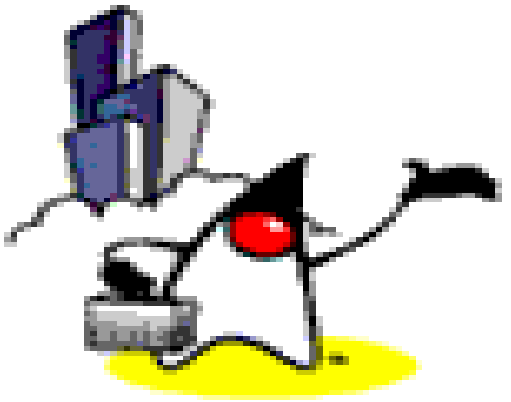
Inheritance

(Importance scale: ****, Very important)

Sang Shin
Michèle Garoche
www.javapassion.com
"Learn with Passion!"

Agenda

- What is and Why Inheritance?
- *Object* class
- How to derive a sub-class?
- Constructor calling chain
- "*super*" keyword
- Overriding methods
- Type casting (very important)
- Final class and final methods



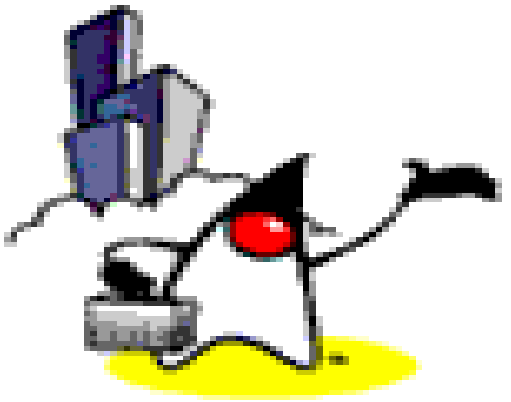
What is Inheritance?

What is Inheritance?

- Inheritance is the concept of a child class automatically inheriting the properties (fields) and methods defined in its parent class
  - Parent class is also called as super class
  - Child class is also called as sub class
- A primary feature of object-oriented programming
  - Along with encapsulation and polymorphism

Why Inheritance? 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 in a super class 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 (methods and properties) between itself and the parent.



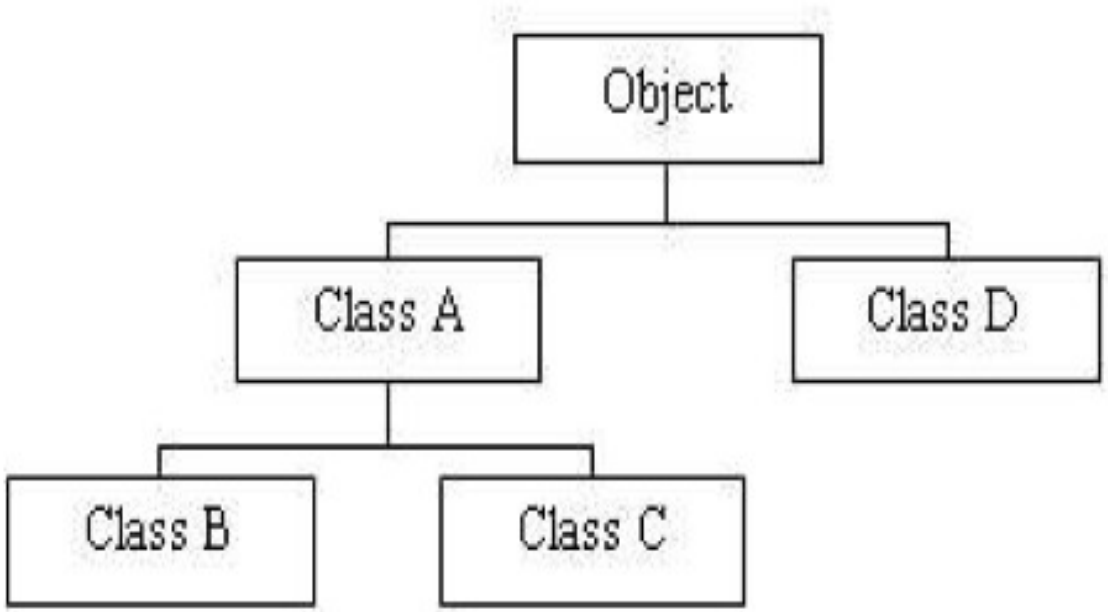
Object Class

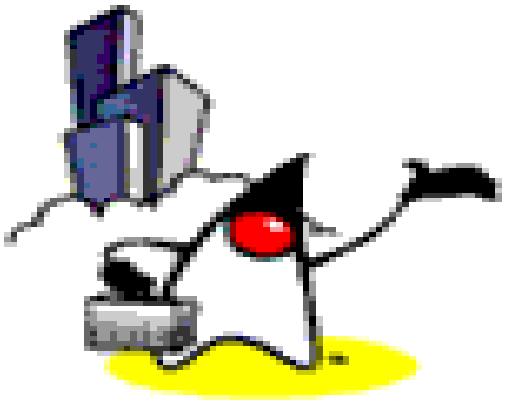
Object Class

- *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
  - *getClass()*
  - *equals()*
  - *toString()*
  - ...

