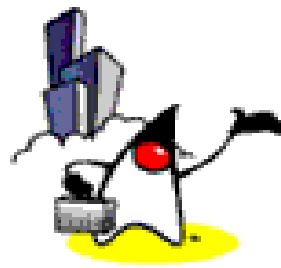




Inheritance



Agenda

- What is and Why Inheritance?
- Object class
- How to derive a sub-class?
- Constructor calling chain
- “super” keyword
- Overriding methods
- Hiding methods
- Hiding fields
- Type casting
- Final class and final methods





What is Inheritance?

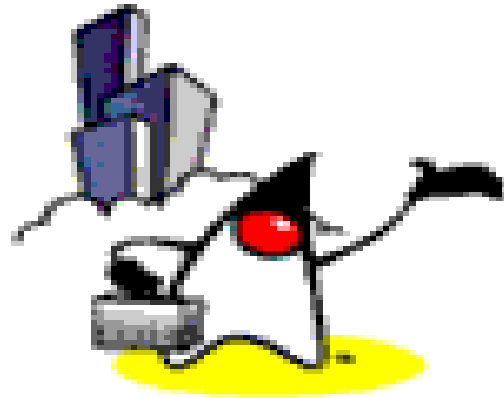
What is Inheritance?

- Inheritance is the concept of a child class automatically inheriting the variables and methods defined in its parent class.
- A primary feature of object-oriented programming

Why Inheritance? Reusability

- Benefits of Inheritance in OOP : Reusability
 - Once a behavior (method) is defined in a super class, that behavior is automatically inherited by all subclasses
 - Thus, you write a method only once and it can be used by all subclasses.
 - Once a set of properties (fields) are defined in a super class, the same set of properties are inherited by all subclasses
 - A class and its children share common set of properties
 - A subclass only needs to implement the differences between itself and the parent.





Object Class

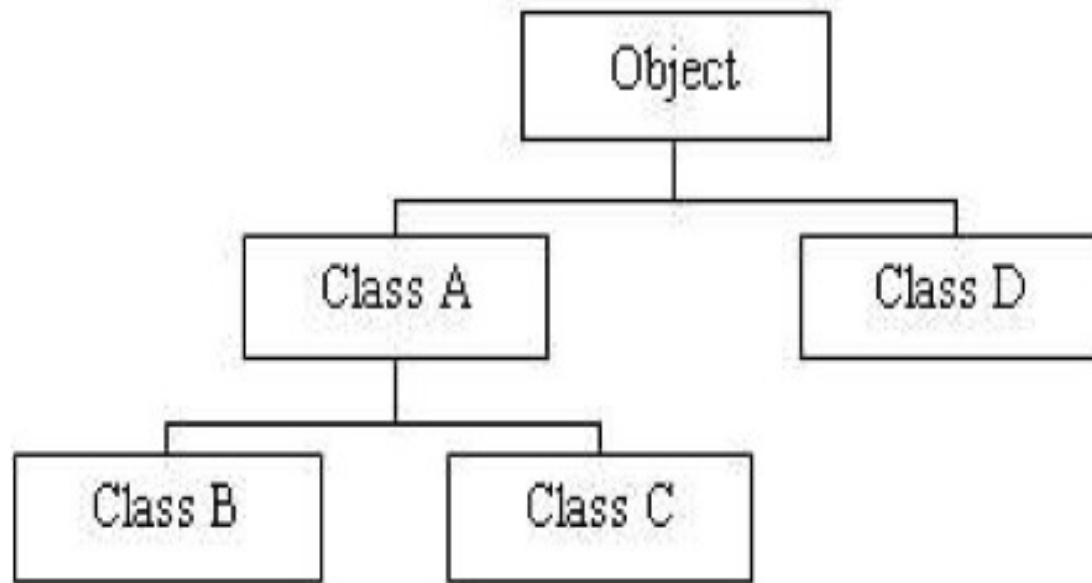
Object Class

- Object class is mother of all classes
 - In Java language, all classes are subclassed (extended) from the **Object** super class
 - Object class is the only class that does not have a parent class
- Defines and implements behavior common to all classes including the ones that you write
 - getClass()
 - equals()
 - toString()
 - ...



