UEF4.3 Object Oriented Programming

Mrs Sadeg and Mrs Bousbia
s_sadeg@esi.dz, n_bousbia@esi.dz
Ecole Nationale Supérieure d'Informatique
(ESI

Chapitre III

Part 1. Inheritance

Inheritance

- Suppose we have the *Point* class discussed earlier in the course and we want to create a *ColPoint* class to represent and manipulate colored points in a two-dimensional plane.
- A colored point shares all the properties of a regular point but has an additional attribute: its color. In other words, a colored point is a specialized version of a point.
- Instead of defining a new class independently, we can apply the inheritance principle of Object-Oriented Programming (OOP) to derive *ColPoint* from *Point* and avoid redundant code.
- By deriving *ColPoint* from the Point, it inherits all the attributes and methods of the Point class. Additionally, it will introduce the color attribute to represent the color of a point and the setColor method to assign or update the color of a point

Inheritance

Class Point

```
class Point {
private int x;
private int y;

public void initialize(int x, int y){
          this.x = x;
     this.y = y;}

public void move (int dx, int dy){
          x = x + dx;
     y = y + dy;}

 public void display() {
       System.out.println("coordinates : " + x +"    "+y); }
}
```

Inheritance

Class ColPoint

```
class ColPoint extends Point {

private String color;

public void setColor (String color){

this.color = color;

}
}
```

Tells the compiler that the ColPoint class derives from the Point class

Inheritance

A class using a ColPoint object

```
class Test {
public static void main(String[] args)  {
  ColPoint cp = new ColPoint ();
  cp.display();
  cp.setColor(" blue ");
  cp.display();
  cp.move(2,5);
  cp.display();
 }
}
```

What will the program display?

Inheritance

A class using a ColPoint object

```
class Test {
public static void main(String[] args)  {
  ColPoint cp = new ColPoint ();
  cp.display();
  cp.setColor(" blue ");
  cp.display();
  cp.move(2,5);
  cp.display();
 }
}
```

invokes the default constructor of the ColPoint class

What will the program display?

Inheritance

A class using a ColPoint object

```
class Test {
public static void main(String[] args)  {
  ColPoint cp = new ColPoint ();
  cp.display();
  cp.setColor(" blue ");
  cp.display();
  cp.move(2,5);
  cp.display();
 }
}
```

invokes the default constructor of the ColPoint class

invokes the method move of the Point class

What will the program display?

Inheritance

A class using a ColPoint object

```
class Test {
public static void main(String[] args)  {
  ColPoint cp = new ColPoint ();
  cp.display();
  cp.setColor(" blue ");
  cp.display();
  cp.move(2,5);
  cp.display();
 }
}
```

invokes the default constructor of the ColPoint class

invokes the method move of the Point class

What will the program display?

Coordinates: 0 0
Coordinates: 0 0
Coordinates: 2 5

Inheritance

A class using a ColPoint object

```
class Test {
public static void main(String[] args)  {
  ColPoint cp = new ColPoint ();
  cp.display();
  cp.setColor(" blue ");
  cp.display();
  cp.move(2,5);
  cp.display();
 }
}
```

invokes the default constructor of the ColPoint class

invokes the method move of the Point class

What will the program display?

Coordinates: 0 0
Coordinates: 0 0
Coordinates: 2 5

An object of a derived class accesses the public members of its base class

Derived class access to base class members

We want to add a method named *displayAll* that displays all the attributes

```
class ColPoint extends Point{

private String color;

public void setColor (String color){
 this.color = color;
}
public void displayAll(){
 System.out.println(" Coordinates : " + x +" "+ y);
 System.out.println(" Coordinates: " + color );
 }
}
 Is this correct ?
```

Derived class access to base class members

We want to add a method named *displayAll* that displays all the attributes

```
class ColPoint extends Point{

private String color;

public void setColor (String color){
 this.color = color;
}
public void displayAll(){
 System.out.println(" Coordinates : " + x +" "+ y);
 System.out.println(" Coordinates: " + color );
 }
}
 Is this correct ?
```

No!

