Classes And Objects Part III

Relationships between classes:
- Inheritance

Access modifiers:
- Public, private, protected

Interfaces: Types Vs. Classes

Abstract classes

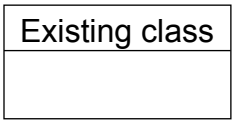
Packages

Design issues for Object-Oriented systems

Object-Oriented design & testing

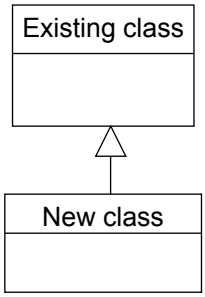
What Is Inheritance?

Creating new classes that are based on existing classes.

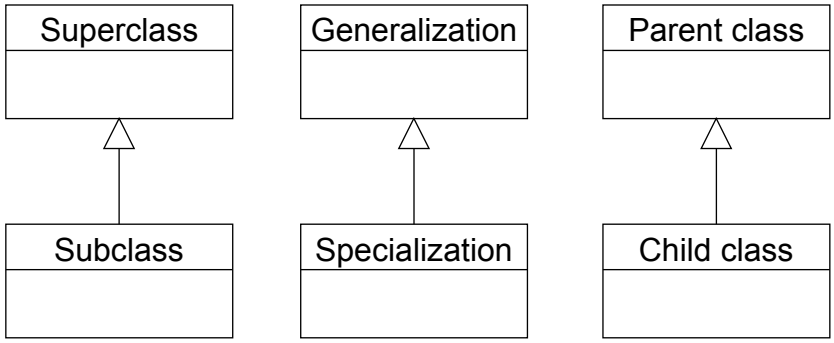


What Is Inheritance?

- 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 the information and behaviors of the existing class (and more).

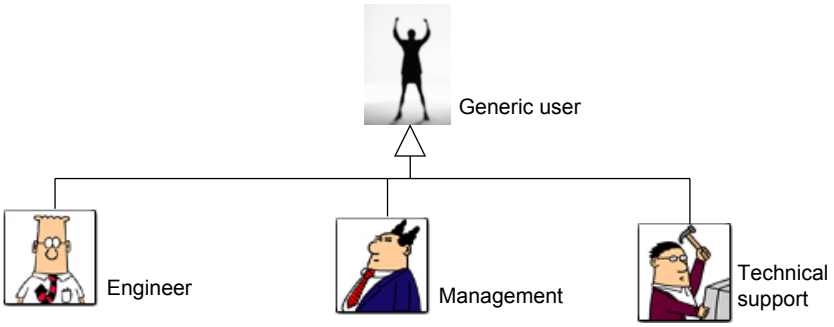


Inheritance Terminology



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
  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

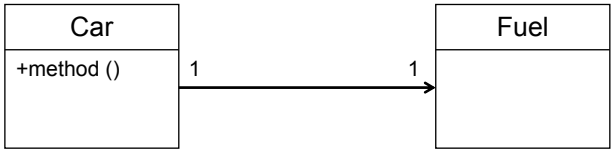
Association ("knows-a")
Aggregation ("has-a")
Inheritance ("is-a")

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) instance of fuel

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

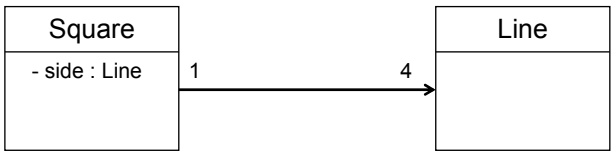


Association "Knows-As" (2)

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 one class is an attribute of another class.

And

The first class is part of the second class (or the second class is an aggregate of the first class)

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

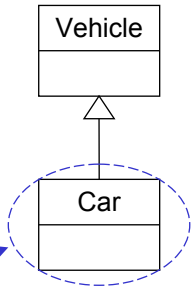
```
class Car
{
      private Engine e;
}
```

Inheritance "Is-A"

An inheritance relation exists between two classes if one class is one type 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 defined.

Protected "#"

- Can access the attribute/method in the methods of the class where the attribute is originally defined 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't access num1 here
}

class Driver
{
   // Can't access num1 here.
}
```


General Rule Of Thumb

- Variable attributes should not have protected access but instead should be private.
- Most methods should be public
- Methods that are used only by the parent and child classes should be made protected.

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
  - e.g.,

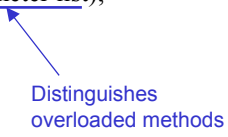
```
class Parent
{
        :           :
    public void method ()
    {
        System.out.println("m1");
    }
}
```

```
class Child extends Parent
{
        :           :
    public void method ()
    {
        num = 1;
    }
}
```


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 program *compile time* (early binding).
- i.e., <reference name>.<method name> (parameter list);



Method Overloading Vs. Method Overriding (2)

Example of method overloading:

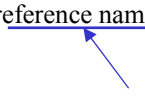
```
class Foo
{
    public void display () { }
    public void display (int i) { }
    public void display (char ch) { }
}
```

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


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 program *run time* (late binding)
- i.e., <reference name>.<method name> (parameter list);



The type of the reference (implicit parameter "this") distinguishes overridden methods

Method Overloading Vs. Method Overriding (4)

Example of method overriding:

```
class Foo
{
   public void display () { … }
      :            :
}
class FooChild extends Foo
{
    public void display () { … }
}

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

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


Polymorph

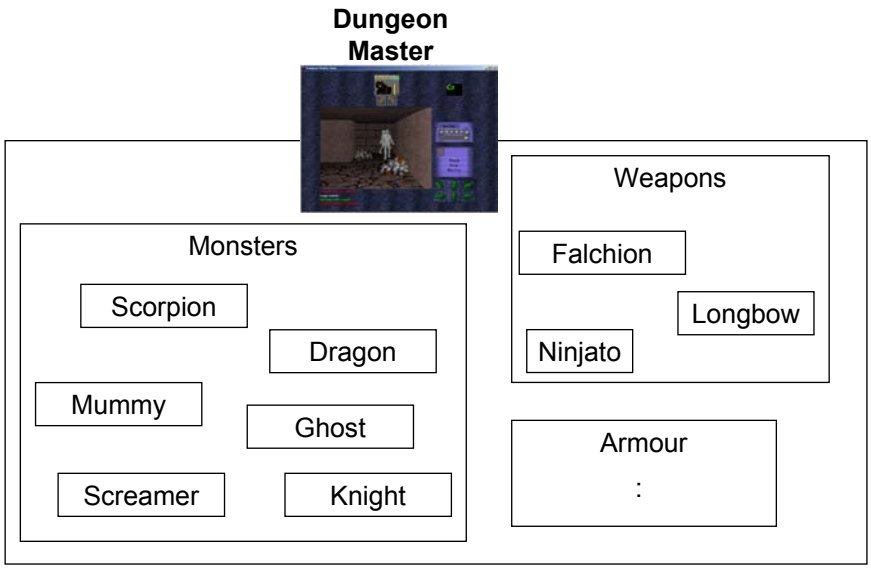
The ability to take on different forms

Images from the game Dungeon Master by FTL

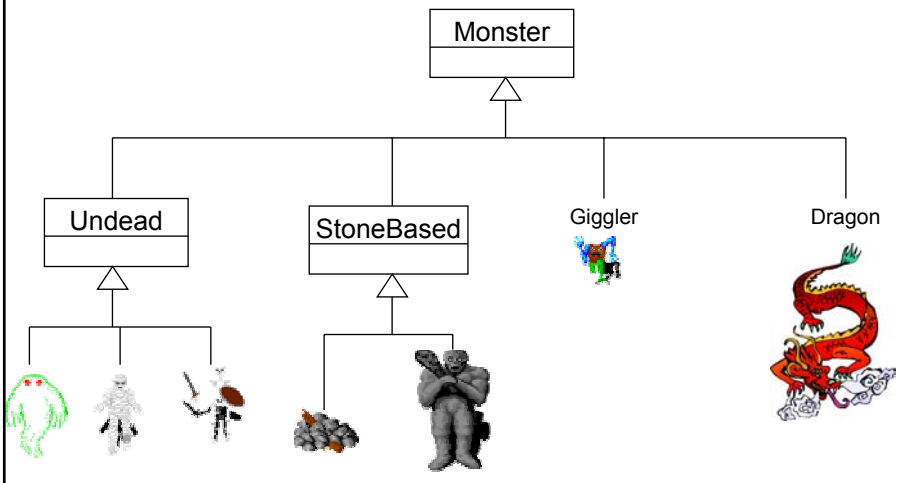
Polymorphism In Object-Orientated Theory

- An overridden method that can take on many forms
- The type of an instance (the implicit parameter) determines at program run-time which method will be executed.

