Object-Oriented Programming

Relationships between classes:

- Inheritance

Access modifiers:

- Public, private, protected

Interfaces: Types Vs. Classes

Abstract classes

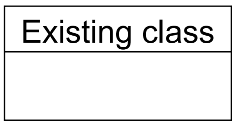
Packages

Design issues

Object-Oriented design & testing

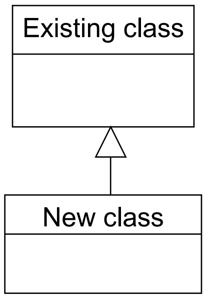
What Is Inheritance?

Creating new classes that are based on existing classes.

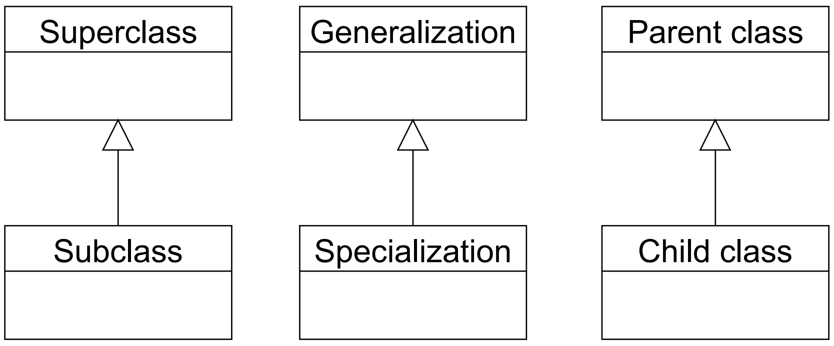


What Is Inheritance?

- Creating new classes that are based on existing classes.
- All non-private data and methods are available to the new class (but the reverse is not true).
- The new class is composed of information and behaviors of the existing class (and more).

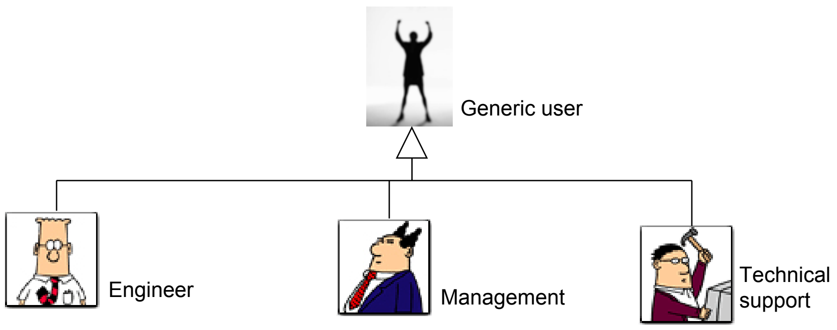


Inheritance Terminology



When To Employ Inheritance

- If you notice that certain behaviors or data is common among a group of candidate classes
- The commonalities may be defined by a superclass
- What is unique may be defined by particular subclasses



Dilbert © United Features Syndicate

Using Inheritance

Format:

```
class <Name of Subclass > extends <Name of Superclass>
{
 // Definition of subclass – only what is unique to subclass
}
```

Example:

```
class Dragon extends Monster
{
   public void displaySpecial ()
   {
      System.out.println("Breath weapon: ");
   }
}
```

The Parent Of All Classes

- You've already employed inheritance
- Class Object is at the top of the inheritance hierarchy (includes Class Array)
- Inheritance from class Object is implicit
- All other classes inherit it's data and methods
- For more information about this class see the url:
  http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html

Review: Relations Between Classes

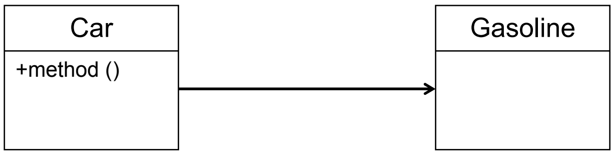
Association ("knows-a")
Aggregation ("has-a")
Inheritance ("is-a")

Association "Knows-a"

A association relation can exist between two classes if within one class' method(s), there exists as a local variable an instance of another class

e.g., A car uses (knows-a) gasoline

```
class Car
{
    public void method ()
    {
        Gasoline = new Gasoline ();
    }
}
```

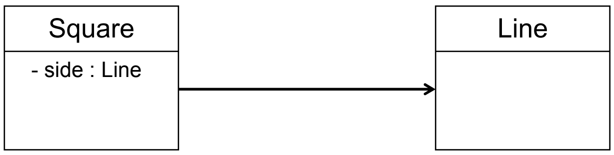


Association "Knows-a" (2)

A association relation can also exist between two classes if an instance of one class is an attribute of another class.

e.g., A square uses (knows-a) line

```
class Square
{
    private Line side;
}
```



Aggregation "Has-a"

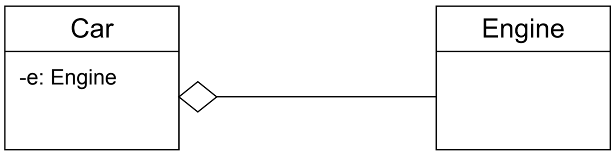
An aggregation relation exists between two classes if a field of one class is an instance of another class.

And

The second class is part of the first class.

e.g., A car has an (has-a) engine

```
class Car
{
    private Engine e;
}
```



Inheritance "Is-a"

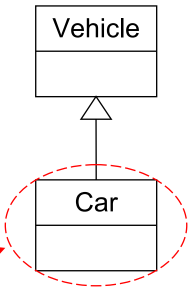
An inheritance relation exists between two classes if one class is the parent class of another class

e.g., A car is a type of (is-a) vehicle

```
class Vehicle
{
}

class Car extends Vehicle
{

}
```



Instances of the subclass can be used in place of instances of the super class

Levels Of Access Permissions

Private "-"

- Can only access the attribute/method in the methods of the class where the attribute is originally listed.

Protected "#"

- Can access the attribute/method in the methods of the class where the attribute is originally listed or the subclasses of that class.

