

# Object-Oriented Principles in Java: Part III

Inheritance

Overloading methods

Shadowing

States

A return to exceptions: creating new exceptions

Interfaces and abstract classes

Packages

## What Is Inheritance?

Creating new classes that are based on existing classes.

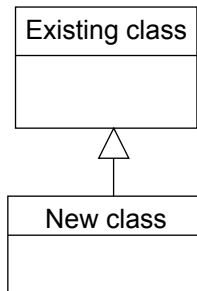
Existing class

## What Is Inheritance?

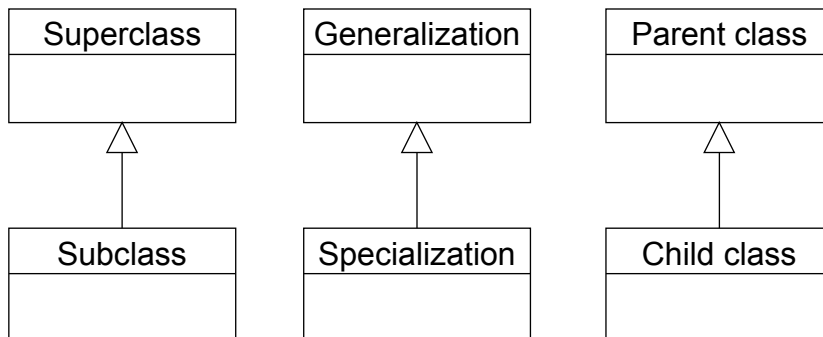
Creating new classes that are based on existing classes.

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

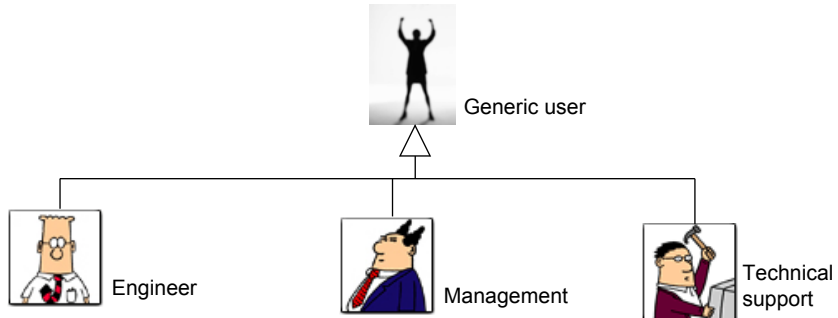


## When To Employ Inheritance

If you notice that certain behaviors or data is common among a group of candidate classes

The commonalities may be defined by a superclass

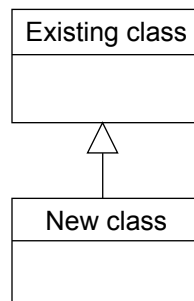
What is unique may be defined by particular subclasses



## Why Employ Inheritance

To allow for code reuse

It may result in more robust code



## Inheritance: Format

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

## Inheritance: An Example

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

## The Parent Of All Classes

Class Object is at the top of the inheritance hierarchy  
(includes Class Array)

All other classes inherit it's data and methods

For more information about this class see the url:

<http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html>

## Review: Relations Between Classes

Composition (“has-a”)

Association (“knows-a”)

Inheritance (“is-a”)

## Composition (“Has-a”)

A composition relation exists between two classes if one classes’ field(s) consist of another class

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

Class Foo

```
{  
    private Bar b;  
}
```



## Association (“Knows-a”)

A composition relation exists 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 Foo

```
{  
    public void method ()  
    {  
        Bar b = new Bar ();  
    }  
}
```



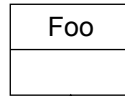
## Inheritance (“Is-a”)

A composition 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 Foo

```
{  
}
```



Class Bar *extends* Bar

```
{  
}
```



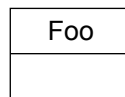
## Inheritance (“Is-a”)

A composition 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

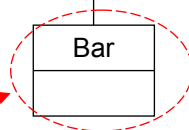
Class Foo

```
{  
}
```



Class Bar *extends* Bar

```
{  
}
```



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

## Method Overloading

- Different versions of a method can be implemented by different classes in an inheritance hierarchy.
- Methods have the same name and parameter list (signature)
- 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 different parameters (type, number or order)
- The method that is actually called is determined at *compile time*.
- i.e., <reference name>. <method name> (parameter list);

Example:

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

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



# Method Overloading Vs. Method Overriding


## Method Overloading

- Multiple method implementations for the same class
- Each method has the same name but different parameters (type, number or order)
- The method that is actually called is determined at *compile time*.
- i.e., <reference name>.<method name> (parameter list);

Example:

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

Distinguishes  
overloaded methods



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

# Method Overloading Vs. Method Overriding

## Method Overriding

- Multiple method implementations between the parent and child classes
- Each method has the same return value, name and parameters (type, number or order)
- The method that is actually called is determined at *run time* (Polymorphism)
- i.e., <reference name>.<method name> (parameter list);

## Method Overloading Vs. Method Overriding

### Method Overriding

- Multiple method implementations between the parent and child classes
- Each method has the same return value, name and parameters (type, number or order)
- The method that is actually called is determined at *run time* (Polymorphism)
- i.e., <reference name>.<method name> (parameter list);

→  
Type of reference  
distinguishes  
overridden methods

## Method Overloading Vs. Method Overriding (2)

Example:

