# AP Computer Science B

## Java Object-Oriented Programming [Ver. 3.0]

## Unit 4: Object-Oriented Programming


CHAPTER 12B: POLYMORPHISM

DR. ERIC CHOU IEEE SENIOR MEMBER

# Objectives

- Polymorphism: Polymorphic Methods, Polymorphic Containers, Polymorphic Iterators
- Dynamic Binding
- Types of polymorphism
  - Overloading/Overriding
  - Coercion Polymorphism
  - Inclusion Polymorphism
  - Parametric Polymorphism

# Objectives

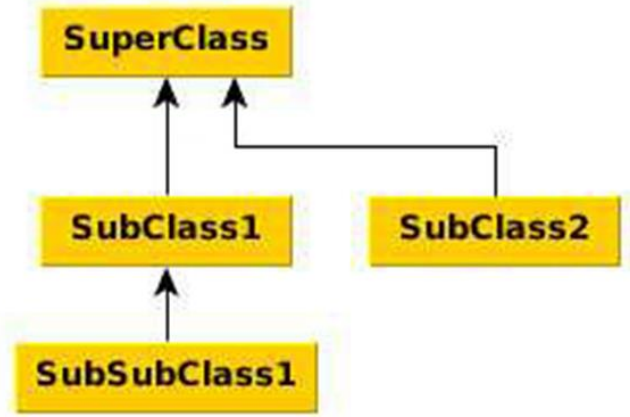
- Equality of reference, object, class, and group
- Polymorphism Example
  - Rectangle, Oval, GeometricObject

# Overview

LECTURE 1

# Polymorphism

- The word polymorphic means "having, assuming, or passing through many or various forms, stages, or the like." In Java object-oriented programming, it means that objects which appear the same by virtue of type behave differently by virtue of what the really are.
- We will rely on this as a running example whose UML diagram looks like this:



# Polymorphism

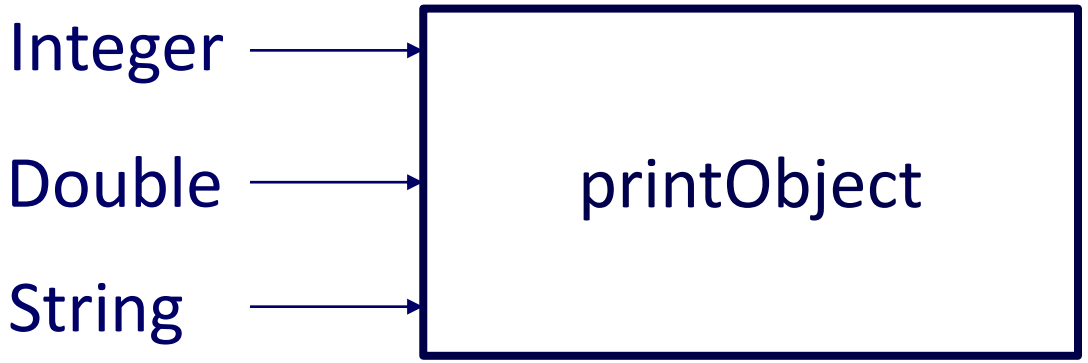
- Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Any Java object that can pass more than one **IS-A** test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the **IS-A** test for their own type and for the class Object.
- It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

# Polymorphism

- Polymorphic Methods
- Polymorphic Containers
- Polymorphic Iterators (Polymorphic Pointers)

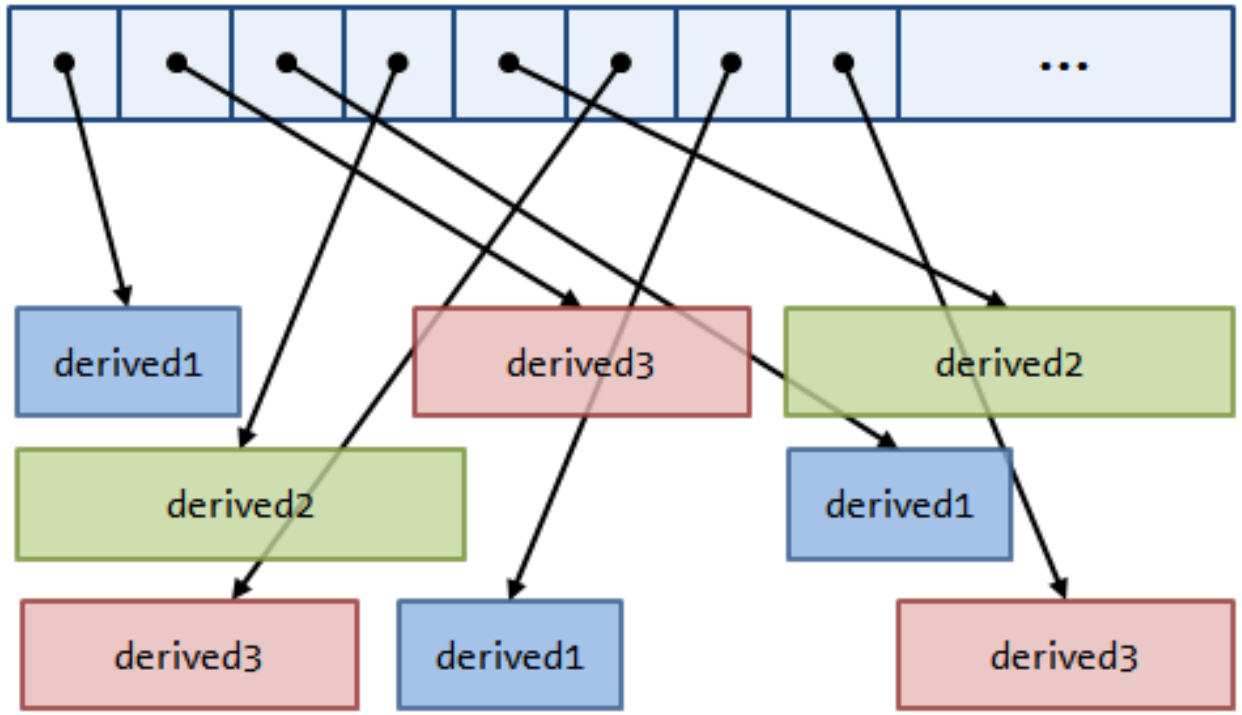
# Polymorphic Method

```
public static void printObject(Object obj){
    System.out.println(obj.toString());
}
```

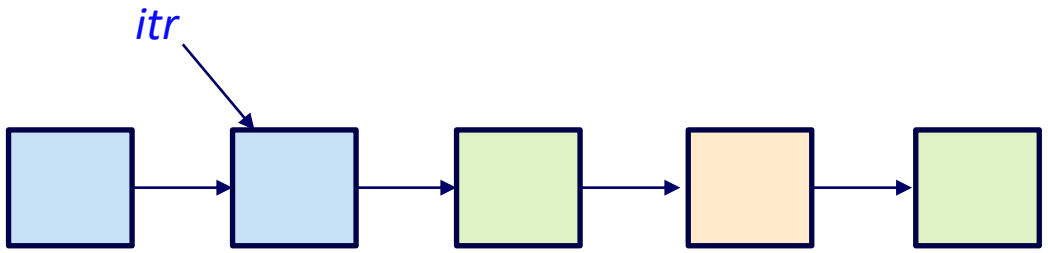


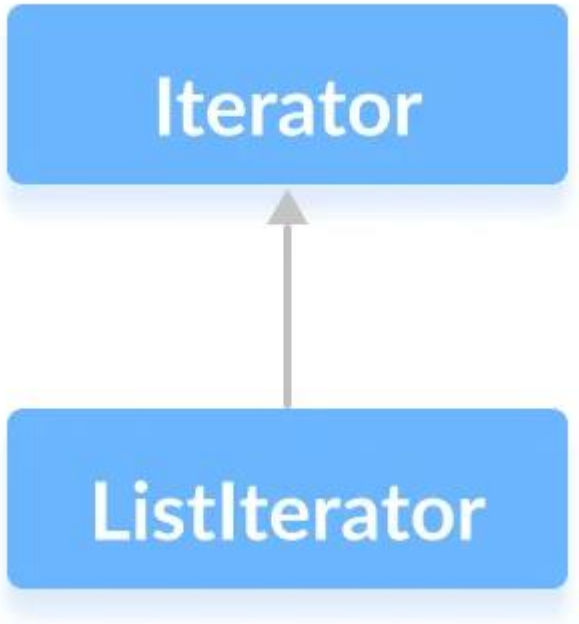
# Polymorphic Containers

```
ArrayList<Object> alist = new ArrayList<Object>();
```