Class Hierarchy

- *Object* class is the root parent class
- Classes A and B are child classes of *Object* class.
- Classes B and C are child classes of Class A





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

What You Can Do in a Sub-class

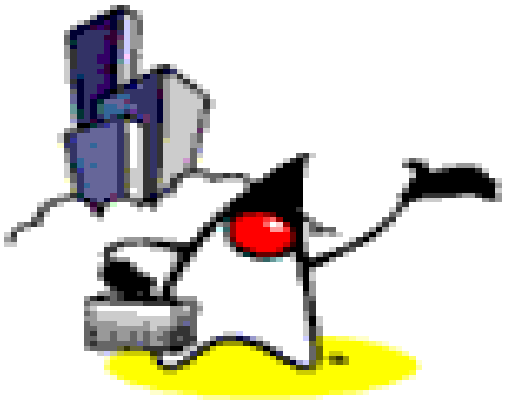
- A subclass inherits all of the “public” and “protected” members (fields or methods) of its parent, no matter what package the subclass is in
- If the subclass is in the same package as its parent, it also inherits the package-private members (fields or methods) of the parent
  - package-private members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

What You Can Do in a Sub-class Regarding Fields (properties)

- 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
- A subclass does not inherit the private members of its parent class. However, note that if the super class has public or protected methods that access the private fields , those methods can 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, thus providing a new behavior other than the one from the super class
- 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 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

- Person class

```
public class Person {
    public Person() {
        System.out.println("Person: constructor is called");
    }
   ..
```

- Student class

```
public class Student extends Person {
    public Student() {
        System.out.println("Student: constructor is called");
    }
   ...
```

- Main class

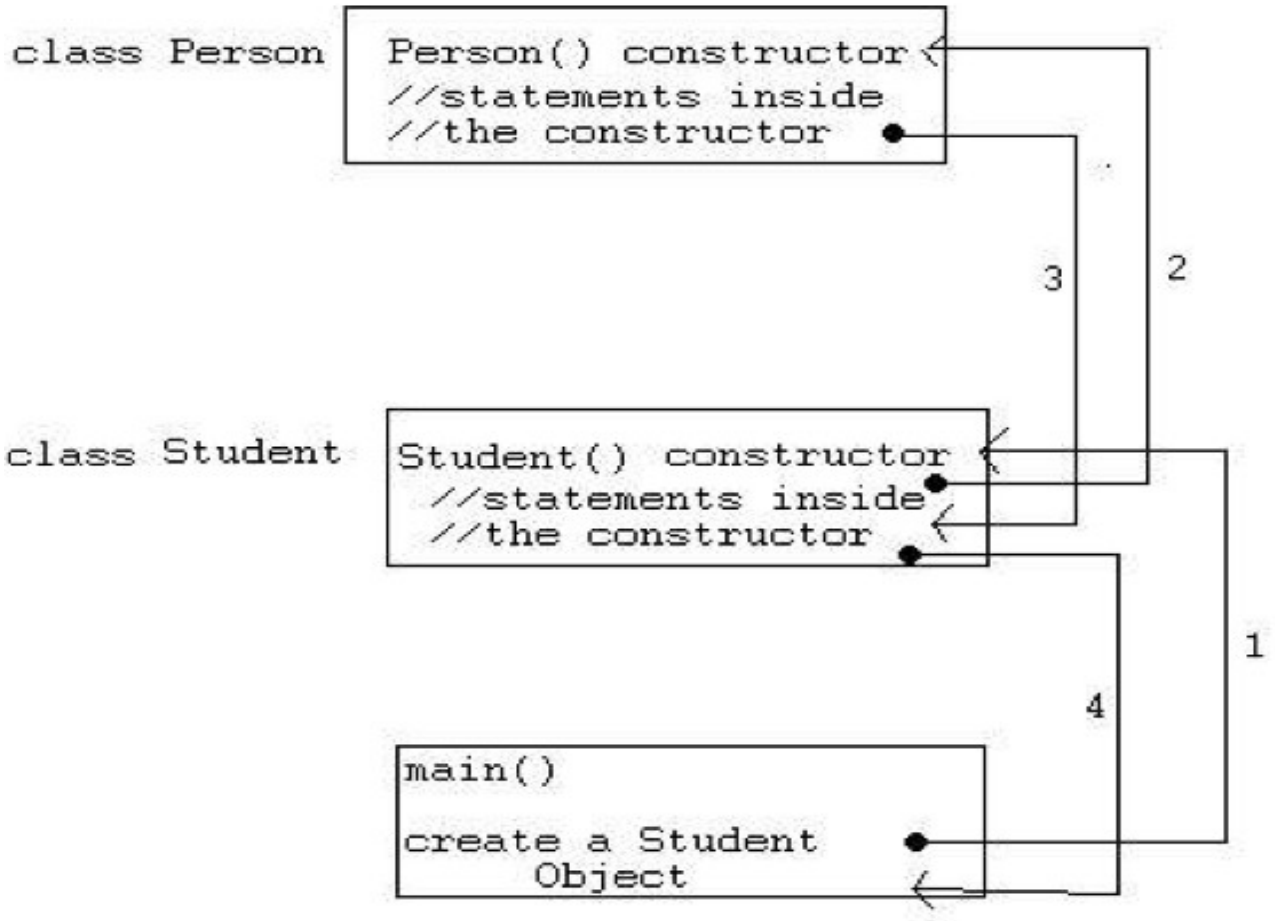
```
public static void main( String[] args ){
    Student anna = new Student();   // Instantiate Student object
}
```