Derived class access to base class members

We want to add a method named *displayAll* that displays all the attributes

```
class ColPoint extends Point{

private String color;

public void setColor (String color){
 this.color = color;
}
public void displayAll(){
 System.out.println(" Coordinates : " + x +" "+ y);
 System.out.println(" Coordinates: " + color );
 }
}
 Is this correct ?
```

No!

A method of a derived class does not access the private members of its base class.

Derived class access to base class members

Solution

```
class ColPoint extends Point{

private String color;

public void setColor (String color){
this.color = color;
}


public void displayAll(){
display();
System.out.println(" color: " + color );
}
}
```

Derived class access to base class members

Solution

```
class ColPoint extends Point{

private String color;

public void setColor (String color){
this.color = color;
}


public void displayAll(){
display();
System.out.println(" color: " + color );
}
}
```

invokes the public display method of the *Point* class

Derived class access to base class members

Other solution

```
class Point {
protected int x;
protected int y;
// the methods
}
class ColPoint extends Point{
private String color
public void displayAll(){
System.out.println("coordinates : "  + x + "    "  + y);
System.out.println(" color: " + color );
}
}
```

protected means that this attribute is accessible from the methods of derived classes.

Derived class access to base class members

Other solution

```
class Point {
protected int x;
protected int y;
// the methods
}
class ColPoint extends Point{
private String color
public void displayAll(){
System.out.println("coordinates : "  + x + "    "  + y);
System.out.println(" color: " + color );
}
}
```

protected means that this attribute is accessible from the methods of derived classes.

An object of a derived class accesses the protected members of its base class

Construction of derived objects

- Reminder:
  - In the case of a simple (non-derived) class, an object is created using *new* by calling the constructor with the required signature (number and types of arguments).
  - If no suitable constructor is available, you'll get a compilation error, unless the class has no constructor at all. In this case, a default constructor is used.
- How are objects constructed in the case of a derived class?

Construction of derived objects

Case 1 Neither the base class nor the derived class has a constructor

```
class Point{
private int x;
private int y;

// No constructor

// other methods
}
```

```
class ColPoint extends Point{
private String color;

// No constructor

// other methods
}
```

Construction of derived objects

Case 1 Neither the base class nor the derived class has a constructor

```
class Point{
private int x;
private int y;

// No constructor

// other methods
}
```

```
class ColPoint extends Point{
private String color;

// No constructor

// other methods
}
```

The default constructor of ColPoint invokes the default constructor of Point

Construction of derived objects

Case 2: both base class and derived class have constructors with arguments

```
class Point {
private int x;
private int y;

public Point(int x, int y){
this.x = x;
This.y = y;}
// other methods
}
```

```
class ColPoint extends Point{
private String color;

public ColPoint(int x, int y,
 String color) {
super(x,y);
this.color = color; }
// other methods
}
```

Construction of derived objects

Case 2: both base class and derived class have constructors with arguments

The ColPoint constructor :

1. invokes the Point constructor to initialize x and y
2. Initializes color attribute

```
class ColPoint extends Point{
private String color;

public ColPoint(int x, int y,
 String color) {
super(x,y);
this.color = color; }
// other methods
}
```

Construction of derived objects

Case 2: both base class and derived class have constructors with arguments

The ColPoint constructor :

1. invokes the Point constructor to initialize x and y
2. Initializes color attribute

```
class ColPoint extends Point{
private String color;

public ColPoint(int x, int y,
 String color) {
super(x,y);
this.color = color; }
// other methods
}
```

When a constructor of a derived class invokes the constructor of its base class, it uses the *super* keyword, which must always be the first statement in the constructor.