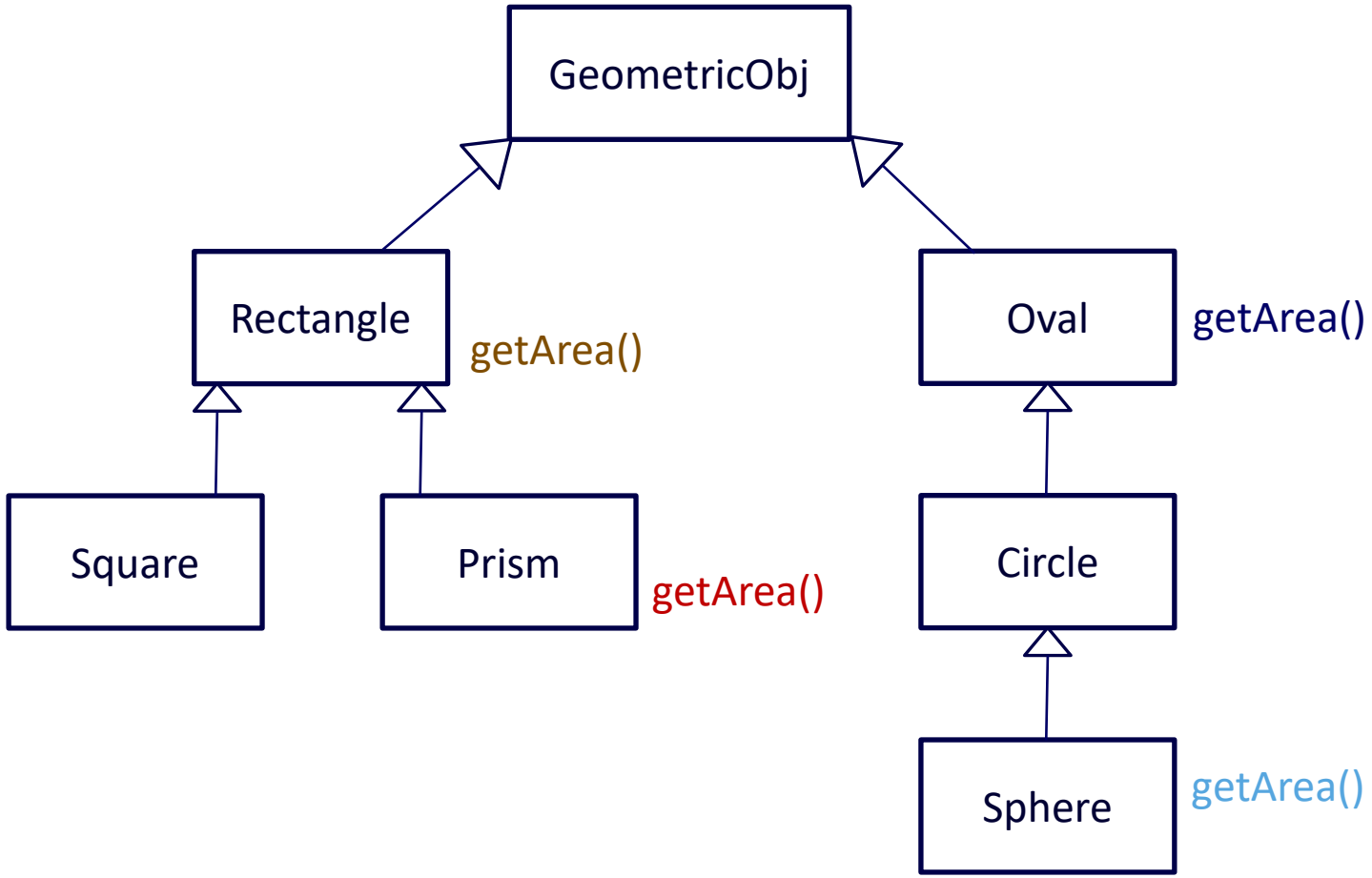


# Polymorphic Iterator



# Dynamic Binding

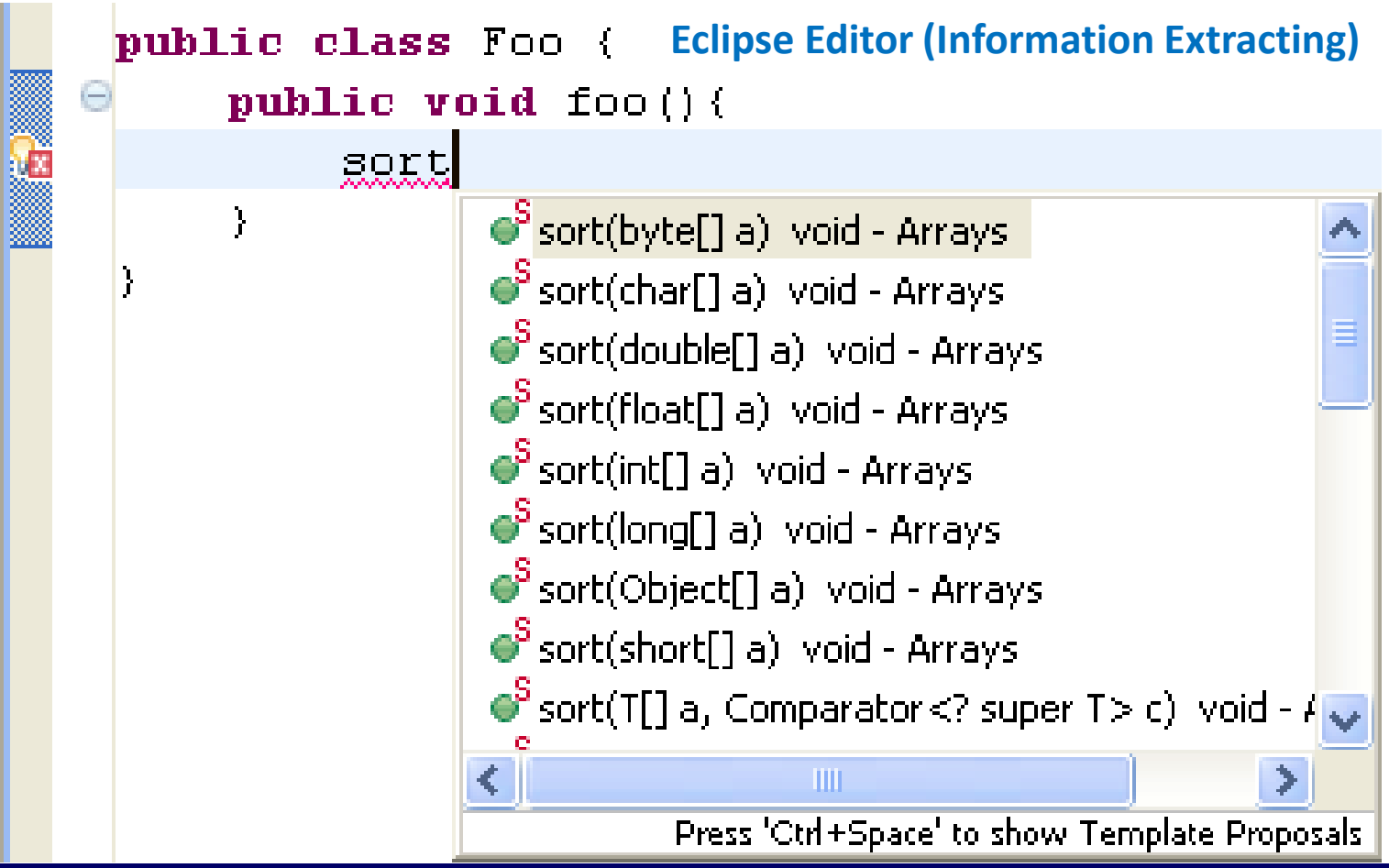


# Dynamic Binding

LECTURE 1

# Method Signature Matching

## Scope, Visibility, Method Signature




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.

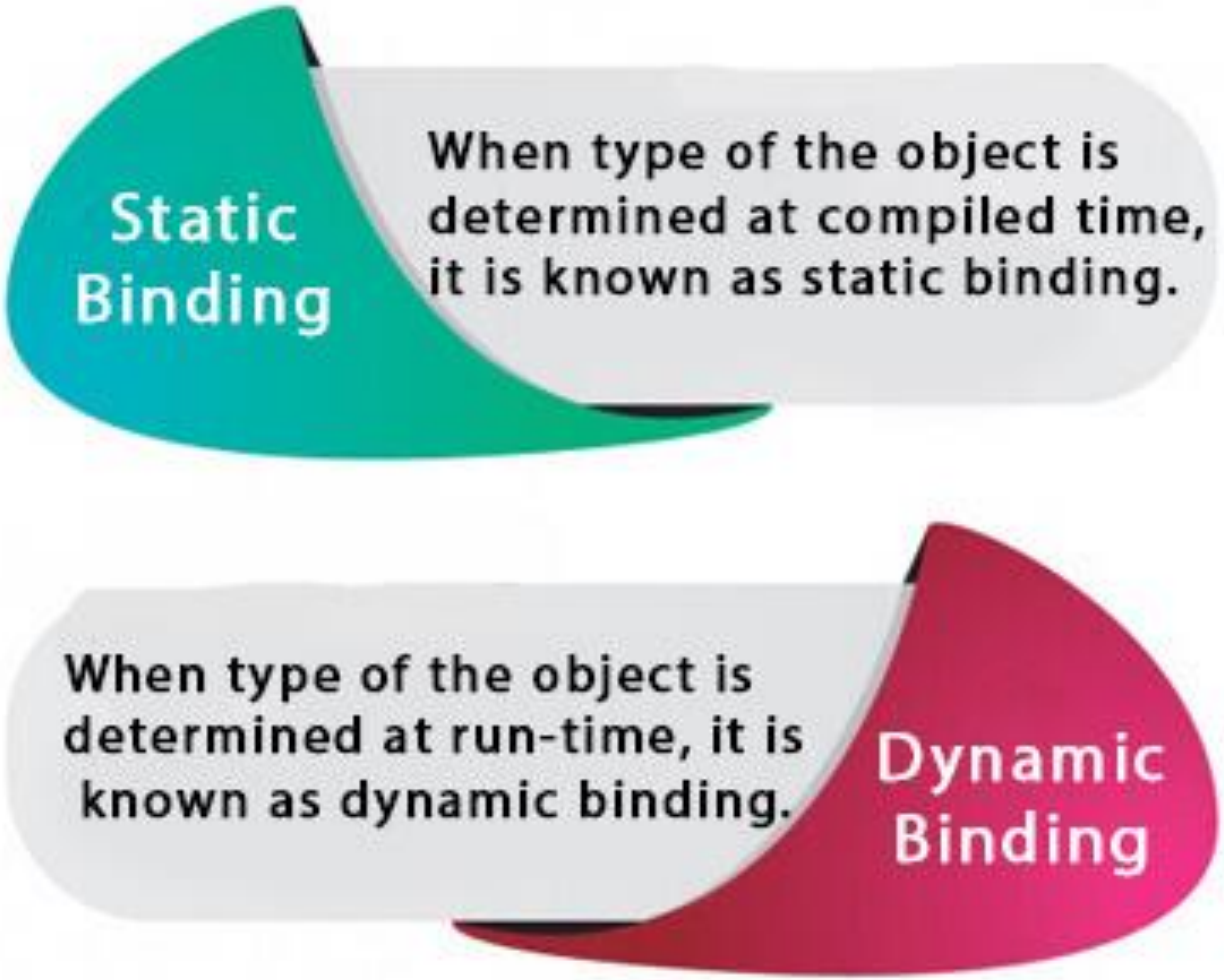
These implements overloading and overriding.