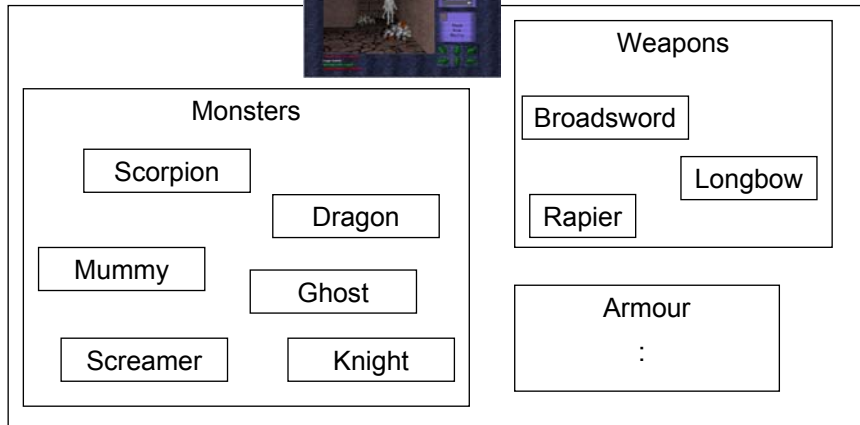
```
Class Foo
{
    display ();
    :
}
Class FooChild
{
    display ();
}
```

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

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

# A Blast From The Past

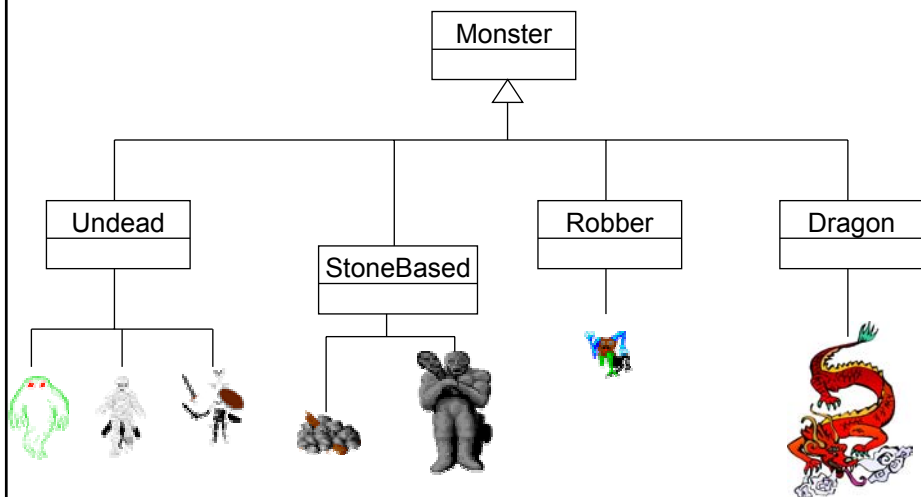
Dungeon  
Master



Object-Oriented Principles in Java: Part III

James Tam

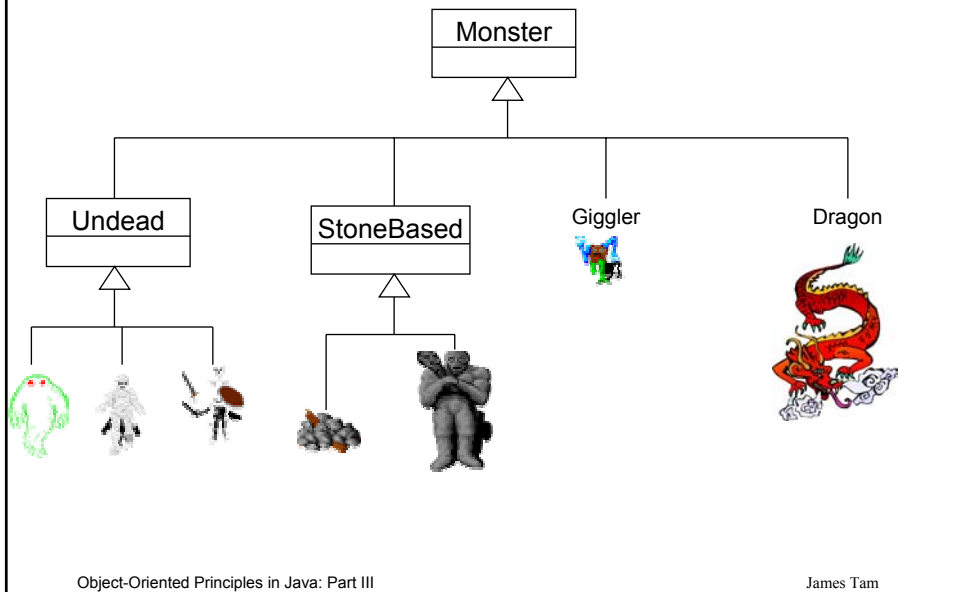
# The Inheritance Hierarchy For The Monsters



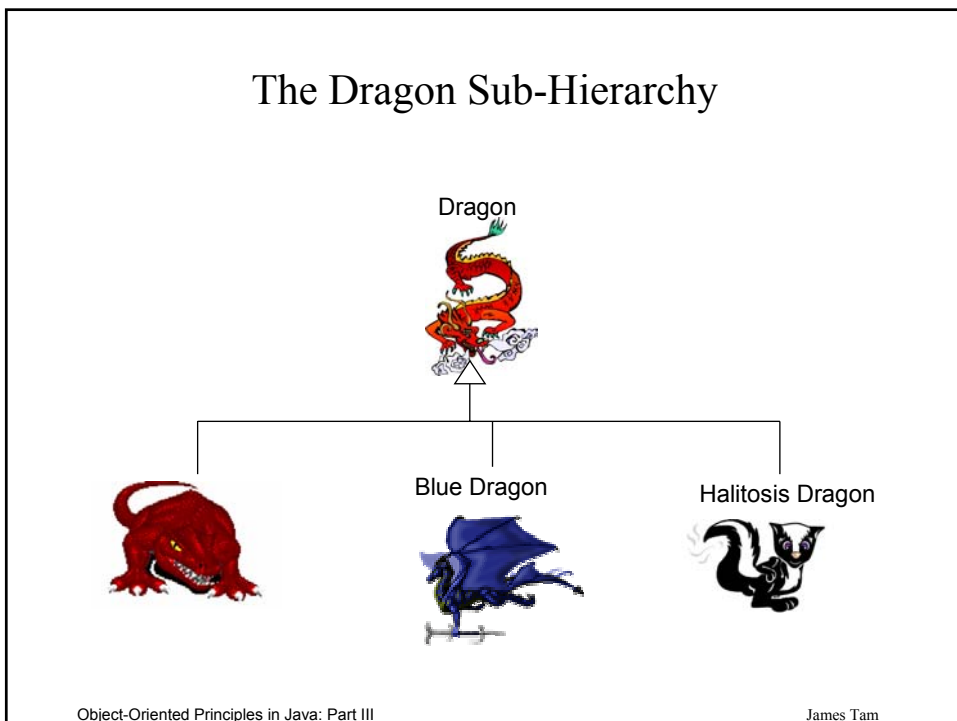
Object-Oriented Principles in Java: Part III

