AP Computer Science B

Java Object-Oriented Programming [Ver. 2.0]

Unit 4: Object-Oriented Design

WEEK 7: CHAPTER 12 INHERITANCE AND POLYMORPHISM (PART 2)

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

- Inclusion Polymorphism
- Equality of reference, object, class, and group
- Dynamic Binding
- Polymorphism Example
  - Base Class
  - People package

Polymorphism

LECTURE 1

Topics

- Overloading/Overriding Polymorphism
  - Overloading Methods/Overriding Methods
  - Dynamic Binding
- Coercion Polymorphism (Type Casting)
  - Implicit Casting
  - Explicit Casting
  - Autoboxing/Auto-unboxing
- Inclusion Polymorphism: Grouping/Subclass Polymorphism
- Parametric Polymorphism [Non-AP topic]

Inclusion Polymorphism

LECTURE 1

Inclusion Polymorphism

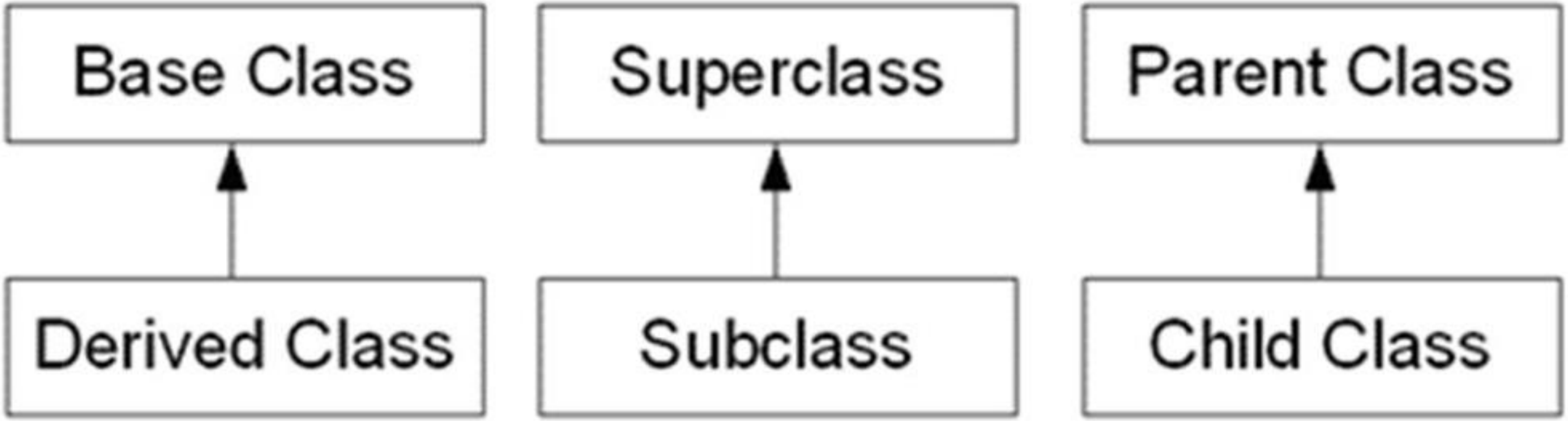
Subclass/Subtype Polymorphism

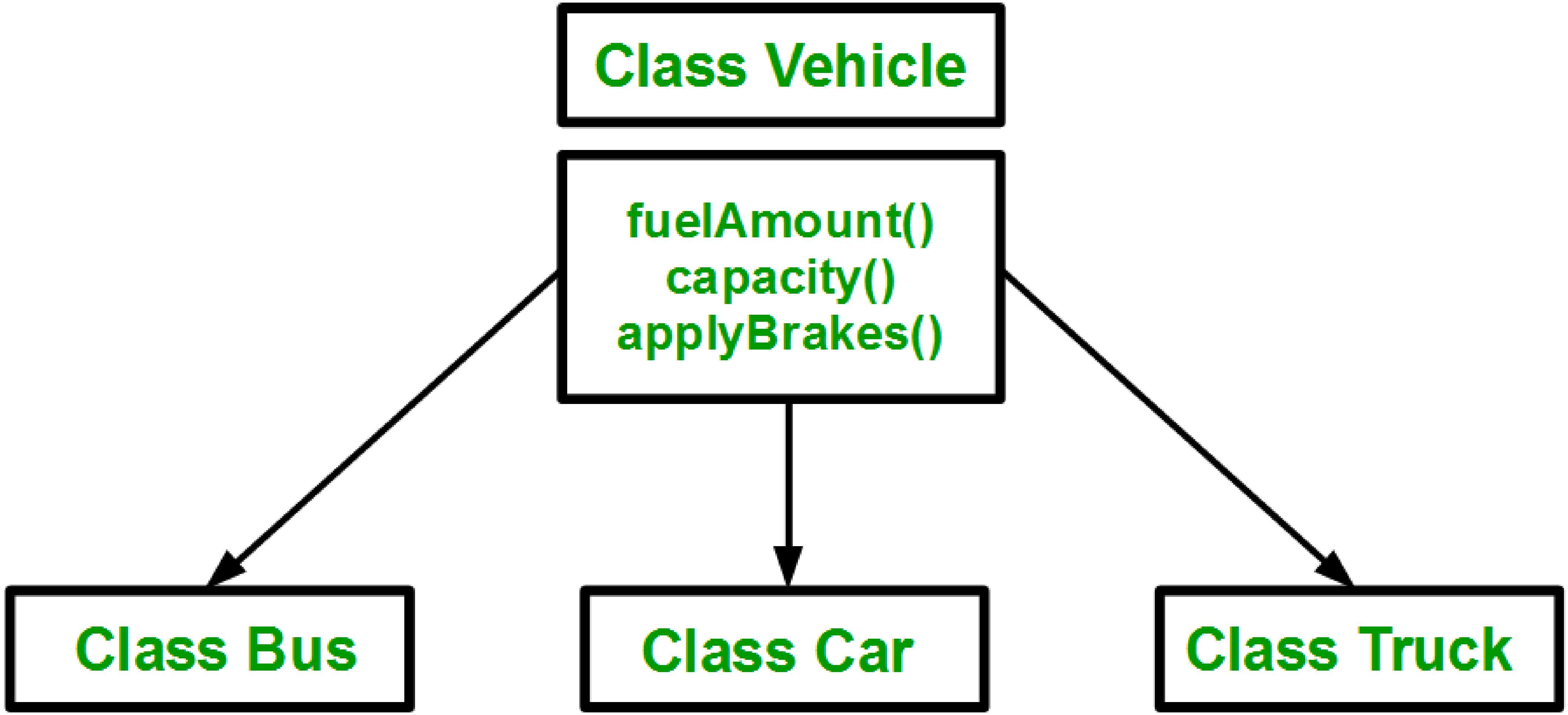
- **Subtype** means that a type can serve as another type's subtype. When a subtype instance appears in a supertype context, executing a supertype operation on the subtype instance results in the subtype's version of that operation executing. For example, consider a fragment of code that draws arbitrary shapes. You can express this drawing code more concisely by introducing a Shape class with a draw() method; by introducing Circle, Rectangle, and other subclasses that override draw(); by introducing an array of type Shape whose elements store references to Shape subclass instances; and by calling Shape's draw() method on each instance.

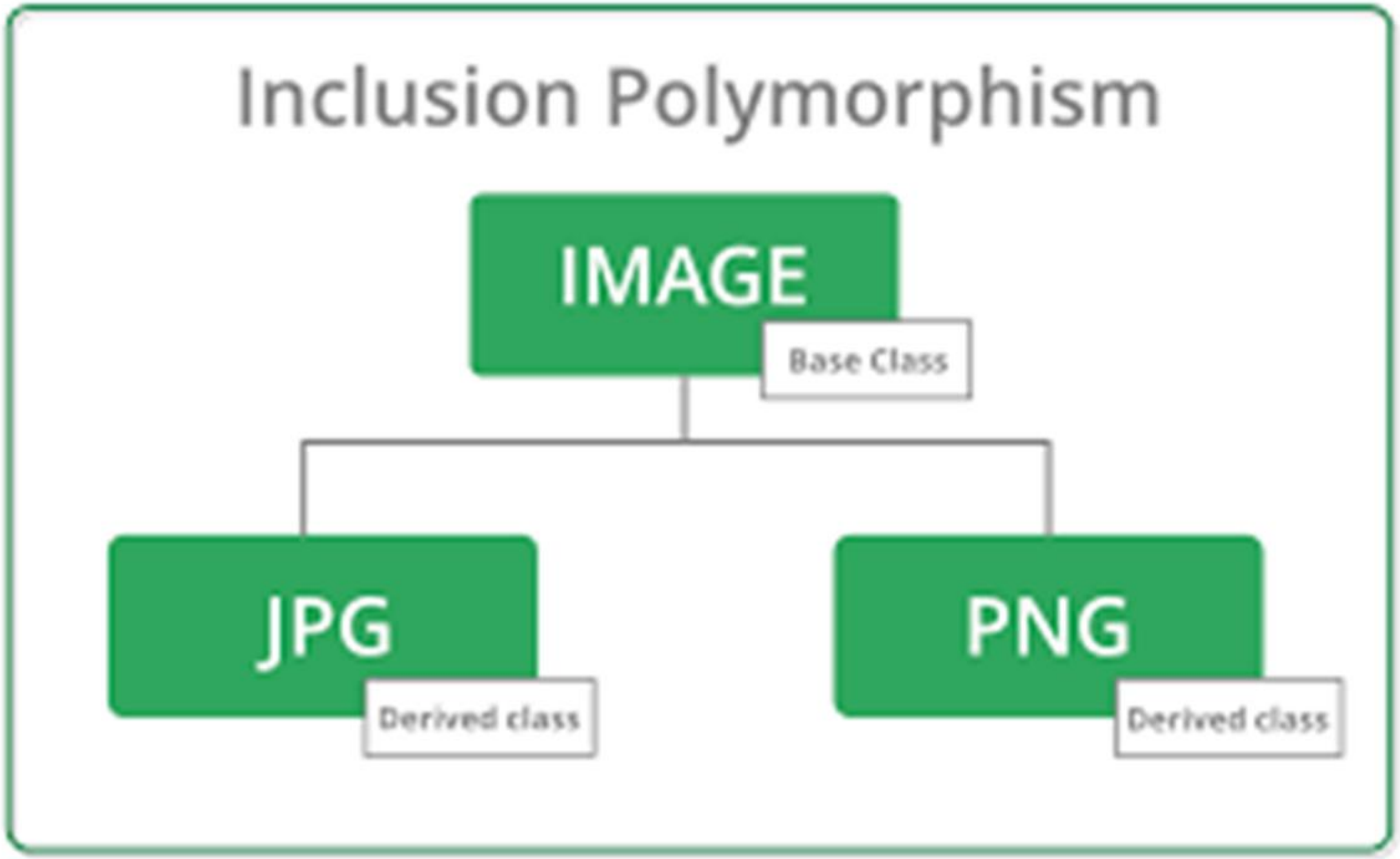
- When you call draw(), it's the Circle's, Rectangle's or other Shape instance's draw() method that gets called. We say that there are many forms of Shape's draw() method.

Inheritance using Java

- Terms
  - source: http://www.learn-java-tutorial.com/Java-Inheritance.cfm




Class Vehicle
fuelAmount()
capacity()
applyBrakes()
Class Bus
Class Car
Class Truck

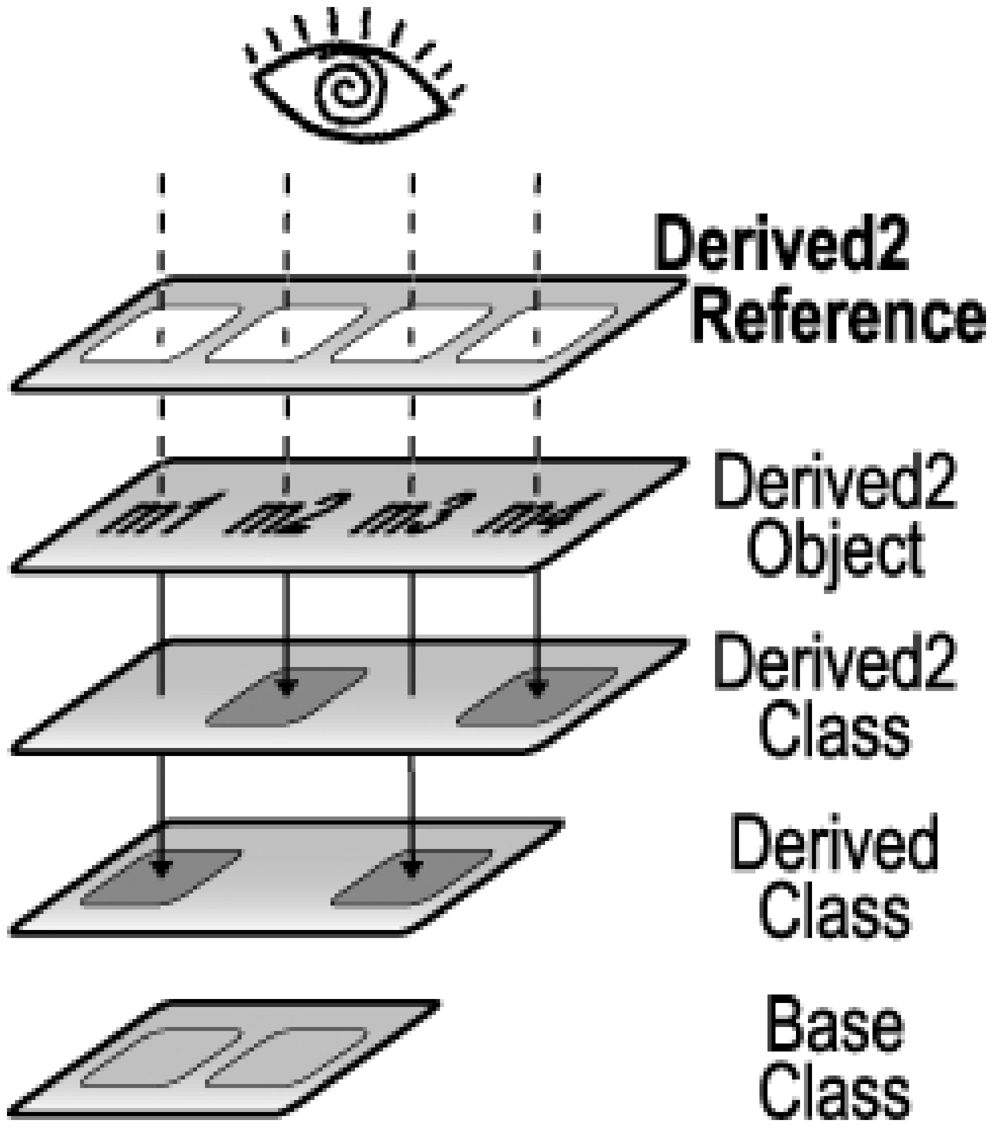

Inclusion Polymorphism
IMAGE
Base Class
JPG
Derived class
PNG
Derived class

```
HondaCRV carList = {car2, car1};

HondaCRV  car1 = new HondaCRV();
HondaCRV  car2 = new HondaCRV();
```

