AP Computer Science B
Java Object-Oriented Programming [Ver. 2.0]

Unit 4: Object-Oriented Design

WEEK 6: CHAPTER 12 INHERITANCE AND POLYMORPHISM (PART 1)

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

- Inheritance: is_A relationship
- Class reference: super keyword
- Design of a sub-class
- Polymorphism and types of polymorphism
- Overloading/Overriding
- Coercion Polymorphism

Overview

LECTURE 1

Inheritance

Inheritance in Java begins with the relationship between two classes defined like this:

```
    class SubClass extends SuperClass
```

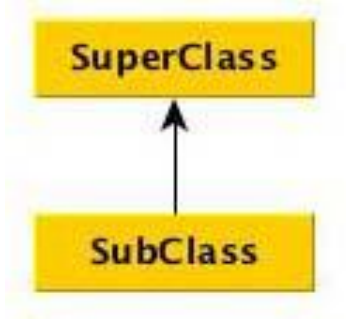
Inheritance expresses the is a relationship in that SubClass is a (**specialization** of) SuperClass. The extends relation has many of the same characteristics of the implements relationship used for interfaces

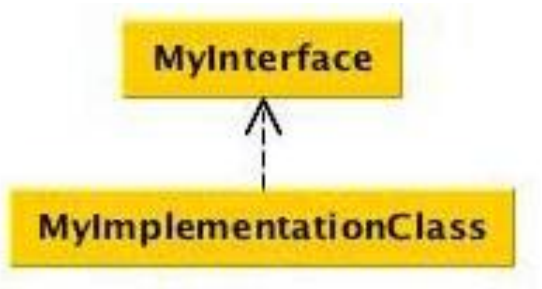
```
    class MyImplementationClass implements MyInterface
```

As with inheritance, we say that **MyImplementationClass** is a **MyInterface**.

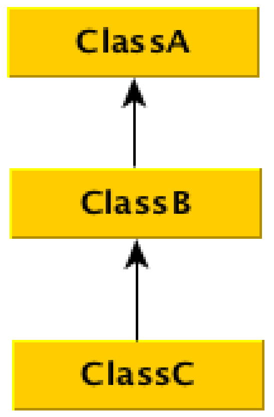
Inheritance

Diagrammatically, these relationships are expressed in UML with the extends as a solid line (white Triangle in some tools) and implements as a dashed line:





The ***is a*** relationship is transitive in that, if we have this hierarchy:



Resources

INHERITANCE.ZIP

DOWNLOAD

Inheritance

in which

ClassB is a ClassA

ClassC is a ClassB

then, by transitivity:

ClassC is a ClassA

The term base class is also used for superclass, and derived class as subclass.

Being a subclass is also transitive in that we can say that:

ClassC is a subclass of ClassA

The term inheritance expresses the fact that the objects of the subclass inherit all the features of the superclass including data members and functions, although the private data members and functions of the superclass are **not** directly accessible.

What does a subclass inherit?

- A **subclass inherits** all the members (fields, methods, and nested classes) from its superclass. **Constructors** are not members, so they are not **inherited** by **subclasses**, but the constructor of the superclass can be invoked from the **subclass**.
- Members of a class that are declared private are not directly accessible by subclasses of that class. Only members of a class that are declared **protected** or **public** are accessed directly by subclasses declared in a package other than the one in which the class is declared.

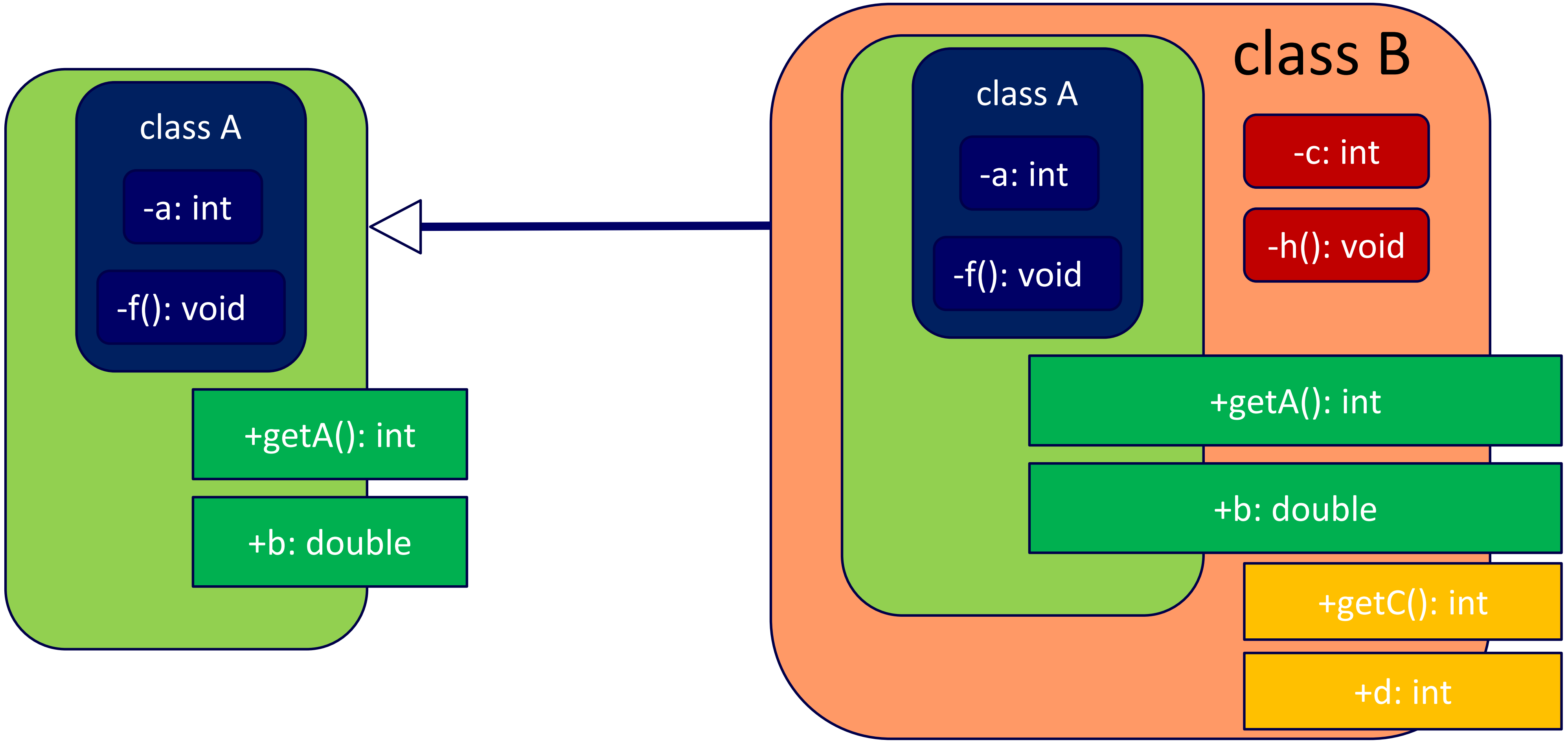
protected: no access by other package but can be inherited.

