

# Code Reuse Through Hierarchies

Part 1: defining commonalities between classes in parent using inheritance.

James Tam

## Real Life Hierarchies

Base entity:

- Attributes: age, height, weight
- Actions: eat(), sleep(), excrete(), multiply()

Derived entity: "martial artist" has all attributes and actions of base entity plus

- Attributes: belt/level
- Actions: ironPalmStrike(), shadowlessKick()

Derived entity: 'spy' has all attributes and actions of base entity plus

- Attributes: territory, code name e.g. 0-0TAM
- Actions: stealth(), codebreaking(), lockPicking()

James Tam

## Review: Association Relation 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.

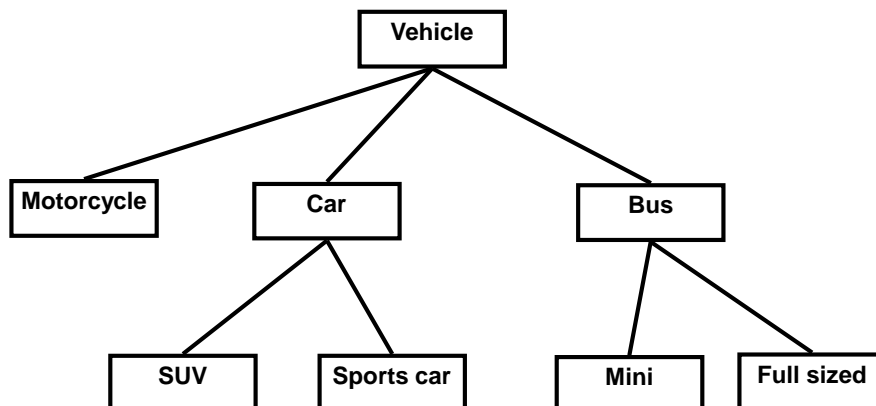
```
public class Car {  
    private Engine anEngine;  
    private Lights carLights;  
    public start() {  
        anEngine.ignite();  
        carLight.turnOn();  
    }  
}
```

```
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 more specific variant type of another class

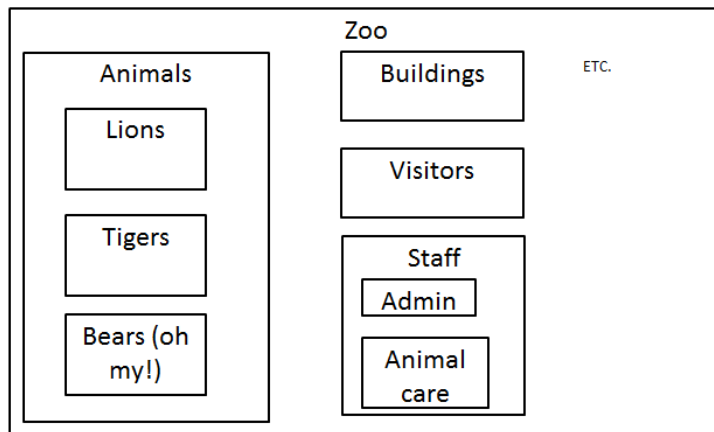


James Tam

## Recall O-O Approach: Finding Candidate Classes

An Example Of The Object-Oriented Approach (Simulation)

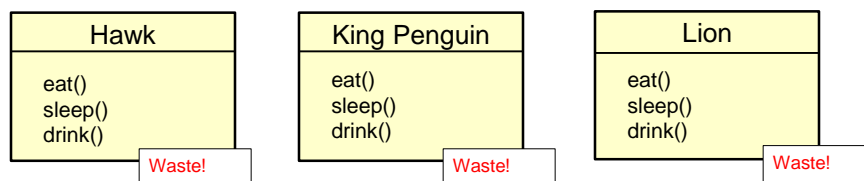
- Break down the program into entities (classes/objects - described with *nouns*)



James Tam

## What If There Are Commonalities Between Classes

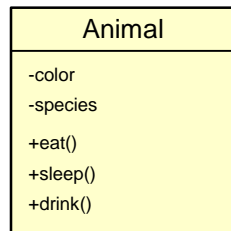
- Examples:
  - All birds 'fly'
  - Some types of animals 'swim': fish, penguins, some snakes, crocodiles, some birds
  - All animals 'eat', 'drink', 'sleep' etc.
  - Under the current approach you would have the same behaviors repeated over and over!



James Tam

## New Technique: Inheritance

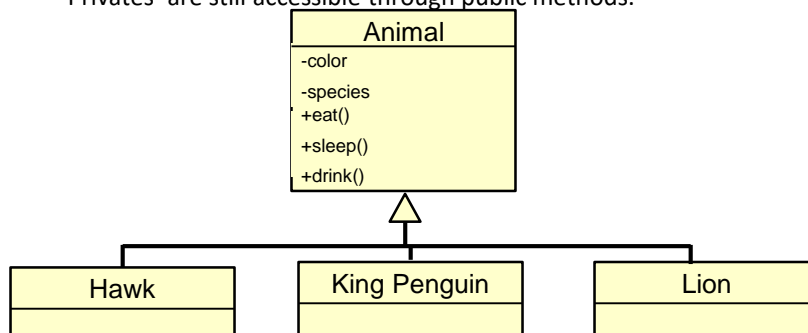
- When designing an Object-Oriented program, look for common behaviors and attributes
  - E.g., color, species, eat(), drink(), sleep()
  - These commonalities are defined in a 'parent' class



James Tam

## New Technique : Inheritance (2)

- These commonalities are defined in a 'parent' class
  - Classes that are derived from (are more specific versions) of the parent class are referred to a 'child' classes.
- As appropriate other 'child' classes will directly include or 'inherit' all the non-private attributes and behaviors of the parent class.
  - 'Privates' are still accessible through public methods.



James Tam

## Defining A Class That Inherits From Another

### Format:

```
public class <Name of child class> extends <Name of parent class>
{
    // Definition of child class - only what is unique to
    // this class
}
```

This means that a **Lion** object **AUTOMATICALLY** has all the capabilities of an **Animal** object

### Example:

```
public class Lion extends Animal
{
    public void roar() {
        System.out.println("Rawr!");
    }
}
```

The only attributes and methods that need to be specified are the ones unique to a lion

James Tam

## First Inheritance Example

- Name of the folder containing the complete example:  
1basicExample

James Tam

## Class Person

```
public class Person
{
    private int age;

    public Person() {
        age = 0;
    }

    public Person(int anAge) {
        age = anAge;
    }

    public void doPersonStuff() {
        System.out.println("Eat, sleep, drink, excrete, be" +
            " fruitful");
    }
}
```



Image of James  
Tam courtesy of  
James Tam

James Tam

## Class Hero: A Hero **Is A** Person

```
public class Hero extends Person {
```

This automatically gives  
instances of class **Hero** all  
the capabilities of an instance  
of class **Person**

```
}
```

James Tam

## Class Hero: A Person But [A Whole Lot More](#)

```
public class Hero extends Person
{
    private int heroicCount;

    public Hero()
    {
        heroicCount = 0;
    }

    public void doHeroStuff()
    {
        System.out.println("Saving the world for: truth!, " +
                           "justice!, and all that good " +
                           "stuff!");

        heroicCount++;
    }
}
```



Image of super  
James Tam courtesy  
of James Tam

James Tam

## The Driver Class: **Person** Vs. **Hero**

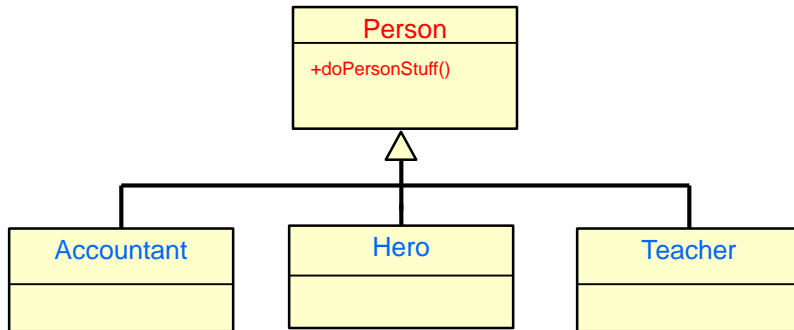
```
public class Driver
{
    public static void main(String [] args)
    {
        Person bob = new Person();
        bob.doPersonStuff(); Eat, sleep, drink, excrete, be fruitful
        System.out.println();