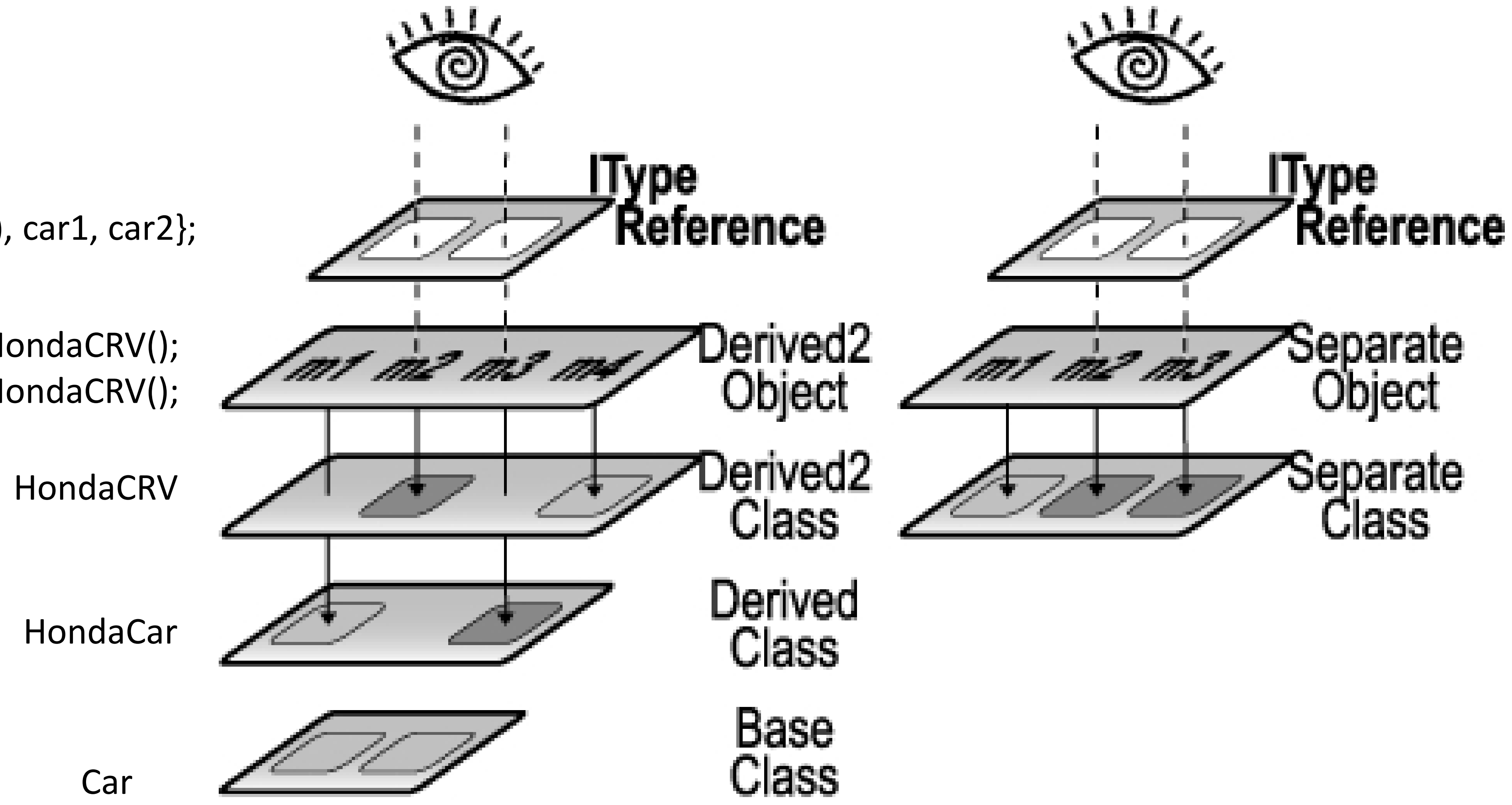
HondaCRV

HondaCar

Car

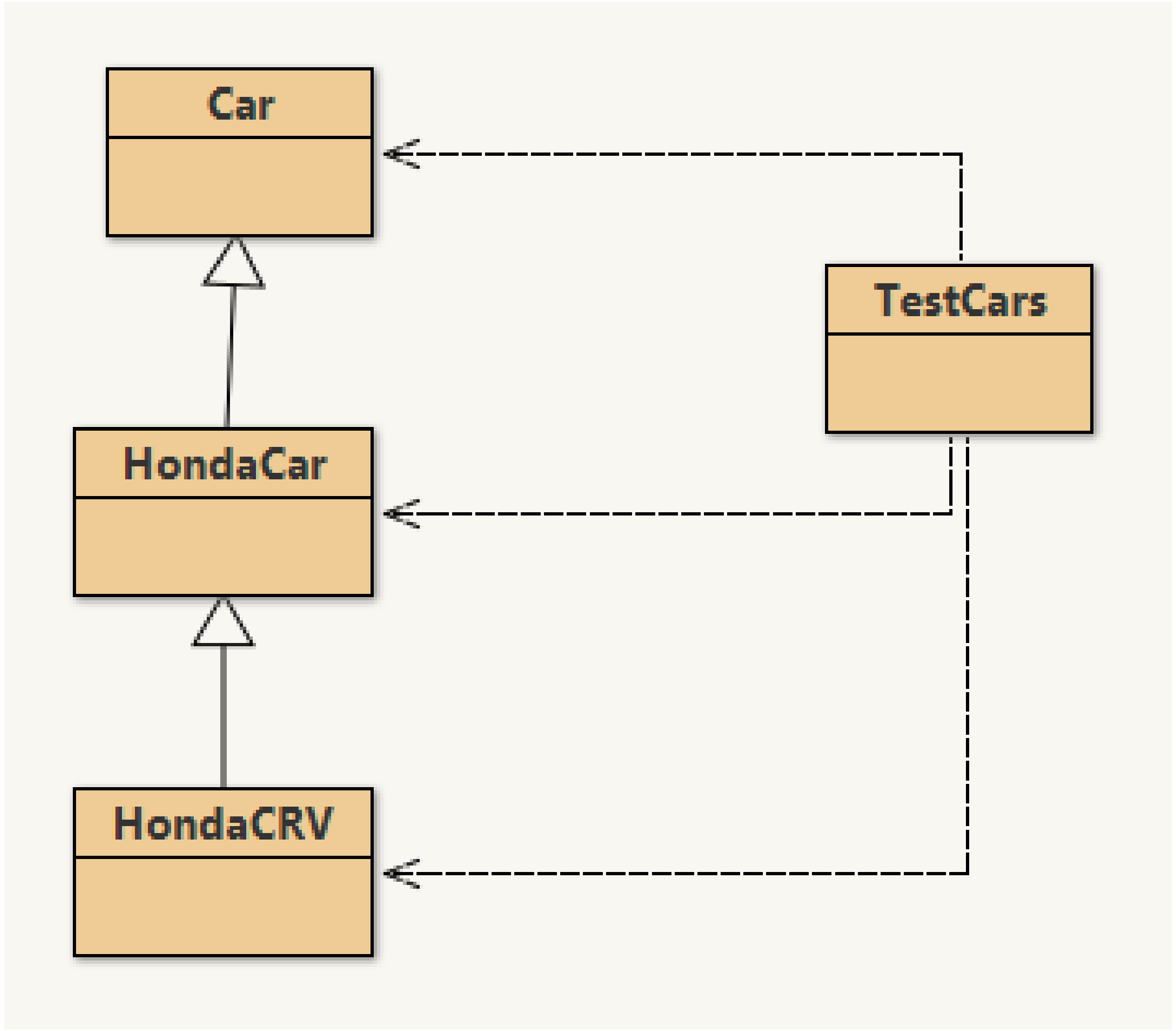


```
Car carList =
  {new Car(), new HondaCar(), car1, car2};

    HondaCRV  car1 = new HondaCRV();
    HondaCRV  car2 = new HondaCRV();
```



Demonstration Program

CAR.JAVA HONDACAR.JAVA HONDACRV.JAVA

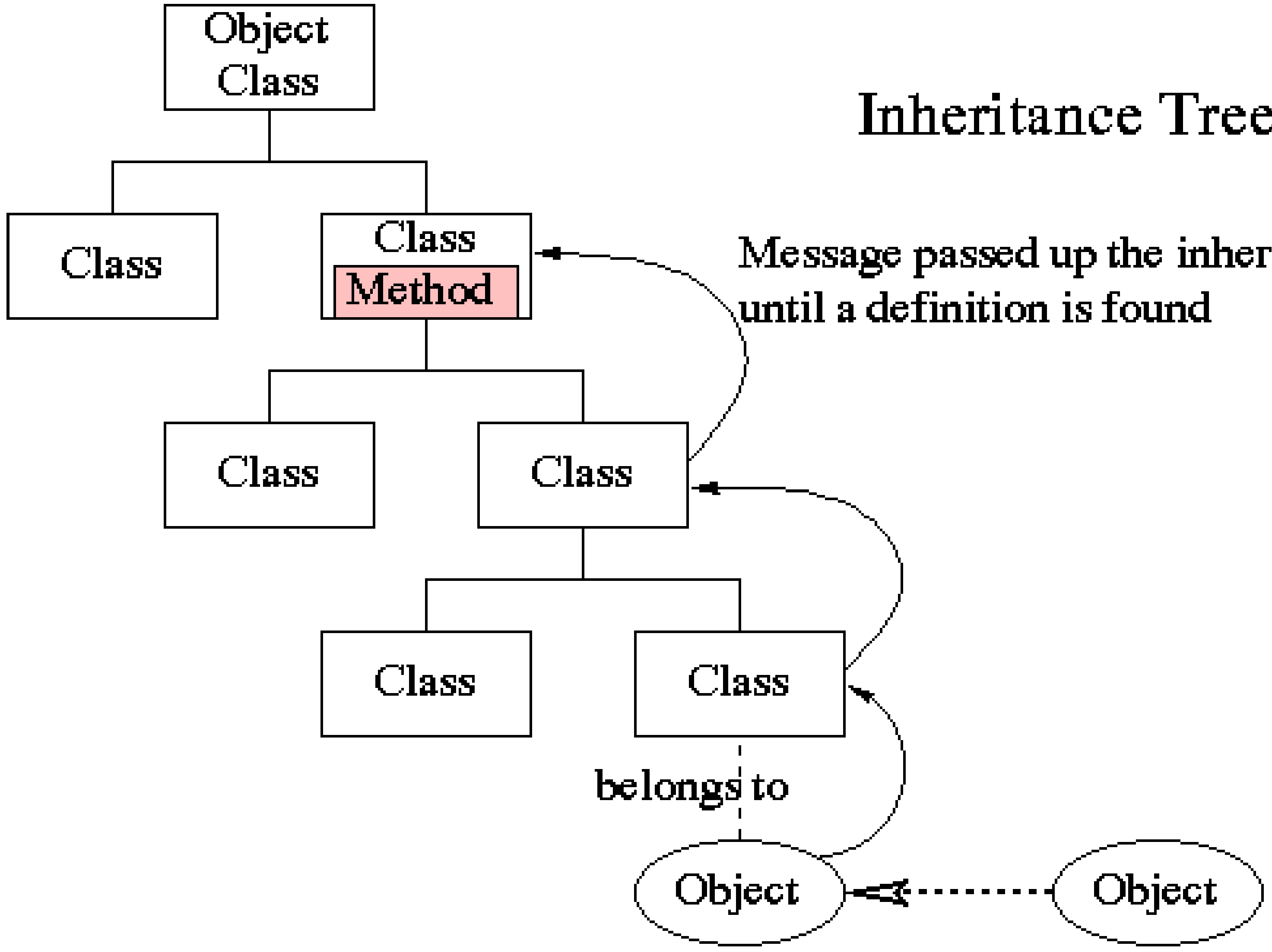

Car
HondaCar
HondaCRV
TestCars

Equality Check:

equality package

LECTURE 1

Inheritance Tree



Message passed up the inheritance tree until a definition is found

Message sent from one object to another

==	equals()	getClass()	instanceof
Same Pointer	Same Contents (By User Definition)	Same Class **obj.getClass().getName()** Then, compare the name String.	Check membership through the class hierarchy Then, two object by instanceof operator

Different Degrees of Identity Check in Java

The equals and getClass methods

Various Identity Level: pointer address, content and class

The equals method is the second most important one (toString(), equals()) to override. It's usage within a class is as the member function:

```
@Override
public boolean equals(Object obj) {
   ...
}
```