Visibility Modifiers

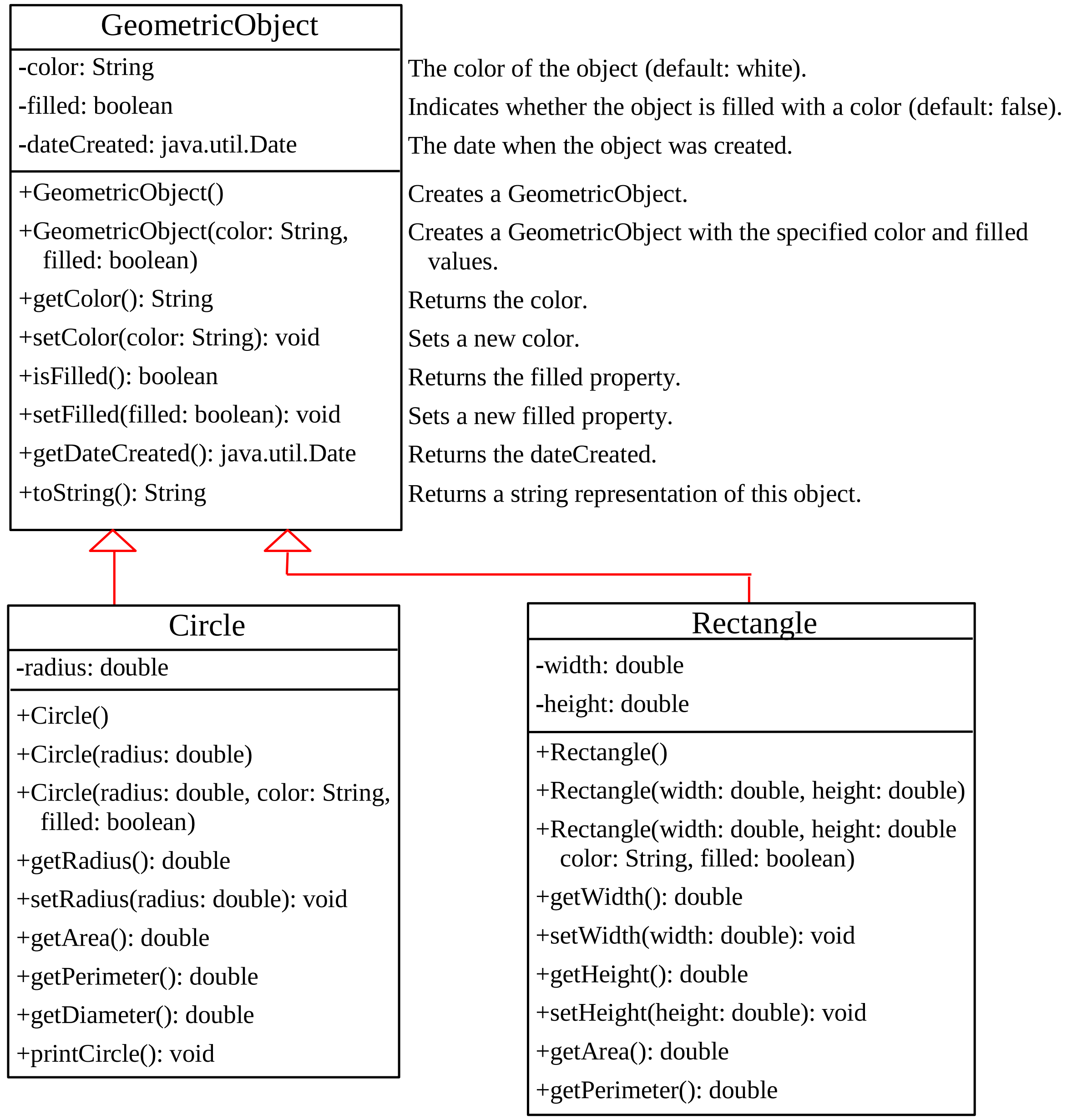
Modifier	Inheritance	Access
+public	All	All
#protected	All	Subclasses
~default (none)	All	Same package
-private	All	Same class

Accessing an Objects of subclasses

(Subclass is another kind of Wrapper Class)



Superclasses and Subclasses



GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Demonstration Program

TESTCIRCLERECTANGLE.JAVA
GEOMETRICOBJECT.JAVA
CIRCLEFROMSIMPLEGEOMETRICOBJECT.JAVA
RECTANGLEFROMSIMPLEGEOMETRICOBJECT.JAVA

Are superclass's Constructor Inherited?

- **No. They are not inherited.**
- **They are invoked explicitly or implicitly.**
- **Explicitly using the super keyword.**

- A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword <u>super</u>. ***If the keyword <u>super</u> is not explicitly used, the superclass's no-arg constructor is automatically invoked.***

Superclass's Constructor Is Always Invoked

- A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts <u>super()</u> as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