Public "+"

- Can access attribute/method anywhere in the program

Levels Of Access Permissions

Access level	Accessible to		
	Same class	Subclass	Not a subclass
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Levels Of Access Permission: An Example

```
class P
{
        private int num1;
        protected int num2;
        public int num3;
        // Can access num1, num2 & num3 here.
}

class C extends P
{
   // Can access num2 & num3 here.
}

class Driver
{
   // Can only access num1 here.
}
```

Method Overriding

- Different versions of a method can be implemented in different ways by the parent and child class in an inheritance hierarchy.
- Methods have the same name and parameter list (identical signature) but different bodies
    - i.e., <method name> (<parameter list>)

Method Overloading Vs. Method Overriding

Method Overloading

- Multiple method implementations for the same class
- Each method has the same name but the type, number or order of the parameters is different (signatures are not the same)
- The method that is actually called is determined at *compile time*.
- i.e., <reference name>.<method name> $\underbrace{\text{(parameter list)}}_{\text{Distinguishes overloaded methods}}$;

James Tam

Method Overloading Vs. Method Overriding (2)

Example of method overloading:

```
class Foo
{
   display ();
   display (int i);
   display (char ch);
     :
}

Foo f = new Foo ();
f.display();
f.display(10);
f.display(‘c’);
```

James Tam

Method Overloading Vs. Method Overriding (3)

Method Overriding

- The method is implemented differently between the parent and child classes
- Each method has the same return value, name and parameter list (identical signatures)
- The method that is actually called is determined at *run time* (Polymorphism)
- i.e., <reference name>.<method name> (parameter list);

Type of reference (implicit parameter) distinguishes overridden methods

Method Overloading Vs. Method Overriding (4)

Example of method overriding:

```
class Foo
{
  display ();
     :
}
class FooChild extends Foo
{
  display ();
}

Foo f = new Foo ();
f.display();

FooChild fc = new FooChild ();
fc.display ();
```

Accessing The Unique Attributes And Methods Of The Parent

- All protected or public attributes and methods of the parent class can be accessed directly in the child class

e.g.

```
class P
{
        protected int num;
}

class C extends P
{
        public void method ()
        {
                this.num = 1;
                // OR
                num = 2;
        }
}
```

Accessing The Non-Unique Attributes And Methods Of The Parent

- An attribute or method exists in both the parent and child class (has the same name in both)
- The method or attribute has public or protected access
- Must prefix the attribute or method with "super" to distinguish it from the child class.
- Format:
  - super.*methodName* ()
  - super.*attributeName* ()

- Note: If you don't preface the method attribute with the keyword "super" then the by default the attribute or method of the child class will be accessed.

Accessing The Non-Unique Attributes And Methods Of The Parent: An Example

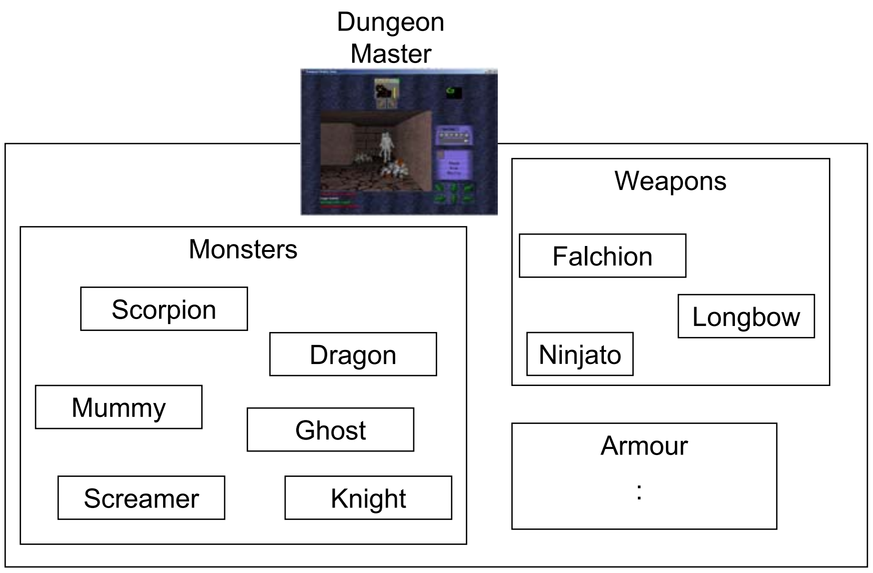
e.g.

```
class P
{
   protected int num;
   protected void method ()
   {
      :
   }
}
```

Accessing The Non-Unique Attributes And Methods Of The Parent: An Example (2)

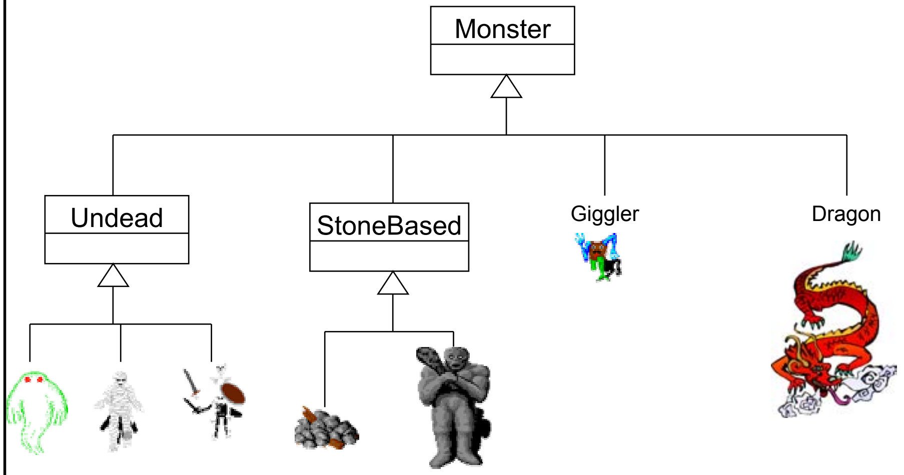
```
class C extends P
{
   protected int num;
   public void method ()
   {
      num = 2;
      super.num = 3;
      super.method();
}
```