A Blast From The Past

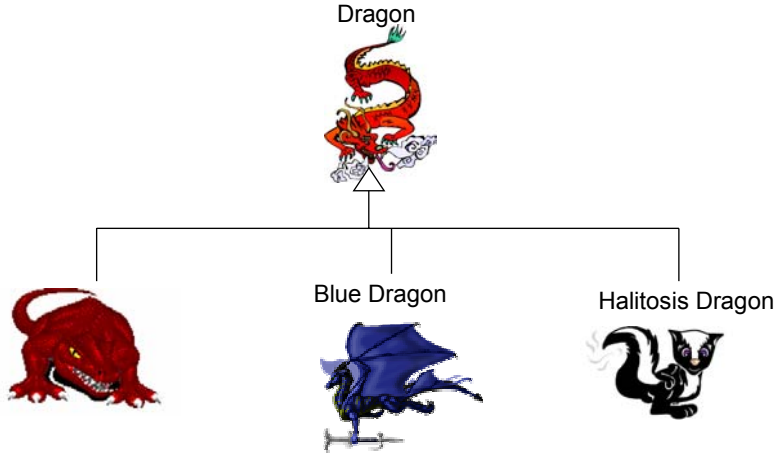


The Inheritance Hierarchy For The Monsters



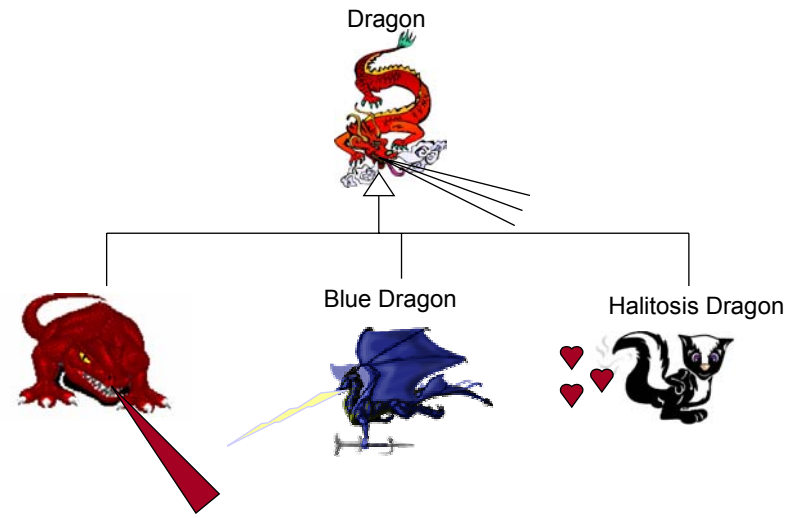
James Tam

The Dragon Sub-Hierarchy



James Tam

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


Updated Scoping Rules

When referring to an identifier in the method of a class

1. Look in the local memory space for that method
2. Look in the definition of the class
3. Look in the definition of the classes' parent

Updated Scoping Rules (2)

```
class P
{
    <<< Third >>>
}
class C extends P
{
    <<< Second >>>
    public void method ()
    {
        <<< First >>>
    }
}
```


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.

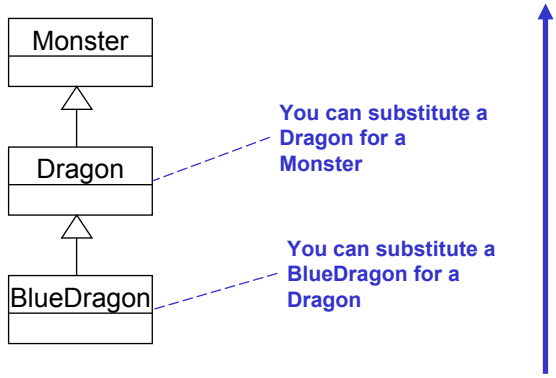
```
class P
{
   protected int num;
   protected void method ()
   {
      :
   }
}
```

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();
}
```

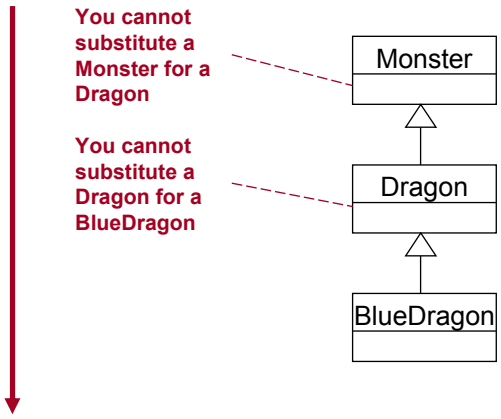

Casting And Inheritance

- Remember: You can substitute instances of a subclass for instances of a superclass.



Casting And Inheritance (2)

- Remember: You cannot substitute instances of a superclass for instances of a subclass



Casting And Inheritance: A Previous Example

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

      :        :       :
   public int getProtection () {return protection;}
      :        :       :
}
```

Casting And Inheritance: An Previous Example

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

   public void fly ()
   {
      System.out.println("Flying");
   }
}
```

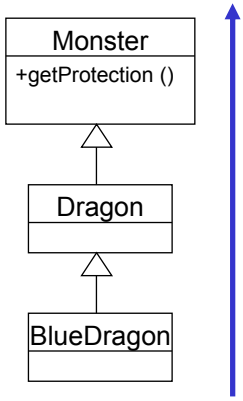

Casting And Inheritance: An Previous Example

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

   public void absorbElectricity ()
   {
      System.out.println("Absorbing electricity.");
   }
}
```

Substituting Sub And Super Classes

- You can substitute an instance of a sub class for an instance of a super class.

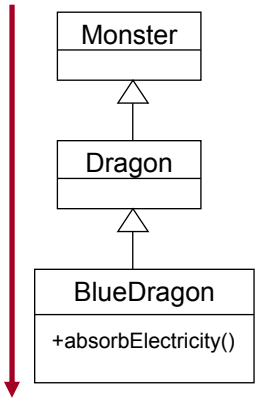
```
BlueDragon electro = new BlueDragon ();
Monster aMonster = new Monster ();

System.out.println(aMonster.getProtection());
System.out.println(electro.getProtection());
```



Substituting Sub And Super Classes

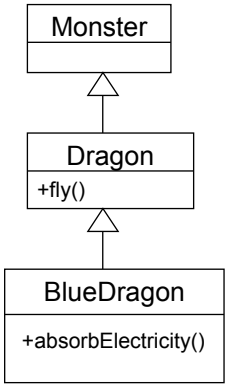
- You cannot substitute an instance of a super class for an instance of a sub class.

```
BlueDragon electro = new BlueDragon ();
Monster aMonster = new Monster ();

electro.absorbElectricity ();
aMonster.absorbElectricity ();
```