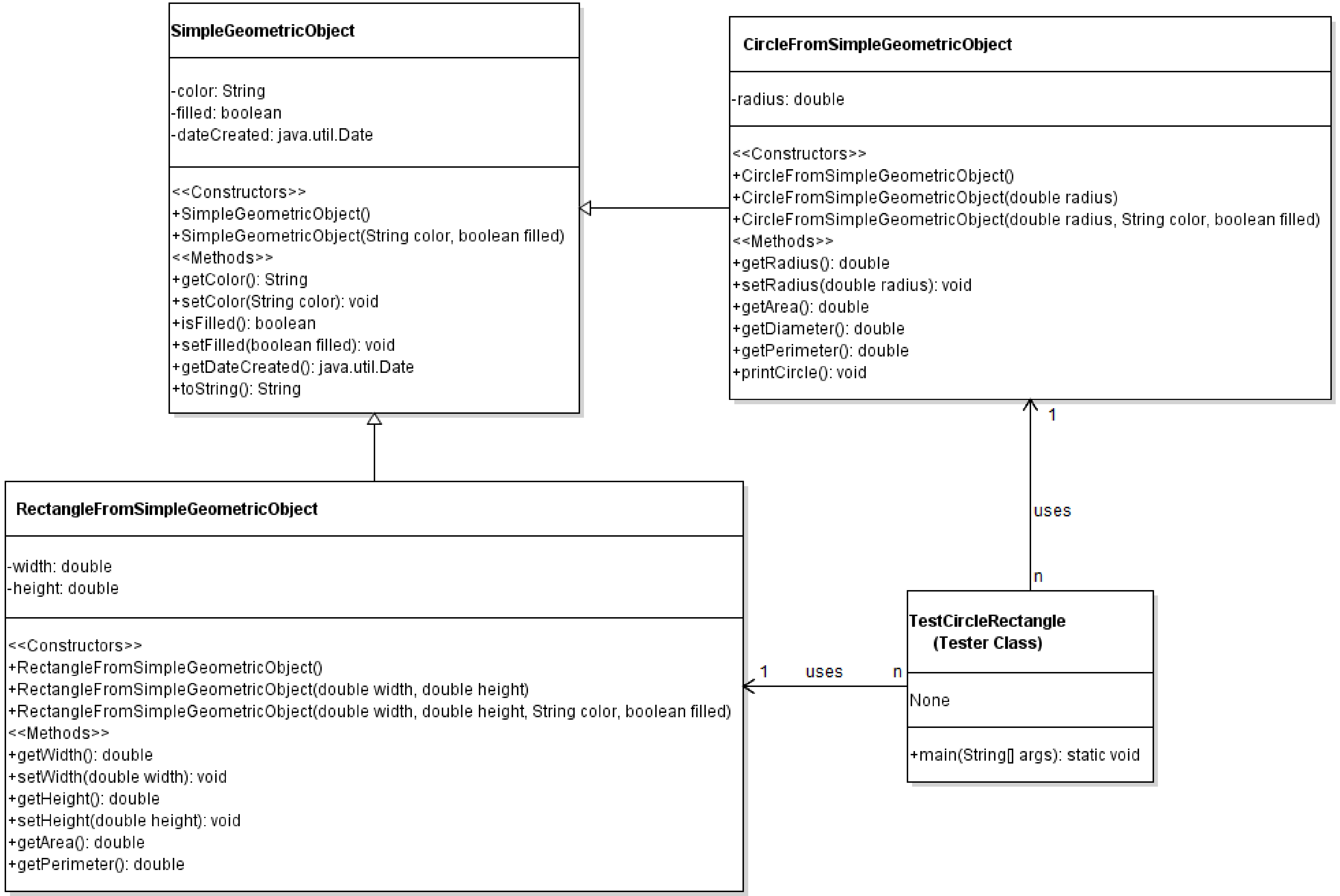
```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Demonstration Program

VIOLET UML FOR INHERITANCE


SimpleGeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
<<Constructors>>
+SimpleGeometricObject()
+SimpleGeometricObject(String color, boolean filled)
<<Methods>>
+getColor(): String
+setColor(String color): void
+isFilled(): boolean
+setFilled(boolean filled): void
+getDateCreated(): java.util.Date
+toString(): String
CircleFromSimpleGeometricObject
-radius: double
<<Constructors>>
+CircleFromSimpleGeometricObject()
+CircleFromSimpleGeometricObject(double radius)
+CircleFromSimpleGeometricObject(double radius, String color, boolean filled)
<<Methods>>
+getRadius(): double
+setRadius(double radius): void
+getArea(): double
+getDiameter(): double
+getPerimeter(): double
+printCircle(): void
1
uses
n
RectangleFromSimpleGeometricObject
-width: double
-height: double
<<Constructors>>
+RectangleFromSimpleGeometricObject()
+RectangleFromSimpleGeometricObject(double width, double height)
+RectangleFromSimpleGeometricObject(double width, double height, String color, boolean filled)
<<Methods>>
+getWidth(): double
+setWidth(double width): void
+getHeight(): double
+setHeight(double height): void
+getArea(): double
+getPerimeter(): double
1
uses
n
TestCircleRectangle
(Tester Class)
None
+main(String[] args): static void

super keyword

LECTURE 2

Using the Keyword super

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

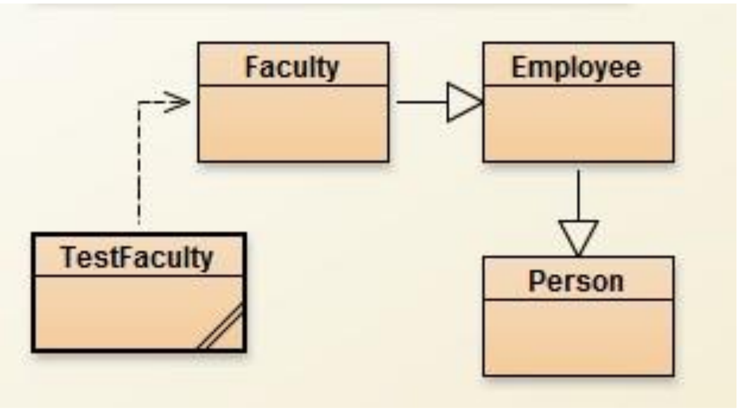
- To call a superclass constructor
- To call a superclass method (access public methods, protected methods, default methods same package only, no private methods)

CAUTION

- You must use the keyword **super** to call the superclass constructor. Invoking a superclass constructor’s name in a subclass causes a syntax error. Java requires that the statement that uses the keyword **super** appear first in the constructor.

Constructor Chaining

Constructing an instance of a class invokes all the superclasses’ constructors along the inheritance chain. This is called *constructor chaining*.