A Blast From The Past



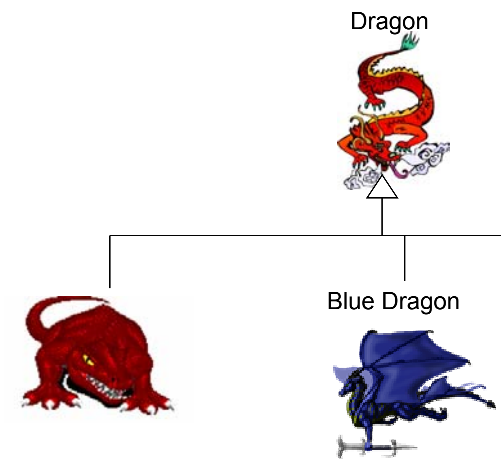
James Tam

The Inheritance Hierarchy For The Monsters

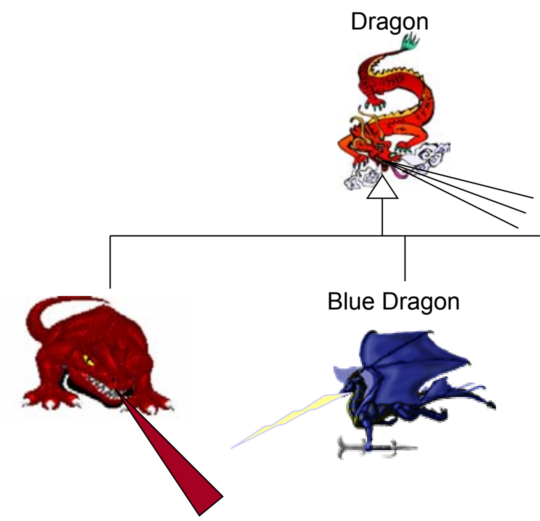


James Tam

The Dragon Sub-Hierarchy



The Dragon Sub-Hierarchy



Class DungeonMaster

Example (The complete example can be found in the directory /home/233/examples/object_programming/DMExample

```
class DungeonMaster
{
   public static void main (String [] args)
   {
      BlueDragon electro = new BlueDragon ();
      RedDragon pinky = new RedDragon ();
      HalitosisDragon stinky = new HalitosisDragon () ;

      electro.displaySpecialAbility ();
      pinky.displaySpecialAbility ();
      stinky.displaySpecialAbility ();
   }
}
```

Class Monster

```
class Monster
{
   private int protection;
   private int damageReceivable;
   private int damageInflictable;
   private int speed;
   private String name;

   public Monster ()
   {
      protection = 0;
      damageReceivable = 1;
      damageInflictable = 1;
      speed = 1;
      name = "Monster name: ";
   }
```

Class Monster (2)

```
public int getProtection () {return protection;}
public void setProtection (int i) {protection = i;}
public int getDamageReceivable () {return damageReceivable;}
public void setDamageReceivable (int i) {damageReceivable = i;}
public int getDamageInflictable () {return damageInflictable;}
public void setDamageInflictable (int i) {damageInflictable = i;}
public int getSpeed () {return speed;}
public void setSpeed (int i) {speed = i;}
public String getName () {return name; }
public void setName (String s) {name = s;}

public void displaySpecialAbility ()
{
   System.out.println("No special ability");
}
```

Class Monster (3)

```
    public String toString ()
    {
       String s = new String ();
       s = s + "Protection: " + protection + "\n";
       s = s + "Damage receivable: " + damageReceivable + "\n";
       s = s + "Damage inflictable: " + damageInflictable + "\n";
       s = s + "Speed: " + speed + "\n";
       s = s + "Name: " + name + "\n";
       return s;
    }
}  // End of definition for class Monster.
```

Class Dragon

```
class Dragon extends Monster
{
   public void displaySpecialAbility ()
   {
      System.out.print("Breath weapon: ");
   }
}
```

Class BlueDragon

```
class BlueDragon extends Dragon
{
   public void displaySpecialAbility ()
   {
      super.displaySpecialAbility ();
      System.out.println("Lightening");
   }
}
```

Class HalitosisDragon

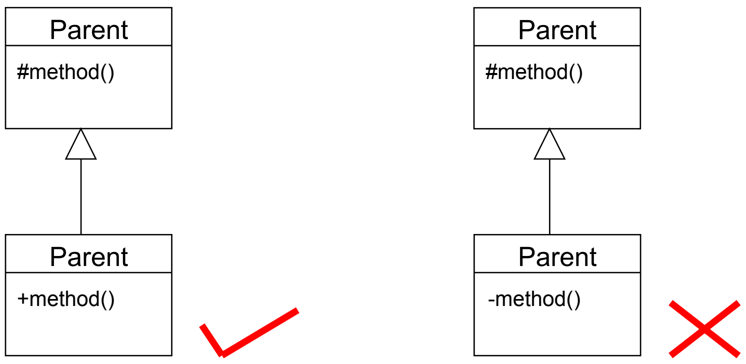
```
class HalitosisDragon extends Dragon
{
   public void displaySpecialAbility ()
   {
      super.displaySpecialAbility();
      System.out.println("Stinky");
   }
}
```

Class RedDragon

```
class RedDragon extends Dragon
{
   public void displaySpecialAbility ()
   {
      super.displaySpecialAbility();
      System.out.println("Fire");
   }
}
```

Changing Permissions Of Overridden Methods

The overridden method must have equal or stronger (less restrictive) access permissions in the child class.



The Final Modifier (Inheritance)

Methods preceded by the final modifier cannot be overridden

e.g., public ***final*** void displayTwo ()

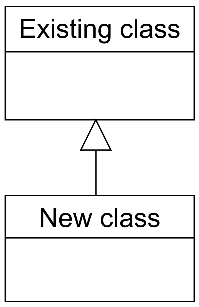
Classes preceded by the final modifier cannot be extended

•e.g., ***final*** class ParentFoo

Why Employ Inheritance

To allow for code reuse

It may result in more robust code



Shadowing

- •Local variables in a method or parameters to a method have the same name as instance fields
- •Fields of the subclass have the same name as fields of the superclass

Fields Of The Subclass Have The Same Names As The SuperClasses' Fields

```
class Foo
{
   private int num;
   public Foo () { num = 1; }
   public int getNum () { return num; }
   public void setNum (int no) {num = no; }
}

class Bar extends Foo
{
   public Bar ()
   {
      num = 10;
   }
}
```