# 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



# 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:**

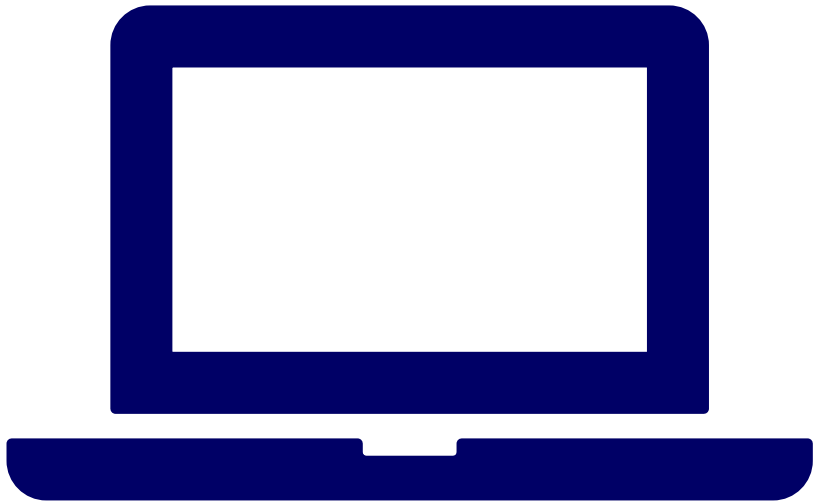
print in superclass.
print in subclass.

# 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***.



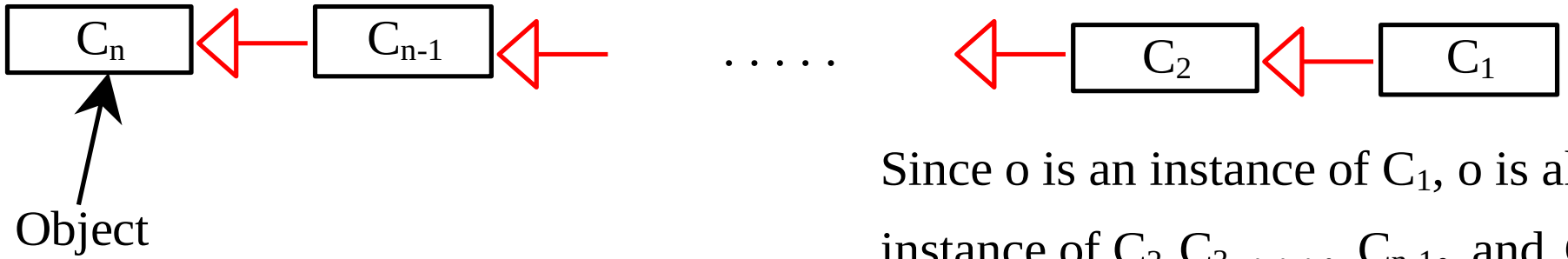
# Demonstration Program

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

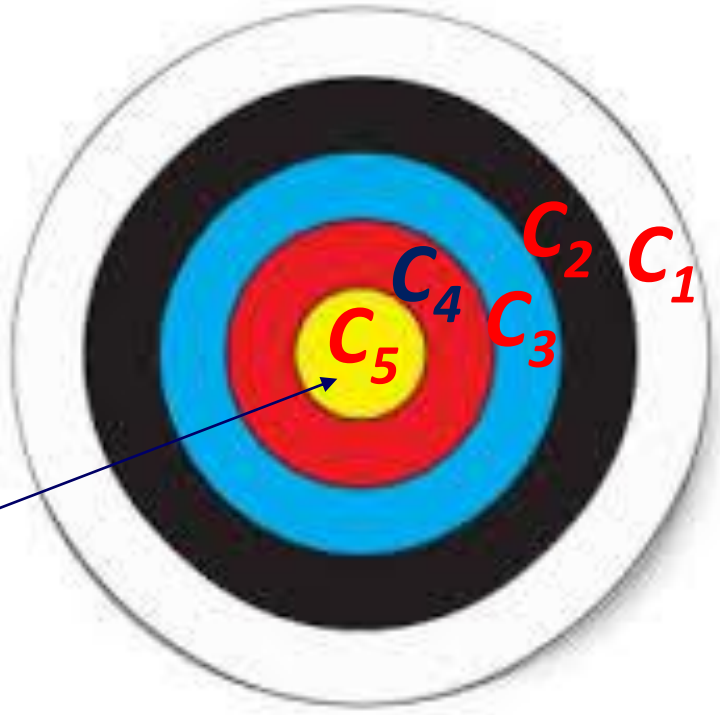



Since o is an instance of $C_1$, o is also an instance of $C_2$, $C_3$, . . ., $C_{n-1}$, and $C_n$

# Dynamic Binding

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

Move up class hierarchy



# Generic Programming

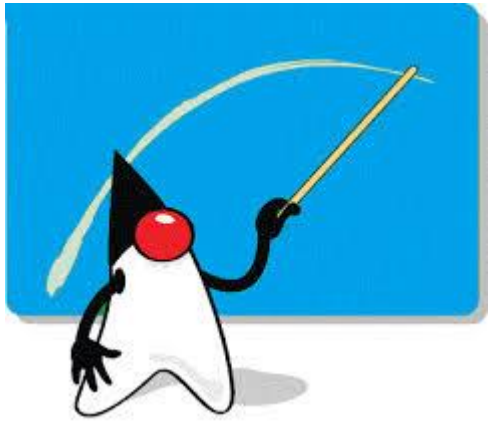
- Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as **generic programming**. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String).
- When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

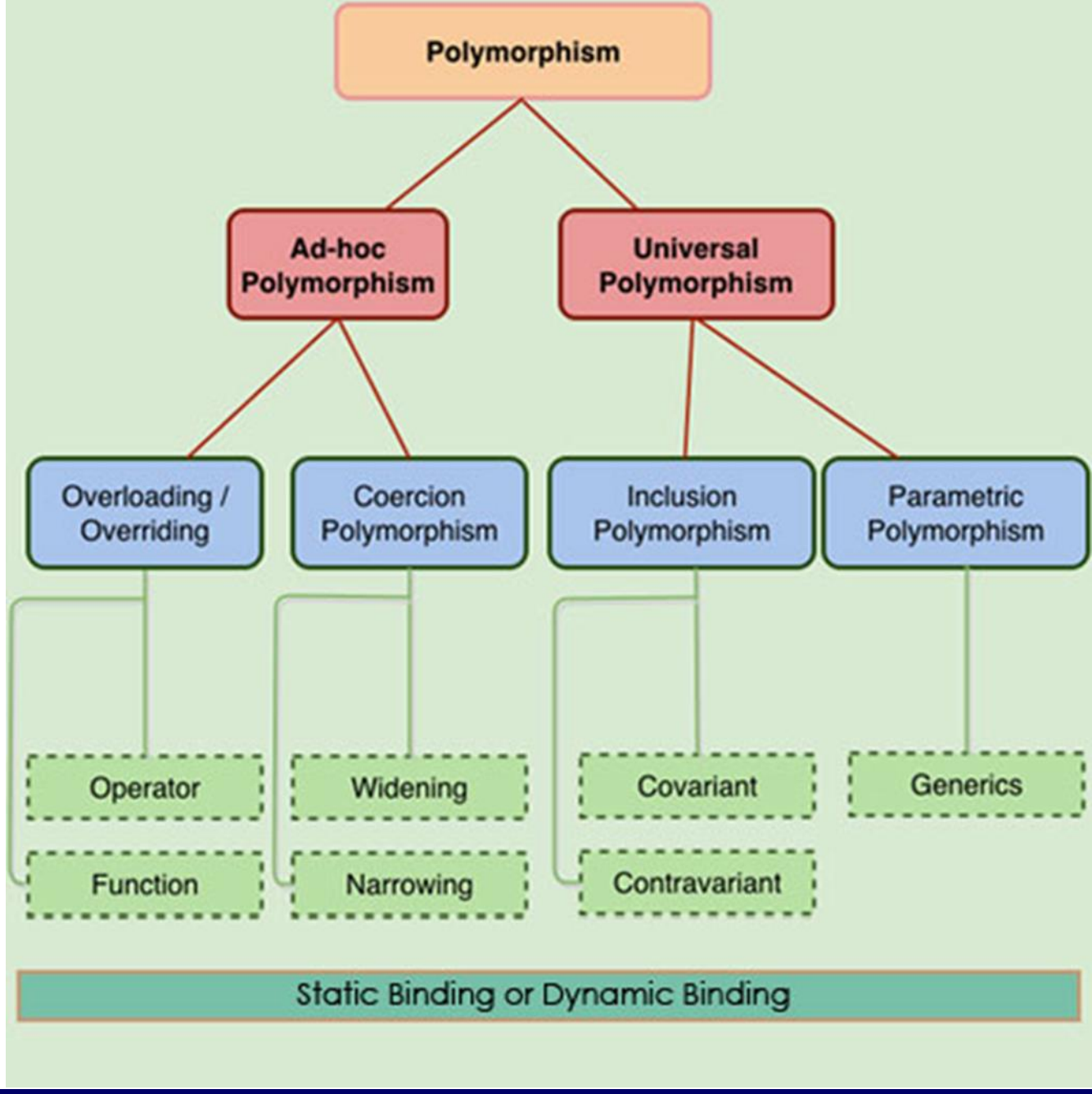
```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

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

# Types of Polymorphism

LECTURE 1



Key Topics in Polymorphism



# Overloading Overriding

LECTURE 1

# Overloading (Chapter 9)

## Two methods of a same name working on different data types

### Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name, Same parameter

### Overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name, Different Parameter

# Method Override

The pervasive method in each subclass overrides the version in the superclass. **NetBeans (Eclipse also)** strongly suggests using the annotation:

**@Override  // in the SubClass1 of basic package**

and it is a very good idea because doing so will help you avoid errors. For example, try this experiment: Edit SubClass1. Comment out the @Override line and simulate a "typing mistake" by changing the pervasive method name to **pervesive** (misspelled):

```
 //@Override
 public void pervesive() {
   System.out.println("SubClass1.pervasive");
 }
```

Re-run Driver and observe the output change in the second group:

====> calls to pervasive

---- obj in SuperClass calls: pervasive in SuperClass

---- obj in SubClass1 calls: pervasive in SuperClass

---- obj in SubClass2 calls: pervasive in SubClass2

---- obj in SubSubClass1 calls: pervasive in SubSubClass1

# Method Override

NetBeans detects the problem at compile time if you un-comment the @Override:

```
  @Override
  public void pervesive() {
    System.out.println("SubClass1.pervasive");
  }
```

The @Override line is now flagged with the error message: method does not override or implement a method from the supertype

Fix the "typing mistake" by changing the method back to its original name pervasive. Re-run Driver to confirm the fix.

# Disscussion for supercalls package

The output of the run is:

```
SubSubClass.foo
SubClass.foo
SuperClass.foo
```

Unlike the usage of super to construct the base class, the member function calls using "super." do not have to be the first statement. Also keep in mind that, without the "super." prefix, the foo() calls in derived objects would be infinitely recursive, e.g.:

```
class SubSubClass extends SubClass {
  @Override
  public void foo() {
    System.out.println("SubSubClass.foo");
    foo();
  }
}
```

The foo() call is the same as this.foo(), but because of the dynamic binding, casting this to a superclass type will not make any difference, i.e.,

**((SubClass)this).foo()** is no different than **foo()**

# Overriding Methods in the Superclass

- A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as ***method overriding***.