```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee’s no-arg constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

Trace Execution

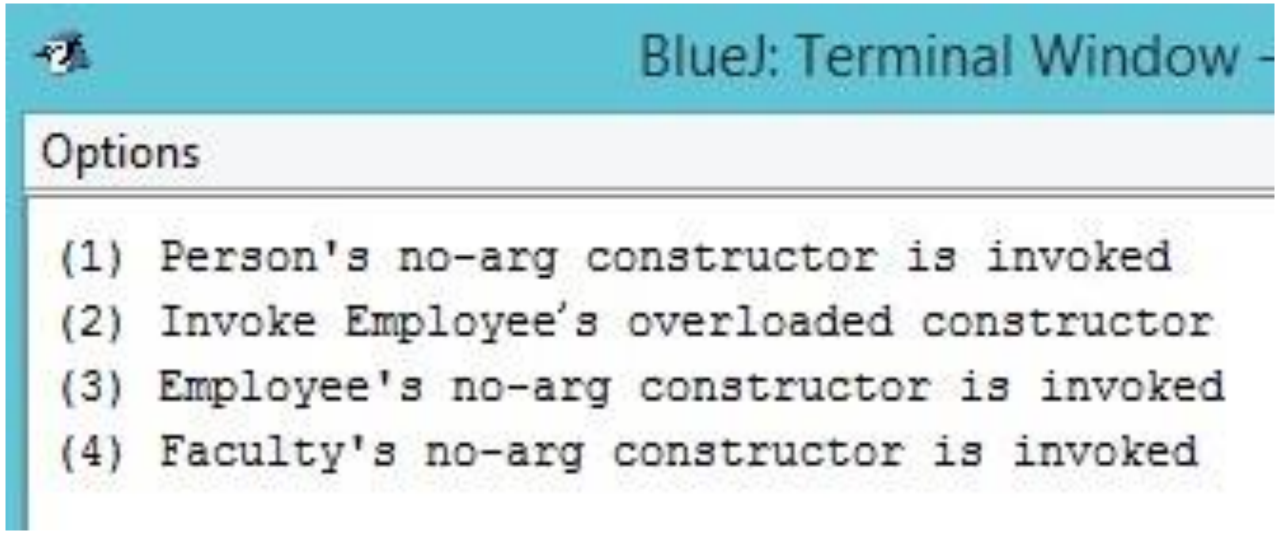
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println



```
(1) Person's no-arg constructor is invoked
(2) Invoke Employee's overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked
```

Results:

Example on the Impact of a Superclass without no-arg Constructor

- Find out the errors in the program:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

Nothing will be called

Design of a Subclass

LECTURE 3

Defining a Subclass

A subclass inherits from a superclass. You can also:

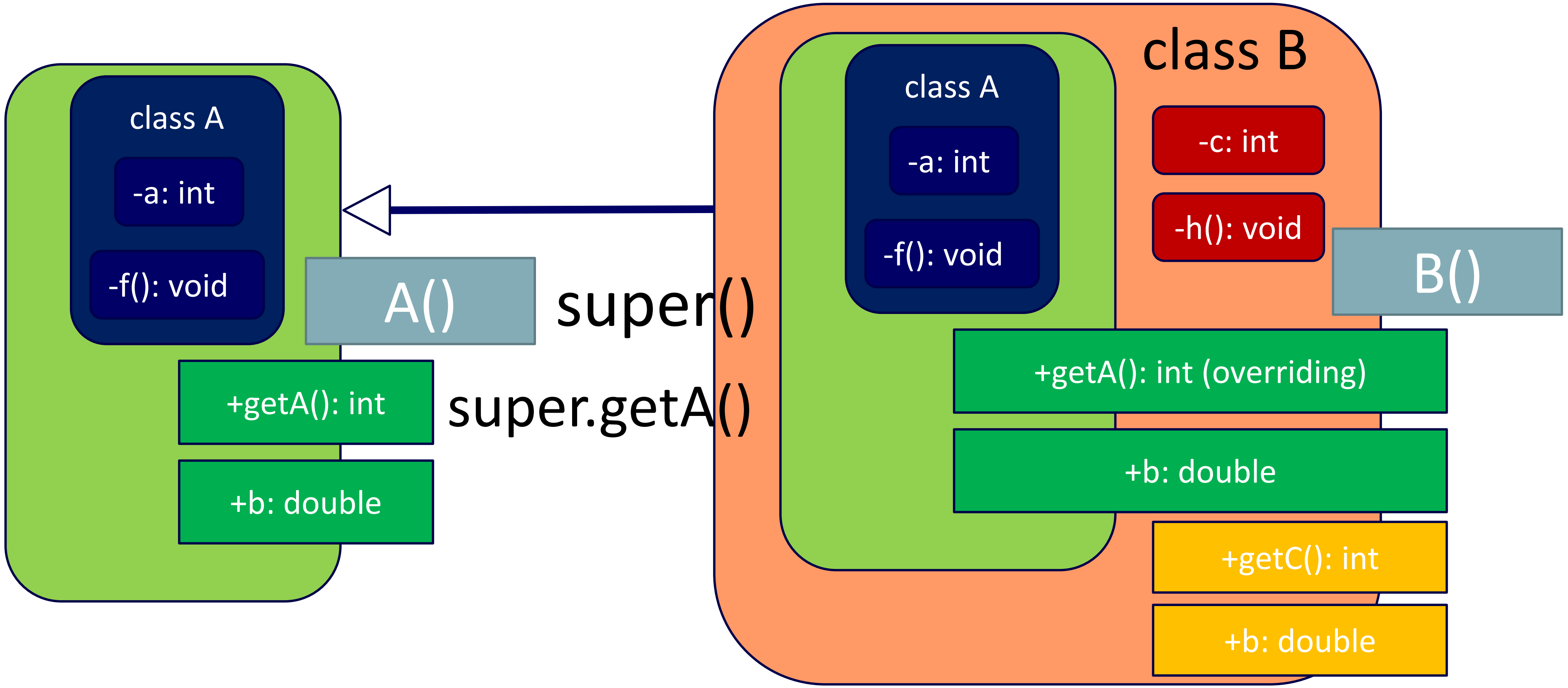
- Add new properties
- Add new methods
- Derived new properties from base class
- Override the methods of the superclass

Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

super keyword



In-Class Demonstration Program

BUILDING A NEW SUBCLASS

Polymorphism

LECTURE 4

The Object Class and Its Methods

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
   ...
}
```

Equivalent

```
public class Circle extends Object {
   ...
}
```

The toString() method in Object
(useful for checking whether two pointers are identical)

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

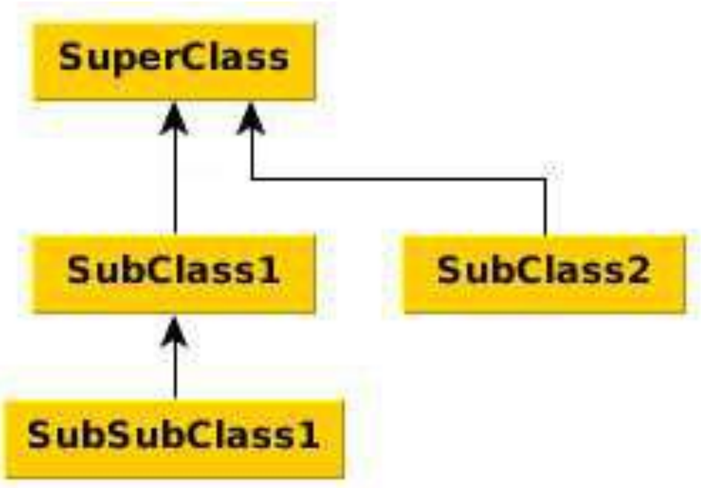
```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@**15037e5** . This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

Polymorphism (Objects/Methods of Many Forms)

Polymorphism is the ability of an object to take on many forms.

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.

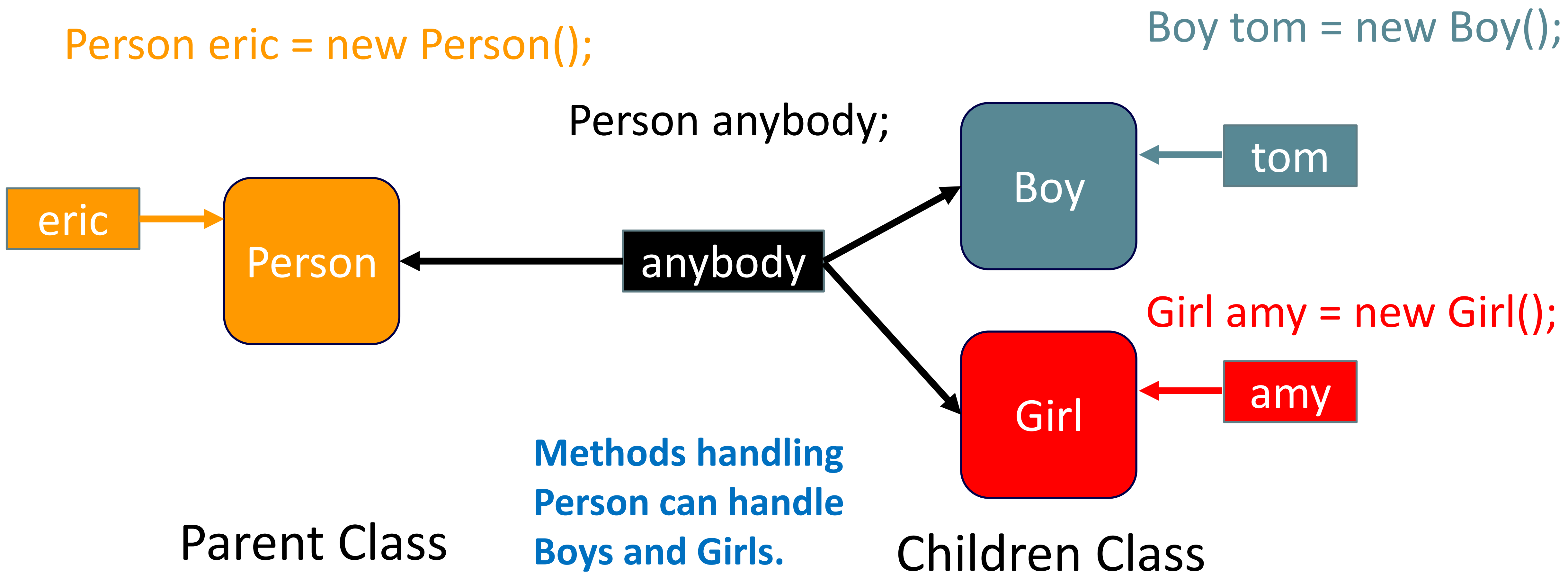
Polymorphism

The **reference variable** can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Polymorphism

One Pointer can be pointing to objects of many forms. Used in methods' parameter list (dynamic linking).



Methods handling Person can handle Boys and Girls.

Polymorphism

Expressed in Java notation:

```
class SubClass1 extends SuperClass { ... }
class SubClass2 extends SuperClass { ... }
class SubSubClass1 extends SubClass1 { ... }
```

The SuperClass can be used to express the type of any object in this hierarchy, thus we can write:

```
SuperClass objs[] = new SuperClass[] {
  new SubClass1(),
  new SubClass2(),
  new SubSubClass1(),
}
```

Polymorphism

Suppose each class in the hierarchy, including the SuperClass, defines the member function:

```
public void foo() { ... }
```

Then it at least makes sense to write:

```
for (SuperClass obj: objs) {
  obj.foo();
}
```

Polymorphism

- The question is: which "foo" is actually called in each case? It turns out that the foo() called is based the object's class, not the declared type, which is consistently **SuperClass**. Calling by class as opposed to type is referred to as **dynamic binding** in that the determination is made at **runtime**, **not compile time**.
- Dynamic binding is the essence of polymorphism in that the foo member function call means something different to each object in the array.

