

## 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 ();
    }
}
```

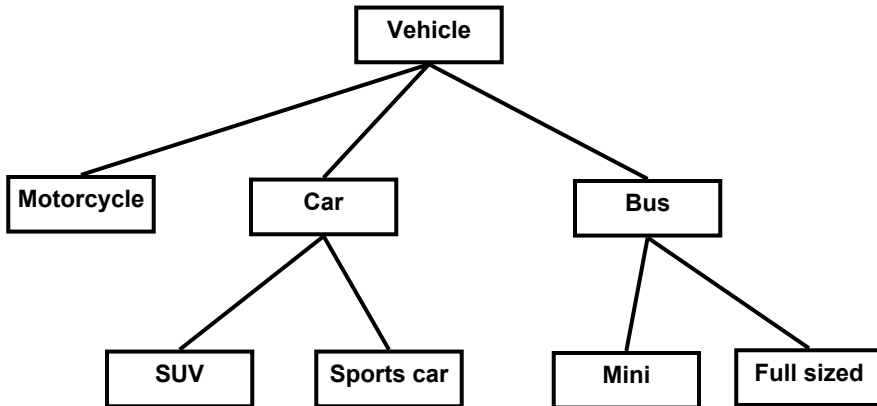
```
public class Engine
{
    public boolean ignite () { .. }
}
```

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

James Tam

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

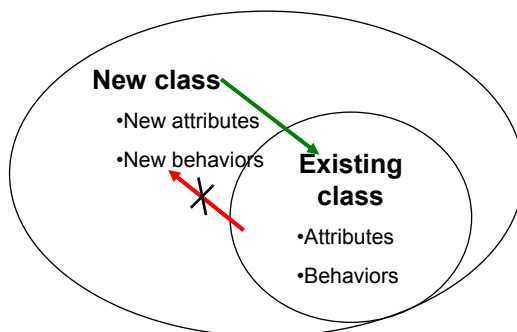
- An inheritance relation exists between two classes if one class is a variant type of another class



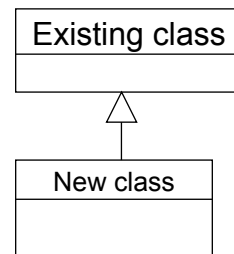
James Tam

## What Is Inheritance?

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



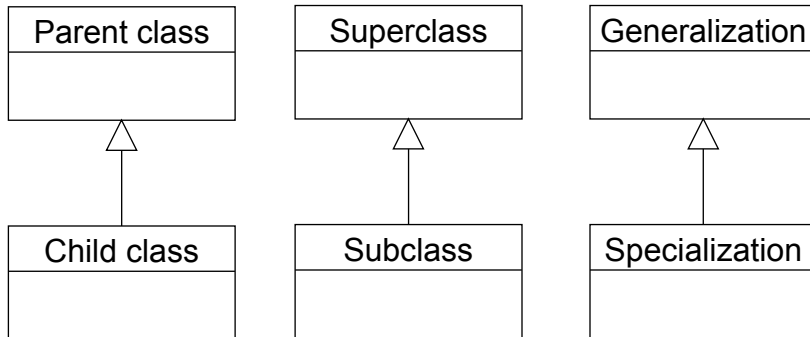
**Inheritance: Represented as sets**



**Inheritance: UML representation**

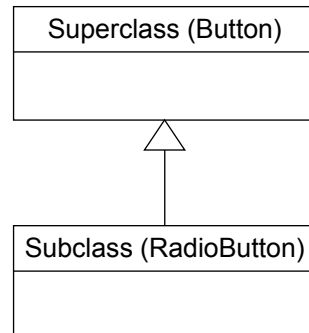
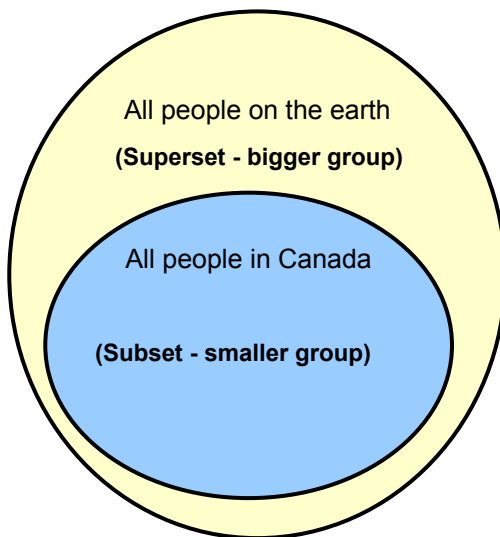