James Tam

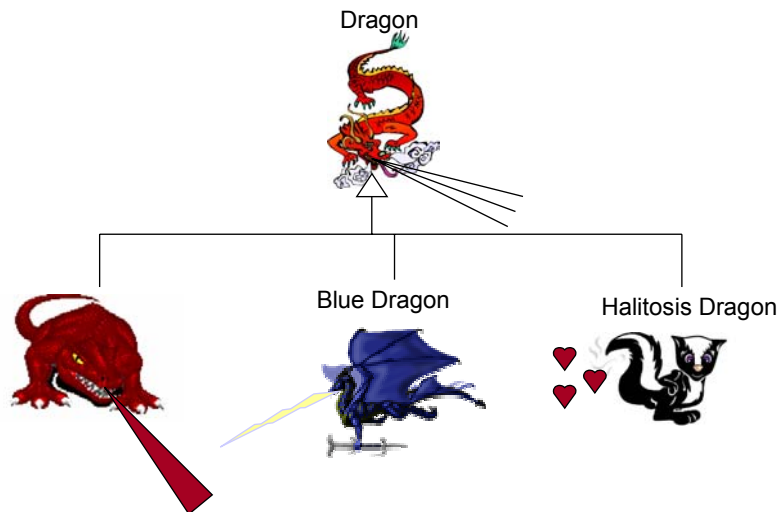
## The Inheritance Hierarchy For The Monsters



## The Dragon Sub-Hierarchy



## The Dragon Sub-Hierarchy



Object-Oriented Principles in Java: Part III

James Tam

## Class Monster

The complete program can be found in the directory:

```
class Monster
{
    protected int protection;
    protected int damageReceivable;
    protected int damageInflictible;
    protected int speed;
    protected String name;
```

Object-Oriented Principles in Java: Part III

James Tam

## Class Monster (2)

```
public String toString ()
{
    String s = new String ();
    s = s + "Protection: " + protection + "\n";
    s = s + "Damage receivable: " + damageReceivable + "\n";
    s = s + "Damage inflictable: " + damageInflictible + "\n";
    s = s + "Speed: " + speed + "\n";
    s = s + "Name: " + name + "\n";
    return s;
}

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

## Class Dragon

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

## Class BlueDragon

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

## Class HalitosisDragon

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

## Class RedDragon

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

## Class DungeonMaster

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

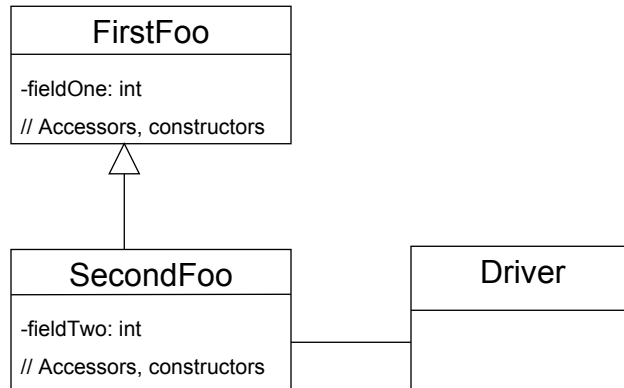
        electro.displaySpecial ();
        pinky.displaySpecial ();
        stinky.displaySpecial ();

    }
}
```



## Inheritance: A Second Example

The source code for this example can be found in the directory:  
/home/profs/tamj/233/examples/inheritance/secondExample



## The Driver Class

```
class Driver
{
    public static void main (String [] argv)
    {
        SecondFoo sf1 = new SecondFoo ();
        System.out.println();

        SecondFoo sf2 = new SecondFoo (20);
        System.out.println();

        SecondFoo sf3 = new SecondFoo (100,200);
        System.out.println();
    }
}
```

## Class SecondFoo

```
class SecondFoo extends FirstFoo
{
    private int fieldTwo;

    public SecondFoo ()
    {
        super();
        System.out.println("Calling default constructor for class SecondFoo");
        fieldTwo = 1;
    }

    public SecondFoo (int f2)
    {
        super();
        System.out.println("Calling one-argument constructor for class SecondFoo");
        fieldTwo = f2;
    }
}
```

## Class SecondFoo (2)

```
public SecondFoo (int f1, int f2)
{
    super(f1);
    System.out.println("Calling two-argument constructor for class SecondFoo");
    fieldTwo = f2;
}

public int getFieldTwo () {return fieldTwo;}
public void setFieldTwo (int i) {fieldTwo = i;}
}
```