        Hero clark = new Hero();
        clark.doPersonStuff(); Eat, sleep, drink, excrete, be fruitful
        clark.doHeroStuff(); Saving the world for: truth!, justice!, and all that good stuff
    }
}
```

James Tam

## Benefit Of Employing Inheritance

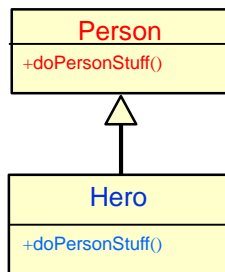
- Code reuse:
  - The common and accessible attributes and methods of the **parent** will automatically be available in all the **children**.



James Tam

## New Terminology: Method Overriding

- Overriding occurs when the **parent** class has a different version of a method than was implemented in the **child** class.

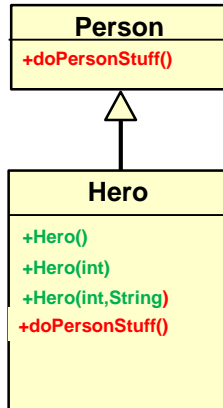


James Tam



## Reminder: **Method Overriding** Vs. Method Overloading

- **Method overriding**: different versions of a method in parent vs. child
- **Method overloading**: different versions of a method in a single class definition



James Tam

## Method Overriding Example

- Name of the folder containing the complete example:  
2overriding

James Tam

## Class Hero

```
public class Hero extends Person
{
    //New method: the rest of the class is the same as the
    //previous version
    public void doPersonStuff()
    {
        System.out.println("Pffff I need not go about " +
                           "mundane niceties such as eating!");
    }
}
```

James Tam

## The Driver Class (Included For Reference)

```
public class Driver
{
    public static void main(String [] args)
    {
        Person bob = new Person();
        bob.doPersonStuff();
        System.out.println();

        Hero clark = new Hero();
        clark.doPersonStuff();

        Pffff I need not go about mundane niceties such as eating!
    }
}
```

James Tam

## Overriding: Who Do We Call?

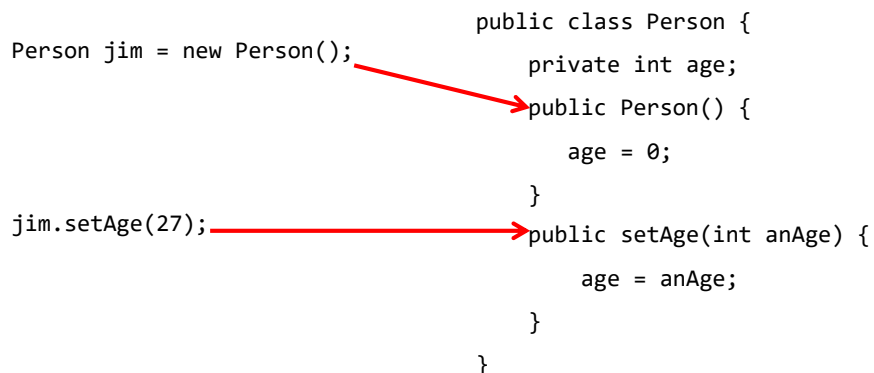
- `bob.doPersonStuff();`
- `clark.doPersonStuff();`

James Tam

## New Term: Binding

- When a reference and a method are specified together, binding determines which version of the method is called.
- If neither method overloading nor method overriding are employed then binding is very easy to determine.