James Tam

## Inheritance Terminology



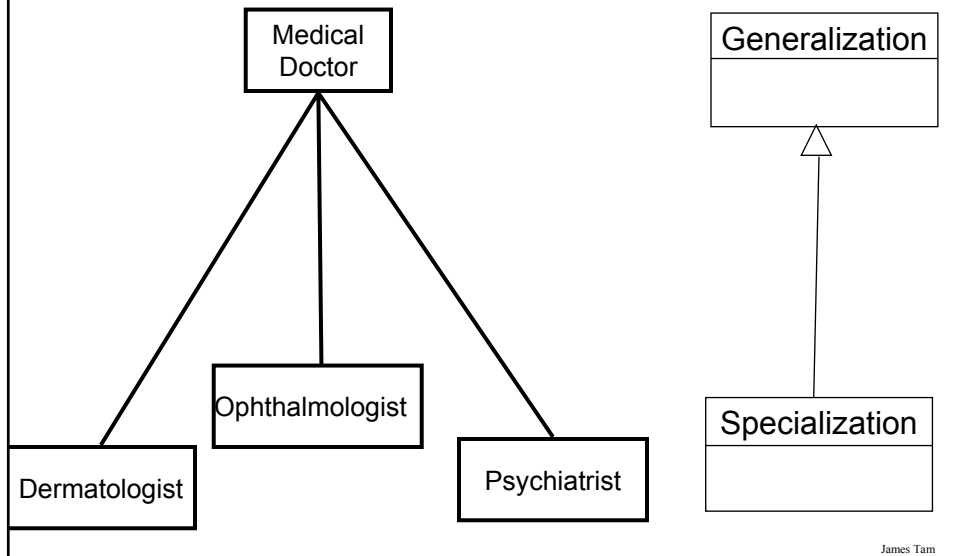
James Tam

## Explanation For Some Of The Terms: Set Theory



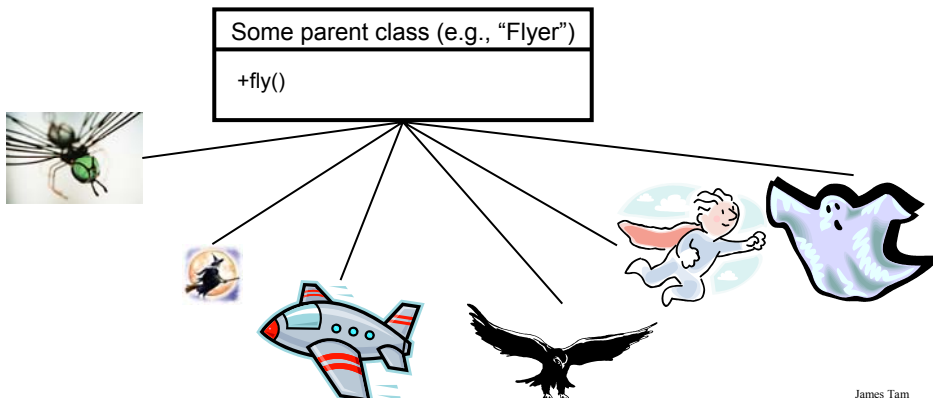
James Tam

## Explanation For Some Of The Terms: Providing An Example

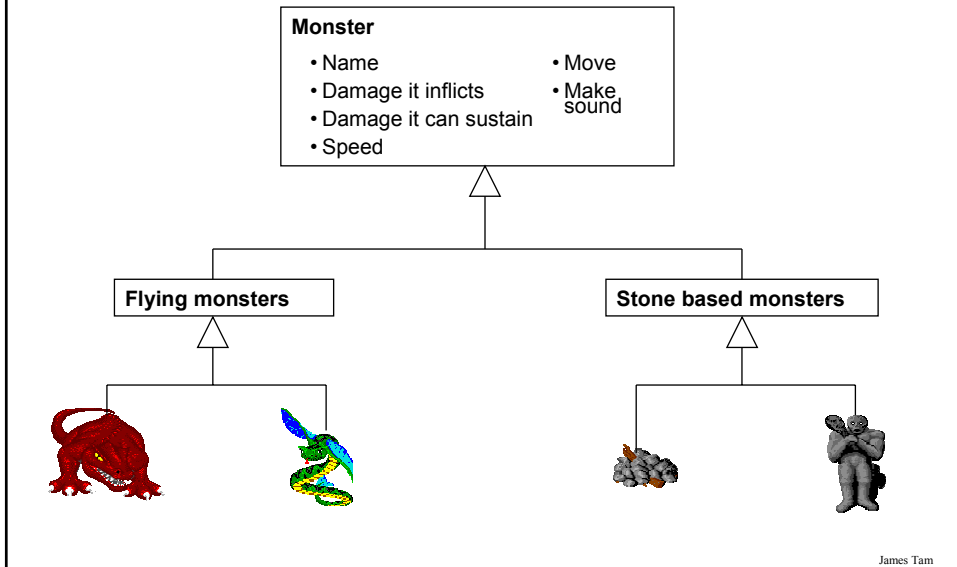


## When To Employ Inheritance

- 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.



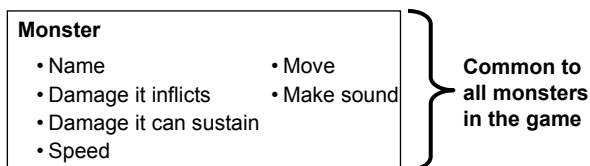
## When To Employ Inheritance: More Detailed Example



## 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 are inappropriate.

### **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 Scorpion extends PoisonousMonster
{
    public void sting()
    {
        System.out.println("I sting you");
    }
}
```