Construction of derived objects

Case 3. The base class has no constructor and the derived class has a constructor

```
class Point{
private int x;
private int y;

// Aucun constructeur

// les autres méthodes
}
```

```
class ColPoint extends Point{
private String color;

public ColPoint(int x, int y, String
 color) {
super();
this.color = color;
}

// les autres méthodes
}
```

Construction of derived objects

Case 3. The base class has no constructor and the derived class has a constructor

```
class Point{
private int x;
private int y;

// Aucun constructeur

// les autres méthodes
}
```

```
class ColPoint extends Point{
private String color;

public ColPoint(int x, int y, String
 color) {
super();
this.color = color;
}

// les autres méthodes
}
```

You can use super or not, it will be automatically inserted by the compiler

Construction of derived objects

Case 4. The derived class has no constructor and the base class has a constructor with arguments

```
class Point{
private int x;
private int y;
public Point(int x,int y){
this.x = x;
this.y = y; }

// other methods
}
```

```
class ColPoint extends Point{
private String color;

// No constructor

// other methods
}
```

Construction of derived objects

Case 4. The derived class has no constructor and the base class has a constructor with arguments

```
class Point{
private int x;
private int y;
public Point(int x,int y){
this.x = x;
this.y = y; }

// other methods
}
```

```
class ColPoint extends Point{
private String color;

// No constructor

// other methods
}
```

Compilation error!
The default constructor of ColPoint tries to call a constructor without arguments in the Point class. However, since the Point class only has a constructor with arguments, the default constructor can no longer be used

Construction des objets dérivés

Case 4. The derived class has no constructor and the base class has a constructor with arguments

```
class Point{
private int x;
private int y;

public Point(){}

public Point(int x,int y){
this.x = x;
this.y = y;
}
// other methods
}
```

```
class ColPoint extends Point{
private String color;

// No constructor

// other methods
}
```

Construction des objets dérivés

Case 4. The derived class has no constructor and the base class has a constructor with arguments

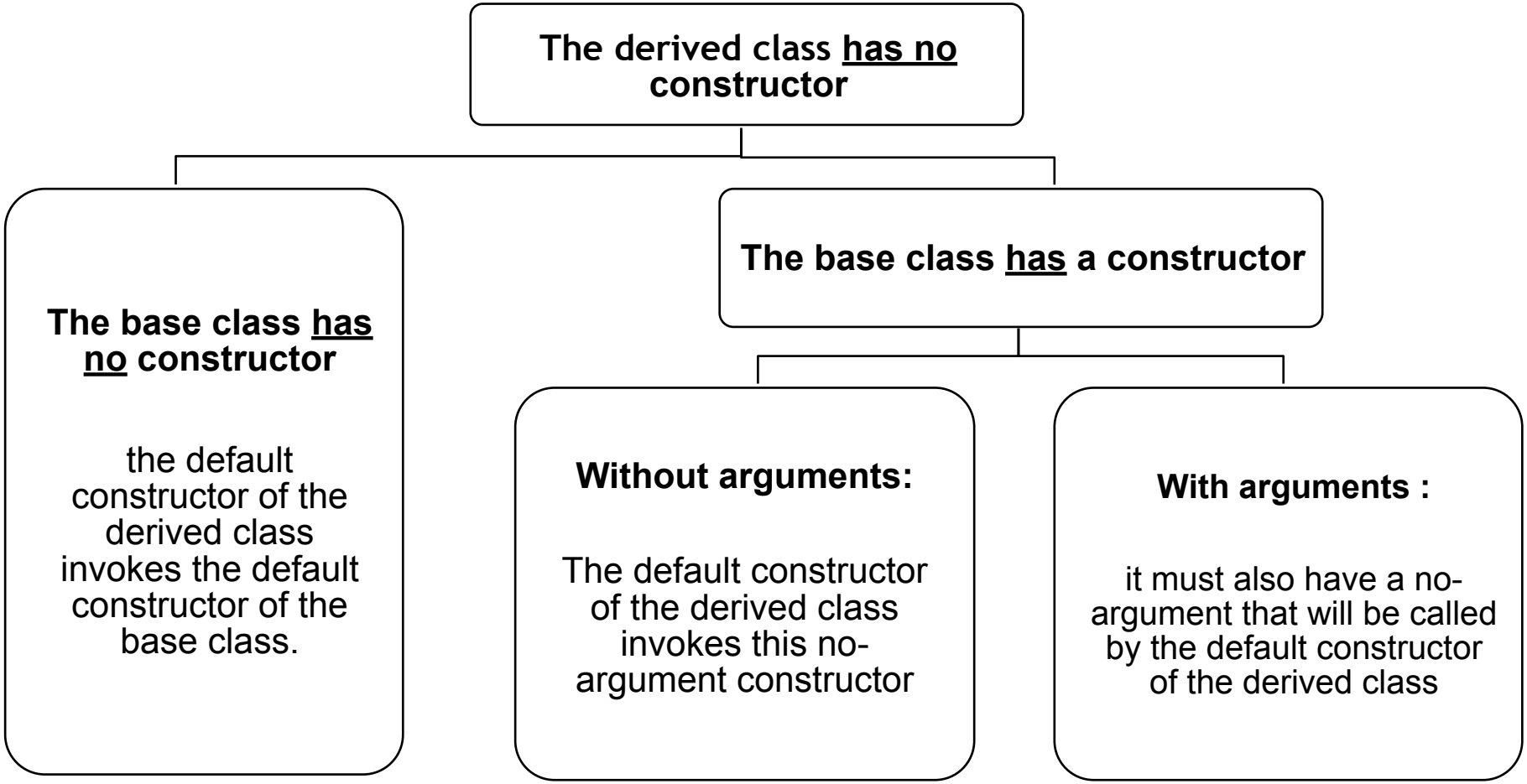
```
class Point{
private int x;
private int y;

public Point(){}

public Point(int x,int y){
this.x = x;
this.y = y;
}
// other methods
}
```

```
class ColPoint extends Point{
private String color;

// No constructor

// other methods
}
```