```
Person jim = new Person();  
  
jim.setAge(27);  
  
public class Person {  
    private int age;  
    public Person() {  
        age = 0;  
    }  
    public setAge(int anAge) {  
        age = anAge;  
    }  
}
```



James Tam

## New Term: Binding (2)

- Early binding (overloading): determined at compile time (by 'javac')
  - Parameter list determines
- Late binding (overriding): determined at run time (by 'java')
  - The type of the implicit parameter ("this" reference) determines

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)

- Examples 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');
```

Binding at compile time (early)

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

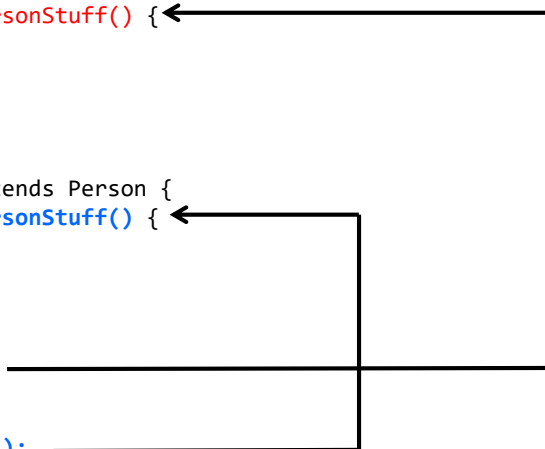
## Example Overriding: The Type Of The Reference Determines The Method Called

```
public class Person {
    public void doPersonStuff() {
        ...
    }
}

public class Hero extends Person {
    public void doPersonStuff() {
        ...
    }
}

// Bob is a Person
bob.doPersonstuff();

// Clarke is a Hero
clark.doPersonStuff();
```



James Tam

## New Terminology: Polymorphism

Poly = many      Morphic = forms

- A polymorphic method has an implementation in the parent class and a different implementation the child class.
- Polymorphism: the specific method called will be automatically be determined without any type checking needed (the type of reference determines which method is called)
- Recall the example:

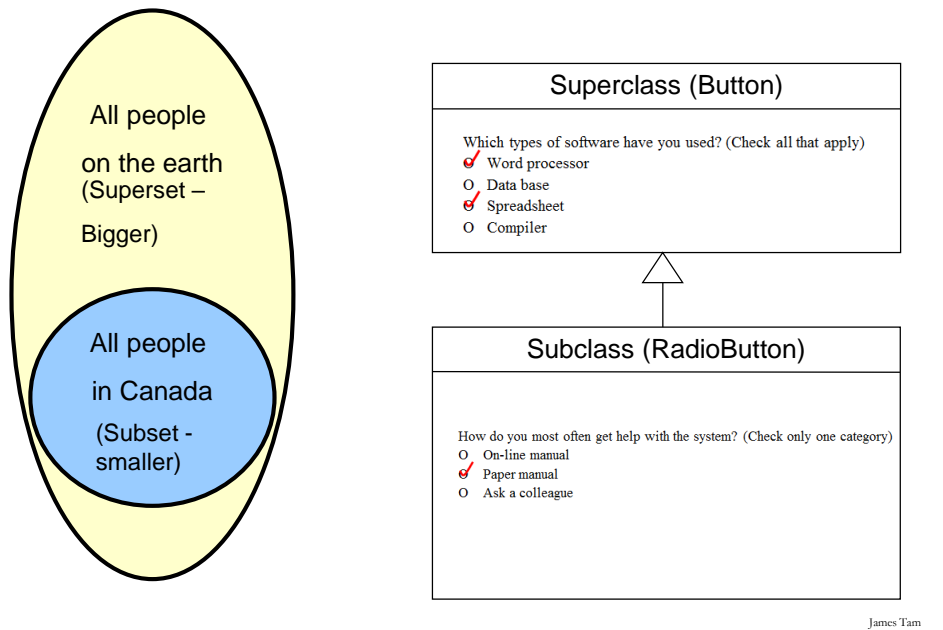
### The Driver Class (Included For Reference)

```
public class Driver
{
    public static void main(String [] args)
    {
        Person bob = new Person();
        bob.doPersonStuff(); // oh, sleep, drink, excrete, be fruitful!
        System.out.println();

        Hero clark = new Hero();
        clark.doPersonStuff();
        // Pffff I need not go about mundane niceties such as eating!
    }
}
```

James Tam

## New Terminology: Super-class Vs. Sub-class



## The 'Super' Keyword

- Used to access the parts of the super-class.

