

Code Reuse Through Hierarchies

You will learn about different ways of creating class hierarchies to better organize and group attributes and methods in order to facilitate code reuse

James Tam

Review: Associations Between Classes

- One type of association relationship is a ‘has-a’ relation (also known as “aggregation”).
 - E.g. 1, A car <has-a> engine.
 - E.g. 2, A lecture <has-a> student.
- Typically this type of relationship exists between classes when a class is an attribute of another class.

```
public class Car
{
    private Engine anEngine;
    private Lights carLights;
    public start ()
    {
        anEngine.ignite ();
        carLight.turnOn ();
    }
}
```

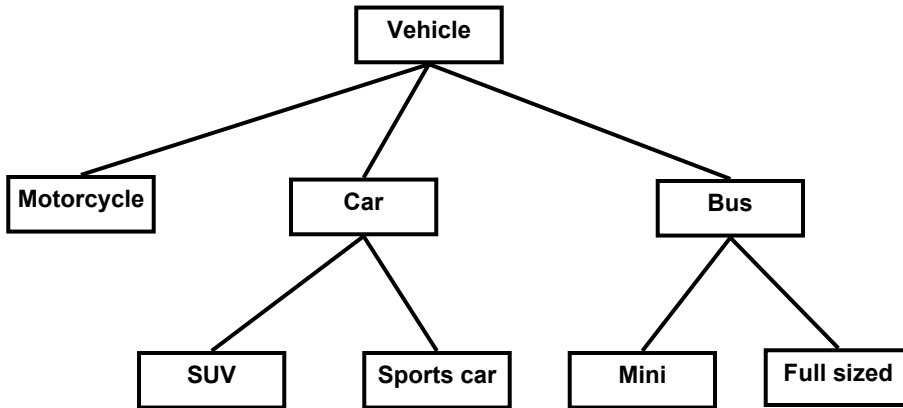
```
public class Engine
{
    public boolean ignite () { .. }
}
```

```
public class Lights
{
    private boolean isOn;
    public void turnOn () { isOn = true;}
}
```

James Tam

A New Type Of Association: Is-A (Inheritance)

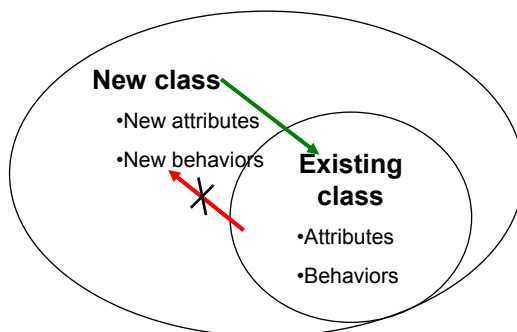
- An inheritance relation exists between two classes if one class is one type of another class



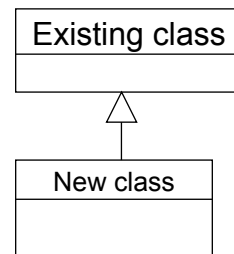
James Tam

What Is Inheritance?

- Taking the attributes/methods of an existing class.
- Extend the existing class with a new class
 - All non-private data and methods of the existing class are available to the new class (but the reverse is not true).