James Tam

## An Example Hierarchy

**Car:**

- 'drive'



**Spy car:**

- 'drive'
- 'fly'



**Ultra spy car:**

- 'drive'
- 'fly'
- 'warp!'



James Bond © MGM/UA Home Entertainment Inc

© Tamco Enterprises

## Example Hierarchy: Example Program

• Location of the example:

- [www.cpsc.ucalgary.ca/~tamj/233/examples/hiearchies/carExample](http://www.cpsc.ucalgary.ca/~tamj/233/examples/hiearchies/carExample)
- [/home/233/examples/hiearchies/carExample](http://home/233/examples/hiearchies/carExample)

## Class Car

```
public class Car
{
    public void drive ()
    {
        System.out.println("Car is traveling on the road");
    }
}
```

James Tam

## Class SpyCar

```
public class SpyCar extends Car
{
    public void fly ()
    {
        System.out.println("Car is flying through the air!");
    }
}
```

James Tam



## Class UltraSpyCar

```
public class UltraSpyCar extends SpyCar
{
    public void warp ()
    {
        System.out.println("Car is warping throughout the galaxy...engage!");
    }
}
```

James Tam

## The Driver Class

```
public class Driver
{
    public static void main (String [] args)
    {
        Car regularCar = new Car();
        SpyCar bondCar = new SpyCar();
        UltraSpyCar tamCar = new UltraSpyCar();

        regularCar.drive();

        bondCar.drive();
        bondCar.fly();

        tamCar.drive();
        tamCar.fly();
        tamCar.warp();
    }
}
```

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:

```
class Foo          class Foo extends Object
{
}
}
```

-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
    // Can access num2, num3
}

public class Driver
{
    // Can't access num1 here and generally can't access num2 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

## Polymorph (Warning: Geek Terminology)



James Tam

## **Definition: Polymorph**

- Poly = many
- Morph = change/changing

James Tam

## **Definition: Method Overriding/Polymorphism**

### Simple definition

- A method that has different forms.

### Detailed definition

- A method that has the same signature (name, parameter list) but a different implementations in the parent vs. child class.

James Tam

## Method Overriding/Polymorphism

- 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
- The type of an instance (the “this” implicit parameter) determines at program run-time which method will be executed.