- **Format:**

`<super>.<method or attribute>`

- **Example:**

```
public void doPersonStuff()  
{  
    System.out.println("Pffff I need not go about mundane"+  
        " niceties such as eating!");  
  
    super.doPersonStuff();  
}
```

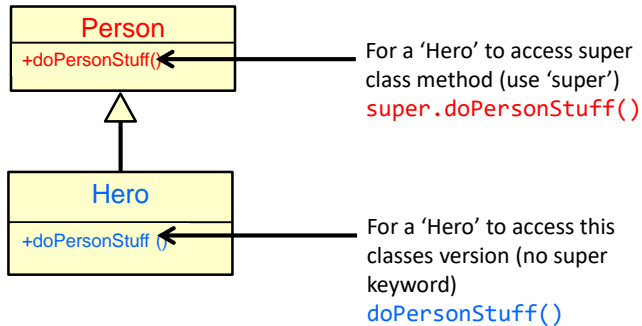
Pffff I need not go about mundane niceties such as eating!  
Eat, sleep, drink, excrete, be fruitful

Parent's version of  
method

James Tam

## Super Keyword: When It's Needed

- You only need this keyword when accessing non-unique methods or attributes (exist in both the super and sub-classes).

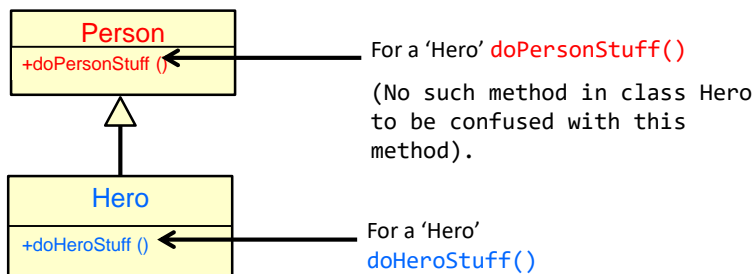


- Without the super keyword then the sub-class will be accessed

James Tam

## Super Keyword: When It's Not Needed

- If that method or attribute exists in only one class definition then there is no ambiguity.



James Tam



## Something Especially Good?

- Note: There Is No Super . Super In Java

James Tam

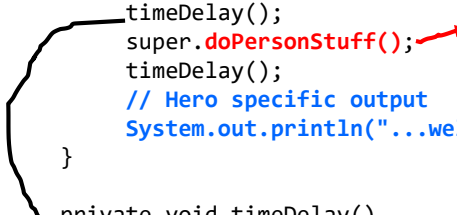
## Using The Super Keyword

- **Name of the folder containing the complete example:**  
3super
  - Note: this example illustrated the use of the super keyword in conjunction with a method "doPersonStuff".
  - As long as access permissions allow it, *any attribute or method in the super class can be accessed in the same way* using the 'super' keyword.

James Tam

## Class Hero: Using The Super Keyword

```
public class Hero extends Person
{
    public void doPersonStuff()
    {
        System.out.println("Pffff I need not go about mundane" +
                           " niceties such as eating!");
        timeDelay();
        super.doPersonStuff();
        timeDelay();
        // Hero specific output
        System.out.println("...well actually I do :$");
    }
    private void timeDelay()
    {
        final long DELAY_TIME = 1999999999;
        for (long i = 0; i <= DELAY_TIME; i++);
    }
}
```



```
public void doPersonStuff()
{
    System.out.println("Eat, sleep, drink, excrete, be" +
                      " fruitful");
}
```

For any Person

James Tam

## Using The Super Keyword Again

- This fourth example illustrates the use of this keyword with the constructor and the (new to this example) toString() method
- Note how the toString() method delegates some behavior to the parent class and implements some of the behaviors in the child class.
- **Name of the folder containing the complete example :**  
4superConstructors