## Class FirstFoo

```
class FirstFoo
{
    private int fieldOne;

    public FirstFoo ()
    {
        System.out.println("Calling default constructor for class FirstFoo");
        fieldOne = 0;
    }

    public FirstFoo (int f1)
    {
        System.out.println("Calling one-argument constructor for class FirstFoo");
        fieldOne = 0;
    }

    public int getFieldOne () {return fieldOne;}
    public void setFieldOne (int i) {fieldOne = i;}
}
```

## Levels Of Access Permissions

### Private “-”

- Can only access field in the methods of the class *where the field is originally listed.*

### Protected “#”

- Can access field in the methods of the class where the field is originally listed or the subclasses of that class

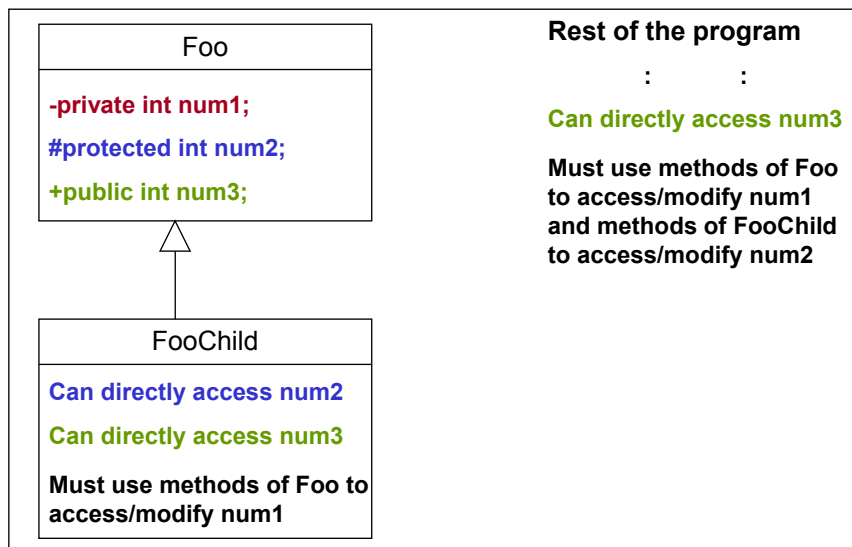
### Public “+”

- Can access field anywhere in the program

## Levels Of Access Permissions (Tabular Form)

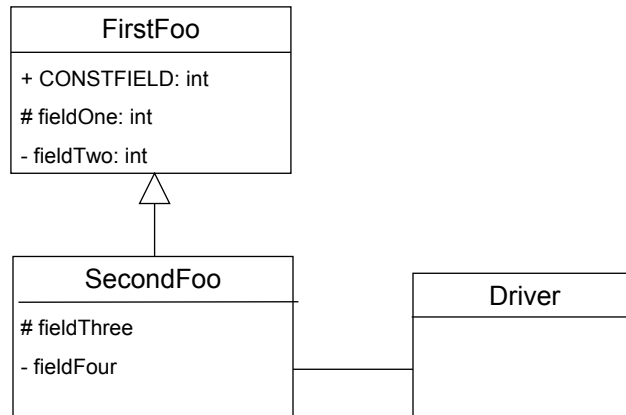
Access level	Accessible to		
	Same class	Subclass	Not a subclass
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

## Levels Of Access Permissions (Graphical Representation)



## Inheritance: A Third Example

The source code for this example can be found in the directory:  
/home/profs/tamj/233/examples/inheritance/thirdExample



Object-Oriented Principles in Java: Part III

James Tam

## The Driver Class

```
class Driver
{
    public static void main (String [] argv)
    {
        SecondFoo f1 = new SecondFoo ();
        System.out.println();
        SecondFoo f2 = new SecondFoo (10,20,30,40);
        System.out.println();

        FirstFoo f3 = (FirstFoo) new SecondFoo ();
        System.out.println(f3.getFieldTwo ());

        //System.out.println(f3.getFieldThree ());

        // SecondFoo f4 = (SecondFoo) new FirstFoo ();
    }
}
```

Object-Oriented Principles in Java: Part III

James Tam

## Class SecondFoo

```
class SecondFoo extends FirstFoo
{
    protected int fieldThree;
    private int fieldFour;
    public SecondFoo ()
    {
        super();
        System.out.println("Calling default constructor for class SecondFoo");
        fieldThree = 3;
        fieldFour = 4;
    }
    public SecondFoo (int f1, int f2, int f3, int f4)
    {
        super (f2);
        System.out.println("Calling four-argument constructor for class SecondFoo");
        fieldOne = f1;
        fieldThree = f3;
        fieldFour = f4;
    }
    :
}
```

## Class FirstFoo

```
class FirstFoo
{
    public static final int CONSTFIELD = 1;
    protected int fieldOne;
    private int fieldTwo;
```

## Class FirstFoo (2)

```
public FirstFoo ()
{
    System.out.println("Calling default constructor for class FirstFoo");
    fieldOne = 0;
    fieldTwo = 0;
}

public FirstFoo (int f2)
{
    System.out.println("Calling one-argument constructor for class FirstFoo");
    fieldOne = 0;
    fieldTwo = f2;
}

public FirstFoo (int f1, int f2)
{
    System.out.println("Calling two-argument constructor for class FirstFoo");
    fieldOne = f1;
    fieldTwo = f2;
}
```

## 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

## Local Variables Shadowing Instance Fields

```
class IntegerWrapper
{
    private int num;

    public IntegerWrapper ()
    {
        num = (int) (Math.random() * 100);
    }

    public IntegerWrapper (int no)
    {
        int num = no;
    }
    :
}
```

## 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;
    }
}
```



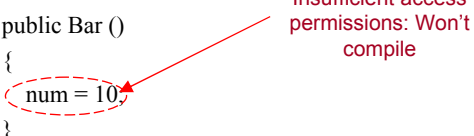
## Fields Of The Subclass Have The Same Names As The SuperClasses' Fields