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

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

Parent aParent = new Parent();
aParent.method();
```

**Type of the reference:**

- Determines which version is called at runtime (dynamic/late binding)

James Tam

## Definition: Dynamic Binding

- The run time binding of a reference to a polymorphic method.

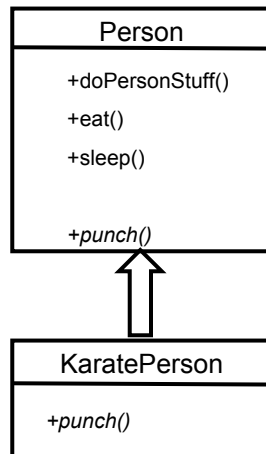
James Tam

## Polymorphism: An Example

- Location of the example:

- [www.cpsc.ucalgary.ca/~tamj/233/examples/hiearchies/polymorphism](http://www.cpsc.ucalgary.ca/~tamj/233/examples/hiearchies/polymorphism)

- [/home/233/examples/hiearchies/polymorphism](http://home/233/examples/hiearchies/polymorphism)



James Tam

## Driver Class

```
public class Driver
{
    public static void main (String [] args)
    {
        Person daniel = new Person ();
        KaratePerson miyagi = new KaratePerson ();
        System.out.println("POLYMORPHIC METHOD: DIFFERENT");
        System.out.println("Daniel punch's: " + daniel.punch()
            + " damage ");
        System.out.println("Mr. Miyagi's punch's: " + miyagi.punch()
            + " damage ");
        System.out.println("");
        System.out.println("SHARED METHODS:");
        daniel.doPersonStuff();
        miyagi.doPersonStuff();
    }
}
```

James Tam

## Class Person

```
public class Person
{
    public void eat () { System.out.println("Munch! Munch! Munch!"); }

    public void sleep () { System.out.println("ZZZ"); }

    public void doPersonStuff () {
        eat();
        sleep();
    }

    public int punch () {
        int power = 1;
        System.out.println("Wimpy punch!");
        return power;
    }
}
```

James Tam

## Class KaratePerson

```
public class KaratePerson extends Person
{
    public int punch ()
    {
        int power = 10;
        System.out.println("Kiai!");
        return power;
    }
}
```

James Tam



## Method Overloading Vs. Method Overriding

- Method Overloading (what you should know)
  - 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

## **Question: Can Miyagi Throw A Regular (“Weak”) Punch?**

- Answer:

- Yes but with some difficulty!
- Unless specific effort is made (using “super”) then the overridden method in the child class is the one that is invoked.

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 the “super” keyword 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

## Example: Accessing Overridden Methods

- Location of the complete example:
  - [www.cpsc.ucalgary.ca/~tamj/233/examples/hiearchies/polymorphismV2](http://www.cpsc.ucalgary.ca/~tamj/233/examples/hiearchies/polymorphismV2)
  - [/home/233/examples/hiearchies/polymorphismV2](http://home/233/examples/hiearchies/polymorphismV2)

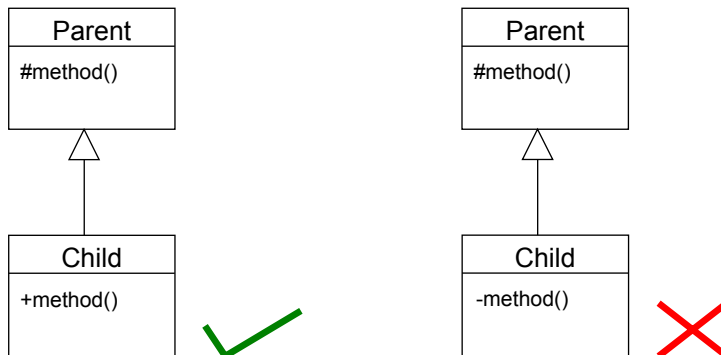
```
public class KaratePerson extends Person
{
    public int punch ()
    {
        int power = super.punch();
        return power;
    }
}
```

Calls the parent classes' method (not so 'super powerful' but it is the 'super class' method)

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

## 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 FooChild extends Foo
{
    public FooChild ()
    {
        num = 10;
    }
}
```