James Tam

## Class Person

```
public class Person {
    private int age;

    public Person() {
        age = 0;
    }

    public Person(int anAge) {
        age = anAge;
    }

    public void doPersonStuff() {
        System.out.println("Eat, sleep, drink, execrate, be" +
            " fruitful");
    }
}
```

James Tam

## Class Person (2)

```
//NEW
public String toString()
{
    String s = "";
    s = s + "Age of the person: " + age;
    return(s);
}
}
```

James Tam

## Class Hero: Using **Super()**

```
public class Hero extends Person
{
    private int heroicCount;

    public Hero() {
        super();
        heroicCount = 0;
    }

    public Hero(int anAge) {
        super(anAge);
        heroicCount = 0;
    }

    public void doHeroStuff() {
        ...
        heroicCount++;
    }
}
```

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

```
public Person(int anAge) {
    age = anAge;
}
```

James Tam

## Class Hero: Using **Super():2**

```
//NEW
public String toString()
{
    String s = super.toString();
    if (s != null)
        s = s + "\n" + "Count of noble and heroic deeds " +
            heroicCount;
    return(s);
}
```

```
// Class Person
public String toString()
{
    String s = "";
    s = s + "Age of the person: " + age;
    return(s);
}
```

James Tam

## The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        Person bob = new Person(55);
        Hero clark = new Hero(25);
    }
}
```

```
public Person(int anAge){
    age = anAge;
}
```

```
public Hero(int anAge){
    super(anAge);
    heroicCount = 0;
}
```

```
public Person(int anAge){
    age = anAge;
}
```

James Tam

## The Driver Class: 2

```
System.out.println("Bob\n" + bob);
```

```
public String toString()
{
    String s = "";
    s = s + "Age of the person: " + age;
    return(s);
}
```

```
System.out.println("Clark\n" + clark);
```

```
}
}
```

```
public String toString() // Hero
{
    String s = super.toString();
    if (s != null)
        s = s + "\n" + "Count of noble and heroic deeds " +
            heroicCount;
    return(s);
}
```

```
public String toString() // Person
{
    String s = "";
    s = s + "Age of the person: " + age;
    return(s);
}
```

James Tam

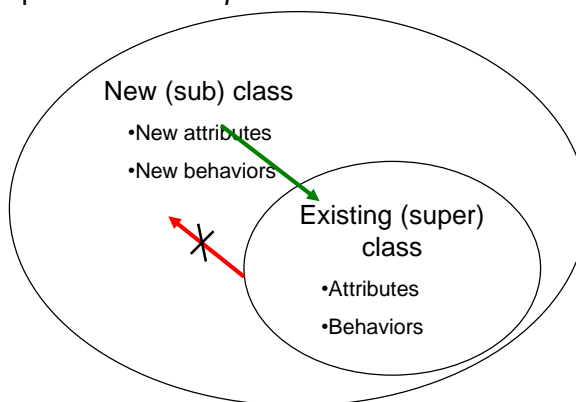
## Example 4 Synopsis

- Using the super keyword to access the parent constructors
- Uses the super keyword to access the parent implementation of toString()
- Both method calls would delegate some of the required behaviors to the parent (access modify parent class attributes) and then the child implement the remaining behavior (access child class attributes)

James Tam

## Keep In Mind: Inheritance Is A One Way Relationship!

- *A Hero is a Person but a Person is not a Hero!*
- That means that while the *sub-class can access the super-class* parts but the *super-class cannot access the sub-class* parts.



James Tam

## Access Modifiers And Inheritance

- Private '-': still works as-is, private attributes and methods can only be accessed within that classes' methods.
  - Child classes, similar to other classes must access private attributes through public methods.
- Public '+': still works as-is, public attributes and methods can be accessed anywhere.
- **New level of access, Protected '#'**: can access the method or attribute in the class or its sub-classes.

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
    // Can access num3 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



## Updated Scoping Rules

- When referring to an identifier in a method of a class
  1. Look in the local memory space for that method
  2. Look in the definition of the class
  3. New: Look in the definition of the parent class

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 >>>
        Reference to an identifier e.g., 'num'

    }
}
```

Similar to how local variables can shadow attributes, the child attributes can shadow parent attributes.