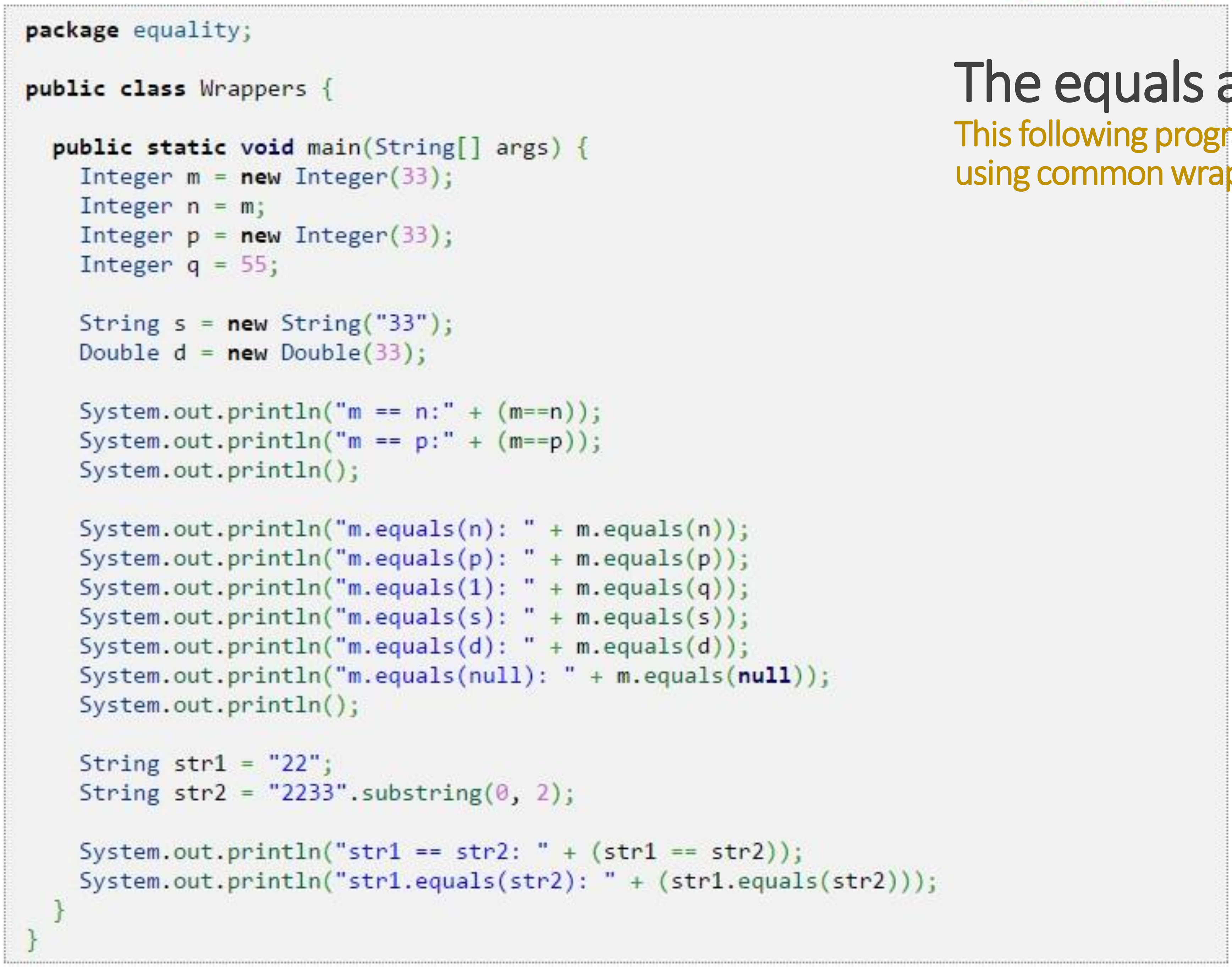
By default, equality testing is based on identity, but classes which use equality testing usually override equals so that the test is somehow based on the object's content. The programs of interest are found in the package

equality

The equals and getClass methods

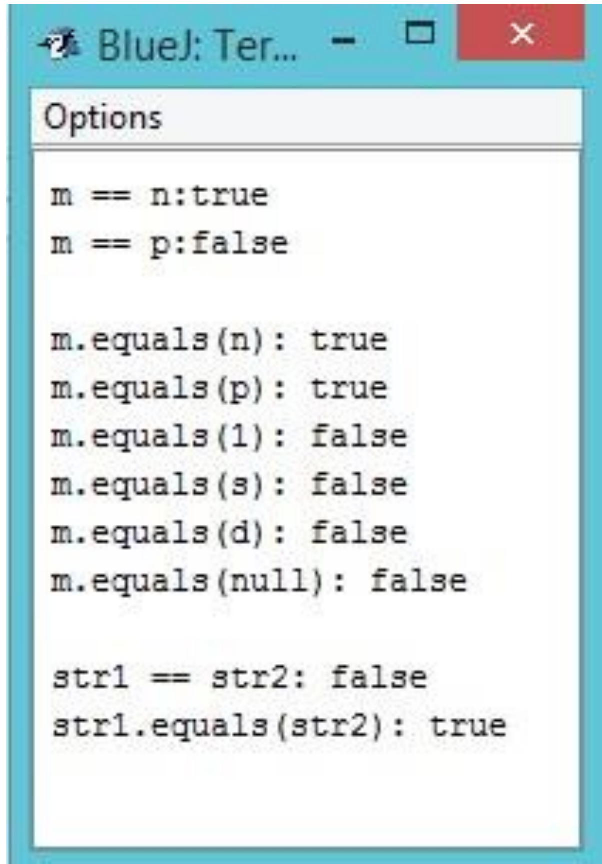
This following program illustrates ".equals" comparisons using common wrapper classes:

equality.Wrappers



```
package equality;

public class Wrappers {

  public static void main(String[] args) {
    Integer m = new Integer(33);
    Integer n = m;
    Integer p = new Integer(33);
    Integer q = 55;

    String s = new String("33");
    Double d = new Double(33);

    System.out.println("m == n:" + (m==n));
    System.out.println("m == p:" + (m==p));
    System.out.println();

    System.out.println("m.equals(n): " + m.equals(n));
    System.out.println("m.equals(p): " + m.equals(p));
    System.out.println("m.equals(1): " + m.equals(q));
    System.out.println("m.equals(s): " + m.equals(s));
    System.out.println("m.equals(d): " + m.equals(d));
    System.out.println("m.equals(null): " + m.equals(null));
    System.out.println();

    String str1 = "22";
    String str2 = "2233".substring(0, 2);

    System.out.println("str1 == str2: " + (str1 == str2));
    System.out.println("str1.equals(str2): " + (str1.equals(str2)));
  }
}
```

Wrapper Class is a Tester Class

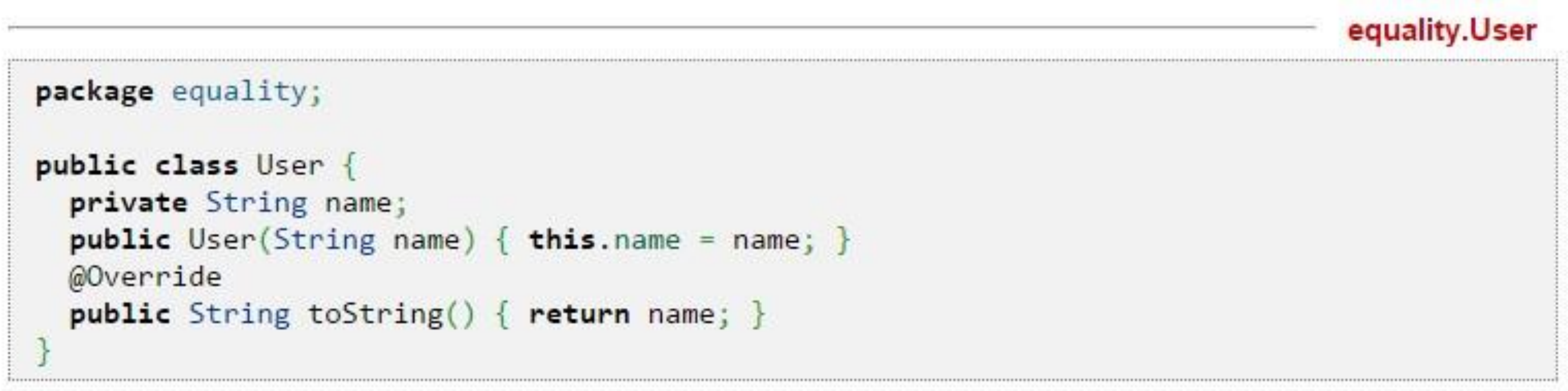


```
m == n:true
m == p:false

m.equals(n): true
m.equals(p): true
m.equals(1): false
m.equals(s): false
m.equals(d): false
m.equals(null): false

str1 == str2: false
str1.equals(str2): true
```

The equals and getClass methods

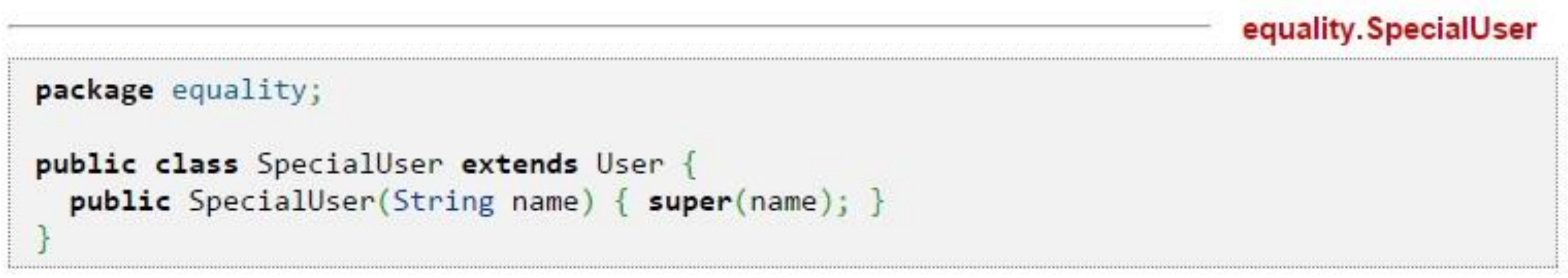
The String class is different from the numbers in that its literals are not primitive types, and so you almost never want comparison via "==".

Suppose we consider the User class defined above. Here are two relevant starter classes used:



equality.User

```
package equality;

public class User {
  private String name;
  public User(String name) { this.name = name; }
  @Override
  public String toString() { return name; }
}
```

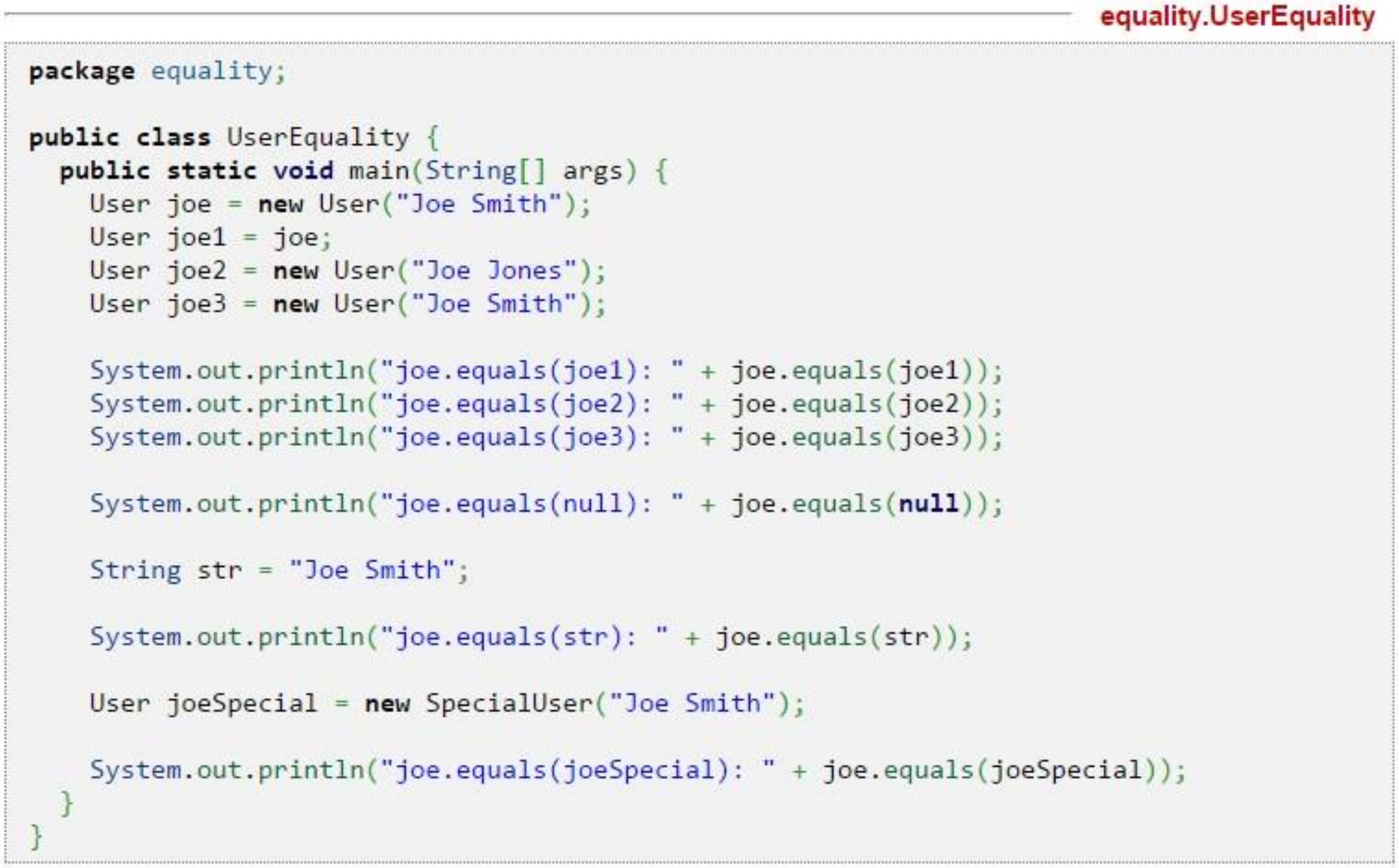


equality.SpecialUser

```
package equality;

public class SpecialUser extends User {
  public SpecialUser(String name) { super(name); }
}
```