Casting And Inheritance

```
BlueDragon electro = new BlueDragon ();
Monster aMonster;

aMonster = electro;
aMonster.fly();
aMonster.absorbElectricity();

aMonster = new Monster ();
electro = aMonster;

electro = (BlueDragon) aMonster;
electro.fly();
electro.absorbElectricity();
```



Casting And Inheritance (2)

- Only use the cast operator if you are sure of the type.

```
BlueDragon electro = new BlueDragon ();
Monster aMonster;
aMonster = electro;

 if (aMonster instanceof BlueDragon)
{
      System.out.println("AMonster is a reference to an instance of a
           BlueDragon");
      electro = (BlueDragon) aMonster;
      electro.fly();
      electro.absorbElectricity();
}
```

Casting And Inheritance (3)

- Only use the cast operator if you are sure of the type.

```
BlueDragon electro = new BlueDragon ();
Monster aMonster;
aMonster = electro;

if (aMonster instanceof BlueDragon)
{
    System.out.println("AMonster is actually a reference to an instance of
        a BlueDragon");
    ((BlueDragon) aMonster).fly();
    ((BlueDragon) aMonster).absorbElectricity();
}
```


Shadowing

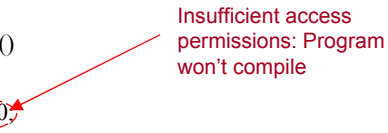
- Local variables in a method or parameters to a method have the same name as instance fields
- Attributes of the subclass have the same name as attributes of the superclass

Attributes Of The Subclass Have The Same Name As The SuperClasses' Attributes

```
class Foo
{
   private int num;
   public Foo () { num = 1; }
   public int getNum () { return num; }
   public void setNum (int newValue) {num = newValue; }
}

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


Attributes Of The Subclass Have The Same Name As The SuperClasses' Attributes

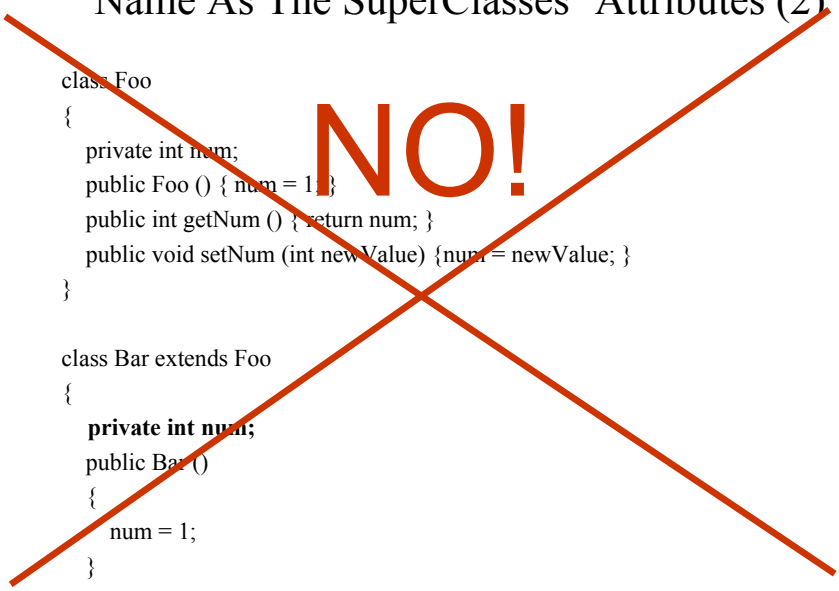
```
class Foo
{
   private int num;
   public Foo () { num = 1; }
   public int getNum () { return num; }
   public void setNum (int newValue) {num = newValue; }
}

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



Attributes Of The Subclass Have The Same Name As The SuperClasses' Attributes (2)

```
class Foo
{
   private int num;
   public Foo () { num = 1; }
   public int getNum () { return num; }
   public void setNum (int newValue) {num = newValue; }
}

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


Attributes Of The Subclass Have The Same Name As The SuperClasses' Attributes (2)




```
class Foo
{
   private int num;
   public Foo () { num = 1; }
   public int getNum () { return num; }
   public void setNum (int newValue) {num = newValue; }
}

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

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());
   }
}
```


Another Scoping Example

```
class ScopingExample
{
   public static void main (String [] args)
   {
      P p1 = new P ();
      C c1 = new C ();
      GC gc = new GC ();
      gc.method1();
      gc.method2();
      gc.method3();
      gc.method();
   }
}
```

Another Scoping Example (2)

```
class GC extends C
{
   private int num1;
   public GC ()
   {
      num1 = 1;
   }
   public void method1 ()
   {
      System.out.println("GC's method 1");
      super.method1();
   }
   public void method2 ()
   {
      System.out.println("GC's method 2");
      super.method2();
   }
```


Another Scoping Example (3)