Fields Of The Subclass Have The Same Names As The SuperClasses' Fields

```
class Foo
{
   private int num;
   public Foo () { num = 1; }
   public int getNum () { return num; }
   public void setNum (int no) {num = no; }
}

class Bar extends Foo
{
   public Bar ()
   {
      num = 10;
   }
}
```

Insufficient access permissions: Won't compile

Fields Of The Subclass Have The Same Names As The SuperClasses' Fields (2)

```
class Foo
{
   private int num;
   public Foo () { num = 1; }
   public int getNum () { return num; }
   public void setNum (int no) {num = no; }
}

class Bar extends Foo
{
   private int num;
   public Bar ()
   {
      num = 1;
   }
}
```

Fields Of The Subclass Have The Same Names As The SuperClasses' Fields (2)

```
class Foo
{
   private int num;
   public Foo () { num = 1; }
   public int getNum () { return num; }
   public void setNum (int no) {num = no; }
}

class Bar extends Foo
{
   private int num;
   public Bar ()
   {
      num = 1;
   }
}
```

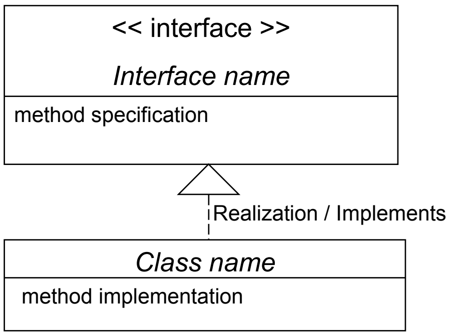
NO!

The Result Of Attribute Shadowing

```
class Bar extends Foo
{
   private int num;
   public Bar ()
   {
      num = 10;
   }
   public int getSecondNum () { return num; }
}
class Driver
{
   public static void main (String [] arv)
   {
      Bar b = new Bar ();
      System.out.println(b.getNum());
      System.out.println(b.getSecondNum());
   }
}
```

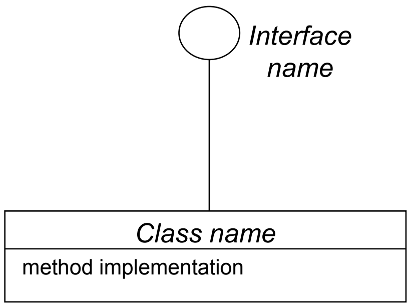
Java Interfaces (A Type)

- Similar to a class
- Provides a design guide rather than implementation details
- Specifies what methods should be implemented but not how
- Cannot be instantiated



Java Interfaces: Lollipop Notation

- Similar to a class
- Provides a design guide rather than implementation details
- Specifies what methods should be implemented but not how
- Cannot be instantiated



Interfaces: Format

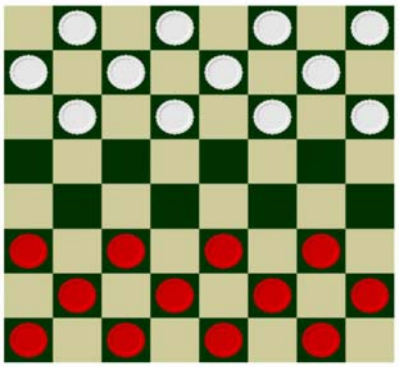
Format for specifying the interface

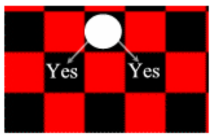
interface <*name of interface*>
{
  constants
  *methods **to be** implemented by the class that realizes this interface*
}

Format for realizing / implementing the interface

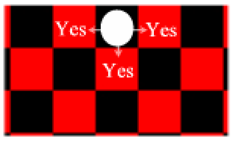
class <*name of class*> implements <*name of interface*>
{
  data fields
  *methods **actually** implemented by this class*
}

Interfaces: A Checkers Example


Basic board



Regular rules



Variant rules

Interface Board

```
interface Board
{
   public static final int SIZE = 8;
   public void displayBoard ();
   public void initializeBoard ();
   public void movePiece ();
   boolean moveValid (int xSource, int ySource, int xDestination,
                              int yDestination);
       :                              :
}
```

Class RegularBoard

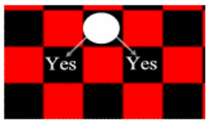
```
class RegularBoard implements Board
{
   public void displayBoard ()
   {
      :
   }

   public void initializeBoard ()
   {
      :
   }
```

Class RegularBoard (2)

```
   public void movePiece ()
   {
      // Get (x, y) coordinates for the source and destination
      if (moveValid == true)
            // Actually move the piece
      else
            // Don't move piece and display error message
   }

   public boolean moveValid (int xSource, int ySource, int xDestination,
                                      int yDestination)
   {
      if (moving diagonally forward)
            return true;
      else
            return false;
   }
}
```



Class VariantBoard

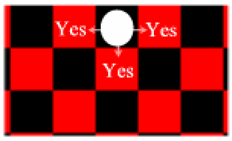
```
class VariantBoard implements Board
{
   public void displayBoard ()
   {
      :
   }

   public void initializeBoard ()
   {
      :
   }
```

Class VariantBoard (2)

```
   public void movePiece ()
   {
      // Get (x, y) coordinates for the source and destination
      if (moveValid == true)
           // Actually move the piece
      else
           // Don't move piece and display error message
   }

   public boolean moveValid (int xSource, int ySource, int xDestination,
                                          int yDestination)
   {
      if (moving straight-forward or straight side-ways)
           return true;
      else
           return false;
   }
}
```



Interfaces: Recapping The Example

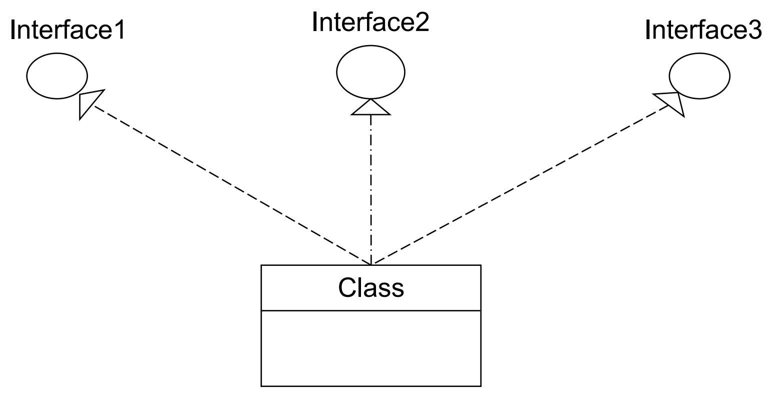
Interface Board

- No state (data) or behavior (body of the method is empty)
- Specifies the behaviors that a board *should* exhibit e.g., clear screen
- This is done by listing the methods that must be implemented by classes that implement the interface.