Demonstration Program

BASIC PACKAGE

Results of Demo Program

The output of the driver program is:

```
====> calls to top
---- obj in SuperClass calls: top in SuperClass
---- obj in SubClass1 calls: top in SuperClass
---- obj in SubClass2 calls: top in SuperClass
---- obj in SubSubClass1 calls: top in SuperClass

====> calls to pervasive
---- obj in SuperClass calls: pervasive in SuperClass
---- obj in SubClass1 calls: pervasive in SubClass1
---- obj in SubClass2 calls: pervasive in SubClass2
---- obj in SubSubClass1 calls: pervasive in SubSubClass1

====> calls to bottom
---- obj in SuperClass
---- obj in SubClass1
---- obj in SubClass2
---- obj in SubSubClass1 calls bottom in SubSubClass1

====> casting changes nothing about pervasive call
---- obj in SubSubClass1 casted to SubClass1 type, calls pervasive
pervasive in SubSubClass1
```

Overloading, Overriding versus Polymorphism

Overloading of Methods and Constructors of Same Class:

```
int max(int a, int b){…}
double max(double a, double b){…}
```

Overriding of Methods and Constructors of a subclass:

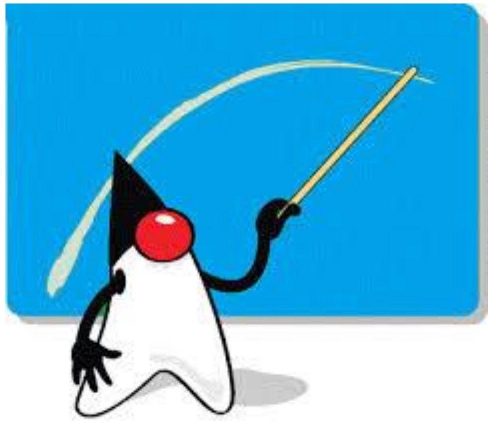
```
int max(){ i=i+1; }
int max(){ i=i-1; }
```

Polymorphism: Methods works on different objects of SuperClass and SubClass:

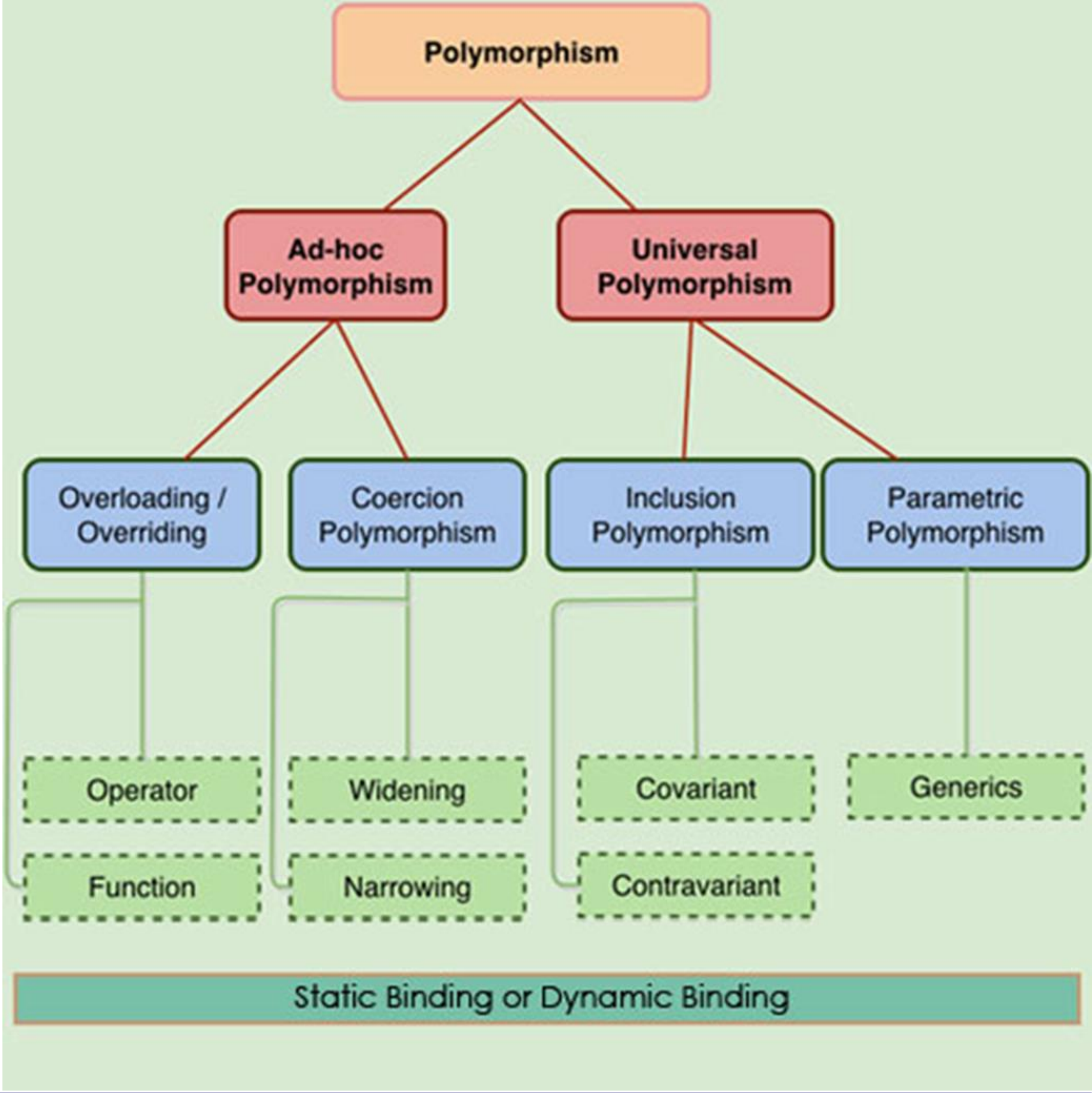
```
Person max(Person a, Person b){….}                                    // polymorphism
Boy b = new Boy();  Girl g = new Girl(); Person p = new Person();
Boy b2 = (Boy) max(b, max(g, p));                                     // casting of objects
```