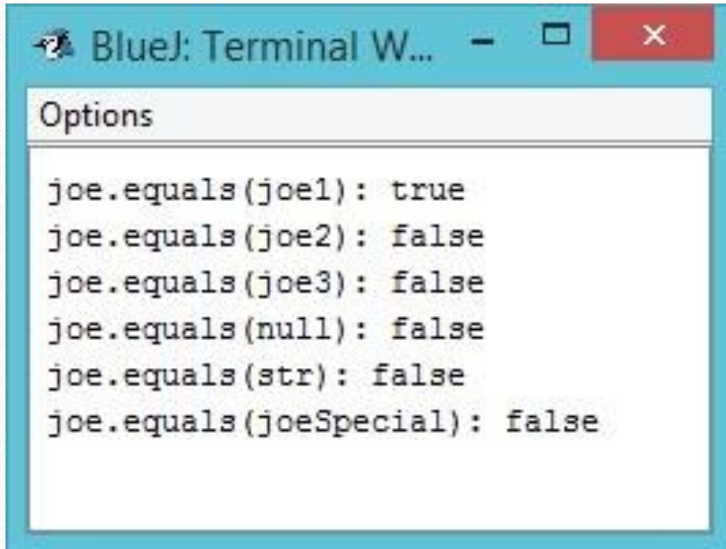
Custom equals() Methods:

We want to override equals so that it is based on name content. Here is a test program:



equality.UserEquality

```java
package equality;

public class UserEquality {
  public static void main(String[] args) {
    User joe = new User("Joe Smith");
    User joe1 = joe;
    User joe2 = new User("Joe Jones");
    User joe3 = new User("Joe Smith");

    System.out.println("joe.equals(joe1): " + joe.equals(joe1));
    System.out.println("joe.equals(joe2): " + joe.equals(joe2));
    System.out.println("joe.equals(joe3): " + joe.equals(joe3));

    System.out.println("joe.equals(null): " + joe.equals(null));

    String str = "Joe Smith";

    System.out.println("joe.equals(str): " + joe.equals(str));

    User joeSpecial = new SpecialUser("Joe Smith");

    System.out.println("joe.equals(joeSpecial): " + joe.equals(joeSpecial));
  }
}
```



```
joe.equals(joe1): true
joe.equals(joe2): false
joe.equals(joe3): false
joe.equals(null): false
joe.equals(str): false
joe.equals(joeSpecial): false
```

Custom equals() Methods:

Running this gives you what you want, except for:

```
joe.equals(joe3)
```

which gives **false** and we want it to be **true** since the two User objects which have identical name fields. We need to override:

```
@Override
public boolean equals(Object obj) {
    ...
}
```

Custom equals() Methods:

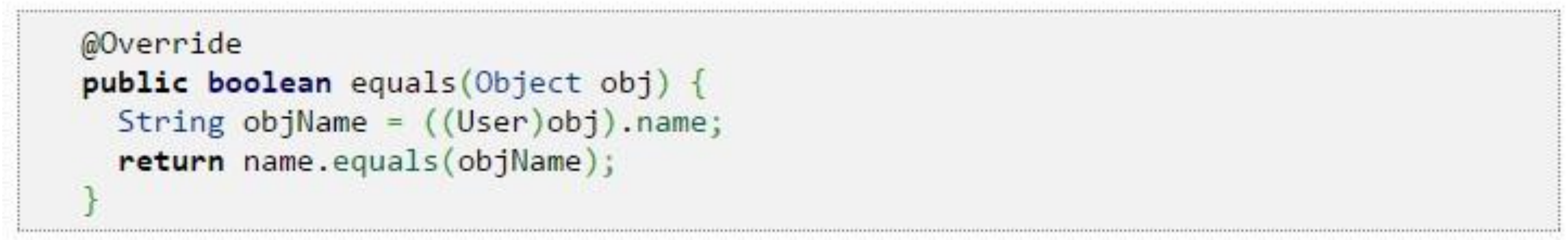
add to the User class

Here is a sequence of steps toward the solution:

1.Initially we might think that we only need the statement:

```
return name.equals(obj.name);
```

However, this won't compile because Object has no name member. We really mean to cast obj to a User which does have a name member.



```
@Override
public boolean equals(Object obj) {
  String objName = ((User)obj).name;
  return name.equals(objName);
}
```

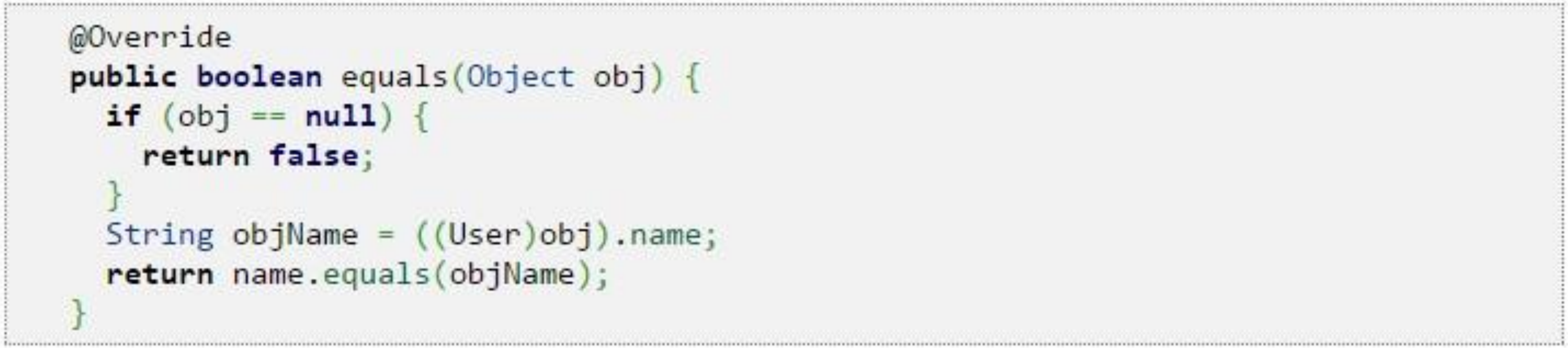
If you run it with this addition, we get the desired match of joe and joe3.

Custom equals() Methods:

2. The first problem appears from testing against null:

```
joe.equals(null)
```

We never want null to equal to a non-null object, so filter it out:



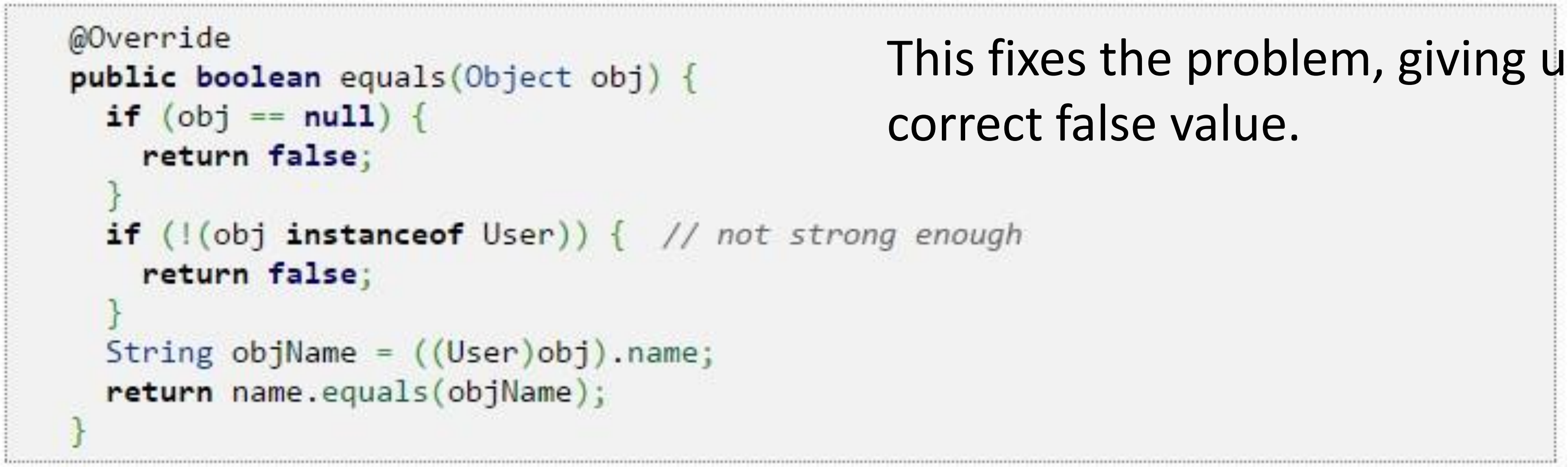
```
@Override
public boolean equals(Object obj) {
  if (obj == null) {
    return false;
  }
  String objName = ((User)obj).name;
  return name.equals(objName);
}
```

Custom equals() Methods:

3. The next problem is that obj may not be a User, seen initially as a class-cast exception using a String:

```
joe.equals(str)
```

Initially we employ the instanceof operator like this:



```
@Override
public boolean equals(Object obj) {
  if (obj == null) {
    return false;
  }
  if (!(obj instanceof User)) {  // not strong enough
    return false;
  }
  String objName = ((User)obj).name;
  return name.equals(objName);
}
```

This fixes the problem, giving us the correct false value.

Custom equals() Methods:

4. The last problem is that we get a true value for the test:

```
joe.equals(joeSpecial)
```

The reason this happens is because instanceof becomes true for the subclass object, joeSpecial of type SpecialUser when matched against the superclass, User, in the expression:

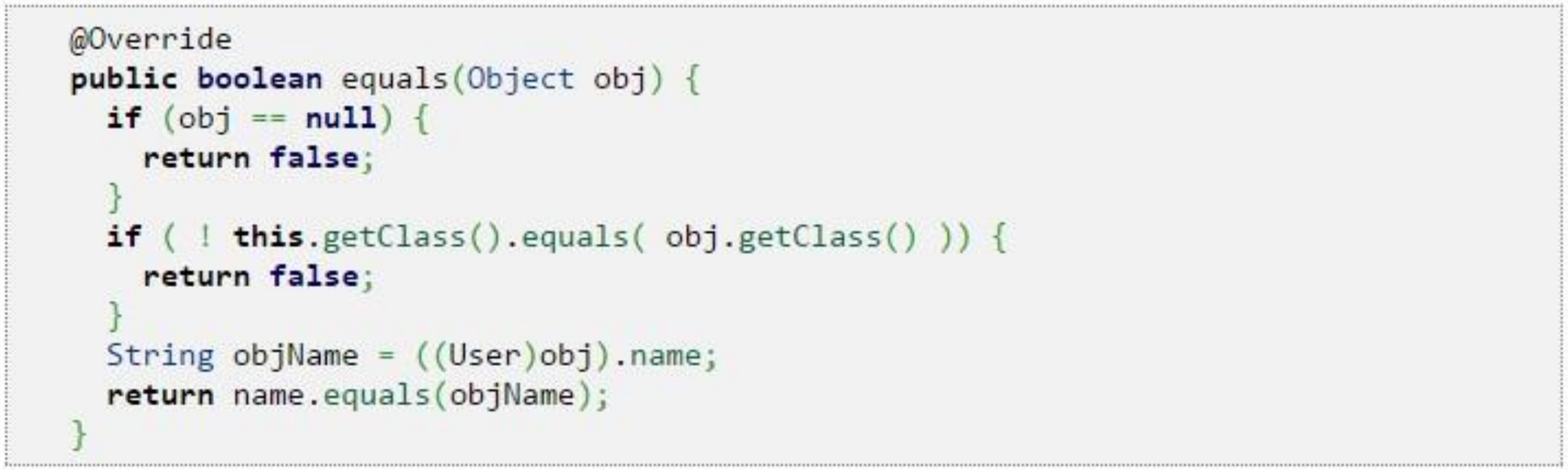
```
joeSpecial instanceof User
```

But we want an exact class match. So the solution is to employ the following member function to compute the class:

```
Class getClass()
```

Custom equals() Methods:

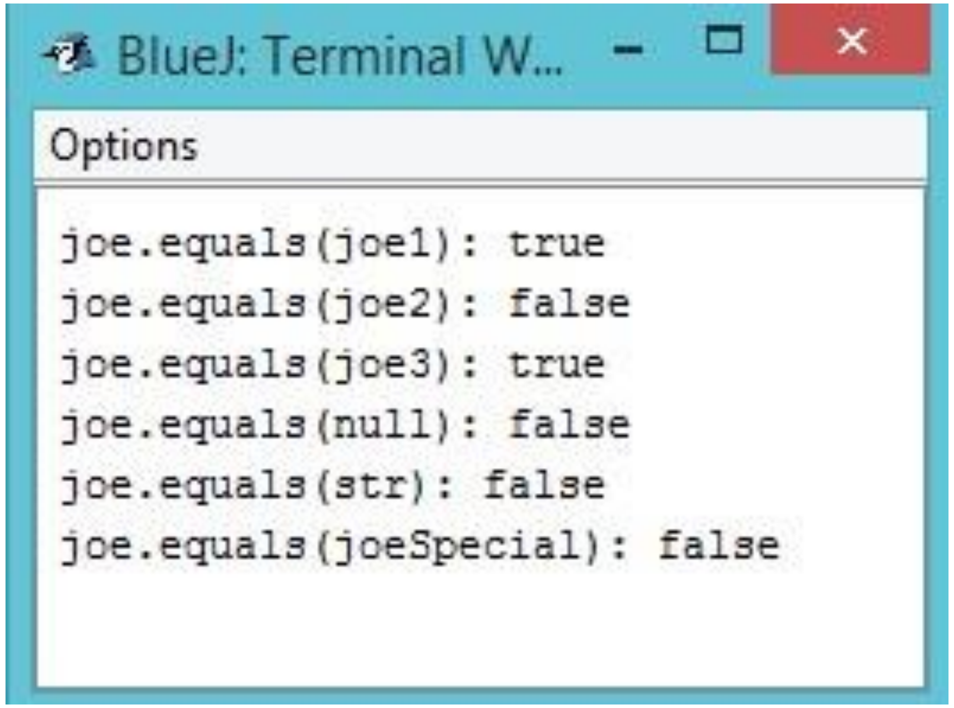
A Class object is just a Java object representing the full package path to the class in question. So our final version is this:



```java
@Override
public boolean equals(Object obj) {
  if (obj == null) {
    return false;
  }
  if ( ! this.getClass().equals( obj.getClass() )) {
    return false;
  }
  String objName = ((User)obj).name;
  return name.equals(objName);
}
```

Test the effectiveness by adding the equals member function to the User, modifying it according each step and running the UserEquality test program.