Class RegularBoard and VariantBoard

- Can have state and methods
- They must implement all the methods specified by interface Board (can also implement other methods too)

Implementing Multiple Interfaces



Implementing Multiple Interfaces

Format:

class <*class name*> implements <*interface name 1*>, <*interface name 2*>, <*interface name 3*>
{

}

Multiple Implementations Vs. Multiple Inheritance

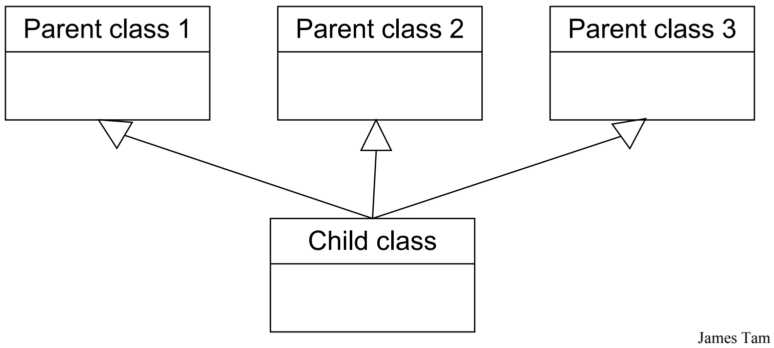
- A class can implement all the methods multiple interfaces
- Classes in Java cannot extend more than one class
- This is not possible in Java (but is possible in some other languages such as C++):

```
class <class name 1> extends <class name 2>, <class name 3>…
{

}
```

Multiple Implementation Vs. Multiple Inheritance (2)

- A class can implement all the methods of multiple interfaces
- Classes in Java cannot extend more than one class
- This is not possible in Java (but is possible in some other languages such as C++):



Abstract Classes

- Classes that cannot be instantiated
- A hybrid between regular classes and interfaces
- Some methods may be implemented while others are only specified
- Used when the parent class cannot define a default implementation (implementation must be specified by the child class).

Format:

```
abstract class <class name>
{
        <public/private/protected> abstract method ();
}
```

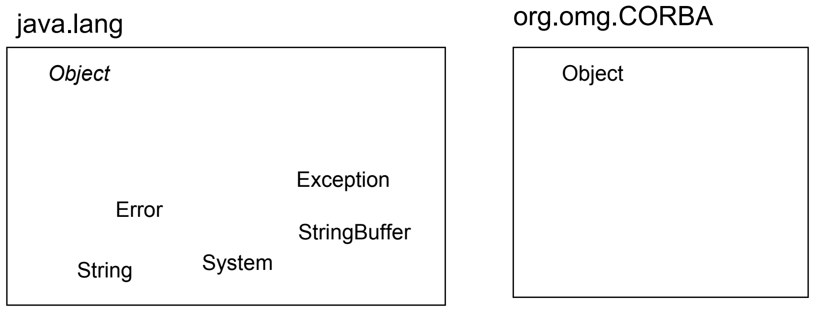
Abstract Classes (2)

Example:

```
abstract class BankAccount
{
   private float balance;
   public void displayBalance ()
   {
      System.out.println("Balance $" + balance);
   }
   public abstract void deductFees () ;
}
```

Packages

- A collection of related classes that are bundled together
- Used to avoid naming conflicts for classes
- Allows for only some implementation details to be exposed to other classes in the package (only some classes can be instantiated outside of the package)



Fully Qualified Names: Matches Directory Structure

$$\overbrace{\text{pack3}}^{\text{package name}}.\underbrace{\text{OpenFoo}}_{\text{class name}}.\underbrace{\text{toString}}_{\text{method name}}()$$

Importing Packages

Importing all classes from a package

Format

import <*package name*>.*;

Example

import java.util.*;

Importing a single class from a package

Format

import <package name>.<class name>;

Example

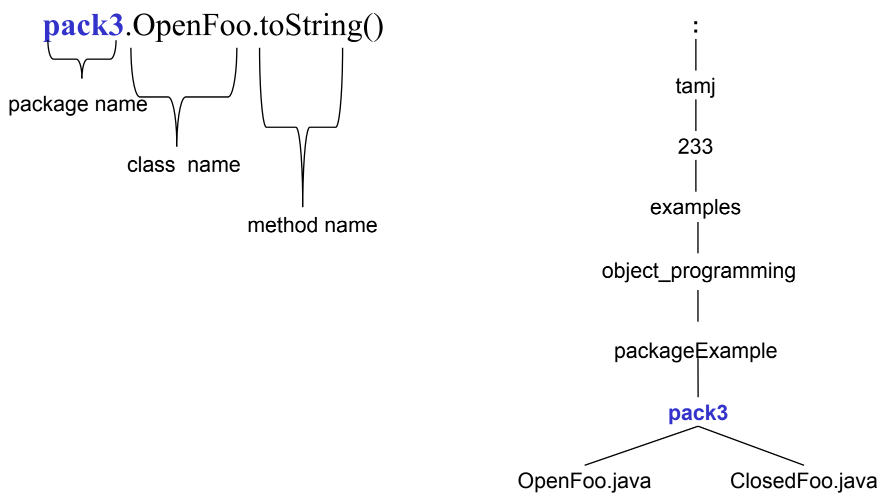
import java.util.Vector;

Importing Packages (2)

When you do not need an import statement:

- When you are using the classes in the java.lang package.
- You do not need an import statement in order to use classes which are part of the same package

Fully Qualified Names: Matches Directory Structure



Where To Match Classes To Packages

1. In directory structure: The classes that belong to a package must reside in the directory with the same name as the package (previous slide).
2. In the classes' source code: At the top class definition you must indicate the package that the class belongs to.