James Tam

## Updated Scoping Rules: A Trace

- Name of the folder containing the complete example :  
5scope

James Tam

## Scoping Rules: Review Code (1 Class)

```
public class Driver {  
    public static void main(String [] args) {  
        System.out.println("REVIEW");  
        System.out.println("-----");  
        P = new P();  
        p.method1();  
        System.out.println();  
    }
```

```
public class P {  
    protected int x = 1;  
    private int y = 2;  
  
    public void method1() {  
        int x = 10;  
        int y = 20;  
        System.out.println("P.method1()");  
        System.out.println("Locals shadow attributes");  
        System.out.println("x/y: " + x + " " + y);  
    }
```

```
P.method2()  
Loc x/y: 10 20  
Attr x/y: 1 2
```

## Scoping Rules: Review Code (1 Class): 2

```
p.method2();  
System.out.println();
```

```
public void method2()  
{  
    int x = 10;  
    int y = 20;  
    System.out.println("P.method2()");  
    System.out.println("Loc x/y: " + x + " " + y);  
    System.out.println("Attr x/y: " + this.x + " " +  
                        this.y);  
}
```

```
P.method2()  
Loc x/y: 10 20  
Attr x/y: 1 2
```

James Tam

## Updated Scoping Rules

```
System.out.println("NEW: INHERITANCE HIERARCHIES");  
System.out.println("-----");  
C c = new C();  
c.method1();
```

```
// Child  
public class C extends P {  
    private int x = 3;  
    private int z = 4;  
  
    public void method1() {  
        System.out.println("C.method1()");  
        System.out.println("Child attributes");  
        System.out.println("x/z: " + this.x + " " +  
                            this.z);  
    }  
}
```

James Tam

## Updated Scoping Rules (2)

```
c.method2();
```

```
// Child
public void method2() {
    int x = 100;
    int y = 200;
    int z = 300;
    System.out.println("C.method2()");
    System.out.println("Local shadows all");
    System.out.println("x/y/z: " + x + " " + y + " " +
                        z);
}
```

James Tam

## Updated Scoping Rules (3)

```
c.method3();
```

```
// Child
public void method3() {
    int x = 100;
    int y = 200;
    int z = 300;
    System.out.println("C.method3()");
    System.out.println("Loc x/y/z: " + x + " " + y + " " + z);

    System.out.println("P(x/y): " + super.x + " " + super.getY());
    // super.y : syntax error, access permission violated
    System.out.println("C(x/z): " + this.x + " " + this.z);
}
```

```
public class P
{
    protected int x = 1;
    private int y = 2;
}
```

James Tam

## The Final Modifier (Inheritance)

- What you know: the keyword `final` means unchanging (used in conjunction with the declaration of constants)
- Methods preceded by the final modifier cannot be overridden  
e.g., `public final void cannot_override()`
- Classes preceded by the final modifier cannot be extended  
-e.g., `final public class CANT_BE_EXTENDED`

James Tam

## You Should Now Know

- What is inheritance, when to employ it, how to employ it in Java
- Defining parent classes by looking for a commonalities
- What are the benefits of employing inheritance
- What is method overriding
  - How does it differ from method overloading
  - What is polymorphism
  - The difference between early vs. late binding and the criteria used to determine the method 'bound' in each case
- How does the 'protected' level of access permission work, how do `private` and `public` access permissions work with an inheritance hierarchy.
  - Under what situations should each level of permission be employed

James Tam

## **You Should Now Know (2)**

- Updated scoping rules (includes inheritance) and how shadowing works with an inheritance hierarchy
- How the 'super' keyword works, when it is and is not needed
- What is the effect of the keyword 'final' on inheritance relationships

James Tam

## **Copyright Notification**

- "Unless otherwise indicated, all images in this presentation are used with permission from Microsoft."

slide 60

James Tam