Expected Results:



```
joe.equals(joe1): true
joe.equals(joe2): false
joe.equals(joe3): true
joe.equals(null): false
joe.equals(str): false
joe.equals(joeSpecial): false
```

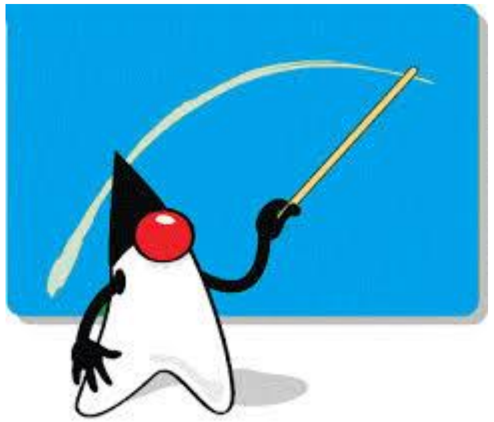
Play around other equals() definition using **==, instanceof and getClass(), getClass().getName()** to check for other equality definitions.

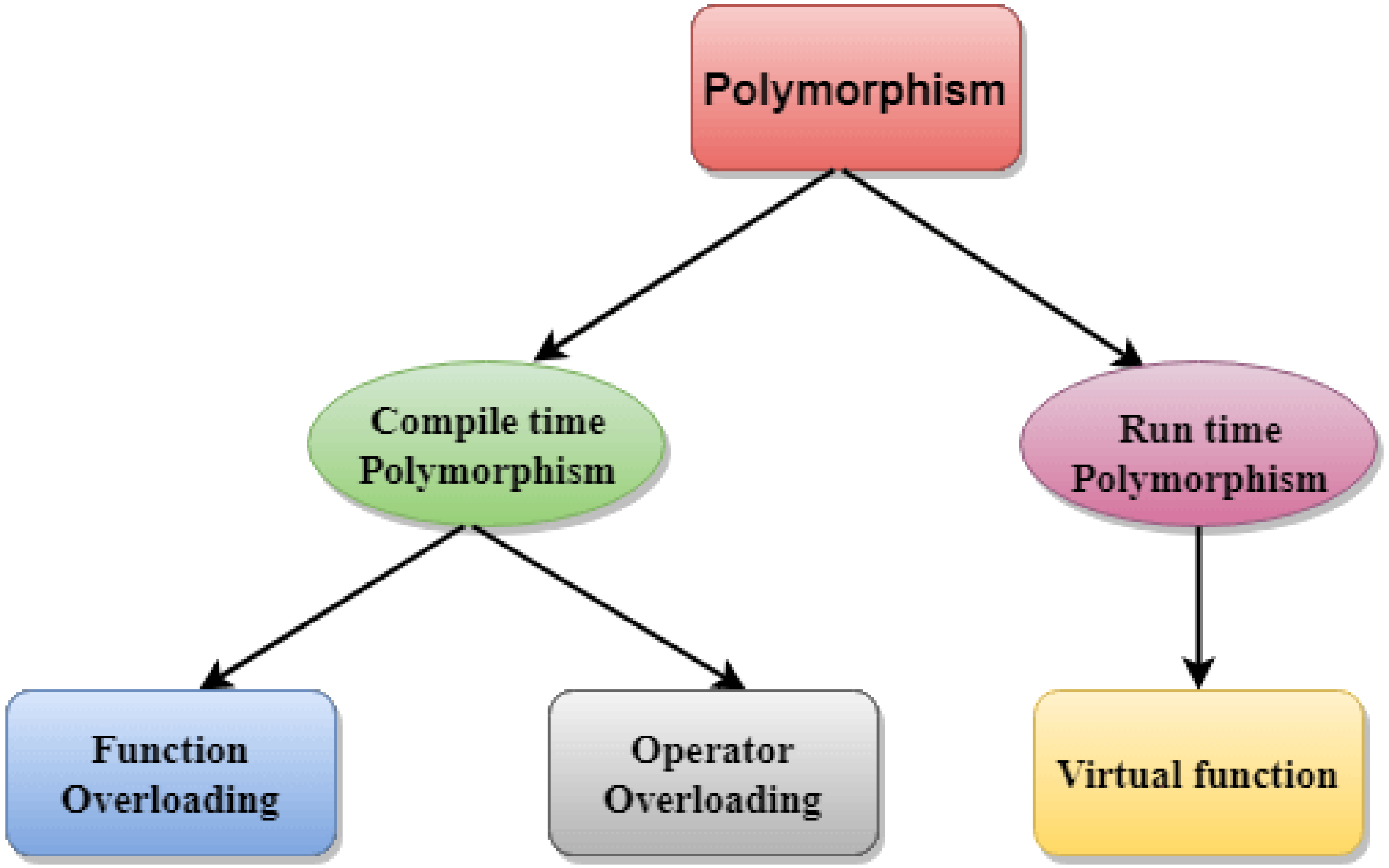
(1) Same name (this one)
(2) Same class
(3) Same memory address (default definition)
(4) Same superclass
(5) Name equals to a string

Dynamic Binding

LECTURE 1

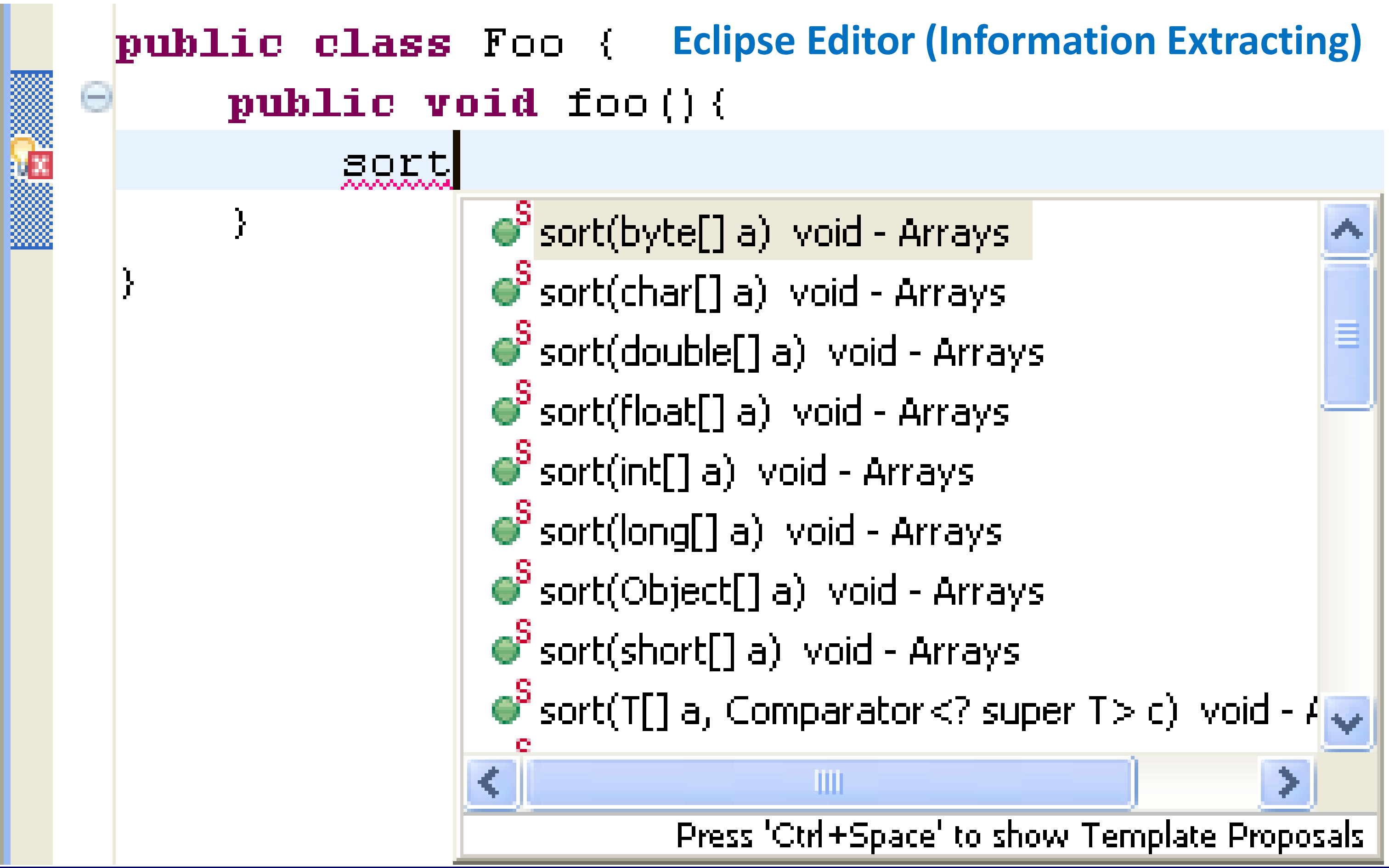
Key Topics in Polymorphism





Method Signature Matching

Scope, Visibility, Method Signature



Eclipse Editor (Information Extracting)

```
public class Foo {
    public void foo(){
        sort
    }
}
```

```
sort(byte[] a)  void - Arrays
sort(char[] a)  void - Arrays
sort(double[] a)  void - Arrays
sort(float[] a)  void - Arrays
sort(int[] a)  void - Arrays
sort(long[] a)  void - Arrays
sort(Object[] a)  void - Arrays
sort(short[] a)  void - Arrays
sort(T[] a, Comparator<? super T> c)  void - A
Press 'Ctrl+Space' to show Template Proposals
```

Scope and Visibility check if a method can be called based on where the method is located.

Method Signature matching decides which method to use.

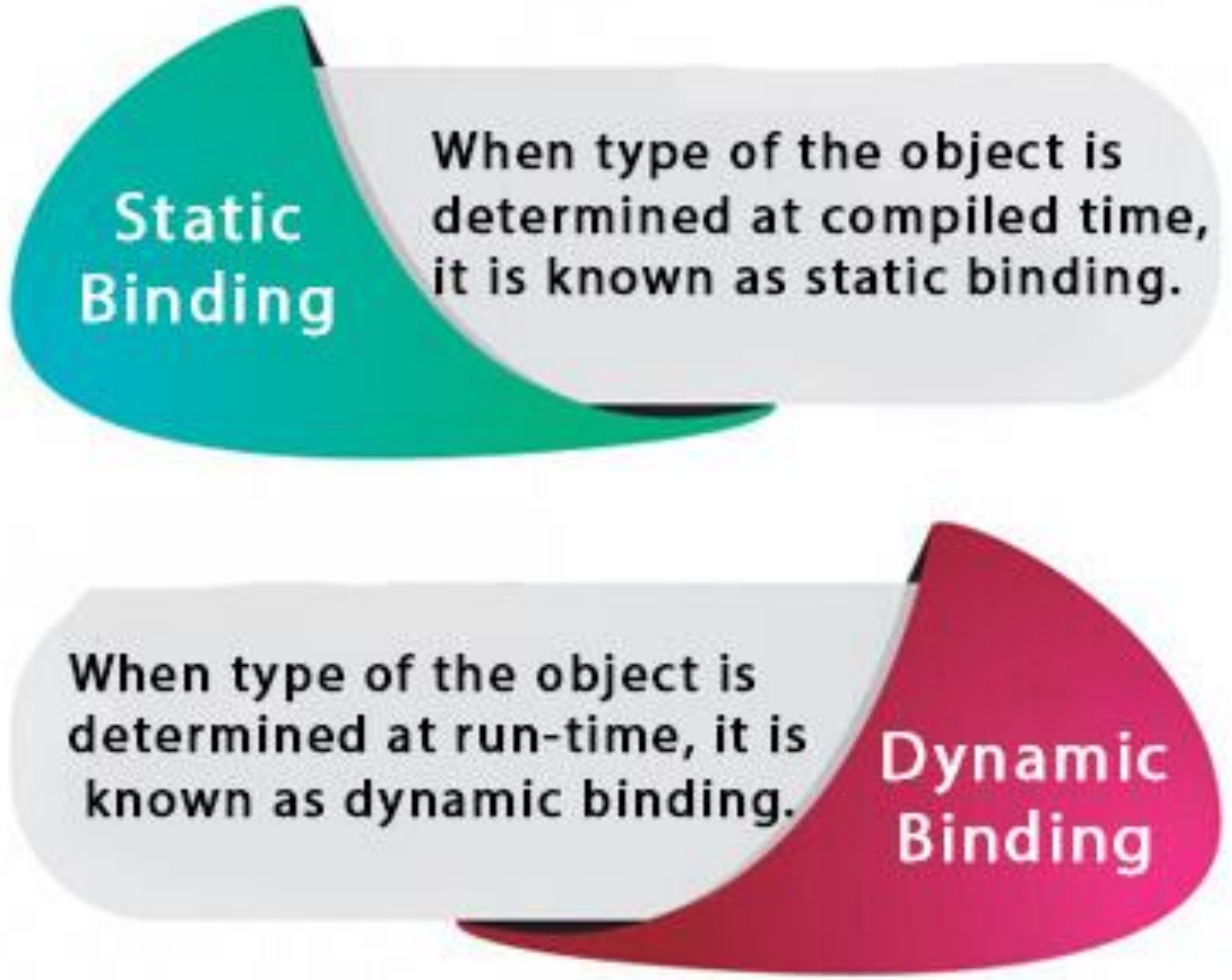
These implements overloading and overriding.