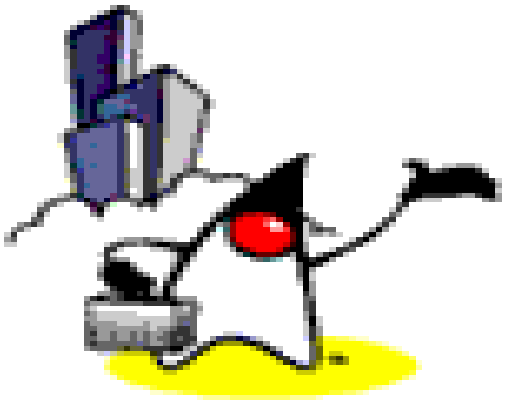
- Result

```
Person: constructor is called
Student: constructor is called
```

Example: Constructor Calling Chain

- The program flow is shown below.





super(..) method & super.<property-name>

super(..) method

- A subclass can also explicitly call a constructor of its immediate super class by calling *super(..)* constructor call.
  - This is in replacement of the default behavior of the constructor call chaining
- 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.

Explicit calling super(..) method

```
public class Person {
    public Person() {
      System.out.println("Person: constructor is called");
    }
    public Person(String name) {
      this.name = name;
      System.out.println("Person: constructor 2 is called");
    }
}

public class Student extends Person {
    public Student() {
      System.out.println("Student: constructor is called");
    }

    public Student(String name, String school, double grade) {
      super(name);        // Call the constructor of the parent class
      this.school = school;
      this.grade = grade;
      System.out.println("Student: constructor 2 is called");
  }
 ..
```

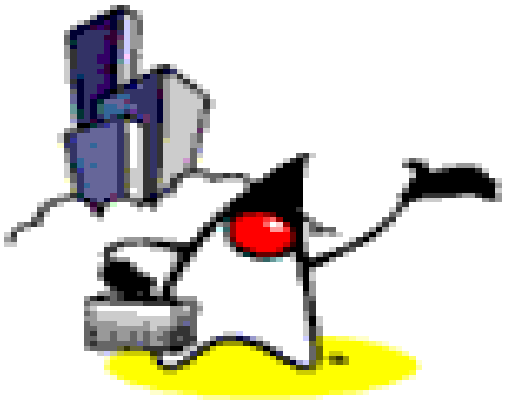
super(..) method

- A couple of 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)

super.<property-name>

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

```
public Student() {
    // Assuming the “name” and “address”
    // are not private members of the super class
    super.name = “somename”;
    super.address = “some address”;
}
```



Overriding Methods

Overriding methods

- If a derived class needs to have a different implementation (meaning different behavior) 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 method signature (same name, number and type of parameters) as the method it overrides
- The overriding method can also return a subtype of the type returned by the overridden method.