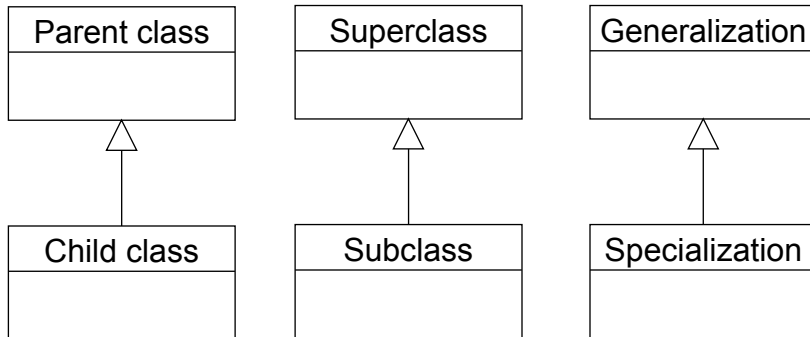
Inheritance: Represented as sets



Inheritance: UML representation

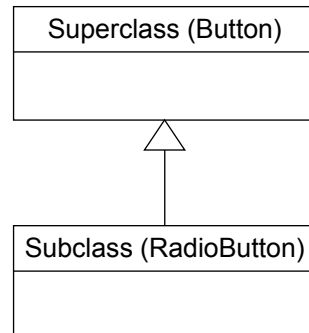
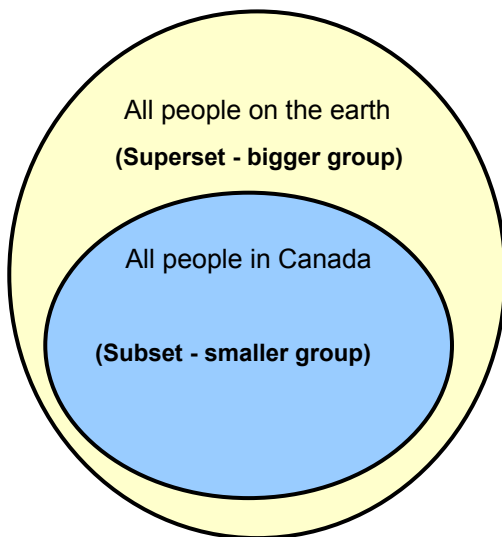
James Tam

Inheritance Terminology



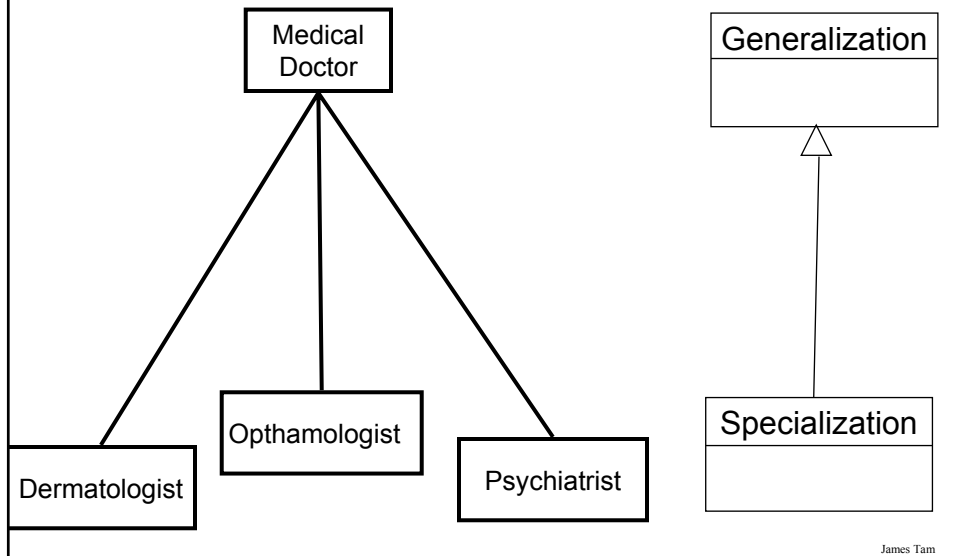
James Tam

Explanation For Some Of The Terms: Set Theory



James Tam

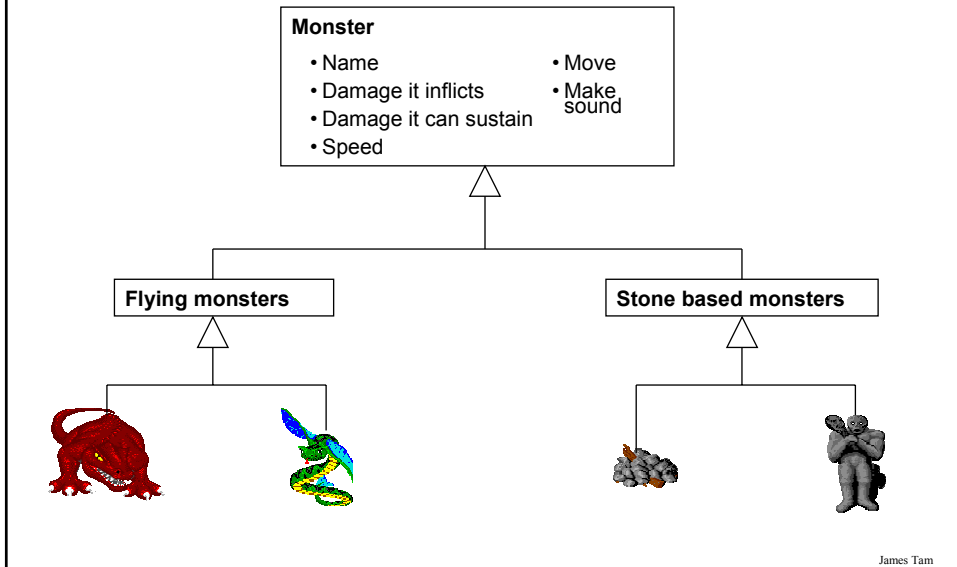
Explanation For Some Of The Terms: Providing An Example