Method Matching vs. Binding

matching(overload)/binding(overriding)

- Matching a method signature and binding a method implementation are two issues. The **compiler** finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.
- A method may be implemented in several subclasses. The **Java Virtual Machine** dynamically **binds** the implementation of the method at runtime.

Static vs Dynamic Binding



Static Binding

When type of the object is determined at compiled time, it is known as static binding.

Dynamic Binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
public class NewClass{
  public static class superclass{
    void print(){
      System.out.println("print in superclass."); }
  }
  public static class subclass extends superclass {
    void print() {
      System.out.println("print in subclass."); }
  }
public static void main(String[] args) {
    superclass A = new superclass();
    superclass B = new subclass();
    A.print();
    B.print();
  }
}
```

Non-Static Methods

Output is:

```
print in superclass.
print in subclass.
```

Polymorphism, Dynamic Binding and Generic Programming (.class (Java byte code), .dll (C/C++))

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

- An object of a subtype can be used wherever its supertype value is required. This feature is known as ***polymorphism***.
- When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as ***dynamic binding***.

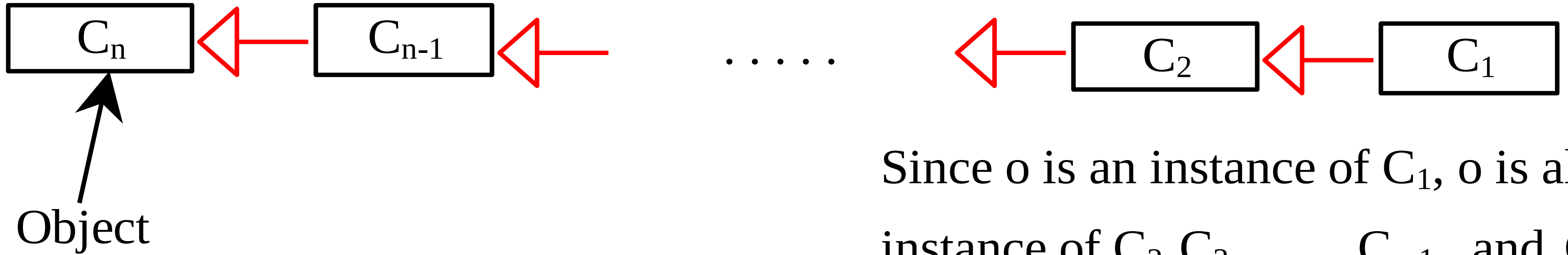
Demonstration Program

DYNAMICBINDINGDEMO.JAVA

Dynamic Binding

- Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class.
- In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in **C_1, C_2, ..., C_{n-1} and C_n, in this order**, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.

Dynamic Binding

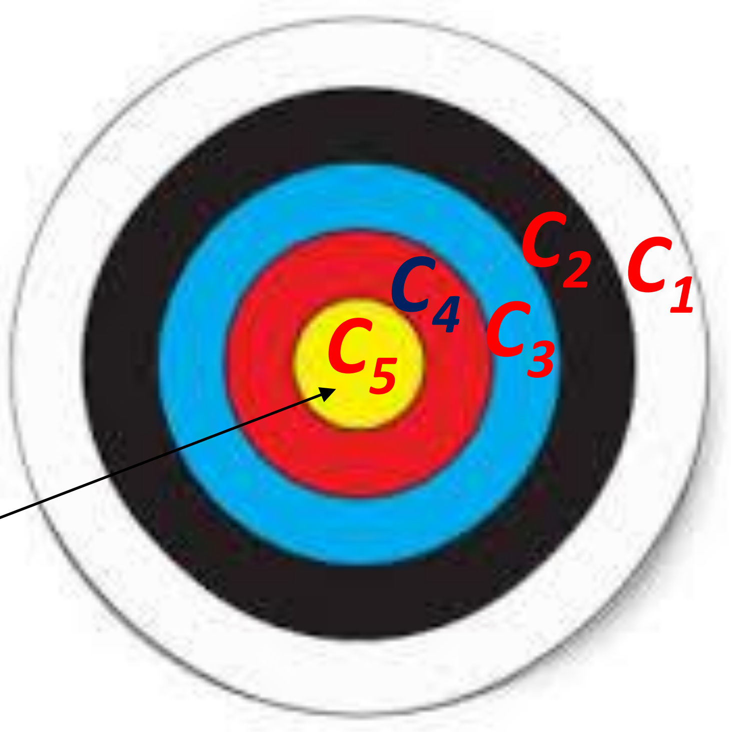


Since o is an instance of C_1, o is also an instance of $C_{2,}$ $C_{3,}$ **. . . ,** C_{n-1}**,** and C_n

Dynamic Binding

Finding the right method from lowest class (outer wrapper class)

Move up class hierarchy



Generic Programming

- Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as **generic programming**. If a method’s parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter’s subclasses (e.g., Student or String).
- When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

Generic programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}
class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Base class Constructor (people package)

LECTURE 1

Base Class Constructors

Used to Initiate Core Data Fields in Base Class

- To test the two programs in this first subsection, I would recommend creating a simple main class

  `NewMain.java`

- Copy the content show and add it within the default (i.e., no) package for simplicity.
- Assume the class relationship:

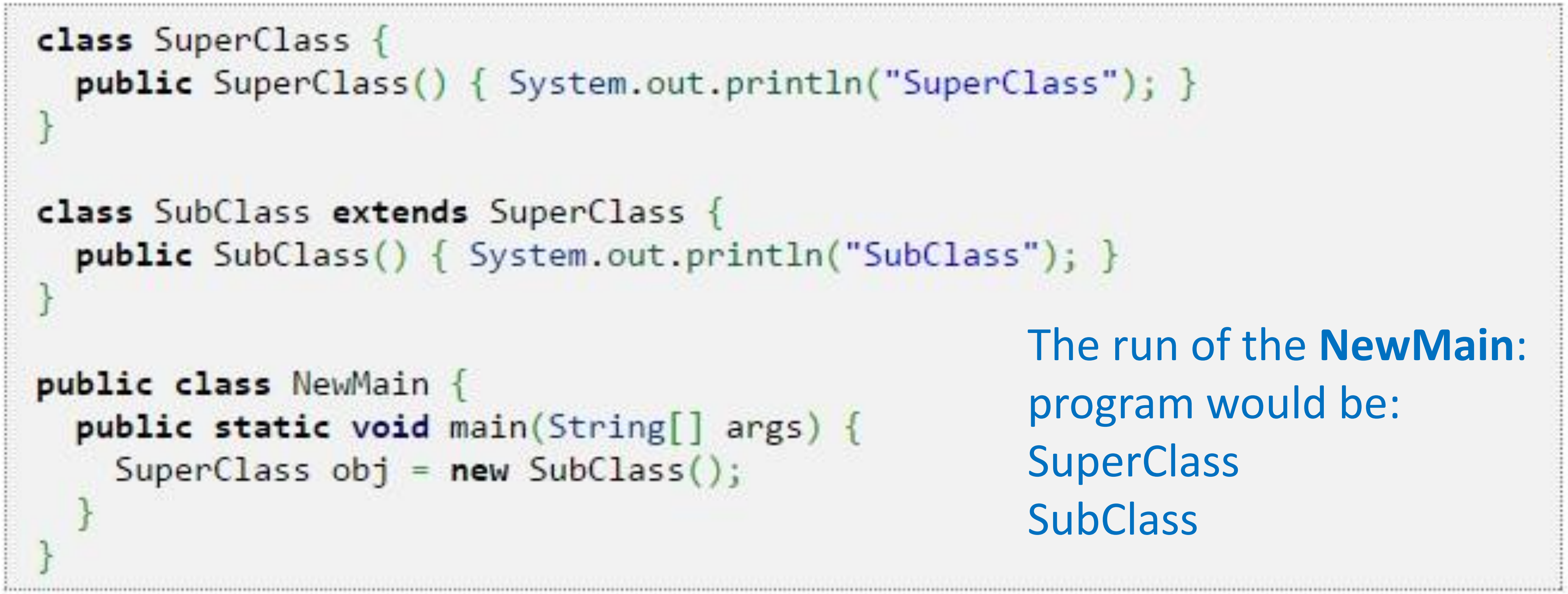
  `class SubClass extends SuperClass`

- Whenever a **SubClass** object is instantiated, the **SuperClass** portion of it must be **first instantiated** through a **SuperClass** constructor. Unless otherwise controlled, the no-argument **SuperClass** constructor will be called. If no constructors are defined, the default constructor is to "**do nothing**."

Base Class Constructors

NewMain.java

Start with this example:



```
class SuperClass {
  public SuperClass() { System.out.println("SuperClass"); }
}

class SubClass extends SuperClass {
  public SubClass() { System.out.println("SubClass"); }
}

public class NewMain {
  public static void main(String[] args) {
    SuperClass obj = new SubClass();
  }
}
```

The run of the **NewMain**: program would be:

SuperClass

SubClass

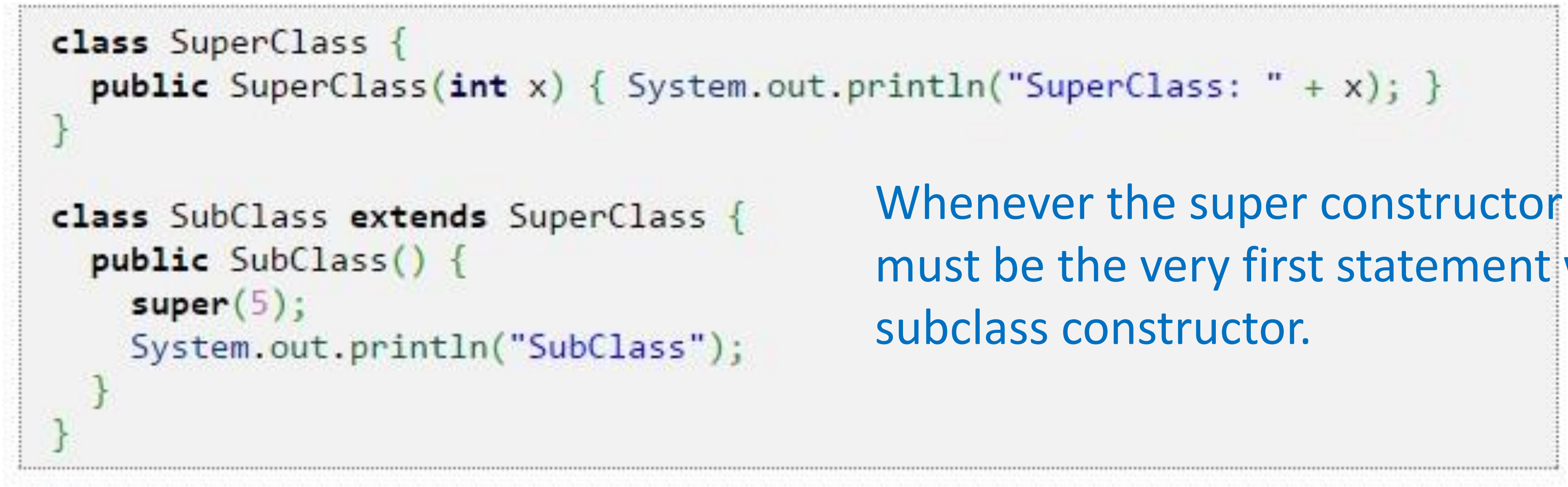
Base Class Constructors

NewMain2.java

- If there is some constructor defined, but not the no-argument constructor, then the SubClass must explicitly call a base class constructor using the Java super keyword. For example, the following code variation would not compile without the added **"super(5)"** statement:

Base Class Constructors

NewMain2.java



```
class SuperClass {
  public SuperClass(int x) { System.out.println("SuperClass: " + x); }
}