```
    public void method3 ()
   {
      int num0 = 0;
      System.out.println("num0=" + num0);
      System.out.println("num1=" + num1);
      System.out.println("num2=" + num2);
      System.out.println("num3=" + num3);
      System.out.println("ch=" + ch);
   }

   public void method ()
   {
      super.method1();
   }
} // End of class GC
```

Another Scoping Example (4)

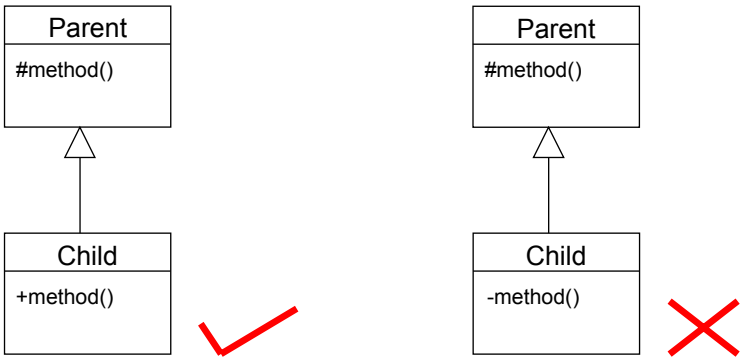
```
class C extends P
{
   protected int num2;
   protected char ch1;

   public C ()
   {
      ch = 'C';
      num2 = 2;
   }
   public void method1 ()
   {
      System.out.println("C's method 1");
   }
   public void method2 ()
   {
      System.out.println("C's method 2");
      super.method2();
   }
} // End of class C
```


Another Scoping Example (5)

```
class P
{
   protected int num3;
   protected char ch;
   public P ()
   {
      ch = 'P';
      num3 = 3;
   }
   public void method1 ()
   {
      System.out.println("P's method 1");
   }
   public void method2 ()
   {
      System.out.println("P's method 2");
   }
} // End of class P
```

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

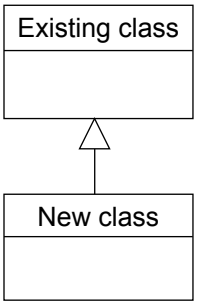
e.g., public ***final*** void displayTwo ()

Classes preceded by the final modifier cannot be extended

- e.g., ***final*** class ParentFoo

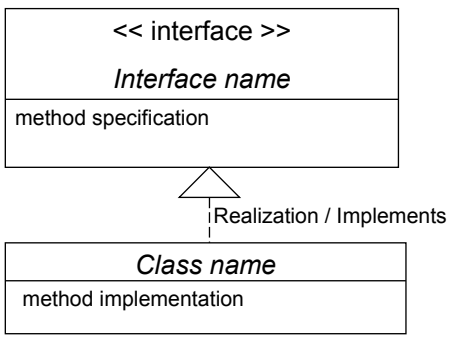
Why Employ Inheritance

- To allow for code reuse
- It may result in more robust code



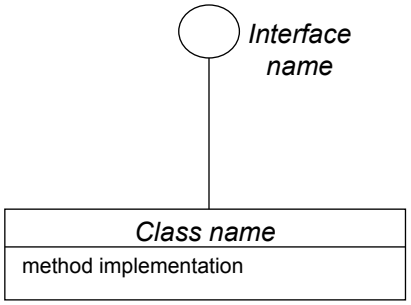
Java Interfaces (Type)

- 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 (Type): 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

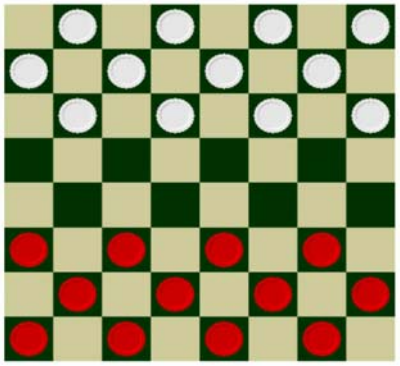
Format for defining an interface

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

Format for realizing / implementing the interface

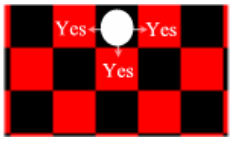
class <*name of class*> implements <*name of interface*>
{
    attributes
    methods ***actually*** *implemented by this class*
}

Interfaces: A Checkers Example



Basic board

Regular rules



Variant rules

Interface Board

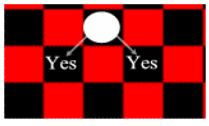
```
interface Board
{
   public static final int SIZE = 8;
   public void displayBoard ();
   public void initializeBoard ();
   public void movePiece ();
   boolean moveValid (int xSource, int ySource, int xDestination,
                      int yDestination);
   public void displayBoard ();
   public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

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 forward diagonally)
            return true;
      else
            return false;
   }
}
```

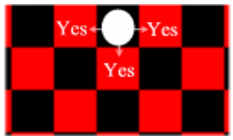


Class VariantBoard