When To Employ Inheritance

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

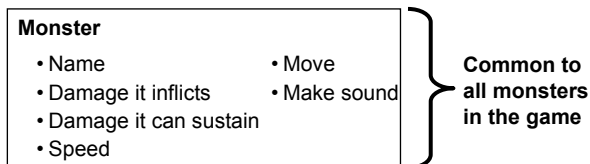
When To Employ Inheritance



Where Should Attributes And Behaviors Be Defined?

•Rule of thumb: Put them high enough in the inheritance hierarchy so that all the appropriate sub-classes have access to the attribute or behavior.

•Example:



Where Should Attributes And Behaviors Be Defined? (2)

- Rule of thumb: Don't put attributes and behaviors higher than they need to be in the inheritance hierarchy otherwise some classes will track information or perform actions that don't make sense.

Monster

- Name
- Damage it inflicts
- Damage it can sustain
- Speed
- Move
- Make sound
- **Fly**



James Tam

Using Inheritance

Format:

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

Example:

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

James Tam

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
 - e.g., "toString" are available to it's child classes
- For more information about this class see the url:
<http://java.sun.com/j2se/1.5/docs/api/java/lang/Object.html>

James Tam

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.

James Tam

Summary: 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

James Tam

Levels Of Access Permission: An Example

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

public class C extends P
{
    // Can't access num1 here
}

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

James Tam

General Rules 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.

James Tam

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


```
public class Parent                public class Child extends Parent
{
    :                               {
    :                               :
    public void method ()          public void method ()
    {                               {
        System.out.println("m1");   num = 1;
    }                               }
}                                   }
```

James Tam

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);


Distinguishes
overloaded methods

James Tam

Method Overloading Vs. Method Overriding (2)

•Example of method overloading:

```
public 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');
```

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

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

James Tam

Method Overloading Vs. Method Overriding (4)

•Example of method overriding:

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

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

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

James Tam

Polymorph

- The ability to take on different forms



Images from the game Dungeon Master by FTL

James Tam

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.