```
public class Circle extends GeometricObject {
  // Other methods are omitted


  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }

}
```

# NOTE

- An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

# NOTE

- Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

# Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

Overriding:

10.0

10.0

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Oveloading:

10

20.0

| Method Overloading | MethodOverriding |
| --- | --- |
| 1. It occurs with in the same class.<br><br>2. Inheritance is not involved.<br><br>3.One method does not hide another.<br><br>4.Parameters must be different.<br><br>5.return type may or may not be same.<br><br>6. Access modifier & Non access modifier can also be changed. | 1. It occurs between two classes i.e., Super class and a subclass.<br>2. Inheritance is involved.<br><br>3. child method hides that of the parent class method.<br>4. Parameters must be same.<br><br>5. return type must be same.<br><br>6. Access modifier should be same or increases the scope of the access modifier.<br>Non access modifier –<br>• **final** : if a method can contain final keyword in a parent class we cannot override.<br>• **static**: if a method can contain static keyword child cannot override parent class methods but hide (child). |

# Overloading Overriding

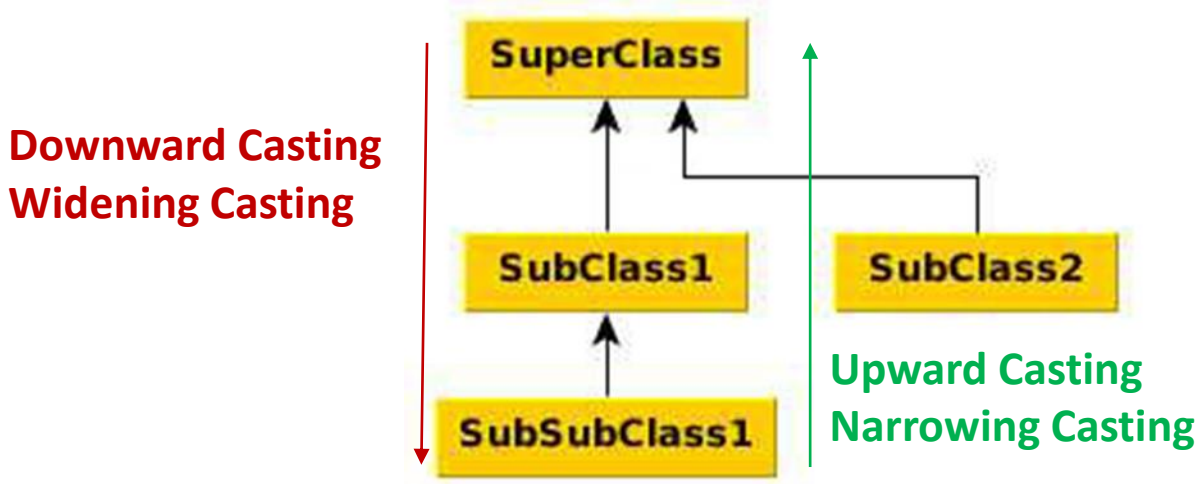
LECTURE 1

# Casting

## Review of Basic Package

- The third section of the output indicates how to access a member function not defined at the top level. Doing so is achieved by casting the object. In our example, SubSubClass1 contains the method bottom which no other class defines.
- The cast operation means simply to alter the type like this:

```
(SubSubClass1) obj
```



# Casting

## Review of Basic Package

- It is usually intended to go down an inheritance path from superclass towards the class of the object. Casting cannot be used to "make a cat bark," so to speak. For example, if we were to use this code

```
SuperClass obj = new SubClass1();
(SubClass2) obj  // or
(SubSubClass1) obj
```

the runtime outcome would be a **ClassCastException**.

# Casting

Casting up toward the superclass (downward casting) is legal, but useless, since it can never change the outcome of a member function, i.e.

```
SuperClass obj = new SubSubClass1();

((SubClass1) obj).pervasive();
```

will not be any different than if the cast were not used:

```
obj.pervasive();
```

# Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. ***Casting*** can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

# Why Casting Is Necessary?

## (Downward Casting can be Implicit. Upward Casting must Match Class Type.)

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

A compile error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student) o;          // Explicit casting
```

# Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

# The instanceof operator

- In the polymorphic setting where objects of subclasses are typed by the superclass, we use the **instanceof** operator to determine the type prior to casting. Thus, casting is often combined with instance of in the manner used in the example:

```
if (obj instanceof SubSubClass1) {
   ((SubSubClass1) obj).bottom();
}
```

# The instanceof operator

- The **instanceof** operator recognizes the "is a" relation of classes in that these two print statements print true values:

```
SuperClass obj1 = new SubSubClass1();
System.out.println("" + (obj1 instanceof SubClass1));
System.out.println("" + (obj1 instanceof
SuperClass));
```

# The instanceof Operator

Use the instanceof operator to test whether an object is an instance of a class:

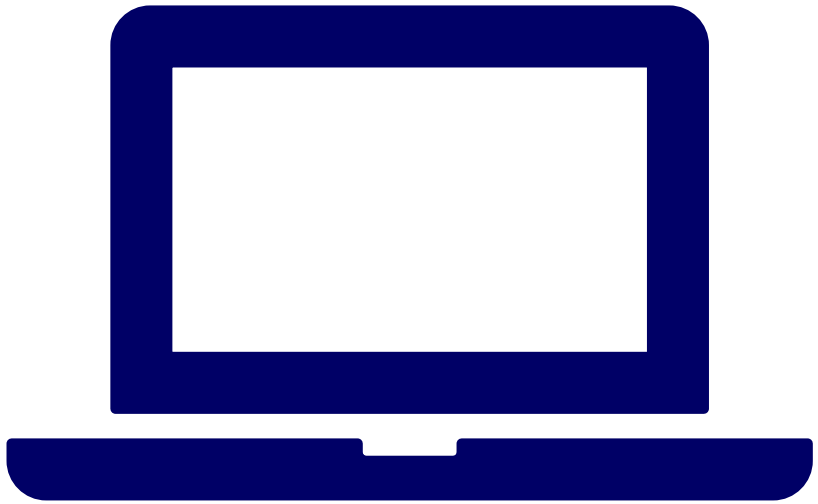
```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

# TIP

- To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange.
- An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit.
- However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

# Example: Demonstrating Polymorphism and Casting

- This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method to display the objects. The displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.



# Demonstration Program

CASTINGDEMO.JAVA

# The equals Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

# NOTE

- The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The equals method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects.
- The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

# Inclusion Polymorphism

LECTURE 1

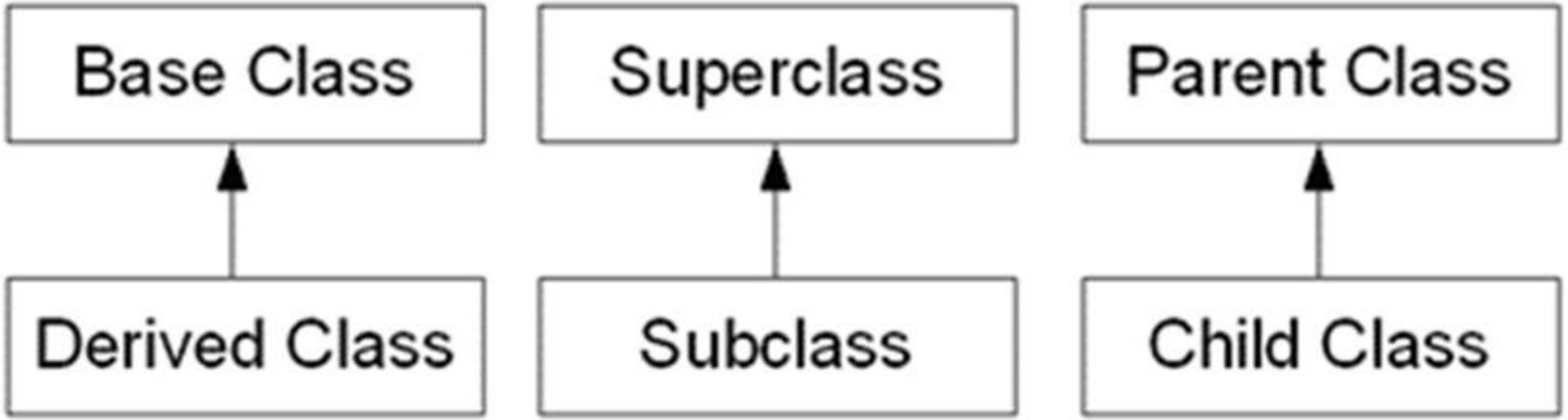
# Inclusion Polymorphism

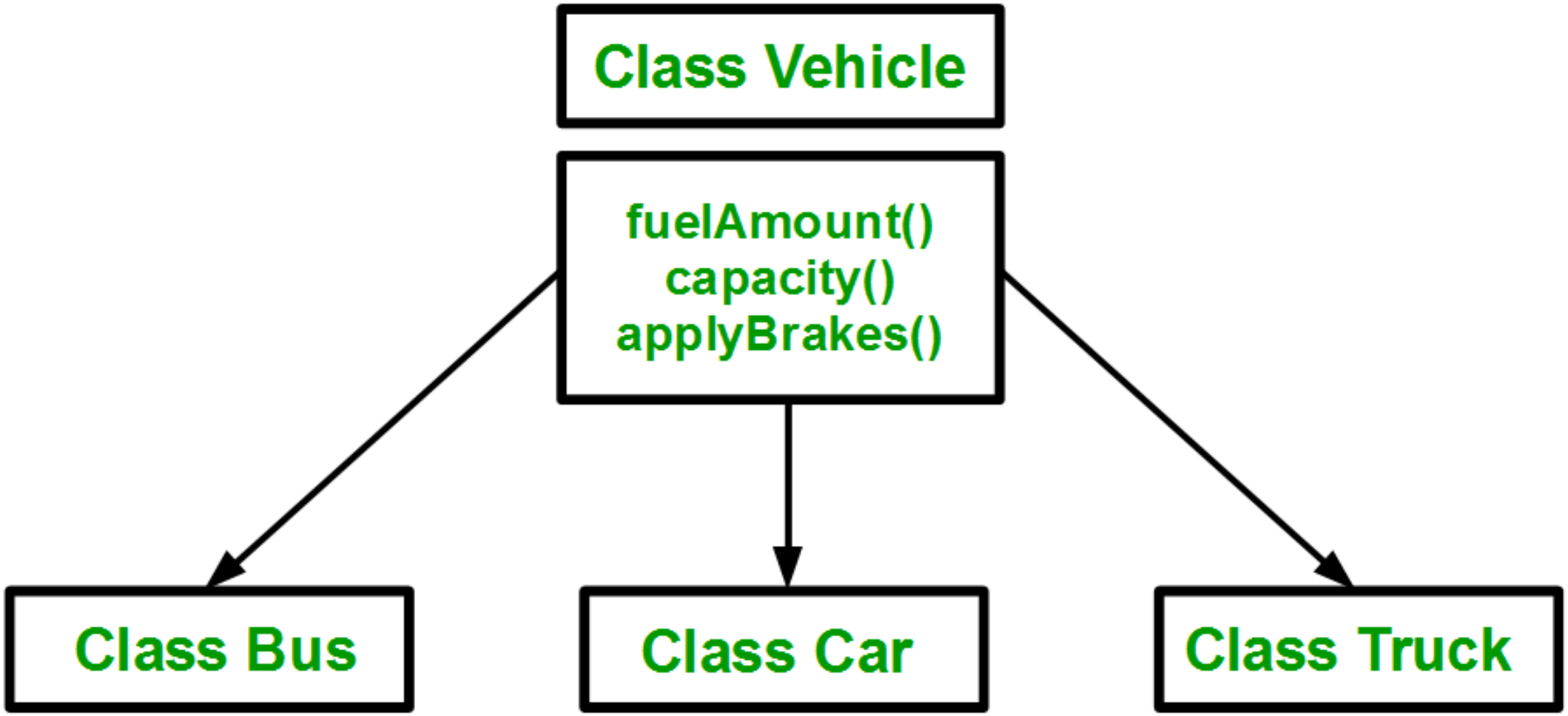
## Subclass/Subtype Polymorphism

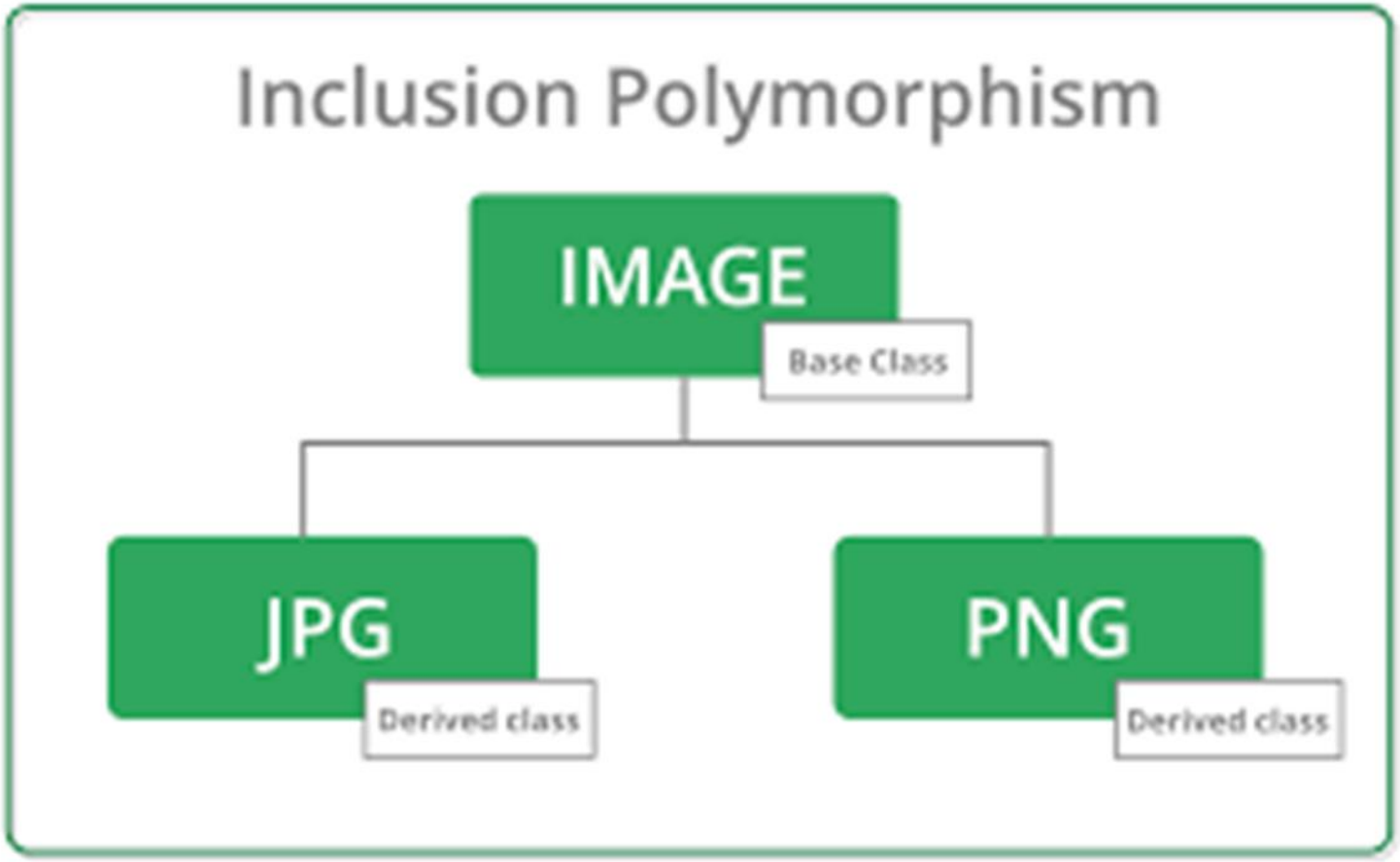