Solution: add a no-argument constructor in Point class so that the default constructor of ColPoint can call it

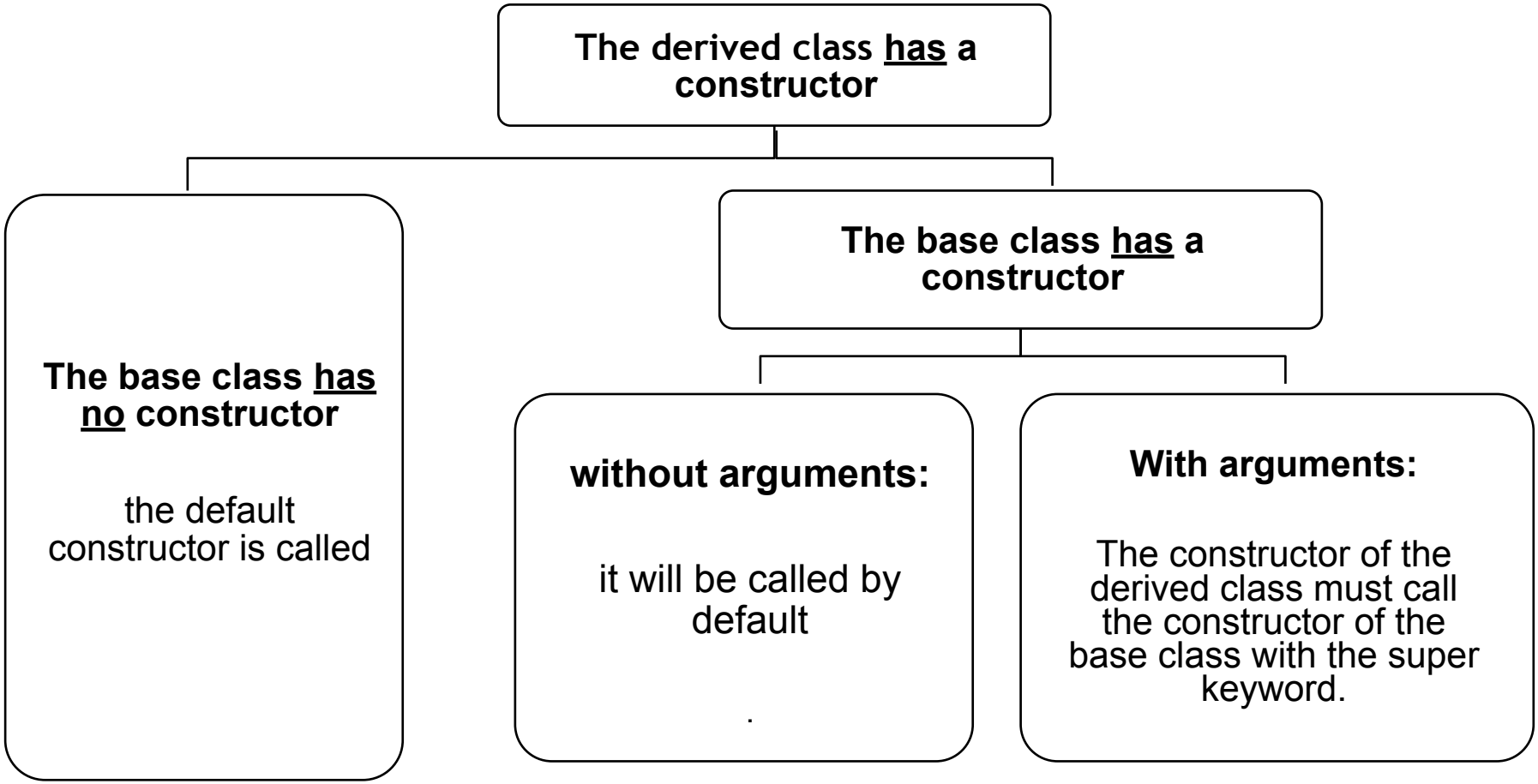
Construction of derived objects

Synthesis (1/2)



Construction of derived objects

Synthesis (2/2)



Initializing a derived object

- In a simple (non-derived) class, object attributes are initialized in the following order
  1. default initialization (implicit)
  2. explicit initialization (if any)
  3. Execution of constructor body

Exemple:

```
class A{
private int n= 10;
private int p;
public A (){…}
……
}
…
A a = new A();
```

1. Implicit initialization of attributes **n** and **p** of object **a** to **0**;
2. Explicit initialization of **n** to the value defined in its declaration (i.e., 10),
3. Execution of the constructor's instructions.

Initializing a derived object

In a derived class (class B extends A {…}), the creation of an object proceeds as follows:

1. Memory allocation for an object of type **B**
2. Implicit initialization of all attributes in **B**
3. Explicit initialization of inherited attributes from **A** (if present)
4. Execution of the constructor body of **A**
5. Explicit initialization of attributes that are specific to **B** (if present)
6. Execution of the constructor body of **B**