```
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;
    }
}
```

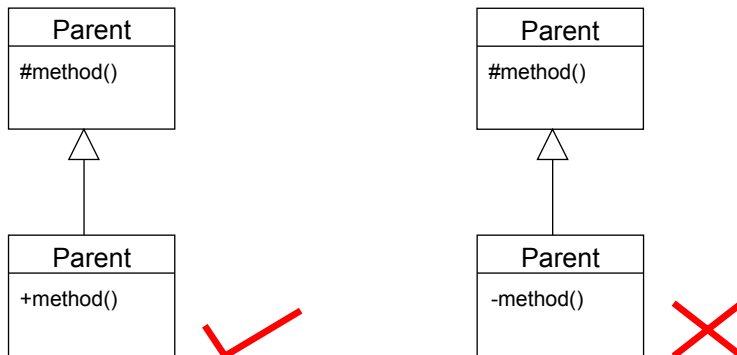
**NO!**

## 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());
    }
}
```

## 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`

## Classes And State

The state of an object is determined by the values of its attributes.

The states of objects can be modeled by State diagrams

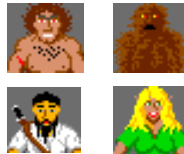
Not all attributes are modeled

- The attribute can only take on a limited range of values
- The attribute has restrictions that determine which values that it may take on.

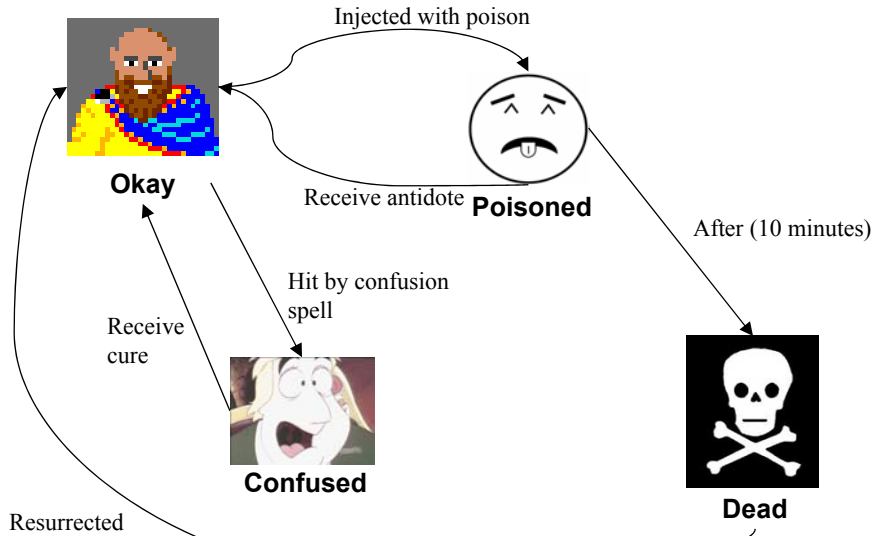
## Example Class: Adventurer

Class Adventurer

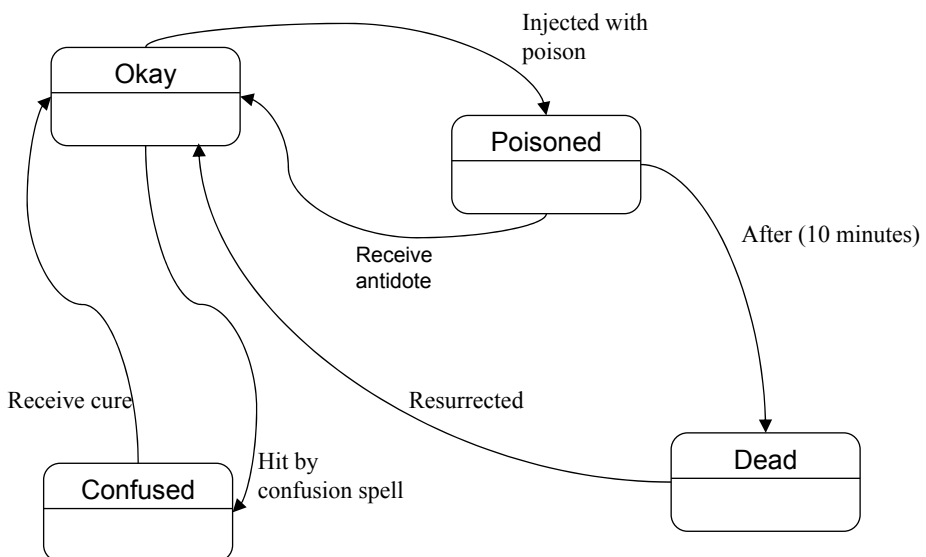
```
{  
    private boolean okay;  
    private boolean poisoned;  
    private boolean confused;  
    private boolean dead;  
    :  
}
```