Format:

```
package <package name>;
<visibility – public or package> class <class name>
{

}
```

Matching Classes To Packages (2)

Example

```
package pack3;
public class OpenFoo
{
    :
}

package pack3;
class ClosedFoo
{
    :
}
```

Matching Classes To Packages (2)

Example

```
package pack3;
public class OpenFoo
{
    :
}
```

Public access: Class can be instantiated by classes that aren't a part of package pack3

```
package pack3;
class ClosedFoo
{
    :
}
```

Package access (default): Class can only be instantiated by classes that are a part of package pack3

Sun's Naming Conventions For Packages

Based on Internet domains (registered web addresses)

e.g., www.tamj.com

```
com.tamj .games
         .productivity
```

Sun's Naming Conventions For Packages

Alternatively it could be based on your email address

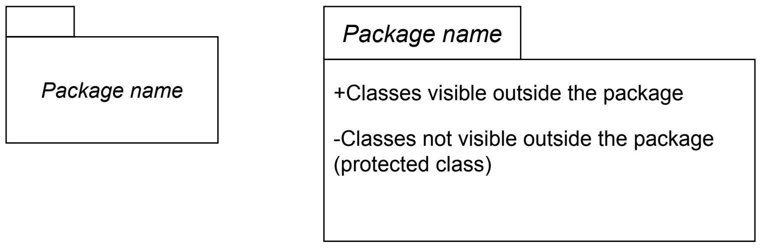
e.g., $\underbrace{\text{tamj@cpsc.ucalgary.ca}}$

```
ca.ucalgary.cpsc.tamj .games
                      .productivity
```

Default Package

- If you do not use a package statement then the class implicitly becomes part of a default package
- All classes which reside in the same directory are part of the default package.

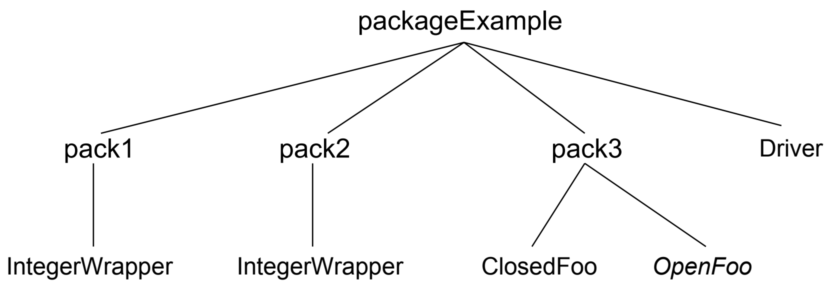
Graphically Representing Packages In UML


Package name
Package name
+Classes visible outside the package
-Classes not visible outside the package (protected class)

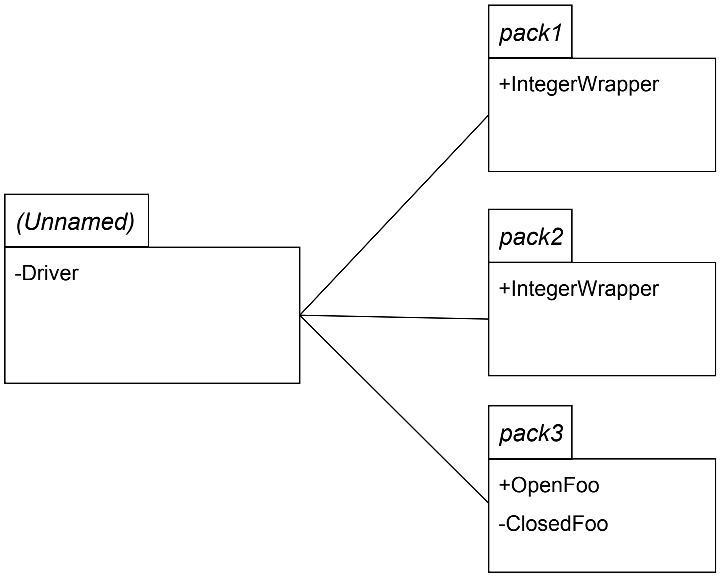
Packages An Example

The complete example can be found in the directory:
/home/233/examples/object_programming/packageExample

(But you should have guessed the path from the package name)



Graphical Representation Of The Example



Package Example: The Driver Class

```
import pack3.*;
class Driver
{
   public static void main (String [] argv)
   {
      pack1.IntegerWrapper iw1 = new pack1.IntegerWrapper ();
      pack2.IntegerWrapper iw2 = new pack2.IntegerWrapper ();
      System.out.println(iw1);
      System.out.println(iw2);

      OpenFoo of = new OpenFoo ();
      System.out.println(of);
      of.manipulateFoo();
      }
}
```

Package Example: Package Pack1, Class IntegerWrapper

```
package pack1;
public class IntegerWrapper
{
   private int num;

   public IntegerWrapper ()
   {
      num = (int) (Math.random() * 10);
   }
   public IntegerWrapper (int no)
   {
      num = no;
   }
   public void setNum (int no)
   {
      num = no;
   }
```

Package Example: Package Pack1, Class IntegerWrapper (2)

```
   public int getNum ()
   {
      return num;
   }

   public String toString ()
   {
      String s = new String ();
      s = s + num;
      return s;
   }
}
```

Package Example: Package Pack2, Class IntegerWrapper

```
package pack2;
public class IntegerWrapper
{
   private int num;

   public IntegerWrapper ()
   {
      num = (int) (Math.random() * 100);
   }
   public IntegerWrapper (int no)
   {
      num = no;
   }
   public void setNum (int no)
   {
      num = no;
   }
```

Package Example: Package Pack2, Class IntegerWrapper (2)

```
   public int getNum ()
   {
      return num;
   }

   public String toString ()
   {
      String s = new String ();
      s = s + num;
      return s;
   }
}
```

Package Example: Package Pack3, Class OpenFoo

```
package pack3;
public class OpenFoo
{
   private boolean bool;
   public OpenFoo () { bool = true; }
   public void manipulateFoo ()
   {
      ClosedFoo cf = new ClosedFoo ();
      System.out.println(cf);
   }
   public boolean getBool () { return bool; }
   public void setBool (boolean b) { bool = b; }
   public String toString ()
   {
      String s = new String ();
      s = s + bool;
      return s;
   }
}
```

Package Example: Package Pack3, Class ClosedFoo

```
package pack3;
class ClosedFoo
{
   private boolean bool;

   public ClosedFoo () { bool = false; }
   public boolean getBool () { return bool; }

   public void setBool (boolean b) { bool = b; }

   public String toString ()
   {
      String s = new String ();
      s = s + bool;
      return s;
   }
}
```

Updated Levels Of Access Permissions: Attributes And Methods

Private "-"

- Can only access the attribute/method in the methods of the class where the attribute is originally listed.