Exercise 1: Read the following program

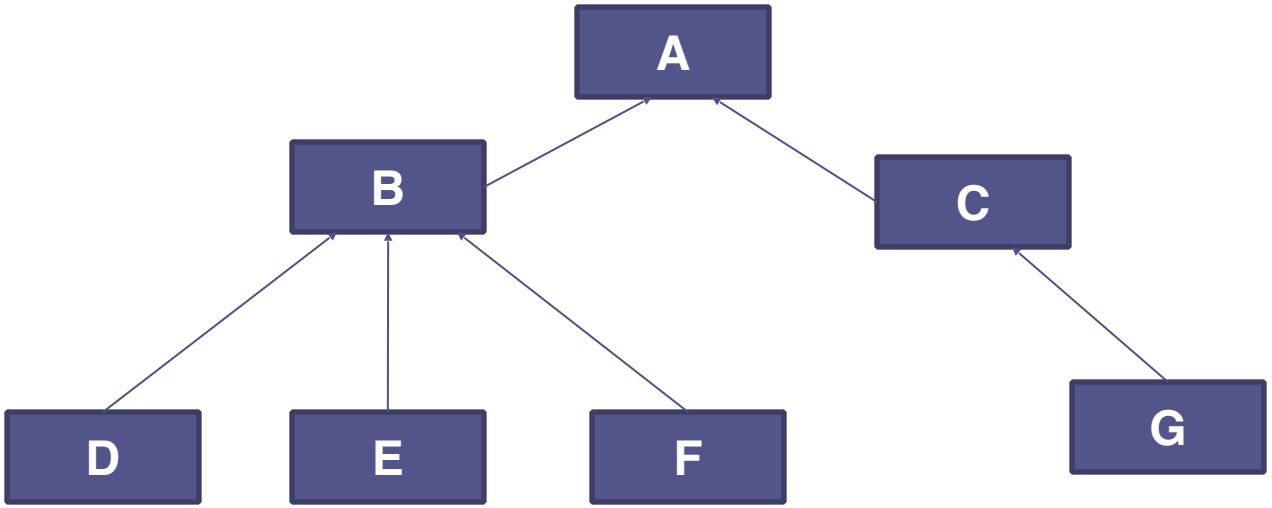
```
class A{
private int n ; private int p=10 ;
public A (int n){
System.out.println ("Entry of constructor A-
value of attribute n=" + n) ;
System.out.println (" Entry of constructor A -
value of attribute p=" + p) ;
this.n = n ;
System.out.println (" Exit of Constructor A -
value of attribute n=" + n) ;
System.out.println ("Exit Constructor A -
value of attribute p=" + p) ;}
}
public class EX1{
public static void main (String args[]){
A a = new A(5) ;
B b = new B(7, 3) ;}
}
```

```
class B extends A {
private int q=25 ;
public B (int n, int pp){
super (n) ;
System.out.println (" Entry of constructor B-
value of attribute n=" + n) ;
System.out.println (" Entry of constructor B-
value of attribute p=" + p) ;
System.out.println (" Entry of constructor B-
attribute value q=" + q) ;
p = pp ; q = 2*n ;
System.out.println (" Exit of Constructor B -
value of attribute n=" + n) ;
System.out.println (" Exit of Constructor B -
value of attribute p=" + p) ;
System.out.println (" Exit of Constructor B -
value of attribute q=" + q) ;}
}
```

1. Are there any errors in this program? Which ones?
2. How can you correct them?
3. What does the program display after correcting the code?

Successive derivations

- In Java, multiple inheritance does not exist. A class can derive from one and only one class.
- Multiple different classes can be derived from the same base class. Consequently, a derived class can serve as the base class for another derived class, forming a hierarchy of inheritance.



Overloading and inheritance

A derived class can **overload** a method from a base class (direct or indirect). The new method will only be accessible by the derived class or its descendants (its derived class) but not by its ancestors.

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public void f (float x) {….}
…
}
A a; B b;
int n, float x;
```

Overloading and inheritance

A derived class can **overload** a method from a base class (direct or indirect). The new method will only be accessible by the derived class or its descendants (its derived class) but not by its ancestors.

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public void f (float x) {….}
…
}
A a; B b;
int n, float x;
```

```
a.f(n);

a.f(x);

b.f(n);

b.f(x);
```

Overloading and inheritance

A derived class can **overload** a method from a base class (direct or indirect). The new method will only be accessible by the derived class or its descendants (its derived class) but not by its ancestors.

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public void f (float x) {….}
…
}
A a; B b;
int n, float x;
```

```
a.f(n);   // invokes f of A
a.f(x);
b.f(n);
b.f(x);
```

Overloading and inheritance

A derived class can **overload** a method from a base class (direct or indirect). The new method will only be accessible by the derived class or its descendants (its derived class) but not by its ancestors.

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public void f (float x) {….}
…
}
A a; B b;
int n, float x;
```

```
a.f(n);   // invokes f of A
a.f(x);   // Compilation error!
b.f(n);
b.f(x);
```

Overloading and inheritance

A derived class can **overload** a method from a base class (direct or indirect). The new method will only be accessible by the derived class or its descendants (its derived class) but not by its ancestors.