class SubClass extends SuperClass {
  public SubClass() {
    super(5);
    System.out.println("SubClass");
  }
}
```

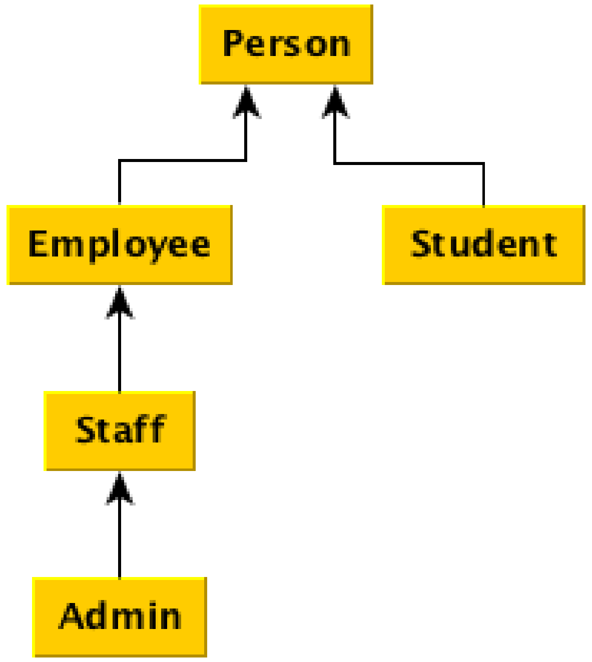
Whenever the super constructor is used, it must be the very first statement within the subclass constructor.

Demonstration Program

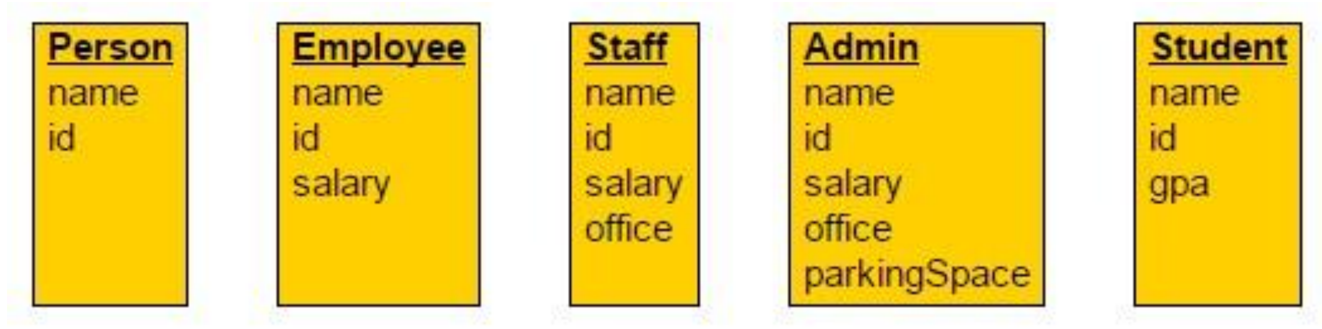
BASE PACKAGE: NEWMAIN.JAVA
NEWMAIN2.JAVA

Demo Program: people package

We'll look at the demo classes in the people package. The UML diagram is:

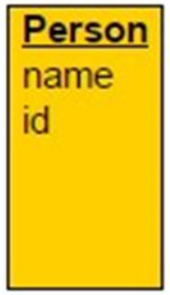


This is the first time that we are using data members within the classes. The data members of the base class are automatically part of the derived class. In this case the data presentation of the objects in this hierarchy are this:



Even though the derived objects have access to the base class data, they cannot use it directly because it is declared as private.

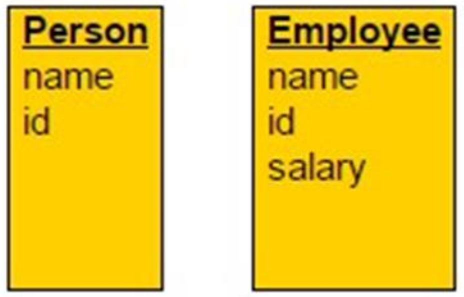
Person

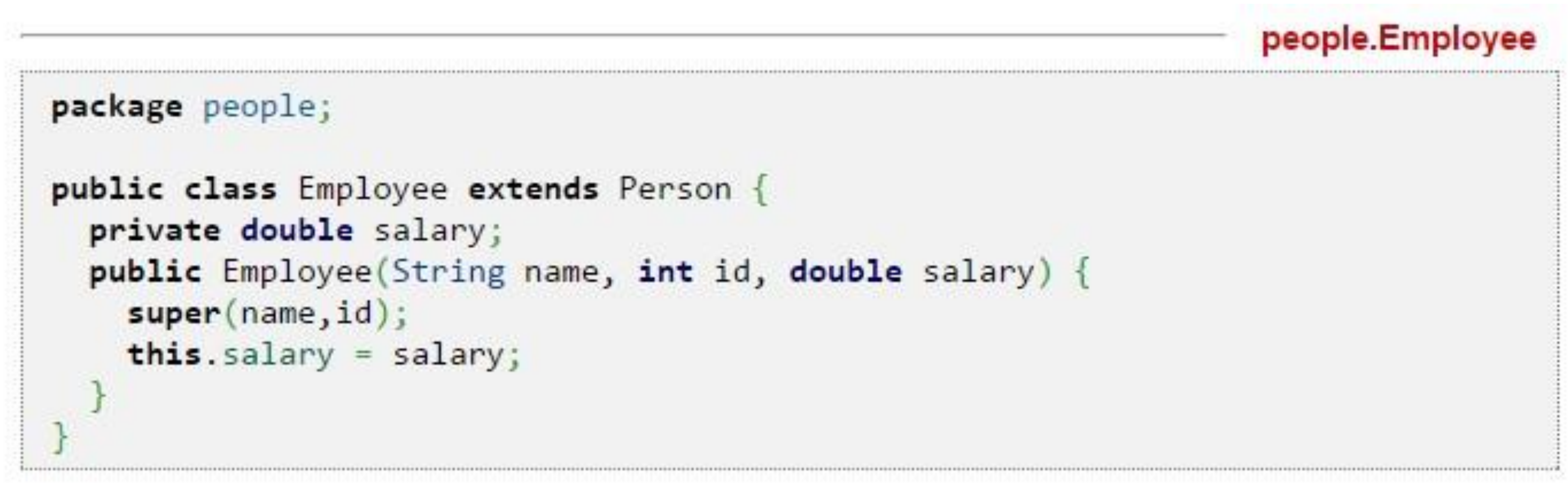


people.Person

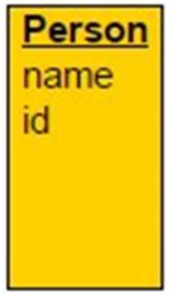
```java
package people;

public class Person {
  private String name;
  private int id;
  public Person(String name, int id) {
    this.name = name;
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public int getId() {
    return id;
  }
  @Override
  public String toString() {
    return String.format("Person(%s,%s)", name, id);
  }
}
```

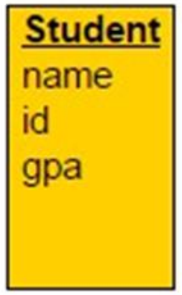
Employee



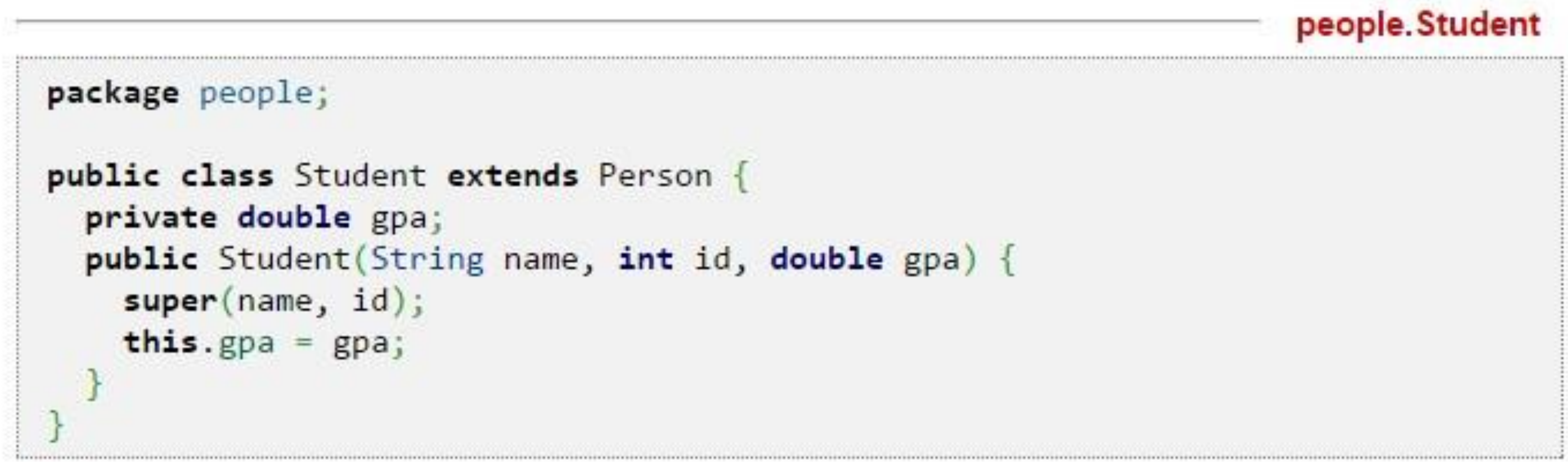

people.Employee

```
package people;

public class Employee extends Person {
  private double salary;
  public Employee(String name, int id, double salary) {
    super(name,id);
    this.salary = salary;
  }
}
```



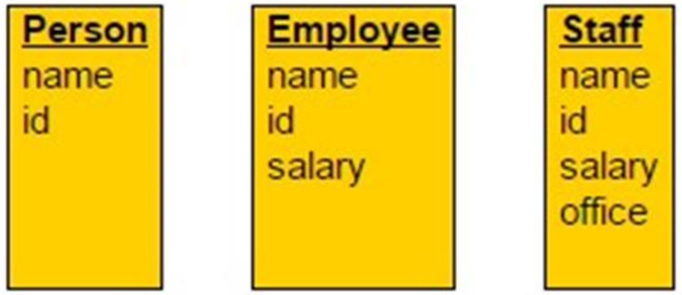


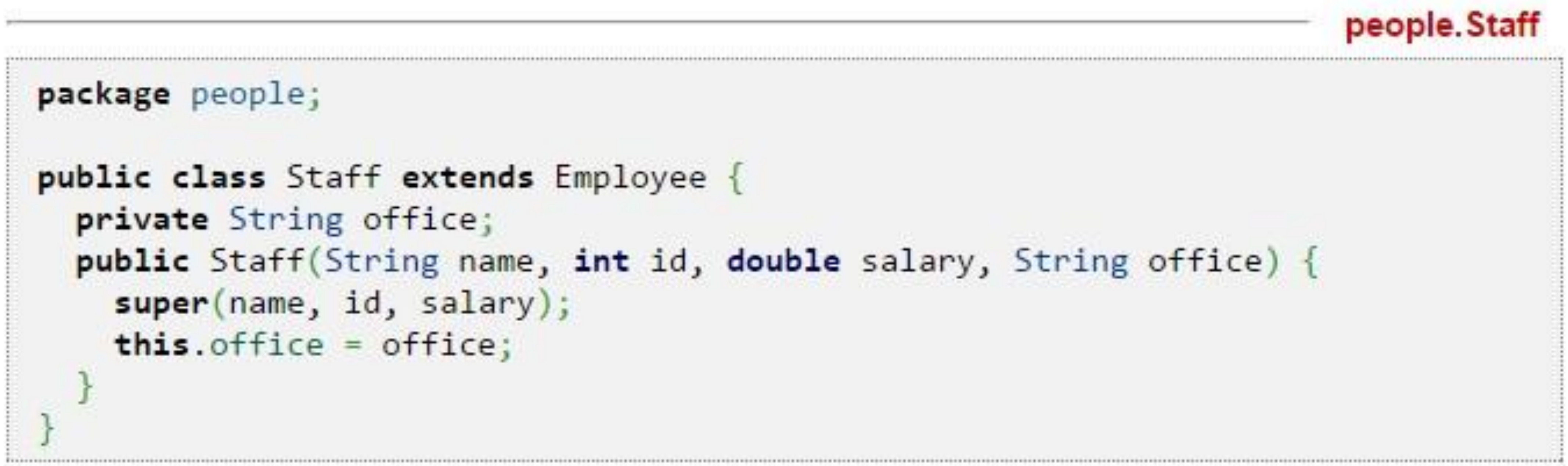
Student


people.Student

```java
package people;

public class Student extends Person {
  private double gpa;
  public Student(String name, int id, double gpa) {
    super(name, id);
    this.gpa = gpa;
  }
}
```

Staff

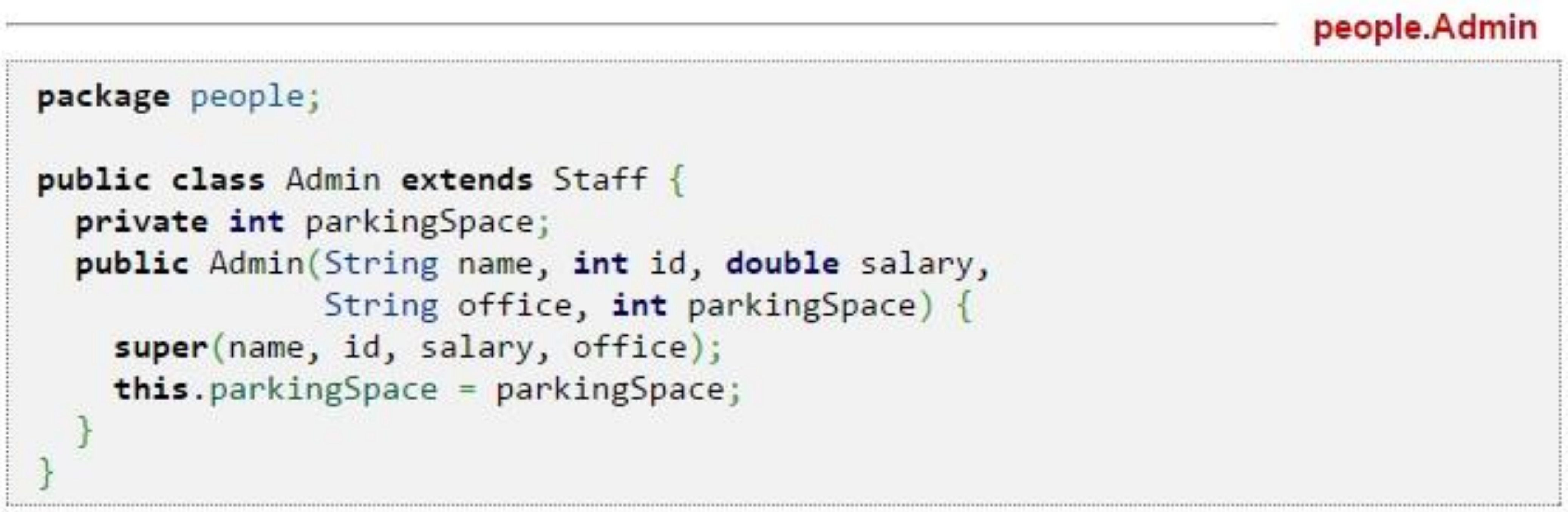



people.Staff

```
package people;

public class Staff extends Employee {
  private String office;
  public Staff(String name, int id, double salary, String office) {
    super(name, id, salary);
    this.office = office;
  }
}
```

Admin

Person	Employee	Staff	Admin
name	name	name	name
id	id	id	id
	salary	salary	salary
		office	office
			parkingSpace



people.Admin