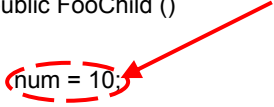
James Tam

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

James Tam

## The Result Of Attribute Shadowing

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

public class Driver
{
    public static void main (String [] argv)
    {
        FooChild fc = new FooChild ();
        System.out.println(fc.getNum());
        System.out.println(fc.getSecondNum());
    }
}
```

James Tam



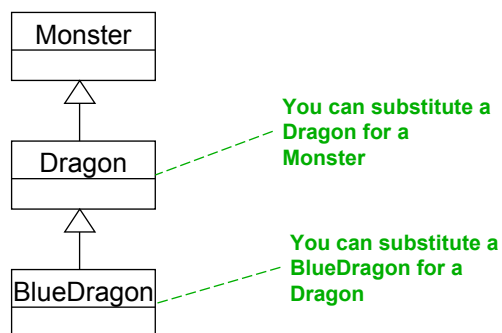
## The Result Of Attribute Shadowing (2)

- Location of the complete example:
  - [www.cpsc.ucalgary.ca/~tamj/233/examples/hiarchies/shadowing](http://www.cpsc.ucalgary.ca/~tamj/233/examples/hiarchies/shadowing)
  - [/home/233/examples/hiarchies/shadowing](http://home/233/examples/hiarchies/shadowing)

James Tam

## Casting And Inheritance

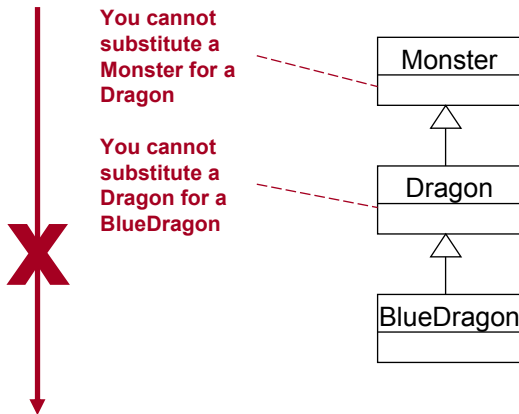
- Because the child class IS-A parent class you can substitute instances of a subclass for instances of a superclass.



James Tam

## Casting And Inheritance (2)

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



James Tam

## 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.");
    }
}
```

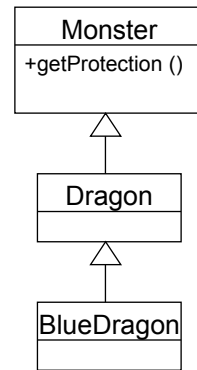
James Tam

## 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());
```



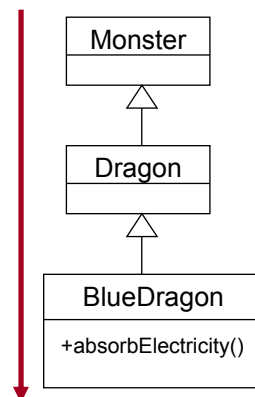
James Tam

## 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 ();
```

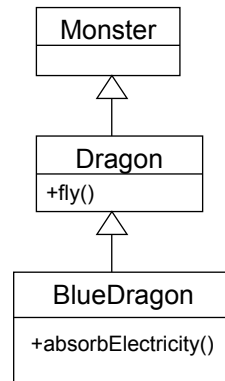


James Tam

## 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

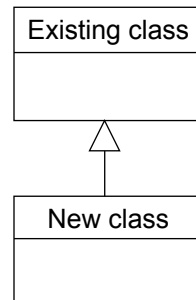
## The Final Modifier (Inheritance)

- 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
  - Parent's code
- 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

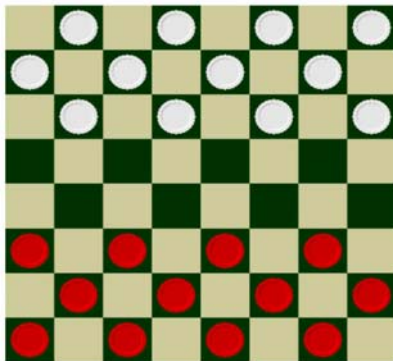
```
public interface <name of interface>
{
    constants
    methods to be implemented by the class that realizes this interface
}
```