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

James Tam

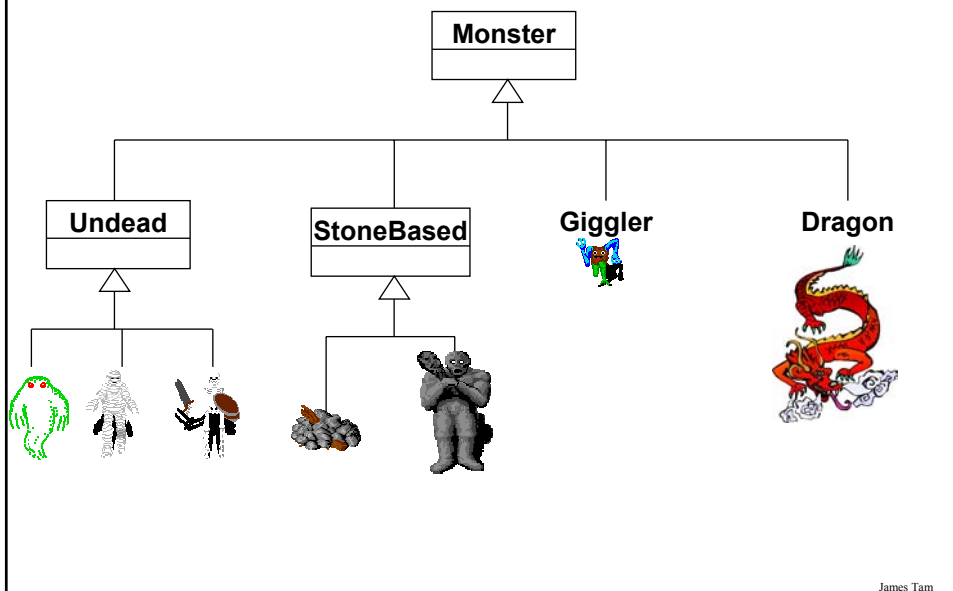
A Blast From The Past

Dungeon Master



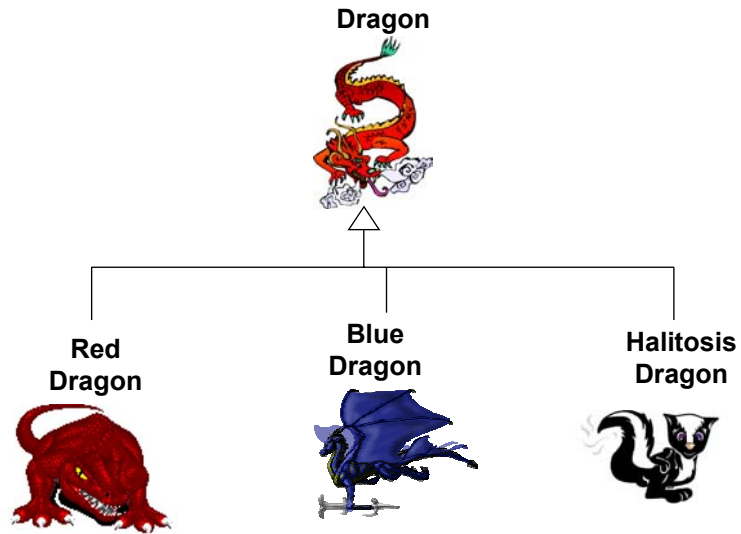
James Tam

The Inheritance Hierarchy For The Monsters



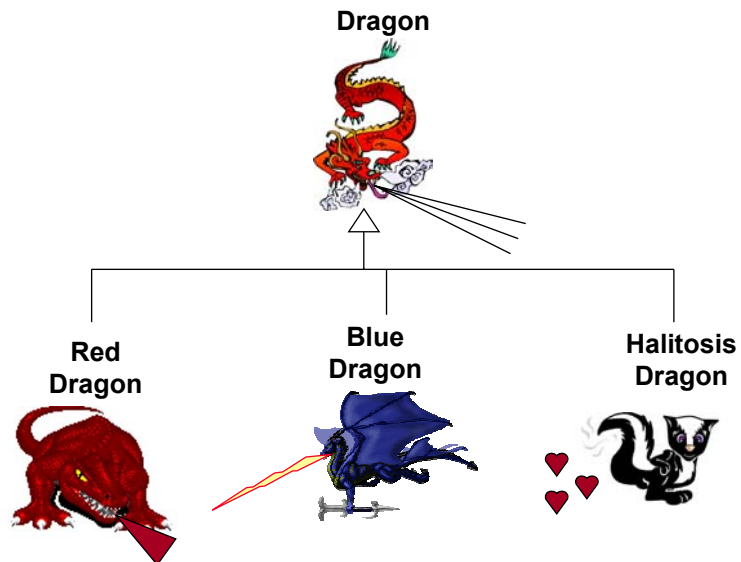
James Tam

The Dragon Sub-Hierarchy



James Tam

The Dragon Sub-Hierarchy



James Tam

Class DungeonMaster

- Example (The complete example can be found in the directory /home/courses/219/examples/hiearchies/DMExample)

```
public 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 ();
    }
}
```

James Tam

Class Monster

```
public class Monster
{
    private int protection;
    private int damageReceivable;
    private int damageInflictible;
    private int speed;
    private String name;
    public Monster ()
    {
        protection = 0;
        damageReceivable = 1;
        damageInflictible = 1;
        speed = 1;
        name = "Monster name: ";
    }
}
```