Class Hierarchy

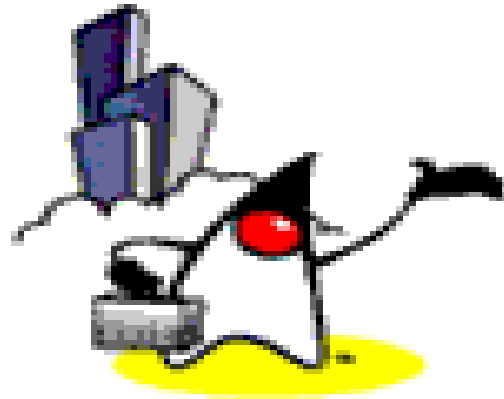
- A sample class hierarchy



Super class & Sub class

- Super class (Parent class)
 - Any class above a specific class in the class hierarchy.
- Sub class (Child class)
 - Any class below a specific class in the class hierarchy.





**How to derive a
sub-class?**

extends keyword

- To derive a child class, we use the **extends** keyword.
- Suppose we have a parent class called **Person**.

```
public class Person {  
    protected String name;  
    protected String address;  
  
    /**  
     * Default constructor  
     */  
    public Person(){  
        System.out.println("Inside Person:Constructor");  
        name = ""; address = "";  
    }  
    . . . .  
}
```



extends keyword

- Now, we want to create another class named **Student**
- Since a student is also a person, we decide to just extend the class **Person**, so that we can inherit all the properties and methods of the existing class **Person**.
- To do this, we write,

```
public class Student extends Person {  
    public Student(){  
        System.out.println("Inside Student:Constructor");  
    }  
    . . . .  
}
```



What You Can Do in a Sub-class

- A subclass inherits all of the “public” and “protected” members (fields or methods) of its parent, no matter what package the subclass is in
- If the subclass is in the same package as its parent, it also inherits the package-private members (fields or methods) of the parent



What You Can Do in a Sub-class Regarding Fields

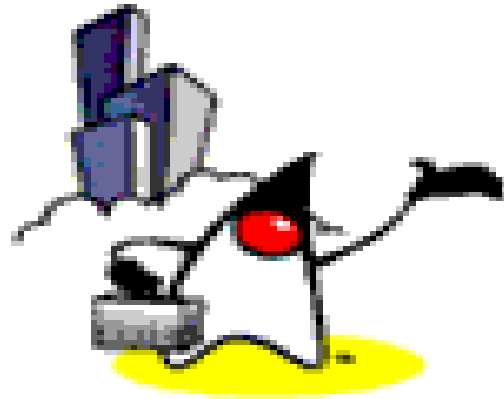
- The inherited fields can be used directly, just like any other fields.
- You can declare new fields in the subclass that are not in the super class
- You can declare a field in the subclass with the same name as the one in the super class, thus hiding it (not recommended).
- A subclass does not inherit the private members of its parent class. However, if the super class has public or protected methods for accessing its private fields, these can also be used by the subclass.



What You Can Do in a Sub-class Regarding Methods

- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the super class, thus overriding it.
- You can write a new static method in the subclass that has the same signature as the one in the super class, thus hiding it.
- You can declare new methods in the subclass that are not in the super class.





Constructor Calling Chain

How Constructor method of a Super class gets called

- A subclass constructor invokes the constructor of the super class implicitly
 - When a **Student** object, a subclass (child class), is instantiated, the default constructor of its super class (parent class), **Person** class, is invoked implicitly before sub-class's constructor method is invoked
- A subclass constructor can invoke the constructor of the super explicitly by using the “super” keyword
 - The constructor of the Student class can explicitly invoke the constructor of the Person class using “super” keyword
 - Used when passing parameters to the constructor of the super class



Example: Constructor Calling Chain

- To illustrate this, consider the following code,

```
public static void main( String[] args ){  
    Student anna = new Student();  
}
```