```
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

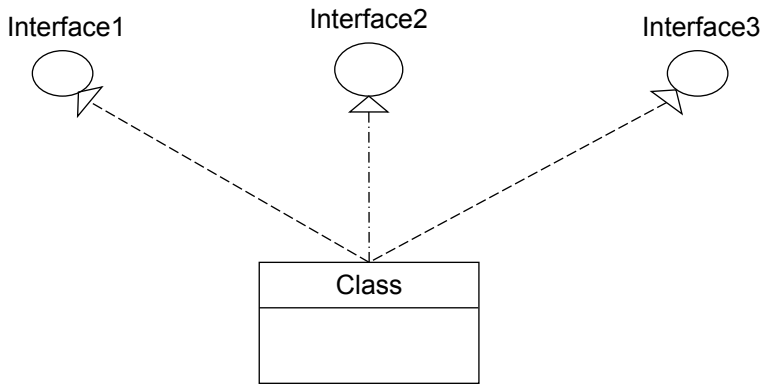
Interface Board

- 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 (but 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*>…
{

}

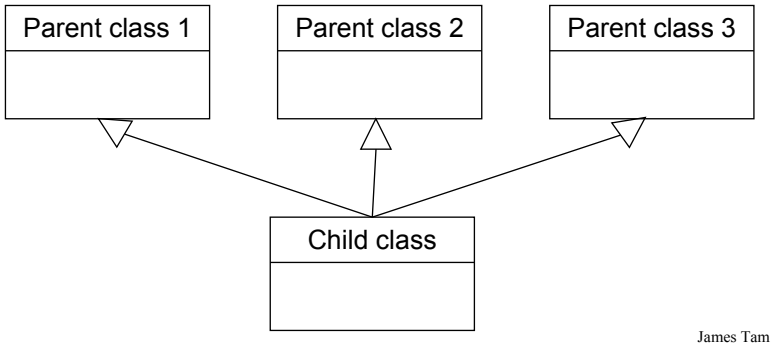
Multiple Implementations Vs. Multiple Inheritance

- 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 Implementations 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 complete default implementation (implementation must be specified by the child class).

Format:

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

Abstract Classes (2)

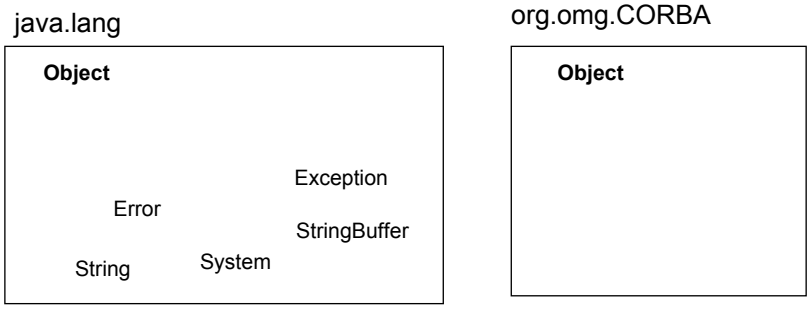
Example[1]:

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

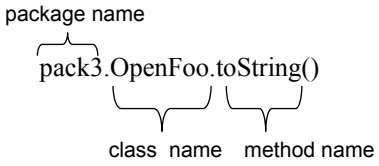
1) From "Big Java" by C. Horstmann pp. 449 – 500.

Packages

- A collection of related classes that are bundled together
- Used to avoid naming conflicts for classes
- Also it 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



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)

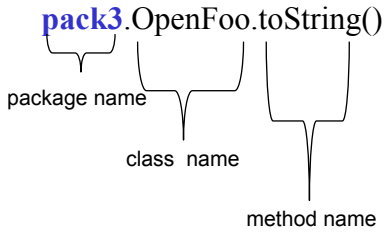
When you do not need an import statement:

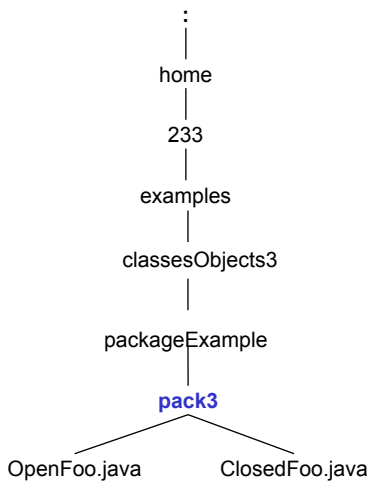
- When you are using the classes in the java.lang package.
- You do not need an import statement in order to use classes which are part of the same package

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 for that program.

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

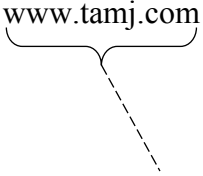
```
package pack3;
public class OpenFoo
{
    :
}

package pack3;
    class ClosedFoo
{
    :
}
```