Example: Overriding Methods

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

```
// This is a parent class, which has
// getName() method
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 method signature

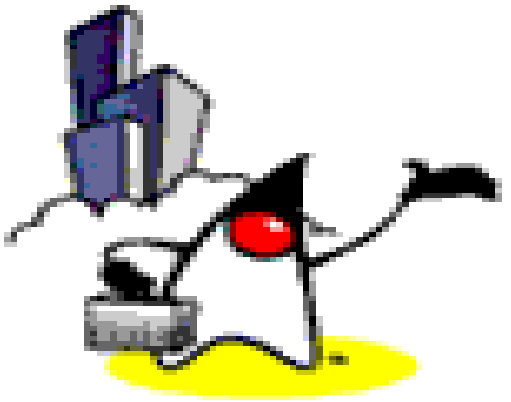
```
// Child class overriding getName() method
public class Student extends Person{
    :
    public String getName(){
        System.out.println("Student: getName");
        return name;
    }
    :
}
```

- Now, when we invoke getName() method of Student object, 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.



Type Casting

(This is very important concept.)

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* or it is *Person* type if *Student* is a child class of *Person*
  - student1 object instance is also type of *Object* because everyone class is subclass of *Object* class

What is the Significance?

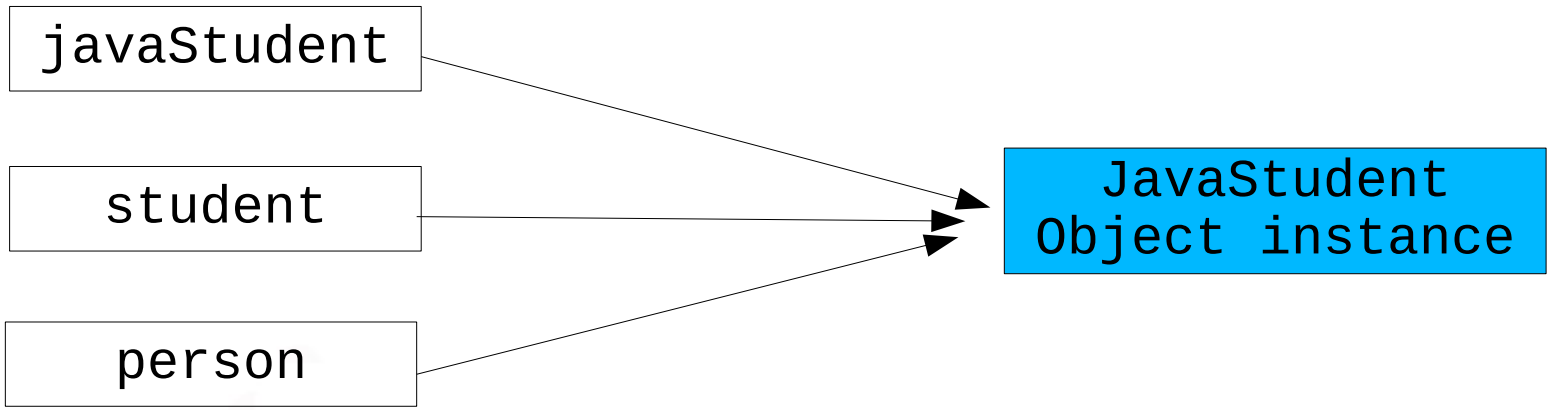
- An object instance of a particular type can be used in any place where an instance of the type and its super type is called for
- Example:
  - Let's say *student1* object instance is a “type” of *JavaStudent*, *Student*, and *Person*
  - Then the *student1* object can be used in any place where object instance of the type of *JavaStudent*, *Student*, or *Person* is called for
- This enables polymorphism (We will cover polymorphism later in detail)

Implicit Type Casting (Very Important)

- 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 JavaStudent class is a child class of Student class

```
JavaStudent javaStudent = new JavaStudent();
Student student = javaStudent; // Implicit type casting
Person person = javaStudent;  // Implicit type casting
Object object = javaStudent; // 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();
// Explicit type casting  required.
Student student1 = (Student) person1;
```

Runtime Type Mismatch Exception

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

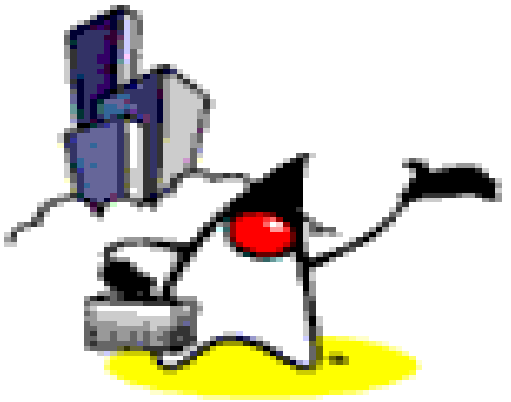
- You can check the type of the object instance using instanceof before the type casting
- Example

```
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;
}
```

Better Use Generics to detect Type Mismatch problem during Compile time

- Generics is introduced from Java SE 5
- Generics is designed to detect Type mismatch problem during compile time not during runtime



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;
}
```

Thank you!

Check JavaPassion.com Codecamps!
http://www.javapassion.com/codecamps
"Learn with Passion!"