- **Subtype** means that a type can serve as another type's subtype. When a subtype instance appears in a supertype context, executing a supertype operation on the subtype instance results in the subtype's version of that operation executing. For example, consider a fragment of code that draws arbitrary 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

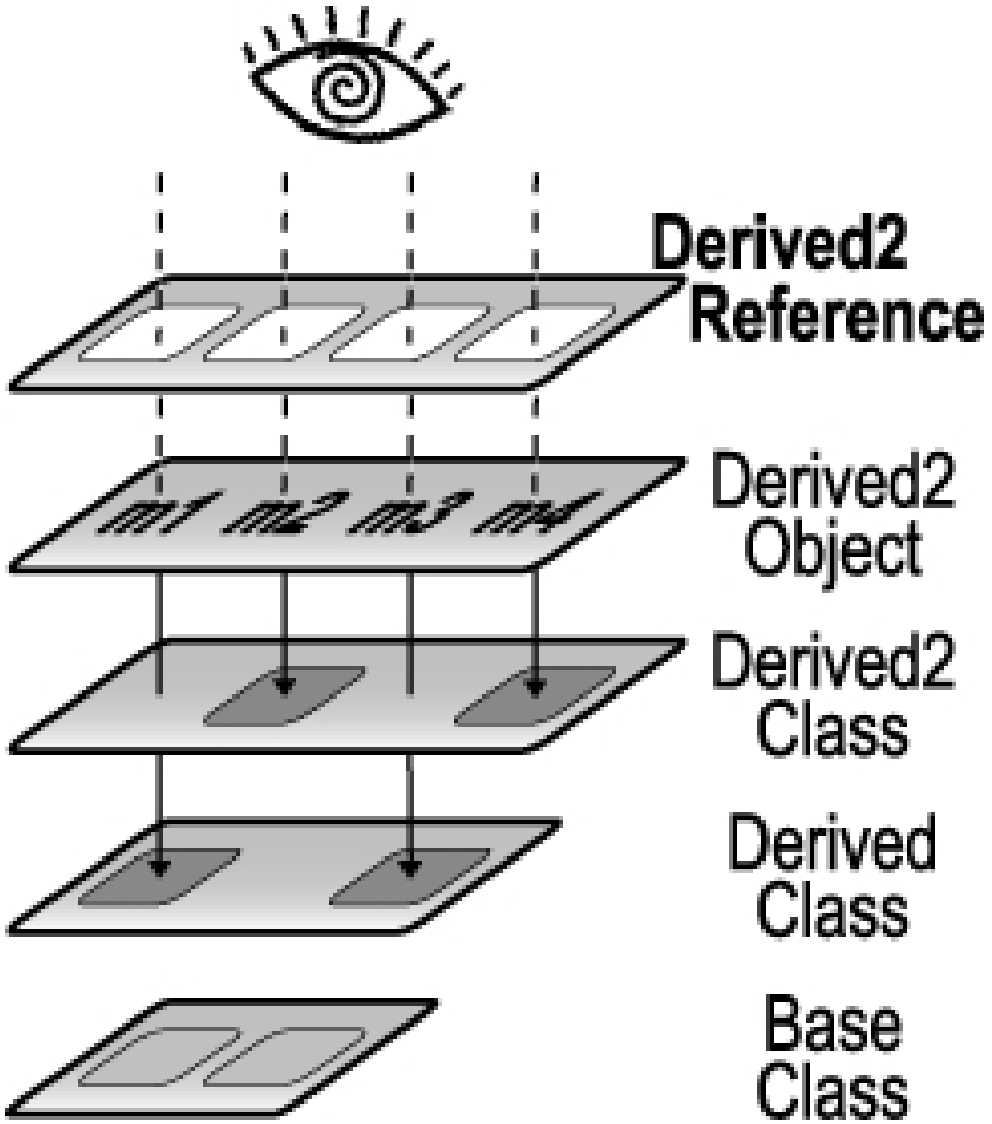
HondaCRV[] carList = {car2, car1};

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

HondaCRV

HondaCar

Car



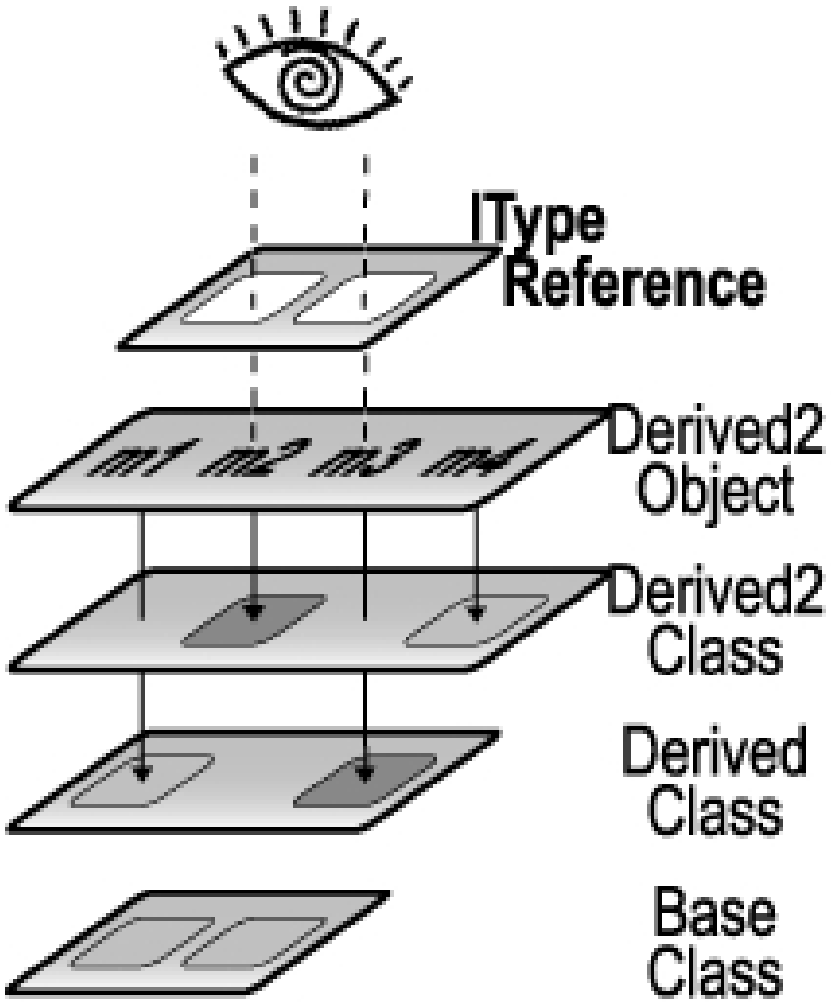
```
Car[] carList =
  {new Car(), new HondaCar(), car1, car2};
```

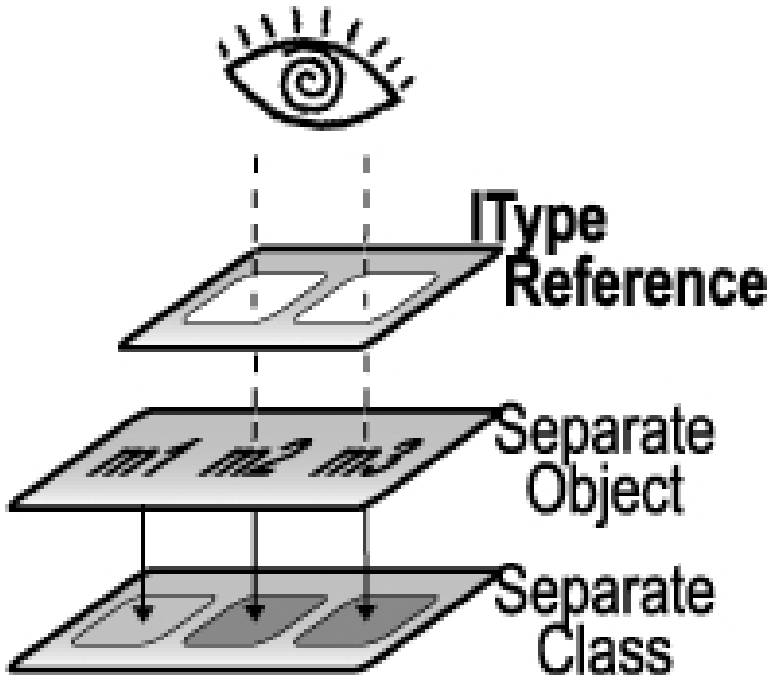
```
HondaCRV  car1 = new HondaCRV();
HondaCRV  car2 = new HondaCRV();
```

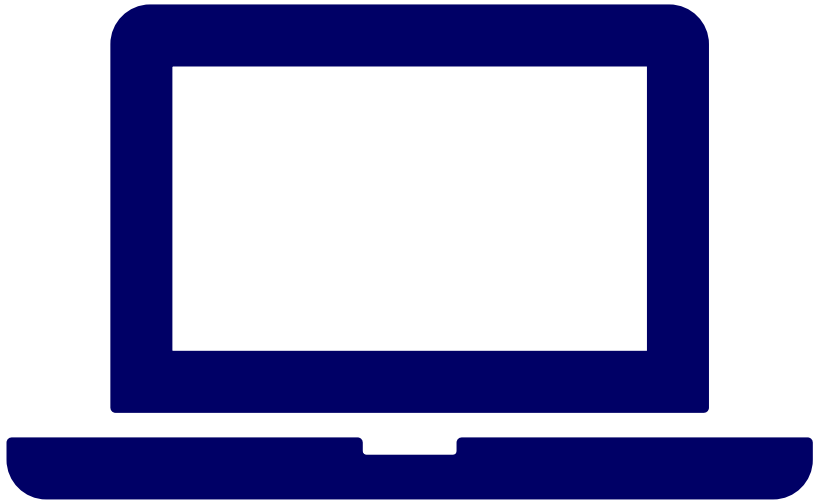
HondaCRV

HondaCar

Car

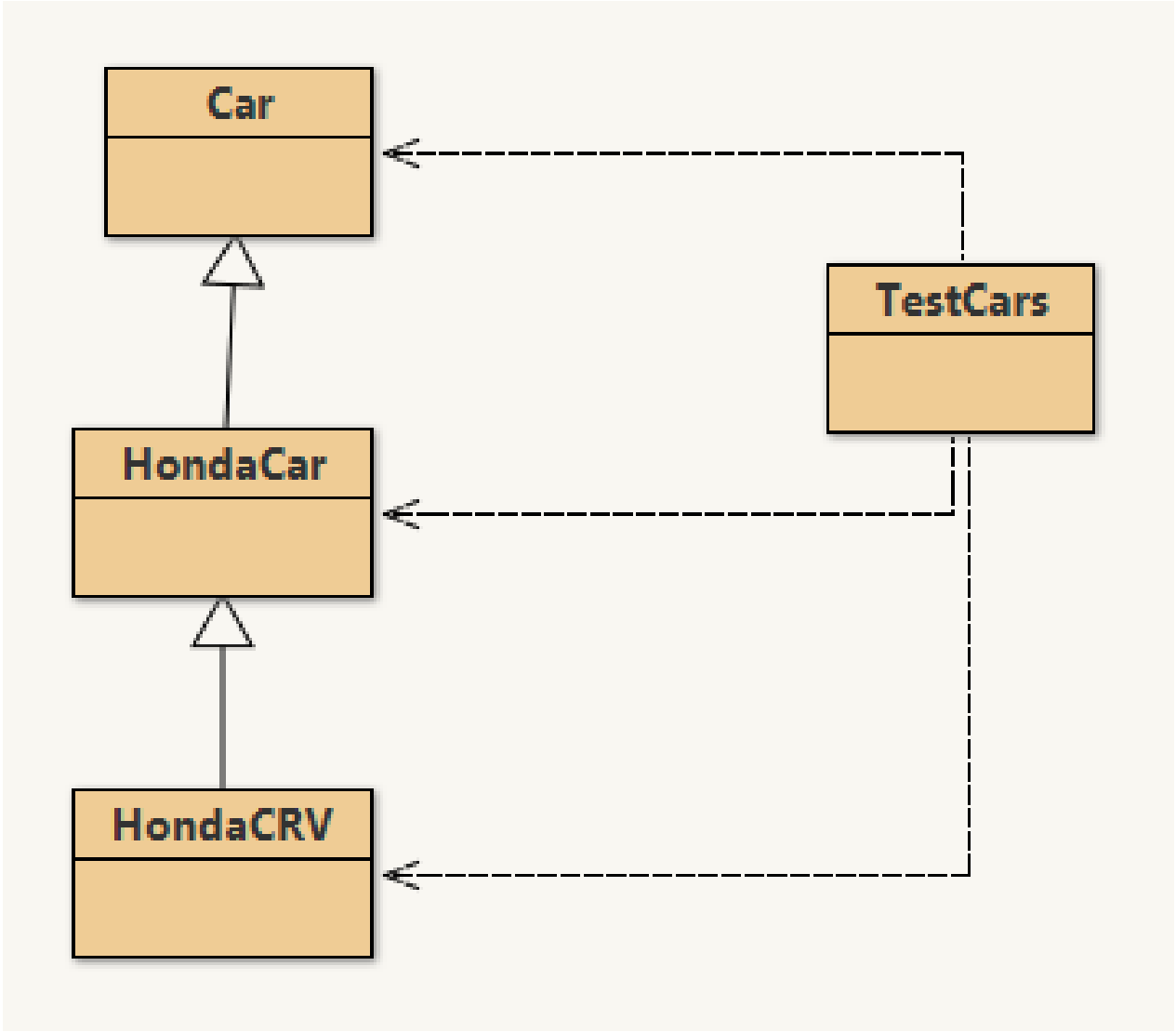




# Demonstration Program

CAR.JAVA HONDACAR.JAVA HONDACRV.JAVA


Car
TestCars
HondaCar
HondaCRV

# Parametric Polymorphism

LECTURE 1

# Type Variable

## <T> in angle brackets

- **<T>** represents a *formal generic type*, which can be replaced later with an *actual concrete type*. Replacing a generic type is called a *generic instantiation*.

- By convention, a single capital letter such as **E** or **T** is used to denote a formal generic type. (**E**ntity and **T**ype)