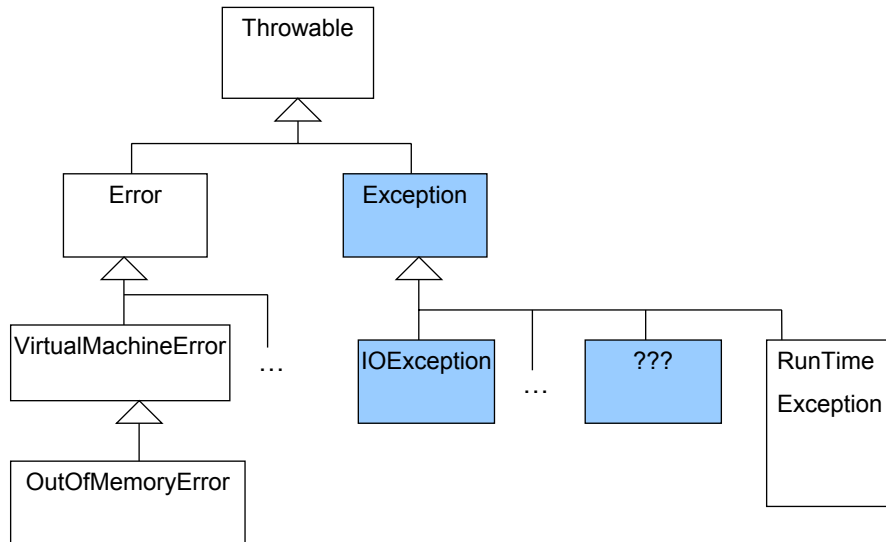
## Class Adventurer: The Set Of States



## Class Adventurer: State Diagram



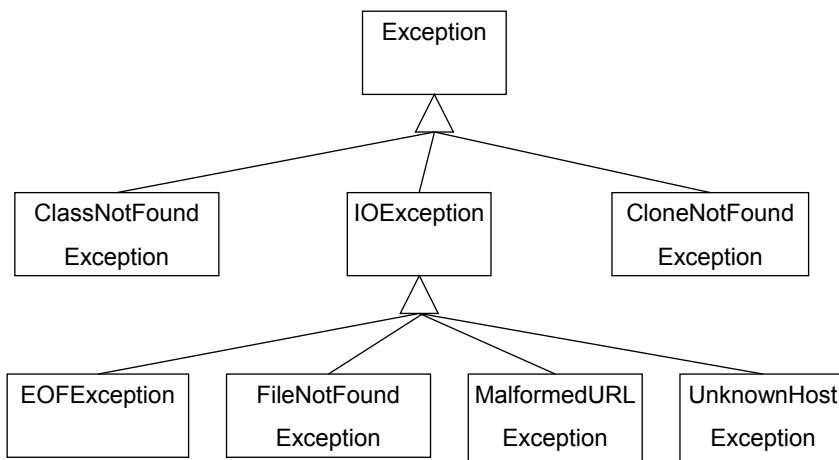
## Creating Your Own Exceptions



Object-Oriented Principles in Java: Part III

James Tam

## Class Exception: Local Inheritance Hierarchy



Object-Oriented Principles in Java: Part III

James Tam

## Creating New Exceptions: An Example

The full example can be found in the directory:  
`/home/profs/tamj/233/examples/exceptions/writingExceptions`

## The Driver Class

```
import tio.*;

class Driver
{
    public static void main (String [] argv)
    {
        int newAge;
        Person jim = new Person ();
        System.out.print("Enter age: ");
        newAge = Console.in.readInt();
    }
}
```

## The Driver Class (2)

```
try
{
    jim.setAge(newAge);
}
catch (Exception e)
{
    System.out.println(e.getMessage());
}
}
```

## Class Person

```
class Person
{
    private int age;
    public static int MINAGE = 0;
    public static int MAXAGE = 120;

    public Person ()
    {
        age = 0;
    }
}
```



## Class Person (2)

```
public Person (int a) throws InvalidAgeException
{
    if ((a < MINAGE) || (a > MAXAGE))
        throw new InvalidAgeException("Invalid Age Exception: Age must be
            between " + MINAGE + " & " + MAXAGE);
    else
        age = a;
}
```

## Class Person (3)

```
public void setAge (int a) throws InvalidAgeException
{
    if ((a < MINAGE) || (a > MAXAGE))
        throw new InvalidAgeException("Invalid Age Exception: Age must be
            between " + MINAGE + " & " + MAXAGE);
    else
        age = a;
}

public int getAge ()
{
    return age;
}
}
```

# Class InvalidAgeException

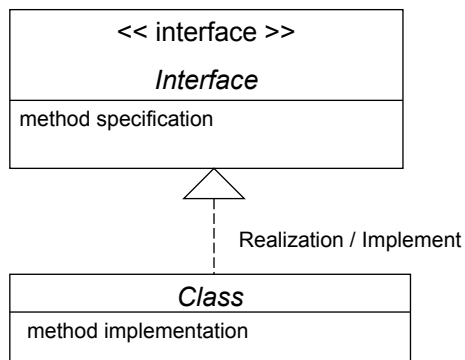
```
class InvalidAgeException extends Exception
{
    InvalidAgeException ()
    {
    }
    InvalidAgeException (String s)
    {
        super(s);
    }
}
```

# Java Interfaces

Similar to a class

Provides a design guide rather than implementation details

Specifies what methods should be implemented but not how

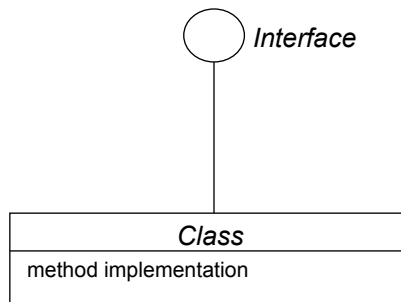


## Java Interfaces: Lollipop Notation

Similar to a class

Provides a design guide rather than implementation details

Specifies what methods should be implemented but not how



## Interfaces: Format

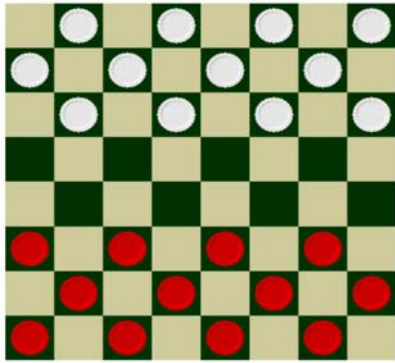
Format for specifying the interface

```
Interface <name of interface>
{
    constants
    methods to be implemented
}
```

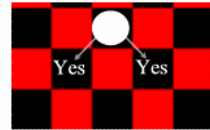
Format for realizing / implementing the interface

```
class <name of class> implements <name of interface>
{
    data fields
    methods actually implemented
}
```