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

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



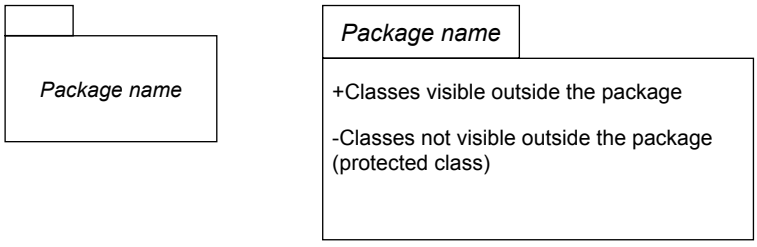
com.tamj .games
.productivity

Sun's Naming Conventions For Packages

Alternatively it could be based on your email address
e.g., tamj@cpsc.ucalgary.ca

ca.ucalgary.cpsc.tamj .games
.productivity

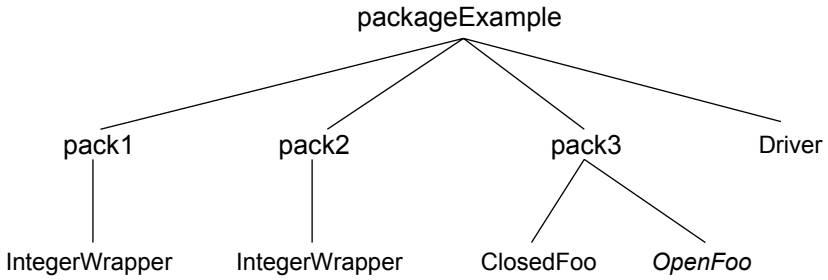
Graphically Representing Packages In UML



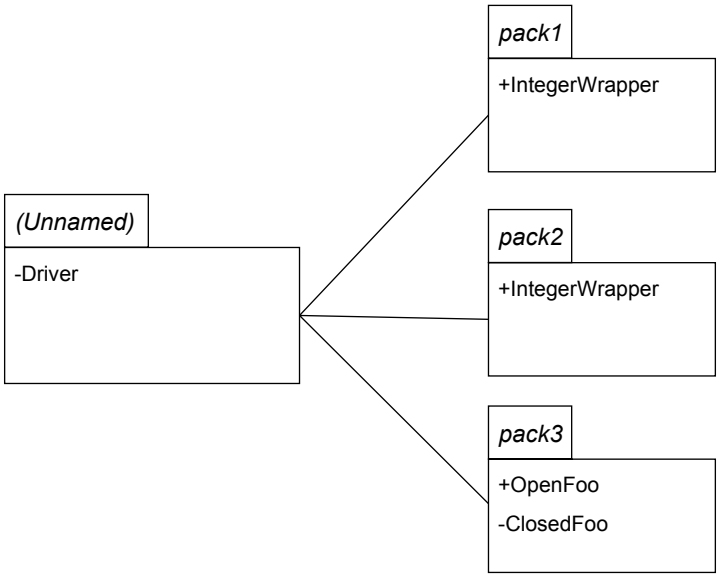
Packages An Example

The complete example can be found in the directory:
/home/233/examples/classesObjects3/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 newValue)
   {
      num = newValue;
   }
   public void setNum (int newValue)
   {
      num = newValue;
   }
```


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 newValue)
   {
      num = newValue;
   }
   public void setNum (int newValue)
   {
      num = newValue;
   }
```


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 newValue) { bool = newValue; }
   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 newValue) { bool = newValue; }

   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 it's originally defined.

Protected "#"

- Can access the attribute/method in the methods of the class where it's originally defined or the subclasses of that class.

Package - no UML symbol for this permission level

- Can access the attribute/method from the methods of the classes within the same package
- *If the level of access is unspecified in a class definition this is the default level of access*

Public "+"

- Can access attribute/method anywhere in the program

James Tam

Updated Levels Of Access Permissions

Access level	Accessible to			
	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

James Tam

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. Be cautious when writing accessor and mutator methods
7. 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.

1. 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

2. 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

3. 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);
```

4. Minimize Modifying Immutable Objects

- Immutable objects
- Once instantiated they cannot change (all or nothing)

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

4. 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

4. 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);
    }
}
```


5. 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.
```

5. Be Cautious In The Use Of References (2)

```
class Foo
{
   private int num;
   public int getNum () { return num; }
   public void setNum (int newValue) { num = newValue; }
}
```


5. 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());

   }
}
```

6. Be Cautious When Writing Accessor And Mutator Methods: First Version

```
class Driver
{
   public static void main (String [] args)
   {
      CreditInfo newAccount = new CreditInfo (10, "James Tam");
      newAccount.setRating(0);
      System.out.println(newAccount);
   }
}
```


6. Be Cautious When Writing Accessor And Mutator Methods: First Version (2)

```
class CreditInfo
{
   private int rating;
   private StringBuffer name;
   public CreditInfo ()
   {
      rating = 5;
      name = new StringBuffer("No name");
   }
   public CreditInfo (int newRating, String newName)
   {
      rating = newRating;
      name = new StringBuffer(newName);
   }
   public int getRating ()
   {
      return rating;
   }
```

6. Be Cautious When Writing Accessor And Mutator Methods: First Version (3)

```
public void setRating (int newRating)
{
   if ((newRating >= 0) && (newRating <= 10))
      rating = newRating;
}

public StringBuffer getName ()
{
   return name;
}

public void setName (String newName)
{
   name = new StringBuffer(newName);
}
```


6. Be Cautious When Writing Accessor And Mutator Methods: First Version (4)

```
 public String toString ()
 {
    String s = new String ();
    s = s + "Name: ";
    if (name != null)
    {
       s = s + name.toString();
    }
    s = s + "\n";
    s = s + "Credit rating: " + rating + "\n";
    return s;
 }
} // End of class CreditInfo
```