- To see the benefits of using generics, let us examine the code in Figure B. The statement in Figure B(a) declares that **c** is a reference variable whose type is **Comparable** and invokes the **compareTo** method to compare a **Date** object with a string. The code compiles fine, but it has a runtime error because a string cannot be compared with a date.

# Type Variable

## <T> in angle brackets

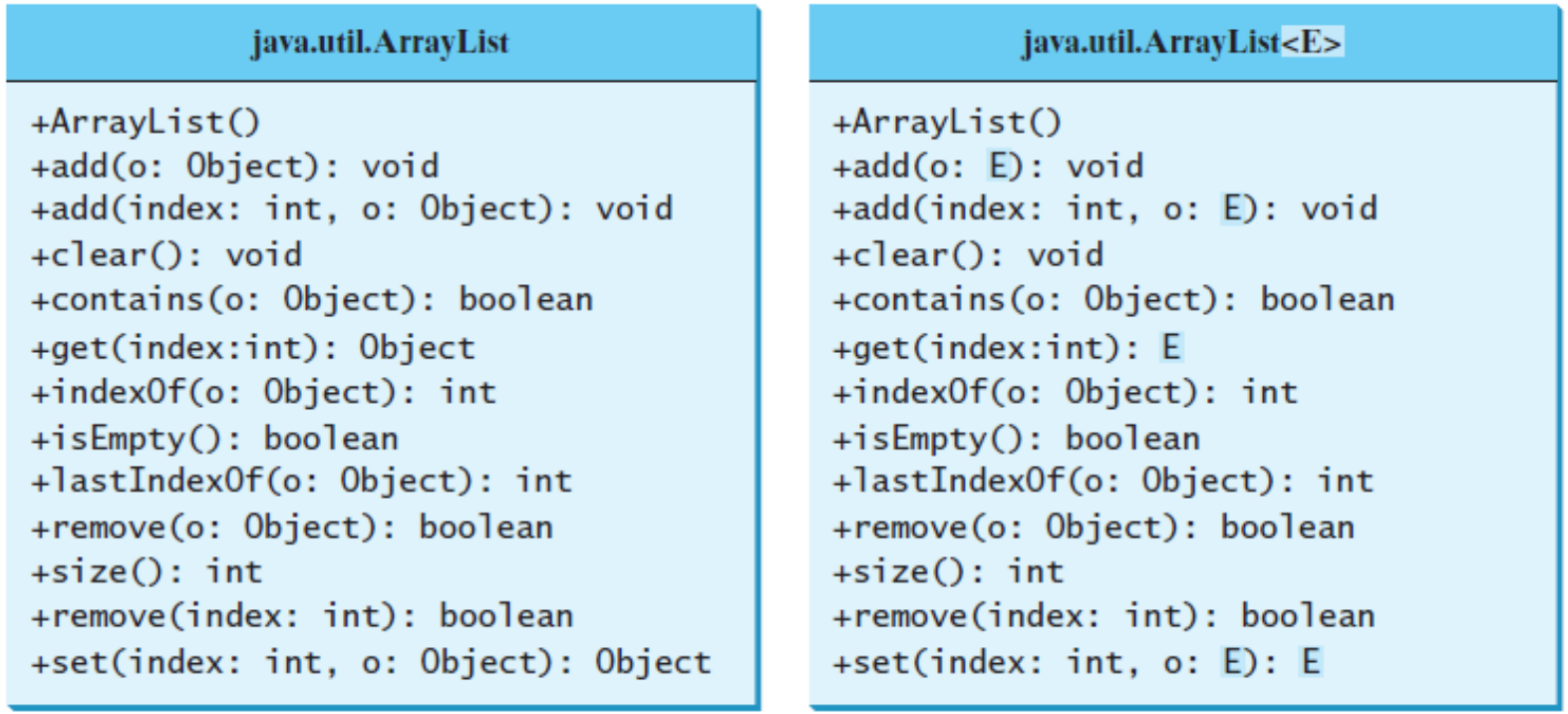
- The statement in Figure B(b) declares that **c** is a reference variable whose type is **Comparable<Date>** and invokes the **compareTo** method to compare a **Date** object with a string.

- This code generates a compile error, because the argument passed to the **compareTo** method must be of the **Date** type. Since the errors can be detected at compile time rather than at runtime, the generic type makes the program more reliable. (The compareTo() can be overrided.)

- The **ArrayList** Class. This class has been a generic class since JDK 1.5.

# Type Variable

## <T> in angle brackets




(a) ArrayList before JDK 1.5

(b) ArrayList since JDK 1.5

Figure C.

# ArrayList as an Example for Generic Container

**Declaration of the Pointer(Reference):**