James Tam

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 getDamageInflictible () {return damageInflictible;}
public void setDamageInflictible (int newValue) {damageInflictible =
    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");
}
```

James Tam

Class Monster (3)

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

James Tam

Class Dragon

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

James Tam

Class BlueDragon

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

James Tam

Class HalitosisDragon

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

James Tam

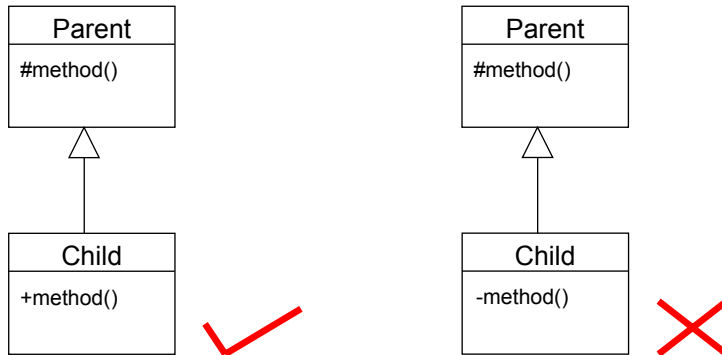
Class RedDragon

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

James Tam

Changing Permissions Of Overridden Methods

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



James Tam

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

James Tam

Updated Scoping Rules (2)

```
public class P
{
    <<< Third: Parent's attribute >>>
}
public class C extends P
{
    <<< Second: Attribute>>>
    public void method ()
    {
        <<< First: Local >>>
    }
}
```

James Tam

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

```
public class P
{
    protected int num;
}

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

James Tam

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.

James Tam

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

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

James Tam

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

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

James Tam

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.

James Tam

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

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

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

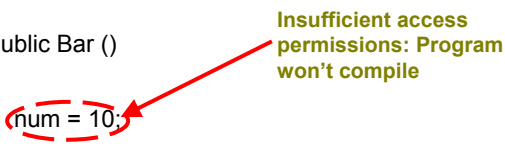
James Tam

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

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

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

Insufficient access permissions: Program won't compile



James Tam

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

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

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

James Tam

The Result Of Attribute Shadowing

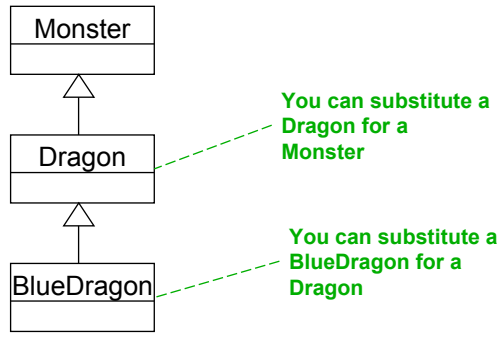
```
public class Bar extends Foo
{
    private int num;
    public Bar ()
    {
        num = 10;
    }
    public int getSecondNum () { return num; }
}

public class Driver
{
    public static void main (String [] arv)
    {
        Bar b = new Bar ();
        System.out.println(b.getNum());
        System.out.println(b.getSecondNum());
    }
}
```

James Tam

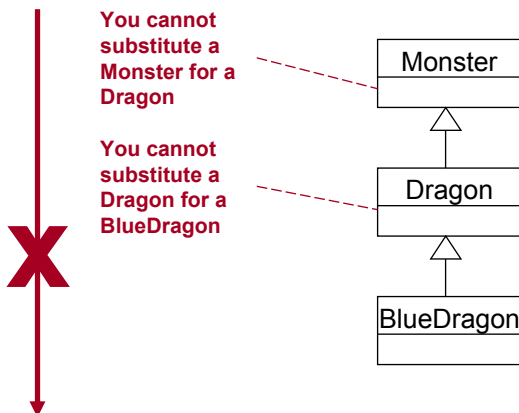
Casting And Inheritance

- Because the child class IS-A parent class you can substitute instances of a subclass for instances of a superclass.



Casting And Inheritance (2)

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



Casting And Inheritance: A Previous Example

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

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

James Tam

Casting And Inheritance: A Previous Example

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

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

James Tam

Casting And Inheritance: A Previous Example

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

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

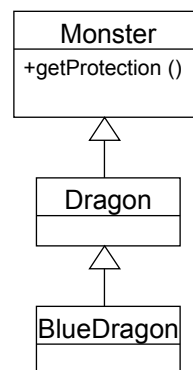
James Tam

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());
```



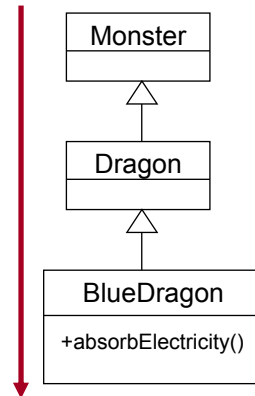
James Tam

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 ();
```