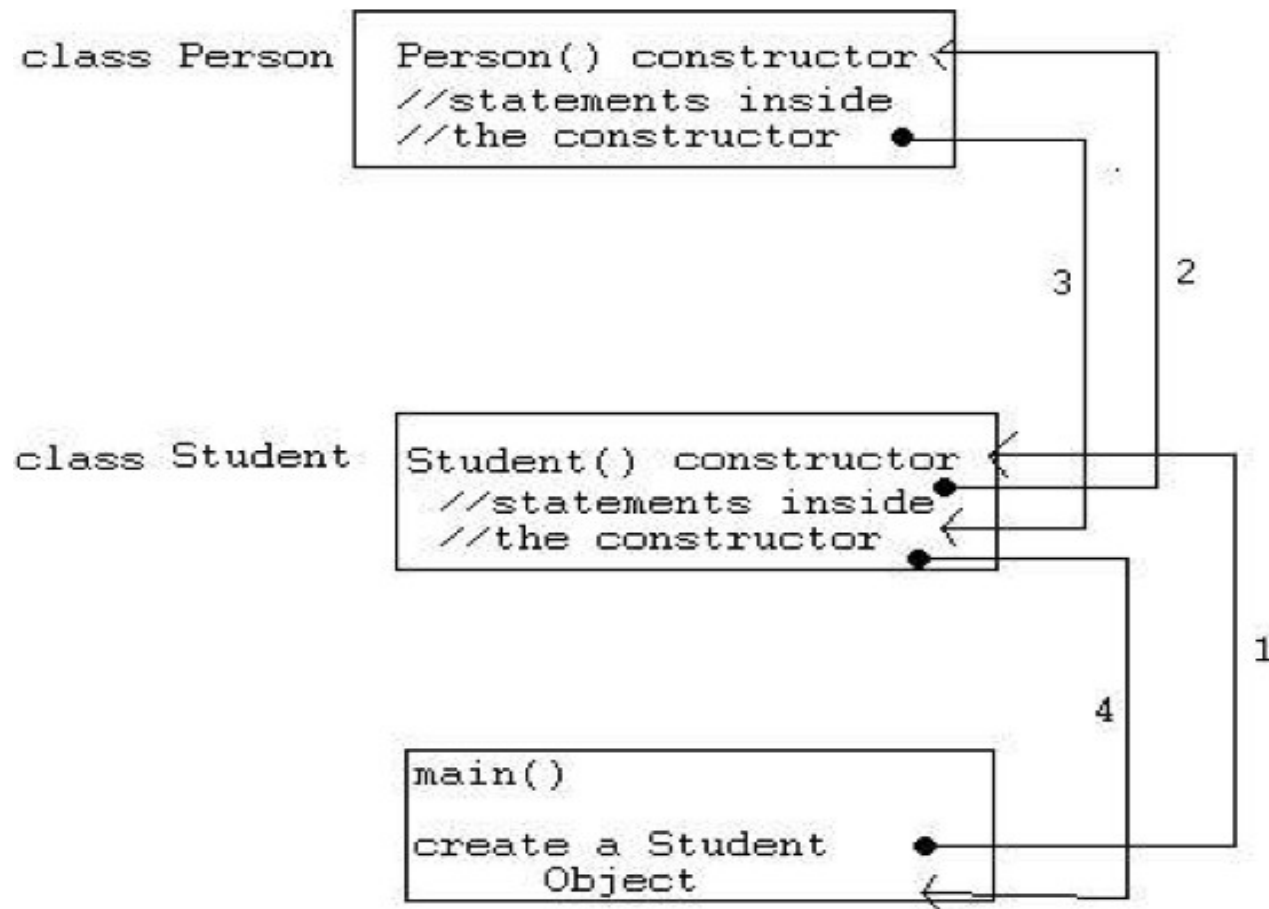
- In the code, we create an object of class **Student**.
The output of the program is,

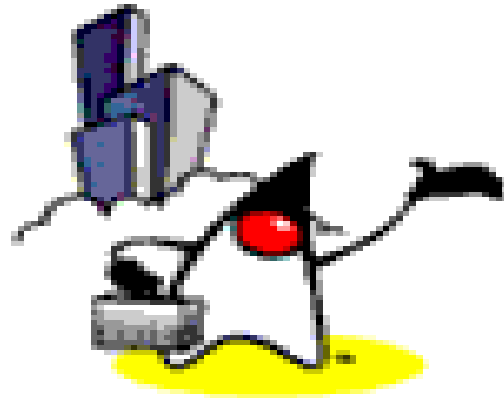
```
Inside Person:Constructor  
Inside Student:Constructor
```



Example: Constructor Calling Chain

- The program flow is shown below.