Protected "#"

- Can access the attribute/method in the methods of the class where the attribute is originally listed or the subclasses of that class.

Package - no UML symbol for this permission level

- Can access the attribute/method from classes within the same package
- If the level of access is unspecified this is the default level of access

Public "+"

- Can access attribute/method anywhere in the program

Updated Levels Of Access Permissions

	Accessible to			
Access level	Same class	Class in same package	Subclass in a different package	Not a subclass, different package
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Package	Yes	Yes	No	No
Private	Yes	No	No	No

Some Principles Of Good Design

1. Avoid going "method mad"
2. Keep an eye on your parameter lists
3. Avoid real values when an integer will do
4. Minimize modifying immutable objects
5. Be cautious in the use of references
6. Consider where you declare local variables

This list was partially derived from "Effective Java" by Joshua Bloch and is by no means complete. It is meant only as a starting point to get students thinking more about why a practice may be regarded as "good" or "bad" style.

Avoid Going Method Mad

- There should be a reason for each method
- Creating too many methods makes a class difficult to understand, use and maintain
- A good approach is to check for redundancies that exist between different methods

Keep An Eye On Your Parameter Lists

Avoid long parameter lists

- Rule of thumb: Three parameters is the maximum

Avoid distinguishing overloaded methods solely by the order of the parameters

Avoid Real Values When An Integer Will Do

```
double db = 1.03 - 0.42;
if (db == 0.61)
    System.out.println("Sixty one cents");
System.out.println(db);
```

Minimize Modifying Immutable Objects

Immutable objects

Once instantiated they cannot change (all or nothing)

```
e.g., String s = "hello";
      s = s + " there";
```

Minimize Modifying Immutable Objects (2)

If you must make changes substitute immutable objects with mutable ones

e.g.,

```
class StringBuffer
{
      public StringBuffer (String str);
      public StringBuffer append (String str);
          :       :          :        :

}
```

For more information about this class

http://java.sun.com/j2se/1.4/docs/api/java/lang/StringBuffer.html

Minimize Modifying Immutable Objects (3)

```
class StringExample
{
   public static void main (String [] args)
   {
      String s = "0";
       for (int i = 1; i < 10000; i++)
          s = s + i;
   }
}
```

```
class StringBufferExample
{
   public static void main (String [] args)
   {
      StringBuffer s = new StringBuffer("0");
      for (int i = 1; i < 10000; i++)
         s = s.append(i);
   }
}
```

Be Cautious In The Use Of References

Similar to global variables:

```
program globalExample (output);
var
  i : integer;

procedure proc;
begin
  for i:= 1 to 100 do;
end;

begin
  i := 10;
  proc;
end.
```

Be Cautious In The Use Of References (2)

```
class Foo
{
   private int num;
   public int getNum () {return num;}
   public void setNum (int no) {num = no;}
}
```

Be Cautious In The Use Of References (3)

```
class Driver
{
   public static void main (String [] argv)
   {
      Foo f1, f2;
      f1 = new Foo ();
      f1.setNum(1);

      f2 = f1;
      f2.setNum(2);

      System.out.println(f1.getNum());
      System.out.println(f2.getNum());

   }
}
```

Consider Where You Declare Local Variables

First Approach: Declare all local variables at the beginning of a method:

```
void methodName (..)
{
    int num;
    char ch;
      :

}
```

Advantage:

- Putting all variable declarations in one place makes them easy to find

Consider Where You Declare Local Variables (2)

Second Approach: declare local variables only as they are needed

```
void methodName (..)
{
    int num;
    num = 10;
      :
    char ch;
    ch = ‘a’;

}
```

Advantage:

- For long methods it can be hard to remember the declaration if all variables are declared at the beginning
- Reducing the scope of a variable may reduce logic errors

Object-Oriented Design And Testing

- Start by employing a top-down approach to design
  - Start by determining the candidate classes in the system
  - Outline a skeleton for candidate classes (methods are stubs)
- Implement each method one at-a-time.
- Create test drivers for methods that have side-effects.
- Fix any bugs in these methods
- Add the working methods to the code for the class.

Determine The Candidate Classes

Example:

A utility company provides three types of utilities:

1. Electricity:

   Bill = No. of kilowatt hours used * $0.01

2. Gas:

   Bill = No. of gigajoules used * $7.50

3. Water

   a) Flat rate: $10.00 + (square footage of dwelling * $0.01)

   b) Metered rate: $1.00 * No. cubic of meters used

Determine The Candidate Classes (2)

Some candidate classes

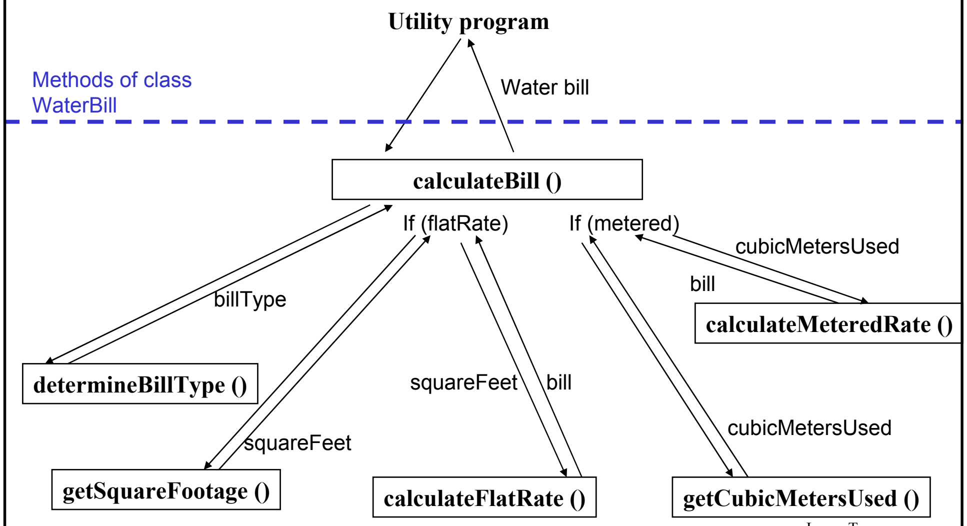
- •ElectricityBill
- •WaterBill
- •GasBill