### Format for realizing / implementing the interface

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

James Tam

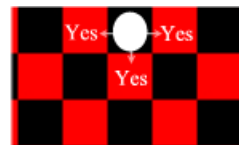
## Interfaces: A Checkers Example



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
```



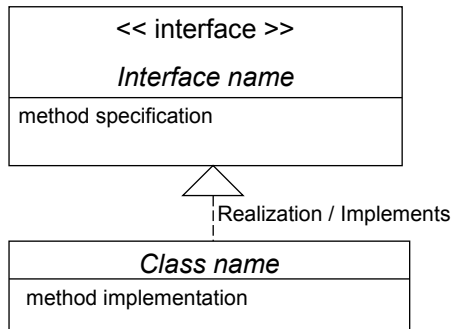
James Tam

## 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)

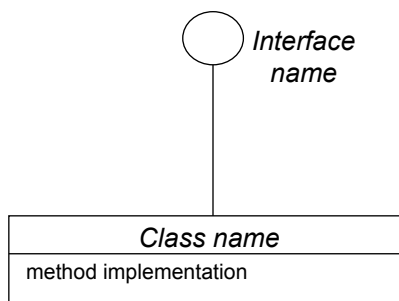
James Tam

## Specifying Interfaces In UML



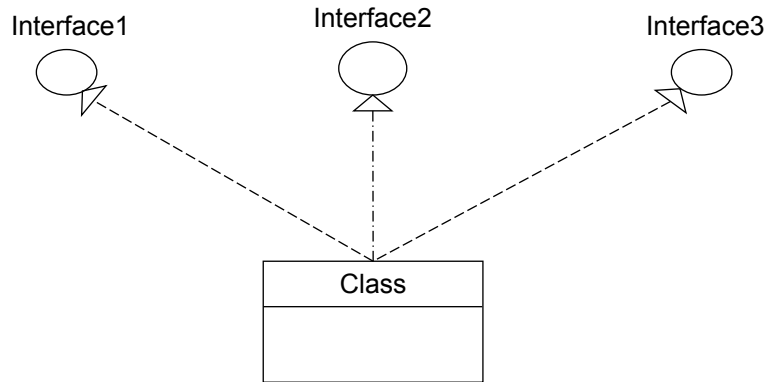
James Tam

## 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

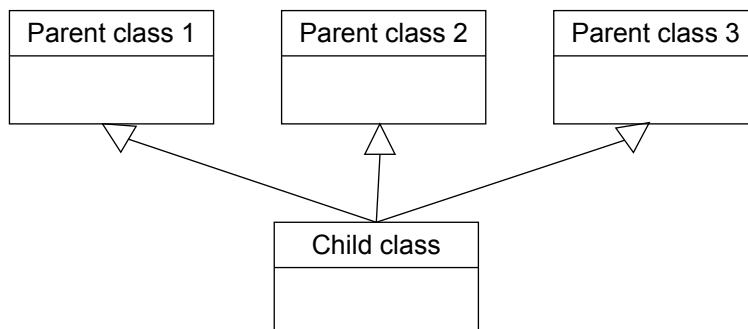
- A class can implement multiple interfaces
- Classes in Java cannot extend more than one class
- This is not possible in Java but is possible in other languages such as C++:

```
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 (Java)

- 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 (Java): 2

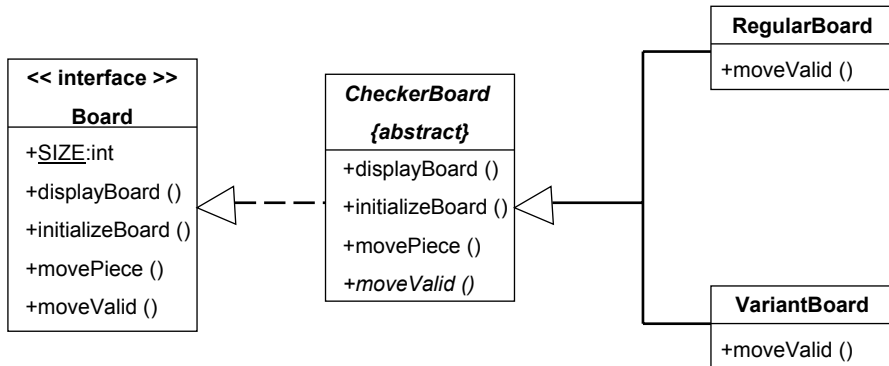
- Example<sup>1</sup>:**

```
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

## Important Terminology

- Is-A relationships/Inheritance
- Parent/Super/Generalization class
- Child/Sub/Specialization class
- Method overriding/polymorphism
- Early binding
- Dynamic/late binding
- Shadowing (in an inheritance hierarchy as well as the version of shadowing you previously learned)
- Casting
- Multiple implementation/realization
- Multiple inheritance

James Tam



## **Important Terminology (2)**

- Abstract classes
- Abstract methods

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.