“super” keyword

The “super” keyword

- A subclass can also **explicitly** call a constructor of its immediate super class.
- This is done by using the **super** constructor call.
- A **super** constructor call in the constructor of a subclass will result in the execution of relevant constructor from the super class, based on the arguments passed.



The “super” keyword

- For example, given our previous example classes **Person** and **Student**, we show an example of a super constructor call.
- Given the following code for Student,

```
public Student(){  
    super( "SomeName", "SomeAddress" );  
    System.out.println("Inside Student:Constructor");  
}
```



The “super” keyword

- Few things to remember when using the super constructor call:
 - The super() call must occur as the first statement in a constructor
 - The super() call can only be used in a constructor (not in ordinary methods)

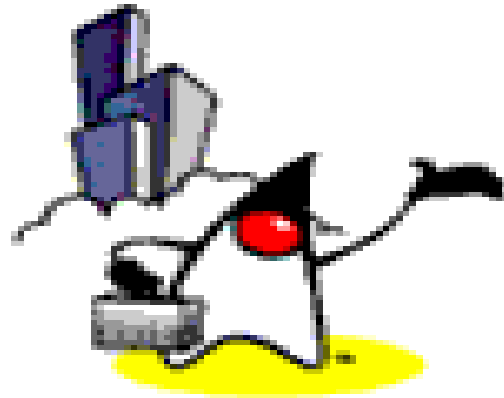


The “super” keyword

- Another use of super is to refer to members of the super class (just like the this reference).
- For example,

```
public Student() {  
    super.name = "somename";  
    super.address = "some address";  
}
```





Overriding Methods

Overriding methods

- If a derived class needs to have a different implementation of a certain instance method from that of the super class, override that instance method in the sub class
 - Note that the scheme of overriding applies only to instance methods
 - For static methods, it is called hiding methods
- The overriding method has the same name, number and type of parameters, and return type as the method it overrides
- The overriding method can also return a subtype of the type returned by the overridden method. This is called a covariant return type



Example: Overriding Methods

- Suppose we have the following implementation for the `getName` method in the `Person` super class,

```
public class Person {  
    :  
    :  
    public String getName(){  
        System.out.println("Parent: getName");  
        return name;  
    }  
}
```



Example: Overriding Methods

- To override the `getName` method of the super class `Person` in the subclass `Student`, reimplement the method with the same signature

```
public class Student extends Person{
    :
    public String getName(){
        System.out.println("Student: getName");
        return name;
    }
    :
}
```

- Now, when we invoke the `getName` method of an object of the subclass `Student`, the `getName` method of the `Student` would be called, and the output would be,

`Student: getName`



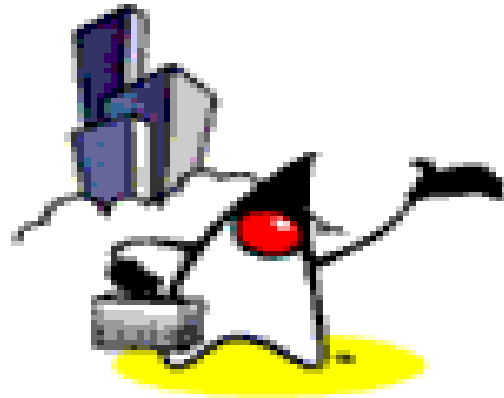
Modifiers in the Overriding Methods

- The access specifier for an overriding method can allow more, but not less, access than the overridden method
 - For example, a protected instance method in the super class can be made public, but not private, in the subclass.
- You will get a compile-time error if you attempt to change an instance method in the super class to a class method in the subclass, and vice versa



Runtime Polymorphism with Overriding Methods

- T



Hiding Methods

Hiding Methods

- If a subclass defines a class method (static method) with the same signature as a class method in the super class, the method in the subclass “**hides**” the one in the super class

Example: Coding of Hiding Static Method

```
class Animal {  
    public static void testClassMethod() {  
        System.out.println("The class method in Animal.");  
    }  
}
```

// The testClassMethod() of the child class hides the one of the
// super class – it looks like overriding, doesn't it? But
// there is difference. We will talk about in the following slide.

```
class Cat extends Animal {  
    public static void testClassMethod() {  
        System.out.println("The class method in Cat.");  
    }  
}
```