```
package people;

public class Admin extends Staff {
  private int parkingSpace;
  public Admin(String name, int id, double salary,
               String office, int parkingSpace) {
    super(name, id, salary, office);
    this.parkingSpace = parkingSpace;
  }
}
```

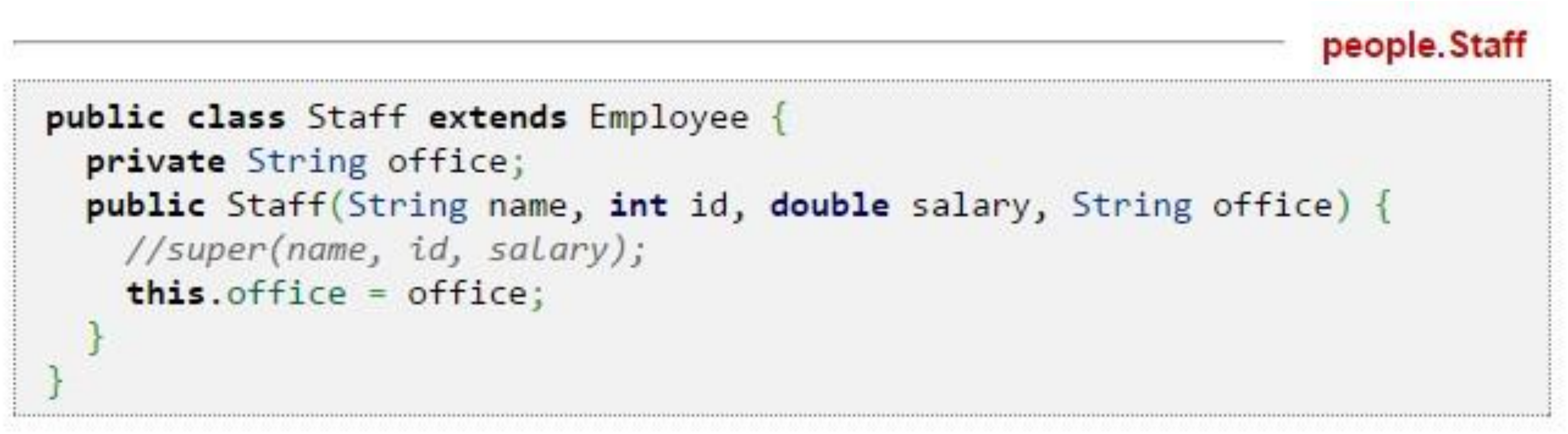
Initialization of Objects through the Class Hierarchy Using super()

- The no-argument constructor is not available in any class throughout this inheritance hierarchy, and so we are consistently forced to use the super constructor:

```
class SubClass extends ... {
   SubClass(...) {
      super( ... );
   }
}
```

Experiment 1: (Up to you, Try your best)

Edit Staff first and comment out the super statement and save:



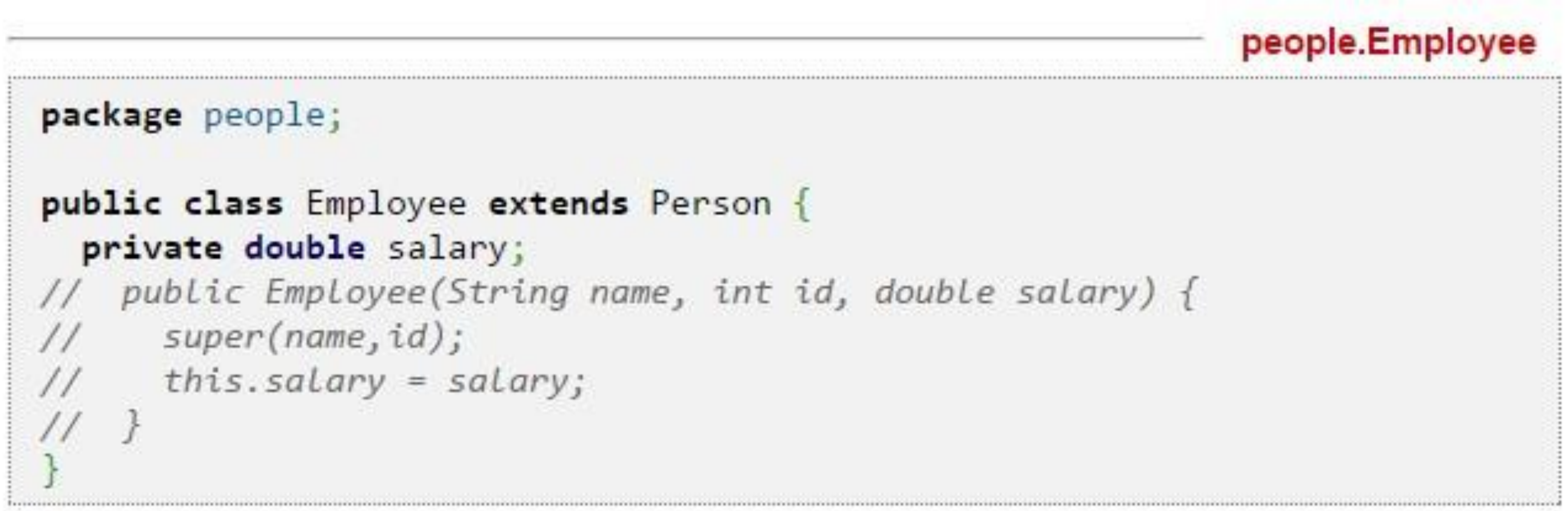
people.Staff

```
public class Staff extends Employee {
  private String office;
  public Staff(String name, int id, double salary, String office) {
    //super(name, id, salary);
    this.office = office;
  }
}
```

The error message given is saying that the no-argument constructor cannot be applied to Employee. The Admin subclass does not register any errors.

Experiment 2:

Now edit Employee, comment out the constructor and save:



people.Employee

```java
package people;

public class Employee extends Person {
  private double salary;
//  public Employee(String name, int id, double salary) {
//     super(name,id);
//     this.salary = salary;
//  }
}
```

You'll see an error of the same nature appear in Employee in that it cannot use the no-argument Person constructor. However, now the previous error in Staff disappears because there it can use the default no-argument constructor in Employee.

Finally undo the changes back to the original state.

The protected Data and Methods

people package: Non-AP Topic

LECTURE 1

The protected Modifier

- The **protected** modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public

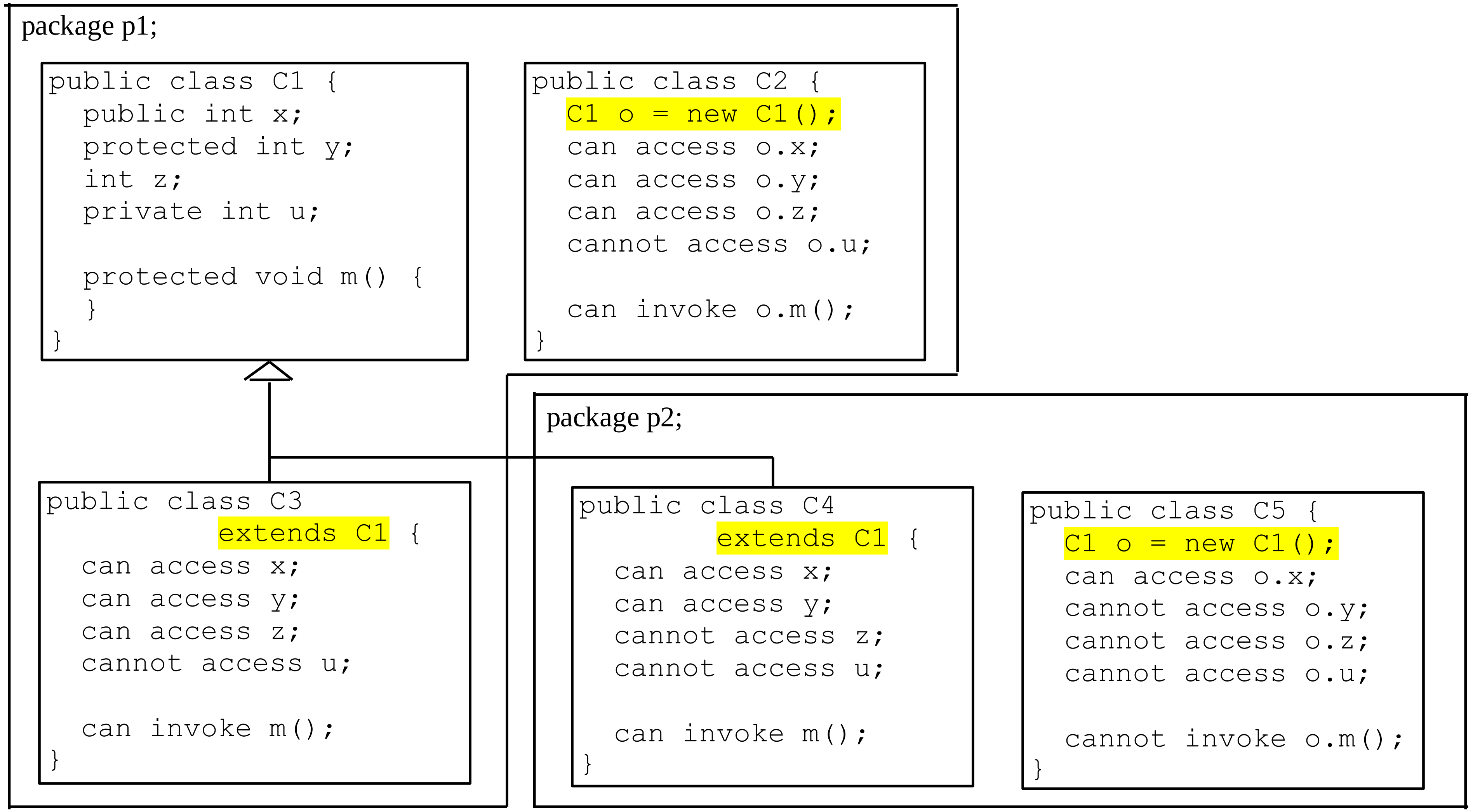
$$\xrightarrow{\texttt{Visibility increases}}$$

`private, none (if no modifier is used), protected, public`

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers



package p1;

```
public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;

  protected void m() {
  }
}
```

```
public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;

  can invoke o.m();
}
```

```
public class C3
          extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}
```

package p2;

```
public class C4
          extends C1 {
  can access x;
  can access y;
  cannot access z;
  cannot access u;

  can invoke m();
}
```

```
public class C5 {
  C1 o = new C1();
  can access o.x;
  cannot access o.y;
  cannot access o.z;
  cannot access o.u;

  cannot invoke o.m();
}
```

A Subclass Cannot Weaken the Accessibility

- A subclass may override a protected method in its superclass and change its visibility to public.
- However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

NOTE

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

The final Modifier

- The **final** class cannot be extended:

```
final class Math {
     ...
   }
```

- The final variable is a constant:

```
final static double PI = 3.14159;
```

- The final method cannot be overridden by its subclasses.

The protected access qualifier

Our current list of access qualifiers is this:

- public: most accessible
- *none*: accessible only within the package
- private: accessible only within class

The protected access qualifier

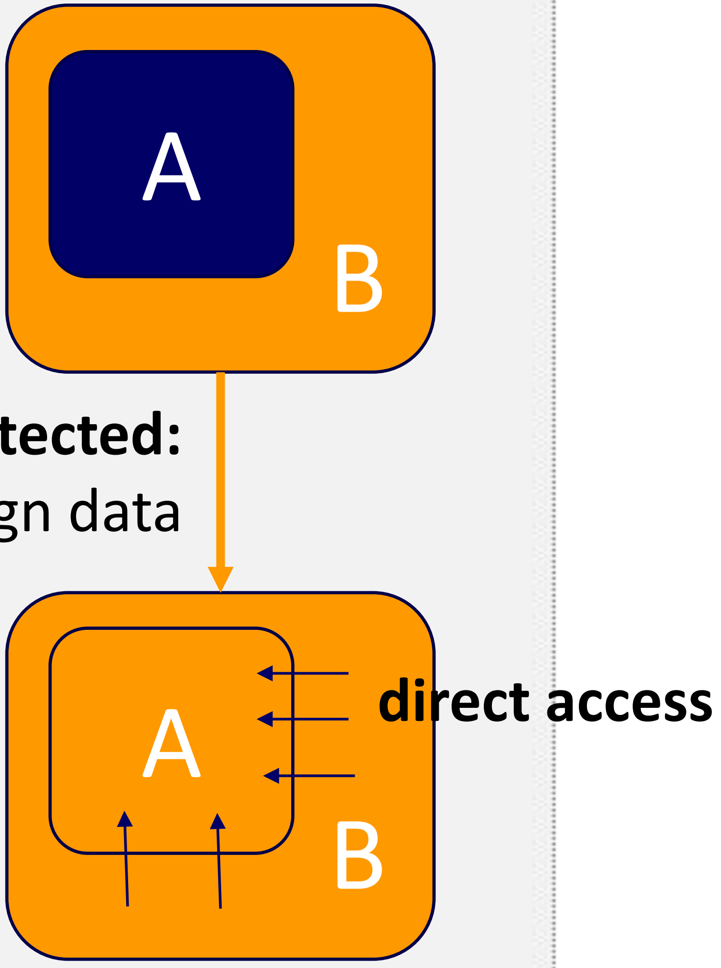
We're going to add the fourth and last qualifierprotected A protected member is one which is accessible outside the class itself only to derived classes, making the complete access hierarchy be:

- public: most accessible
- *none*: accessible only within the package
- protected: accessible to derived classes
- private: accessible only within class

Let's return to our people example by changing private fields to protected. This time, use the people1 package.

```
package people1;

public class Person {
  protected String name;
  protected int id;
  protected Person() { }

  // the rest is the same as before
  public Person(String name, int id) {
    this.name = name;
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public int getId() {
    return id;
  }
  @Override
  public String toString() {
    return String.format("Person(%s,%s)", name, id);
  }
}
```

change private to protected:
naturalization of foreign data



Demonstration Program

PEOPLE1 PACKAGE

Summary

LECTURE 1

Summary:

- Making the fields protected allows them to be **set** in constructor of **derived classes**. We still, however, have to deal with the need of derived constructors activating some base class constructor. In this example, we simply provide the no-argument, empty constructor in the superclass as:

```
        protected SuperClass() { }
```

- **Making it protected means it cannot be used outside the hierarchy to create objects with un-initialized fields.**