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public void f (float x) {….}
…
}
A a; B b;
int n, float x;
```

```
a.f(n);   // invokes f of A
a.f(x);   // Compilation error!
b.f(n);   // invokes f of A
b.f(x);
```

Overloading and inheritance

A derived class can **overload** a method from a base class (direct or indirect). The new method will only be accessible by the derived class or its descendants (its derived class) but not by its ancestors.

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public void f (float x) {….}
…
}
A a; B b;
int n, float x;
```

```
a.f(n);   // invokes f of A
a.f(x);   // Compilation error!
b.f(n);   // invokes f of A
b.f(x);   // invokes f of B
```

Method overriding

```
class Point{

private  int  x, y ;

public  void display(){
System.out.println("Coordinates"
+ x + " " + y);
}

}
```

```
class ColPoint extends Point{

private String color;

public void setColor (String
color){
this.color = color;
}

public void displayAll(){
display();
System.out.println(" color: "
  + color );
}

}
```

Method overriding

```
class Point{

private  int  x, y ;

public  void display(){
System.out.println("Coordinates"
+ x + " " + y);
}

}
```

```
class ColPoint extends Point{

private String color;

public void setColor (String
color){
this.color = color;
}

public void displayAll(){
display();
System.out.println(" color: "
  + color );
}

}
```

Since both *display* and *displayAll* methods serve to display the object's state it is logical to use the same method name.

Method overriding

```
class Point{
private  int  x, y ;
public void display(){
System.out.println(" Coordinate"
+ x+ " " + y);}
}
```

```
class ColPoint extends
Point{

private String color;

public void setColor (String
color){
this.color = color;
}
public void display(){
super.display ();
System.out.println(" color:
" + color );
}
}
```

Method overriding

```
class Point{
private  int  x, y ;
public void display(){
System.out.println(" Coordinate"
+ x+ " " + y);}
}
```

The super keyword must be used; otherwise, it would result in a recursive call to the display method of the ColPoint class.

```
class ColPoint extends
Point{

private String color;

public void setColor (String
color){
this.color = color;
}
public void display(){
super.display ();
System.out.println(" color:
" + color );
}
}
```

Method overriding

```
class Point{
private  int  x, y ;
public void display(){
System.out.println(" Coordinate"
+ x+ " " + y);}
}
```

The super keyword must be used; otherwise, it would result in a recursive call to the display method of the ColPoint class.

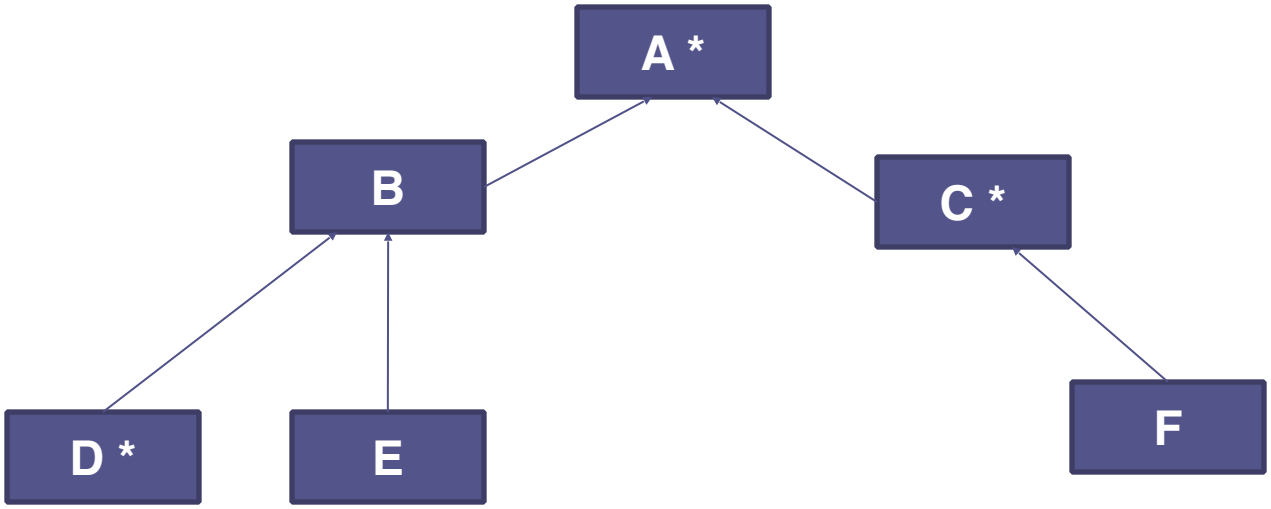
```
class ColPoint extends
Point{

private String color;

public void setColor (String
color){
this.color = color;
}
public void display(){
super.display ();
System.out.println(" color:
" + color );
}
}
```