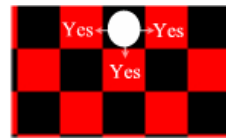
## Interfaces: A Checkers Example



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

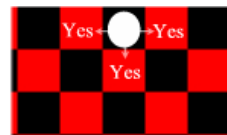
```
class VariantBoard implements Board
{
    public void displayBoard ()
    {
        :
    }

    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

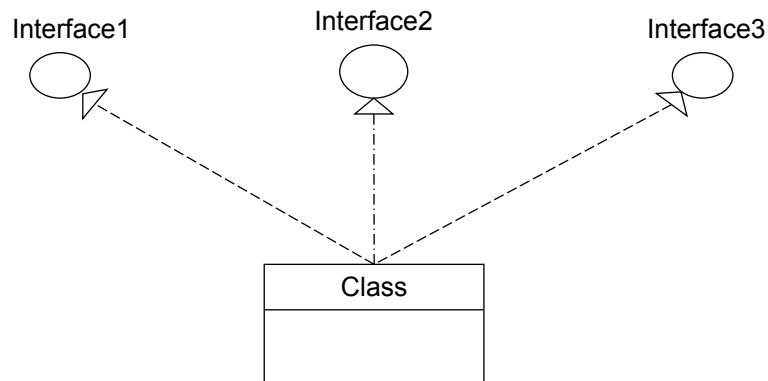
### Interface Board

- No state (data) or behavior (method bodies) listed
- Specifies the behaviors that a board *should* exhibit
- 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>...  
{  
  
}
```

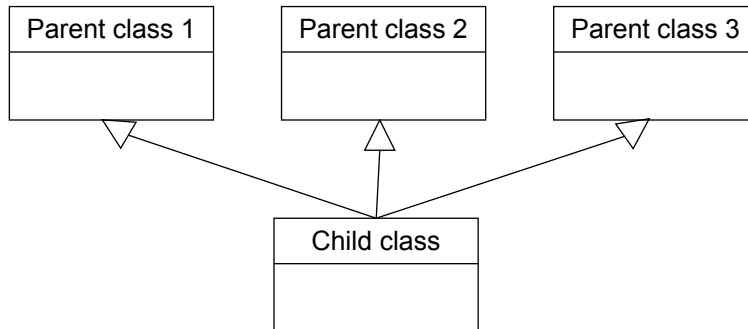


## Multiple Implementations Vs. Multiple Inheritance (2)

A class can implement all the methods multiple interfaces

Classes in Java cannot extend more than one class

This is not possible in Java:



## 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 (must be specified by the child class).

Format:

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

## Abstract Classes (2)

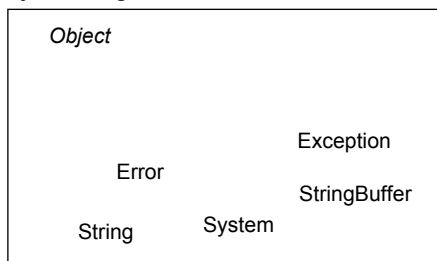
Example:

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

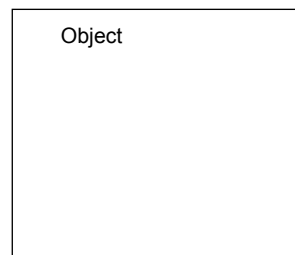
## Packages

A collection of related classes that are bundled together  
To allow for some implementation details to be exposed only  
to other classes in the package  
Used to avoid naming conflicts for classes

java.lang



org.omg.CORBA



## Fully Qualified Names: Matches Directory Structure

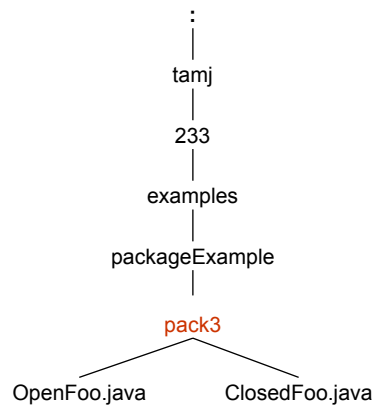
`packageExample.pack1.OpenFoo.toString()`

package name      class name      method name

## Fully Qualified Names: Matches Directory Structure

`pack3.OpenFoo.toString()`

package name      class name      method name



## Matching Classes To Packages

1. The classes that belong to a package must reside in the directory with the same name as the package (previous slide).
2. The definition of the class 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

```
public class OpenFoo  
{  
:  
}  
  
class ClosedFoo  
{  
:  
}
```

## Matching Classes To Packages (2)

### Example

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

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

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

*Package (default): Class can only be instantiated by classes that are members 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

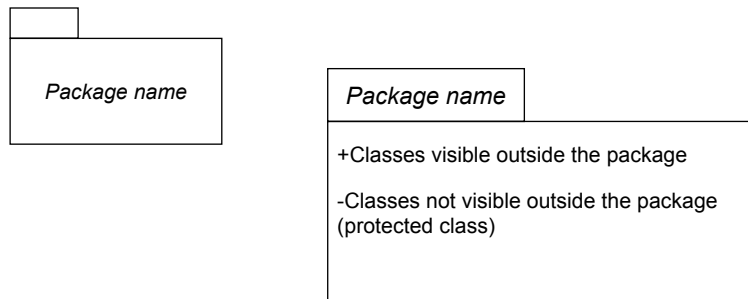
## Sun's Naming Conventions For Packages

