *signature* (number and types of parameters).

Static methods can override static methods, instance methods can override instance methods. No 'mixing' of overriding is allowed.

What methods does a subclass inherit?

Superclass methods declared as public or protected, and not overridden in subclass.

Hidden variable: superclass variable whose name is used in subclass to define another variable. Superclass and subclass variables do not have to have same type. No restriction regarding static vs. instance variables.

What variables are inherited by a subclass?

Superclass variables declared as public or protected, and not hidden by subclass definition of a variable with same name.

What is object-oriented programming?

Encapsulation + Inheritance

Used intelligently, OO-programming enables the writing of more modular code than is possible with C or Pascal (where you can fake encapsulation but not inheritance).