Overriding Method vs. Hiding Method

- Hiding a static method of a super class looks like overriding an instance method of a super class
- The difference comes during runtime
 - When you override an instance method, you still get the benefits of run-time polymorphism
 - When you override an static method, there is no run-time polymorphism



Example: Overriding Method vs. Hiding Method during Runtime

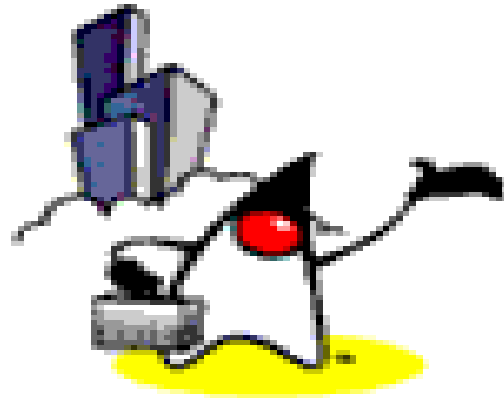
```
// Create object instance of Cat.  
Cat myCat = new Cat();
```

```
// The object instance is Cat type  
// and assigned to Animal type variable.  
Animal myAnimal2 = myCat;
```

```
// For static method, the static method of  
// the super class gets called.  
Animal.testClassMethod();
```

```
// For instance method, the instance method  
// of the subclass is called even though  
// myAnimal2 is a super class type. This is  
// run-time polymorphism.  
myAnimal2.testInstanceMethod();
```



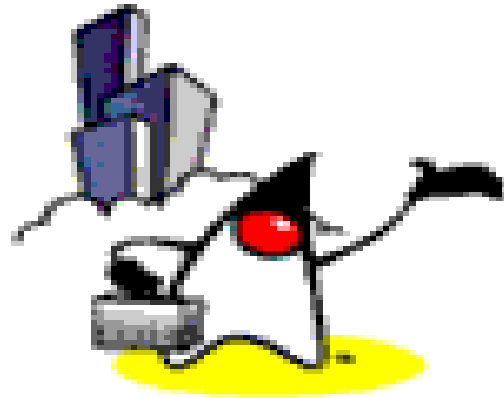


Hiding Fields

Hiding Fields

- Within a sub class, a field that has the same name as a field in the super class hides the super class' field, even if their types are different
- Within the subclass, the field in the super class cannot be referenced by its simple name
 - Instead, the field must be accessed through **super** keyword
- Generally speaking, hiding fields is not a recommended programming practice as it makes code difficult to read





Type Casting

What is “Type”?

- When an object instance is created from a class, we say the object instance is “type” of the class and its super classes

- Example:

`Student student1 = new Student();`

- student1 object instance is the type of Student or it is Student type
- student1 object instance is also type of Person if Student is a child class of Person
- student1 object instance is also type of Object



What is Significance?

- An object instance of a particular type can be used in any place where an instance of the type and its super type is called for
- Example:
 - student1 object instance is a “type” of TuftsStudent, Student, and Peson
 - student1 object can be used in any place where object instance of the type of TuftsStudent, Student, or Person is called for
- This enables polymorphism



Implicit Type Casting

- An object instance of a subclass can be assigned to a variable (reference) of a parent class through implicit type casting – this is safe since an object instance of a subclass “is” also the type of the super class
- Example
 - Let's assume `Student` class is a child class of `Person` class
 - Let's assume `TuftsStudent` class is a child class of `Student` class

```
TuftsStudent tuftstudent = new TuftsStudent();
```