```
ArrayList<String> alist = new ArrayList<String>();
```

**Addition of Element (body):**

```
alist.add(new String(1));
```

**Generic Container only for Reference Type:**

```
ArrayList<int> alist = new ArrayList<int>();
```

The primitive type is not allowed here.

Casting is not needed to retrieve a value from a list with a specified element type, because the compiler already knows the element type. For example, the following statements create a list that contains strings, add strings to the list, and retrieve strings from the list.

```
ArrayList<String> alist = new ArrayList<>();
alist.add("Red");
alist.add("White");
String s = list.get(o); // No casting needed.
```

# Defining Generic Classes and Interfaces

A generic type can be defined for a class or interface. A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.

This example creates a stack to hold integers and adds three integers to the stack.

```
GenericStack<Integer> stack2 = new GenericStack<>();
stack2.push(1);  // autoboxing
stack2.push(2);
stack2.push(3);
```

Instead of using a generic type, you could simply make the type element Object, which can accommodate any object type. However, using generic types can improve software reliability and readability, because certain errors can be detected at compile time rather than at runtime. For example, because stack1 is declared GenericStrck<String>, only strings can be added to the stack. It would be a compile error if you attempted to add an integer to stack1.

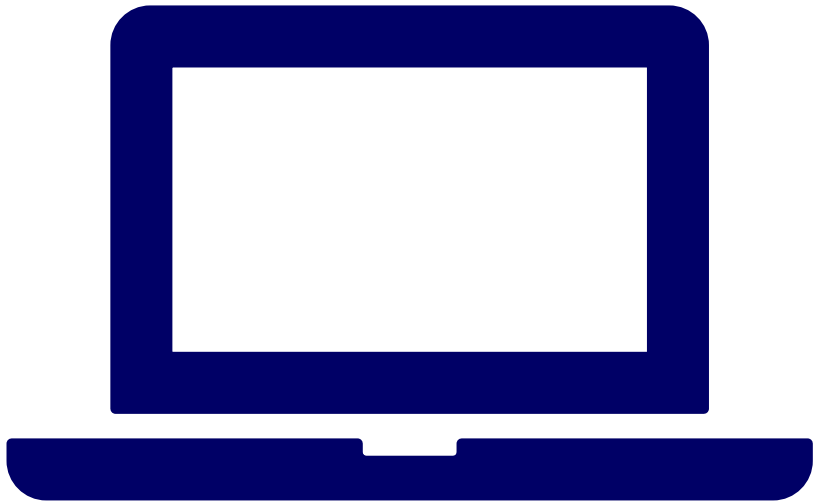
# Note:

- Multiple type variables for a generic class definition. For example, **<E1, E2, E3>**
- To create a stack of strings, you can **new GenericStack<String>()** or **new GenericStack()**. This could mislead you into thinking that the constructor of GenericStack should be defined as

  **public GenericStack<E>()**

This is wrong. It should be defined as

**public GenericStack()**



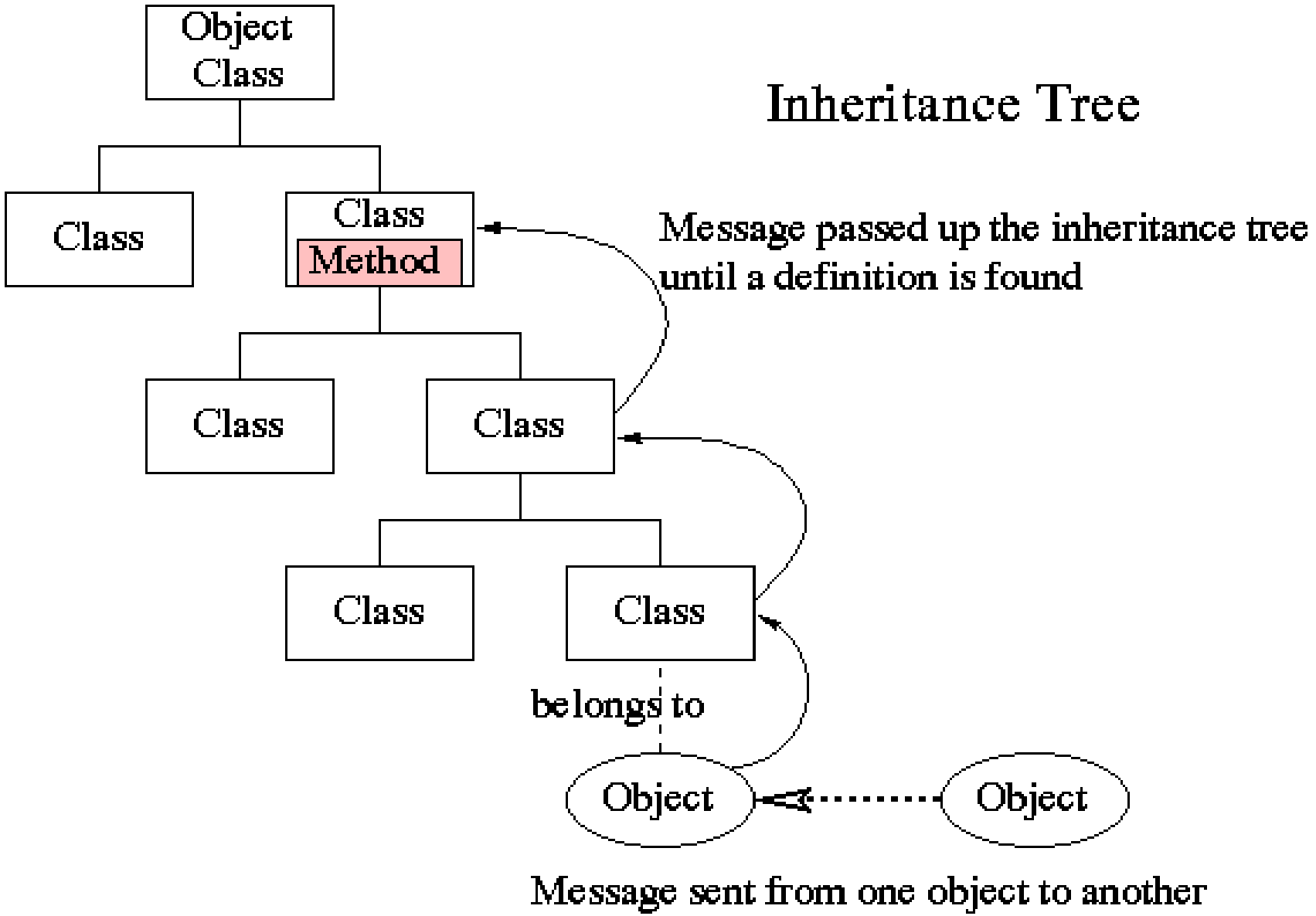
# Demonstration Program

GENERICSTACK.JAVA +
TESTGENERICSTACK.JAVA

# Equality Check:

## equality package

LECTURE 1


Object
Class
Inheritance Tree
Class
Class
Method
Message passed up the inheritance tree
until a definition is found
Class
Class
Class
Class
belongs to
Object
Object
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**

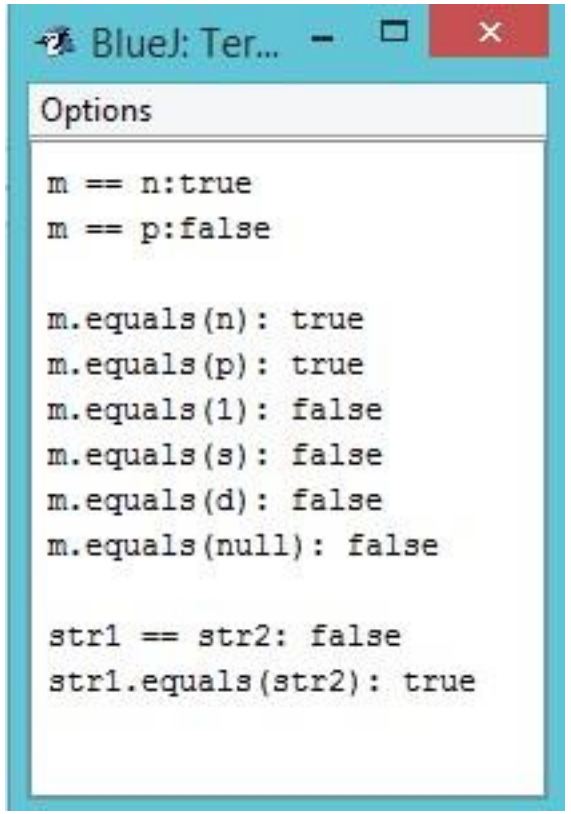
equality.Wrappers

```java
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)));
  }
}
```

# The equals and getClass methods

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

**Wrapper Class is a Tester Class**



# 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

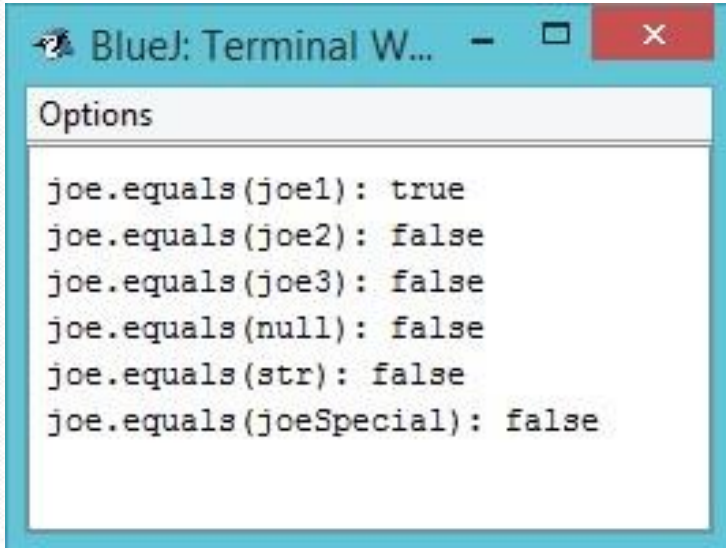
```
package equality;

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

```
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));
  }
}
```



# 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()
```

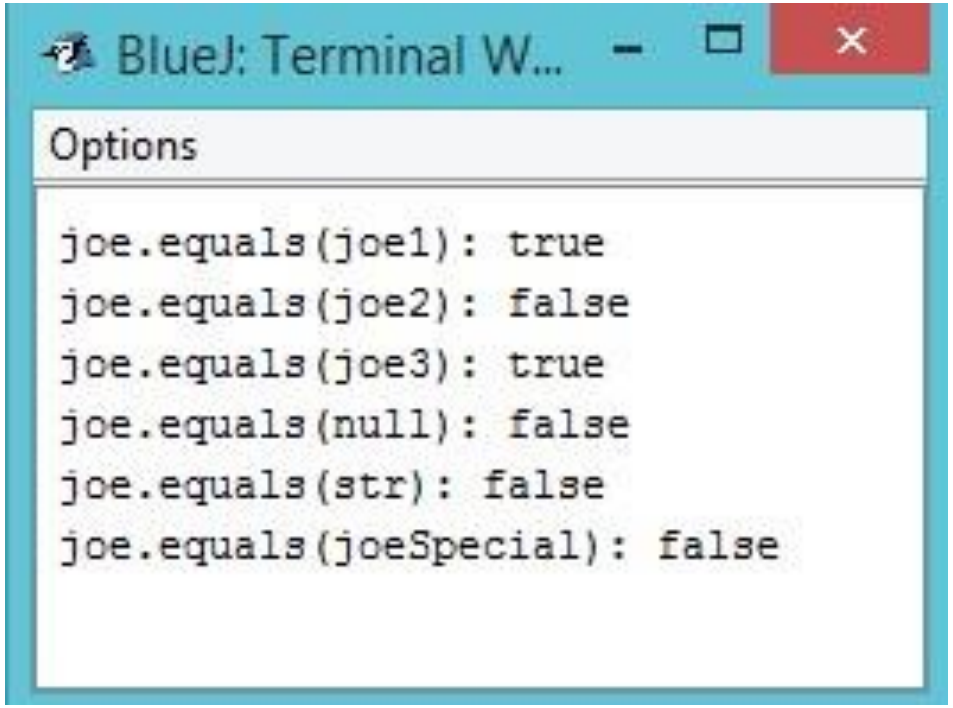
# Custom equals() Methods:

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

```
@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:

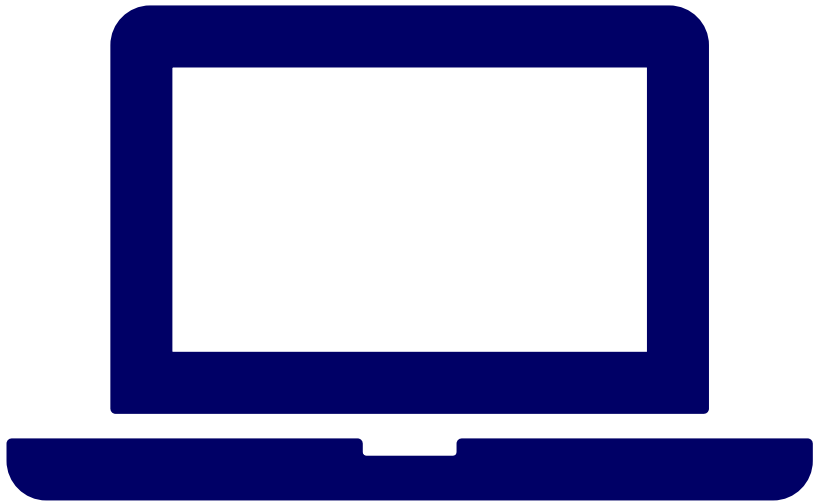



Play around other equals() definition using **==, instanceof and getClass(), getClass().getName()** to check for other equality definitions.

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

# Polymorphism Example

LECTURE 1



# Demonstration Program

GEOMETRIC OBJECTS PROJECT

# Summary

LECTURE 1

# Summary:

- Polymorphism provides program mechanism to handle data of different type.  It can be applied to polymorphic methods, polymorphic containers, and polymorphic iterators.
- This lecture explain the meaning of polymorphism and its applications.