Skeleton For Class WaterBill

```
class WaterBill
{
    private char billType;
    private double bill;
    public static final double RATE_PER_SQUARE_FOOT = 0.01;
    public static final double BASE_FLAT_RATE_VALUE = 10.0;
    public static final double RATE_PER_CUBIC_METER = 1.0;

   public WaterBill ()
   {
   }
     :          :         :
```

Determining The Remaining Methods



Remaining Skeleton For Class WaterBill (2)

```
public double calculateBill () { return 1.0;}
public void determineBillType () { }
public int getSquareFootage () { return 1; }
public double calculateFlatRate (int squareFootage) { return 1.0; }
public double cubicMetersUsed () { return 1.0; }
public double calculateMeteredRate (double cubicMetersUsed) { return; }
```

Implementing The Bodies For The Methods

1. calculateBill
2. determineBillType
3. getSquareFootage
4. calculateFlatRate (*to be tested*)
5. cubicMetersUsed
6. calculateMeteredRate (*to be tested*)

Body For Method CalculateBill

```
public double calculateBill ()
{
   int squareFootage;
   double cubicMetersUsed;
   determineBillType();
   if (billType == 'f')
   {
      squareFootage = getSquareFootage ();
      bill = calculateFlatRate (squareFootage);
   }
   else if (billType == 'm')
   {
      cubicMetersUsed = getCubicMetersUsed();
      bill = calculateMeteredRate (cubicMetersUsed);
   }
   else
   {
      System.out.println("Bill must be either based on a flat rate or metered.");
   }
   return bill;
}
```

Body For DetermineBillType

```
public void determineBillType ()
{
   System.out.println("Please indicate the method of billing.");
   System.out.println("(f)lat rate");
   System.out.println("(m)etered billing");
   billType = (char) Console.in.readChar();
   Console.in.readChar();
}
```

Body For GetSquareFootage

```
public int getSquareFootage ()
{
   int squareFootage;
   System.out.print("Enter square footage of dwelling: ");
   squareFootage = Console.in.readInt();
   Console.in.readChar();
   return squareFootage;
}
```

Body For CalculateFlatRate

```
public double calculateFlatRate (int squareFootage)
{
   double total;
   total = BASE_FLAT_RATE_VALUE + (squareFootage *
            RATE_PER_SQUARE_FOOT);
   return total;
}
```

James Tam

Creating A Driver For CalculateFlatRate

```
class DriverCalculateFlatRate
{
   public static void main (String [] args)
   {
      WaterBill water = new WaterBill ();
      double bill;
      int squareFootage;

      squareFootage = 0;
      bill = water.calculateFlatRate(squareFootage);
      if (bill != 10)
         System.out.println("Incorrect flat rate for 0 square feet");
      else
         System.out.println("Flat rate okay for 0 square feet");
```

James Tam

Creating A Driver For CalculateFlatRate (2)

```
        squareFootage = 1000;
        bill = water.calculateFlatRate(squareFootage);
        if (bill != 20)
            System.out.println("Incorrect flat rate for 1000 square feet");
        else
            System.out.println("Flat rate okay for 1000 square feet");
    }
} // End of Driver
```

Body For GetCubicMetersUsed

```
    public double getCubicMetersUsed ()
    {
        double cubicMetersUsed;
        System.out.print("Enter the number of cubic meters used: ");
        cubicMetersUsed = Console.in.readDouble();
        Console.in.readChar();
        return cubicMetersUsed;
    }
```

Body For CalculateMeteredRate

```
public double calculateMeteredRate (double cubicMetersUsed)
{
   double total;
   total = cubicMetersUsed * RATE_PER_CUBIC_METER;
   return total;
}
```

Driver For CalculateMeteredRate

```
class DriverCalculateMeteredRate
{
   public static void main (String [] args)
   {
      WaterBill water = new WaterBill ();
      double bill;
      double cubicMetersUsed;

      cubicMetersUsed = 0;
      bill = water.calculateMeteredRate(cubicMetersUsed);
      if (bill != 0 )
         System.out.println("Incorrect metered rate for 0 cubic meters consumed.");
      else
         System.out.println("Metered rate for 0 cubic meters consumed is okay.");
```

Driver For CalculateMeteredRate (2)

```
        cubicMetersUsed = 100;
        bill = water.calculateMeteredRate(cubicMetersUsed);
        if (bill != 100 )
           System.out.println("Incorrect metered rate for 100 cubic meters
              consumed.");
        else
           System.out.println("Metered rate for 100 cubic meters consumed is
              okay.");
    }
}
```

General Rule Of Thumb: Test Drivers

- Write a test driver class if you need to verify that a method does what it is supposed to do.
    - e.g., When a method performs a calculation
- Benefits of writing test drivers:

1) Ensuring that you know precisely what your code is supposed to do.
2) Making code more robust (test it before adding it the code library).

You Should Now Know

How the inheritance relationship works

- When to employ inheritance and when to employ other types of relations
- What are the benefits of employing inheritance
- How to create and use an inheritance relation in Java
- What is the effect of the keyword "final" on inheritance relationships

What is method overloading?

- How does it differ from method overriding
- What is polymorphism

What are interfaces/types

- How do types differ from classes
- How to implement and use interfaces in Java

You Should Now Know (2)

What are abstract classes in Java and how do they differ from non-abstract classes and interfaces

UML notations for inheritance and packages

How do packages work in Java

- How to utilize the code in pre-defined packages
- How to create your own packages

How the 4 levels of access permission work

Some general design principles

- What constitutes a good or a bad design.

How to write test drives and what are the benefits of using test drivers in your programs