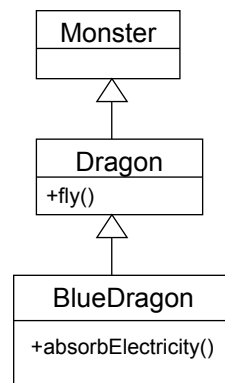
James Tam

Casting And Inheritance

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

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

```
aMonster = new Monster ();  
X electro = aMonster;  
  
X electro = (BlueDragon) aMonster;  
X electro.fly();  
X electro.absorbElectricity();
```



James Tam

Casting And Inheritance (2)

- When casting between classes 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();
}
```

James Tam

Casting And Inheritance (3)

- When casting between classes 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();
}
```

James Tam

Another Scoping Example

The complete example can be found in UNIX under:
/home/courses/219/examples/scope:

```
public class ScopingExample
{
    public static void main (String [] args)
    {
        P p1 = new P ();
        C c1 = new C ();
        GC gc1 = new GC ();
        gc1.method1();
        gc1.method2();
        gc1.method3();
        gc1.method();
    }
}
```

James Tam

Another Scoping Example (2)

```
public 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();
    }
}
```

James Tam

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
```

James Tam

Another Scoping Example (4)

```
public 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
```

James Tam

Another Scoping Example (5)

```
public 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
```

James Tam

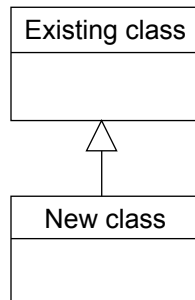
The Final Modifier (Inheritance)

- 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** public class ParentFoo

James Tam

Why Employ Inheritance

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



James Tam

Java Interfaces (Type)

- Similar to a class
- Provides a design guide rather than implementation details
- Specifies what methods should be implemented but not how
 - An important design tool and agreement for the interfaces should occur very early before program code has been written.
 - (Specify the signature of methods so each part of the project can proceed with minimal coupling between classes).
- It's a design tool so they cannot be instantiated

James Tam

Interfaces: Format

Format for defining an interface

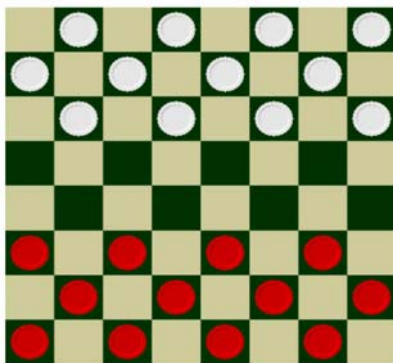
```
public interface <name of interface>
{
    constants
    methods to be implemented by the class that realizes this interface
}
```

Format for realizing / implementing the interface

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

James Tam

Interfaces: A Checkers Example



Basic board



Regular rules



Variant rules

James Tam

Interface Board

```
public 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);
    :
    :
    :
}
```

James Tam

Class RegularBoard

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

    public void initializeBoard ()
    {
        :
    }
}
```

James Tam

Class RegularBoard (2)

```
public void movePiece ()
{
    // Get (x, y) coordinates for the source and destination
    if (moveValid (xS, yS, xD, yD) == 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;
}
} // End of class RegularBoard
```



James Tam

Class VariantBoard

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

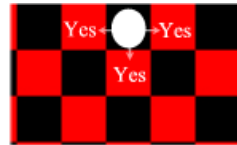
    public void initializeBoard ()
    {
        :
    }
}
```

James Tam

Class VariantBoard (2)