Alternatively it could be based on your email address

e.g., tamj@cpsc.ucalgary.ca

ca.ucalgary.cpsc.tamj

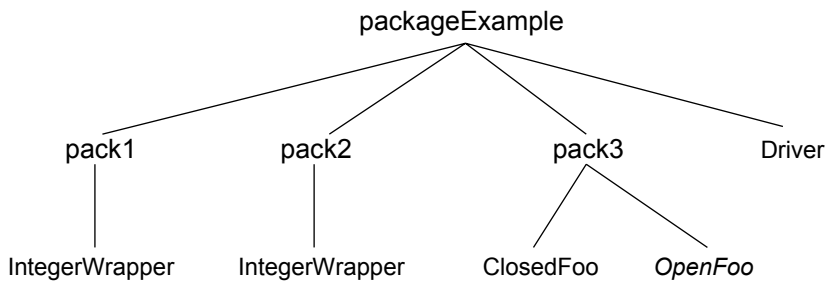
## Graphically Representing Packages In UML



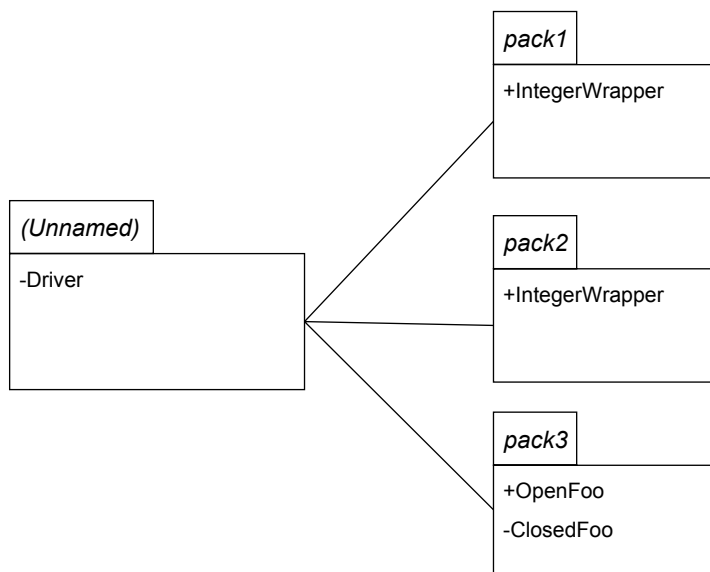
# Packages An Example

The complete example can be found in the directory:  
/home/profs/tamj/examples/packageExample

(But you should have guessed that from the previous slides)



# Graphical Representation Of The Example



## The Driver Class

```
import packageExample.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 Pack1: IntegerWrapper

```
package pack1;

public class IntegerWrapper
{
    private int num;
    public IntegerWrapper ()
    {
        num = (int) (Math.random() * 10);
    }

    // Also includes one argument constructor, accessor and mutator methods
    public String toString ()
    {
        String s = new String ();
        s = s + num;
        return s;
    }
}
```



## Package Pack2: IntegerWrapper

```
package pack2;

public class IntegerWrapper
{
    private int num;
    public IntegerWrapper ()
    {
        num = (int) (Math.random() * 100);
    }

    // Also includes one argument constructor, accessor and mutator methods
    public String toString ()
    {
        String s = new String ();
        s = s + num;
        return s;
    }
}
```

## 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);
    }
    // Also includes accessor and mutator methods along with a toString () method.
}
```

## Package Pack3: Class ClosedFoo

```
package pack3;

class ClosedFoo
{
    private boolean bool;

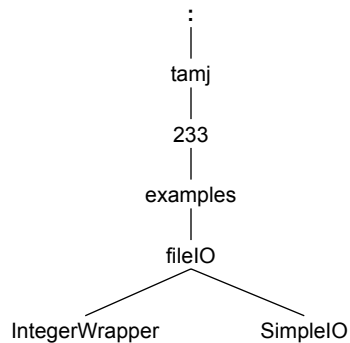
    public ClosedFoo ()
    {
        bool = false;
    }

    // Also includes accessor and mutator methods along with a toString () method.

}
```

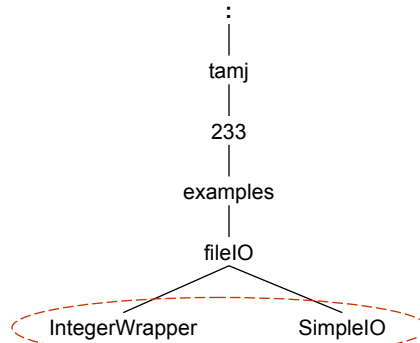
## Classes That Aren't Declared As Part Of A Package

Classes that aren't explicitly associated with a particular package are implicitly part of the same package as the other classes that reside in the same directory as that class.



## Classes That Aren't Declared As Part Of A Package

Classes that aren't explicitly associated with a particular package are implicitly part of the same package as the other classes that reside in the same directory as that class.



## Updated Levels Of Access Permissions

### Private “-”

- Can only access field in the methods of the class *where the field is originally listed.*

### Protected “#”

- Can access field in the methods of the class where the field is originally listed or the subclasses of that class

### Package - no UML symbol for this permission level

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

### Public “+”

- Can access field anywhere in the program

## Updated Levels Of Access Permissions (Tabular Form)

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

## Summary

You should now know:

- Inheritance

- Why use inheritance? When to use inheritance?
- How does inheritance work in Java?
- What other types of relations (besides inheritance) can exist between classes?
- Method overloading vs. method overriding
- Protected levels of access permissions
- Shadowing data fields
- Casting classes
- Calling methods of the parent class vs. calling methods of the child
- The effect of the final modifier in terms of inheritance

## Summary (2)

- State
  - What determines the state of a class?
  - How are states represented with State diagram?
- Creating new exceptions by inheriting existing exception classes
- Interfaces
  - Interfaces vs. Classes
  - How Interfaces are used in the design process?
  - Similarities and differences between abstract classes and interfaces
- Packages
  - What is the purpose of using packages?
  - How are classes associated with particular packages?
  - Updated levels of access permissions (4 levels).