6. Be Cautious When Writing Accessor And Mutator Methods: Second Version

- (All mutator methods now have private access).

```
class Driver
{
   public static void main (String [] args)
   {
      CreditInfo newAccount = new CreditInfo (10, "James Tam");

      StringBuffer badGuyName;
      badGuyName = newAccount.getName();

      badGuyName.delete(0, badGuyName.length());
      badGuyName.append("Bad guy on the Internet");

      System.out.println(newAccount);
   }
}
```


6. Be Cautious When Writing Accessor And Mutator Methods: Second Version (2)

```
class CreditInfo
{
   private int rating;
   private StringBuffer name;

   public CreditInfo ()
   {
      rating = 5;
      name = new StringBuffer("No name");
   }

   public CreditInfo (int newRating, String newName)
   {
      rating = newRating;
      name = new StringBuffer(newName);
   }
```

6. Be Cautious When Writing Accessor And Mutator Methods: Second Version (3)

```
public int getRating ()
{
   return rating;
}
private void setRating (int newRating)
{
   if ((newRating >= 0) && (newRating <= 10))
      rating = newRating;
}
public StringBuffer getName ()
{
   return name;
}
private void setName (String newName)
{
   name = new StringBuffer(newName);
}
```


6. Be Cautious When Writing Accessor And Mutator Methods: Second Version (4)

```
public String toString ()
{
   String s = new String ();
   s = s + "Name: ";
   if (name != null)
   {
      s = s + name.toString();
   }
   s = s + "\n";
   s = s + "Credit rating: " + rating + "\n";
   return s;
}
}
```

6. Be Cautious When Writing Accessor And Mutator Methods: Third Version

```
class Driver
{
   public static void main (String [] args)
   {
      CreditInfo newAccount = new CreditInfo (10, "James Tam");
      String badGuyName;
      badGuyName = newAccount.getName();

      badGuyName = badGuyName.replaceAll("James Tam", "Bad guy on
            the Internet");
       System.out.println(badGuyName + "\n");
       System.out.println(newAccount);
   }
}
```


6. Be Cautious When Writing Accessor And Mutator Methods: Third Version (2)

```
class CreditInfo
{
   private int rating;
   private String name;
   public CreditInfo ()
   {
      rating = 5;
      name = "No name";
   }
   public CreditInfo (int newRating, String newName)
   {
      rating = newRating;
      name = newName;
   }
   public int getRating ()
   {
      return rating;
   }
```

6. Be Cautious When Writing Accessor And Mutator Methods: Third Version (3)

```
 private void setRating (int newRating)
{
   if ((newRating >= 0) && (newRating <= 10))
      rating = newRating;
}

public String getName ()
{
   return name;
}

private void setName (String newName)
{
   name = newName;
}
```


6. Be Cautious When Writing Accessor And Mutator Methods: Third Version (4)

```
 public String toString ()
{
   String s = new String ();
   s = s + "Name: ";
   if (name != null)
   {
      s = s + name;
   }
   s = s + "\n";
   s = s + "Credit rating: " + rating + "\n";
   return s;
}
}
```

7. 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

7. 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 perform calculations.
- 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)

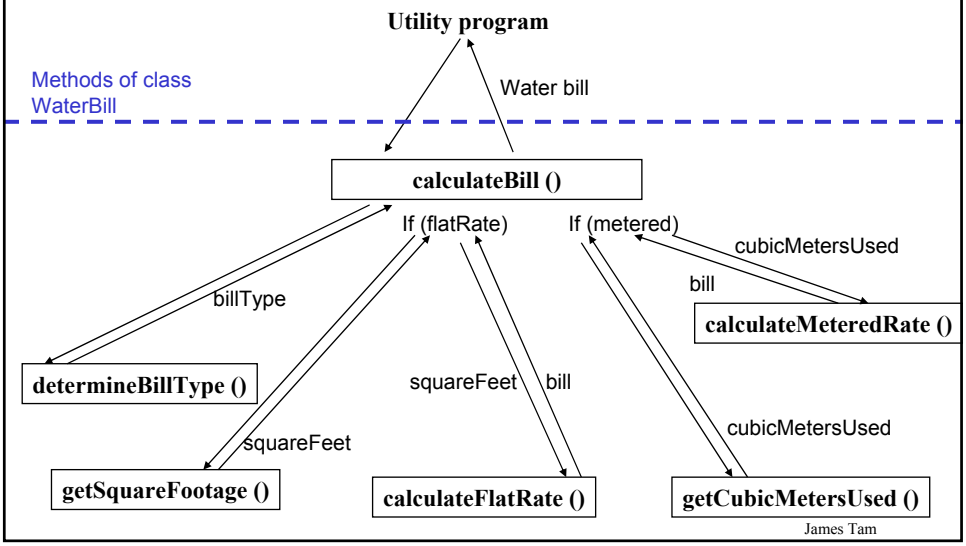
Some candidate classes

- 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;
}
```


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");
```

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 (is it correct).
  - 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
  - How casting works within an inheritance hierarchy
  - What is the effect of the keyword "final" on inheritance relationships
  - Issues related to methods and attributes when employing inheritance
- 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