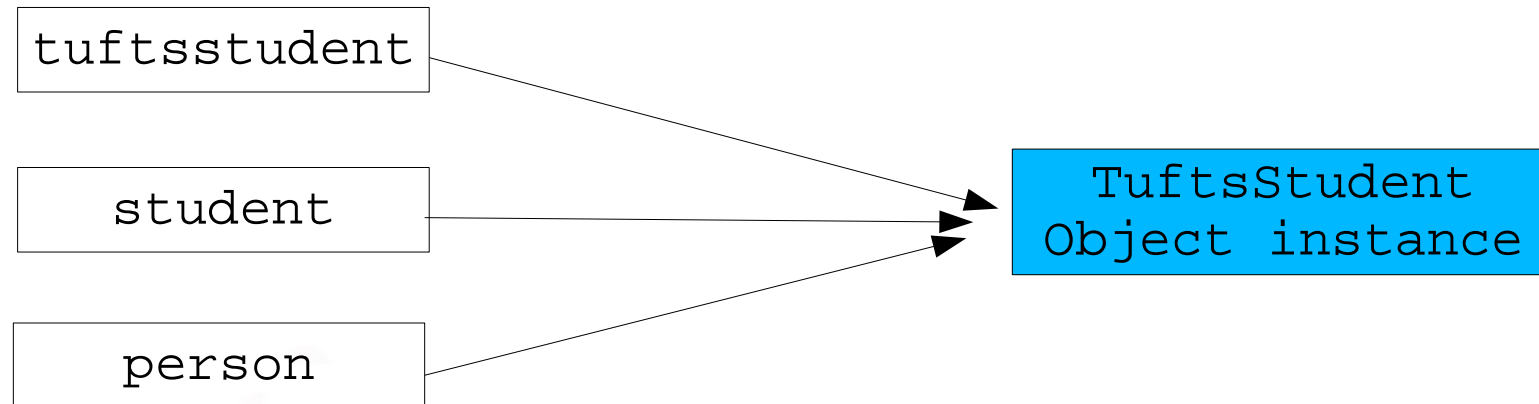
```
Student student = tuftsstudent; // Implicit type casting
```

```
Person person = tuftsstudent; // Implicit type casting
```

```
Object object = tuftsstudent; // Implicit type casting
```



Type Casting between Objects



Explicit Type Casting

- An object instance of a super class must be assigned to a variable (reference) of a child class through explicit type casting
 - Not doing it will result in a compile error since the type assignment is not safe
 - Compiler wants to make sure you know what you are doing
- Example
 - Let's assume **Student** class is a child class of **Person** class

```
Person person1 = new Student();  
Student student1 = (Student) person1; // Explicit type casting
```



Runtime Type Mismatch Exception

- Even with explicit casting, you could still end up having a runtime error
- Example
 - Let's assume **Student** class is a child class of **Person** class
 - Let's assume **Teacher** class is also a child class of **Person** class

```
Person person1 = new Student();
```

```
Person person2 = new Teacher();
```

```
Student student1 = (Student) person1; // Explicit type casting
```

```
// No compile error, but runtime type mismatch exception
```

```
Student student2 = (Student) person2;
```



Use instanceof Operator to Prevent Runtime Type Mismatch Error

- You can check the type of the object instance using **instanceof** before the type casting
- Example

```
Person person1 = new Student();
```

```
Person person2 = new Teacher();
```

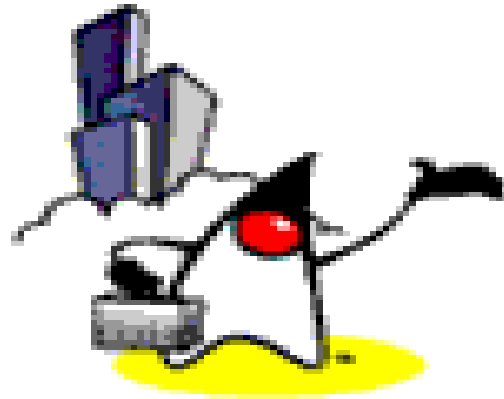
```
// Do the casting only when the type is verified
```

```
if (person2 instanceof Student) {
```

```
    Student student2 = (Student) person2;
```

```
}
```





Final Class & Final Method

Final Classes

- Final Classes

- Classes that cannot be extended
- To declare final classes, we write,

```
public final ClassName{  
    . . .  
}
```

- Example:

```
public final class Person {  
    . . .  
}
```

- Other examples of final classes are your wrapper classes and String class
 - You cannot create a subclass of String class



Final Methods

- Final Methods

- Methods that cannot be overridden
- To declare final methods, we write,

```
public final [returnType] [methodName]([parameters]){  
    . . .  
}
```

- Static methods are automatically final



Example: final Methods

```
public final String getName(){  
    return name;  
}
```



Inheritance