```
public void movePiece ()
{
    // Get (x, y) coordinates for the source and destination
    if (moveValid (xS, yS, xD, yD) == 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;
}
} // End of class VariantBoard
```



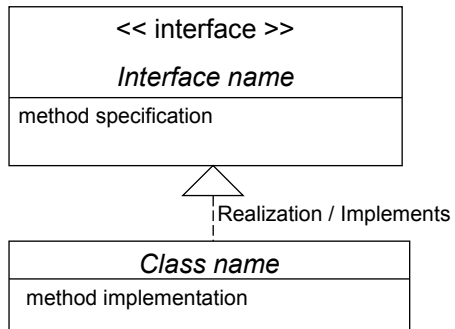
James Tam

Interfaces: Recapping The Example

- Interface Board
 - No state (variable 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)

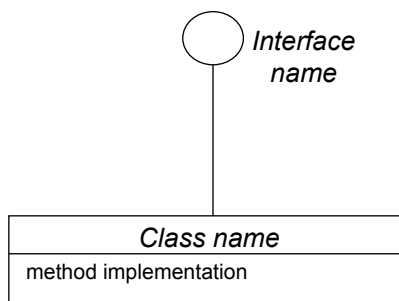
James Tam

Specifying Interfaces In UML



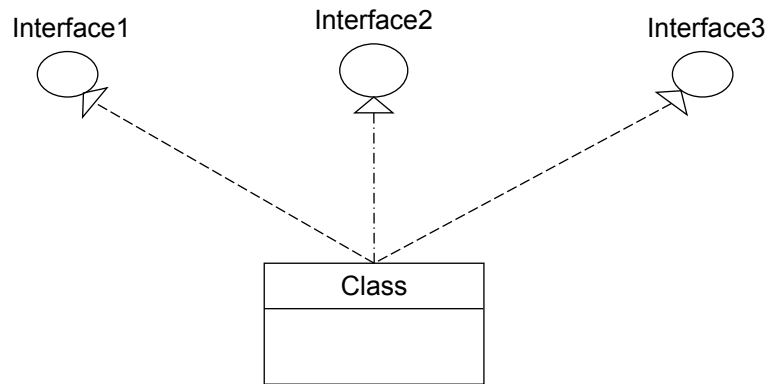
James Tam

Alternate UML Representation (Lollipop Notation)



James Tam

Implementing Multiple Interfaces



James Tam

Implementing Multiple Interfaces

Format:

```
public class <class name> implements <interface name 1>,
    <interface name 2>, <interface name 3>...
{
}

```

James Tam

Multiple Implementations Vs. Multiple Inheritance

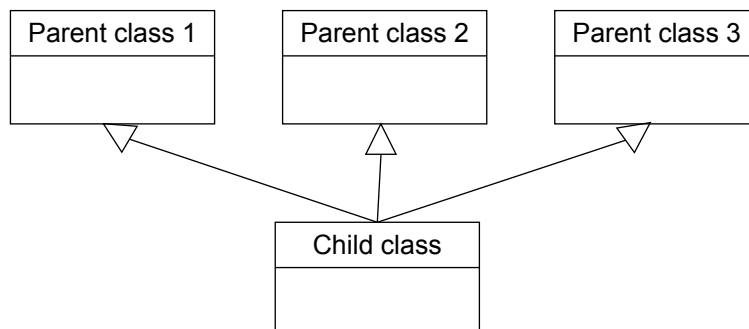
- A class can implement multiple interfaces
- Classes in Java cannot extend more than one class
- This is not possible in Java but is possible in other languages such as C++:

```
class <class name 1> extends <class  
name 2>, <class name 3>...  
{  
  
}
```

James Tam

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 other languages such as C++:



James Tam

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:**

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

James Tam

Abstract Classes (2)

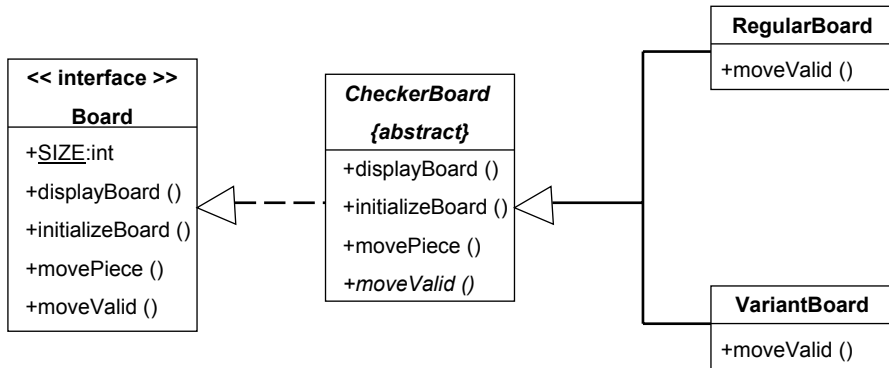
- Example¹:**

```
public 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.

James Tam

Another Example For An Abstract Class



James Tam

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

James Tam

You Should Now Know (2)

- What are interfaces/types
 - How do types differ from classes
 - How to implement and use interfaces in Java
- What are abstract classes in Java and how do they differ from non-abstract classes and interfaces.
- How to read/write UML notations for inheritance and interfaces.