Types of Polymorphism

LECTURE 5

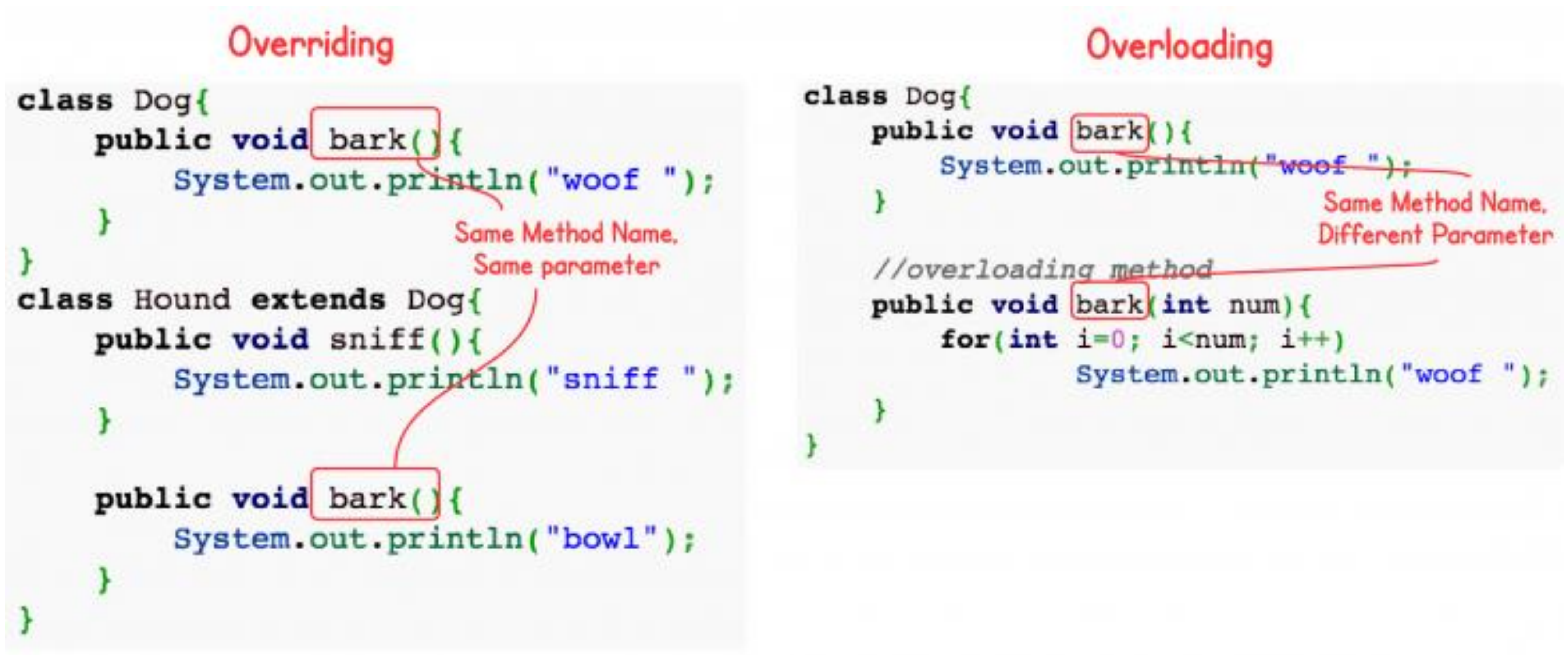


Key Topics in Polymorphism



Overloading (Chapter 9)

Two methods of a same name working on different data types



```
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");
    }
}
```

```
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 ");
    }
}
```

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.

Demonstration Program

CALLING OVERRIDDEN METHODS (MODIFYING BASIC PACKAGE, AND CHECK SUPERCALLS PACKAGE)

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

```java
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
```

```java
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
```

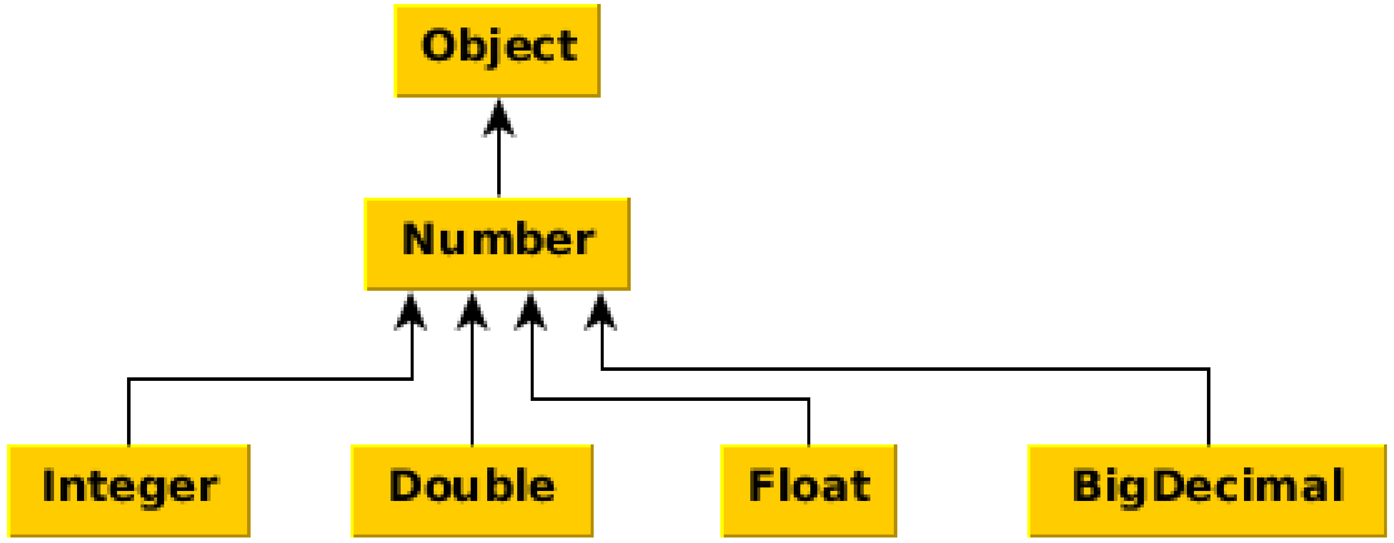
Method Overloading	MethodOverriding
1. It occurs with in the same class.	1. It occurs between two classes i.e., Super class and a subclass.
2. Inheritance is not involved.	2. Inheritance is involved.
3.One method does not hide another.	3. child method hides that of the parent class method.
4.Parameters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

Demo Program: numbers package

LECTURE 6

Numbers in java.lang

- All the numeric classes in **java.lang** extend the **Number** class as well as the ones in **java.math**: **java.math.BigDecimal**, **java.math.BigInteger**. Here is an excerpt:



Numbers in java.lang

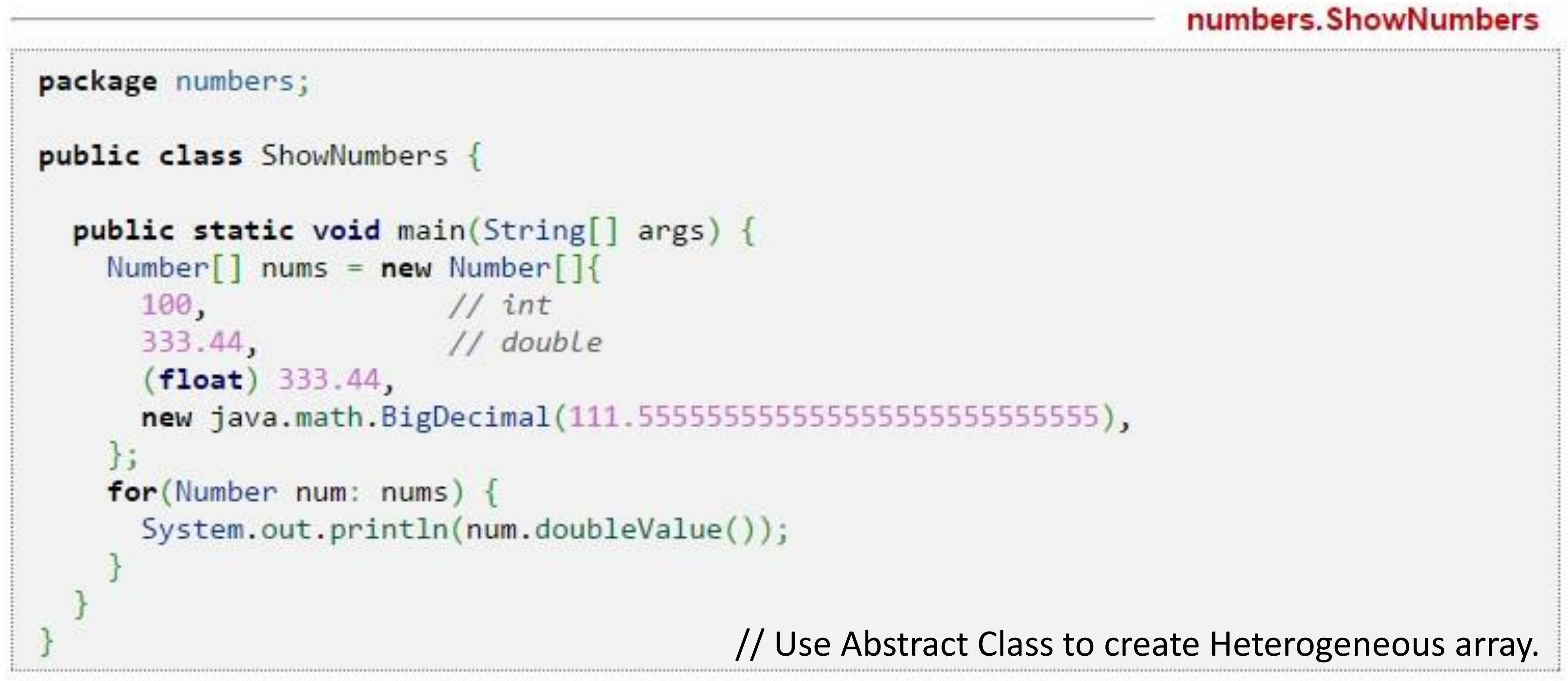
- The Number class is **abstract**. The numeric wrapper classes Integer, Double, etc. are **final**, but not the others, such as BigDecimal are not.

Note:

1. abstract class cannot be instantiated but can be inherited
2. final class cannot be inherited but can be instantiated
3. interface can be inherited but with only methods and wrapping concrete classes up.

- Here is a program which illustrates a polymorphic collection of Numbers to which the function **doubleValue**() is applied

ShowNumbers.java in numbers package



numbers.ShowNumbers

```java
package numbers;

public class ShowNumbers {

  public static void main(String[] args) {
    Number[] nums = new Number[]{
      100,                // int
      333.44,             // double
      (float) 333.44,
      new java.math.BigDecimal(111.555555555555555555555555555),
    };
    for(Number num: nums) {
      System.out.println(num.doubleValue());
    }
  }
}
```

// Use Abstract Class to create Heterogeneous array.

ShowNumbers.java

- You can always interchange, say, 100 with new Integer(100) and 333.44 with new Double(333.44). The transformation of putting a number in its wrapper is called "boxing," and the opposite of taking it out of its wrapper is "unboxing."

Demonstration Program

NUMBERS PACKAGE

Coercion Polymorphism: Casting and instanceof Operator

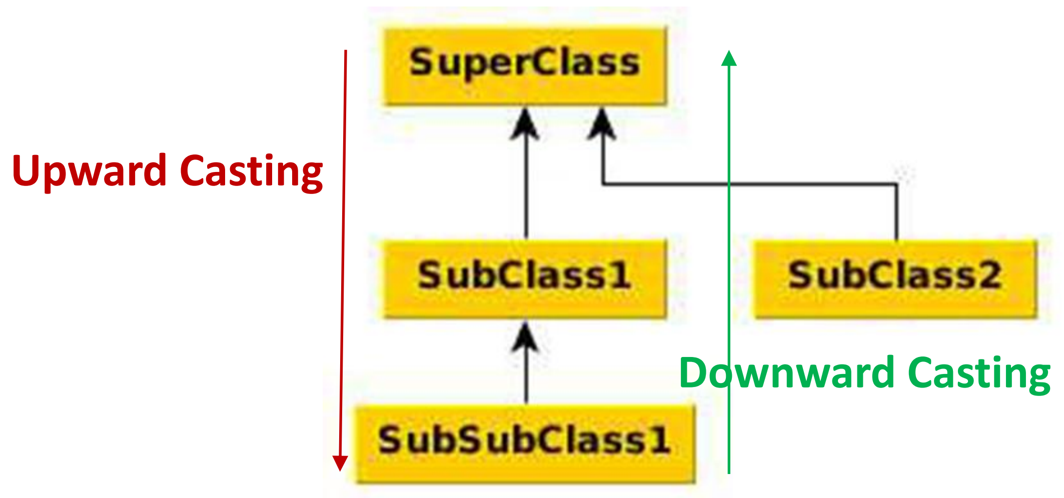
LECTURE 7

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.

Assignment

GEOMETRIC 4 ASSIGNMENT

SUBMIT YOUR PROGRAM TO MOODLE COURSE UPLOAD LINK