the overriding method in the derived class replaces the corresponding overridden method in the base class.

Method overriding and successive derivations

Let *f* be a method defined in A and overridden in class C and D.

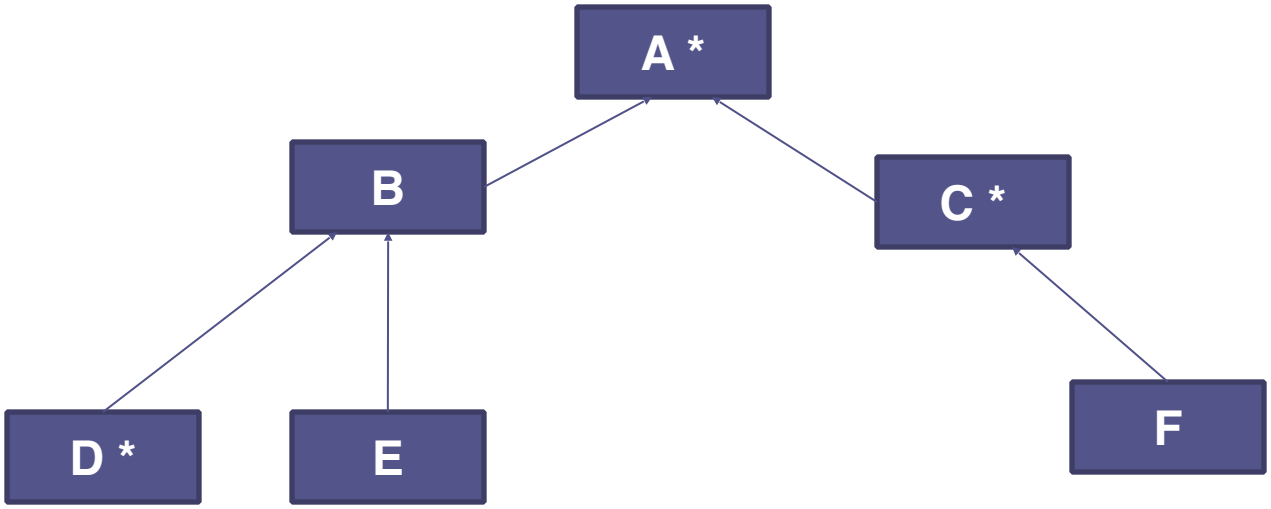
- Calling *f* in A invokes *f* from ...
- Calling *f* in B invokes *f* from ...
- Calling *f* in C invokes *f* from ...
- Calling *f* in D invokes *f* from ...
- Calling *f* in E invokes *f* from ...
- Calling *f* in F invokes *f* from ...



Method overriding and successive derivations

Let *f* be a method defined in A and overridden in class C and D.

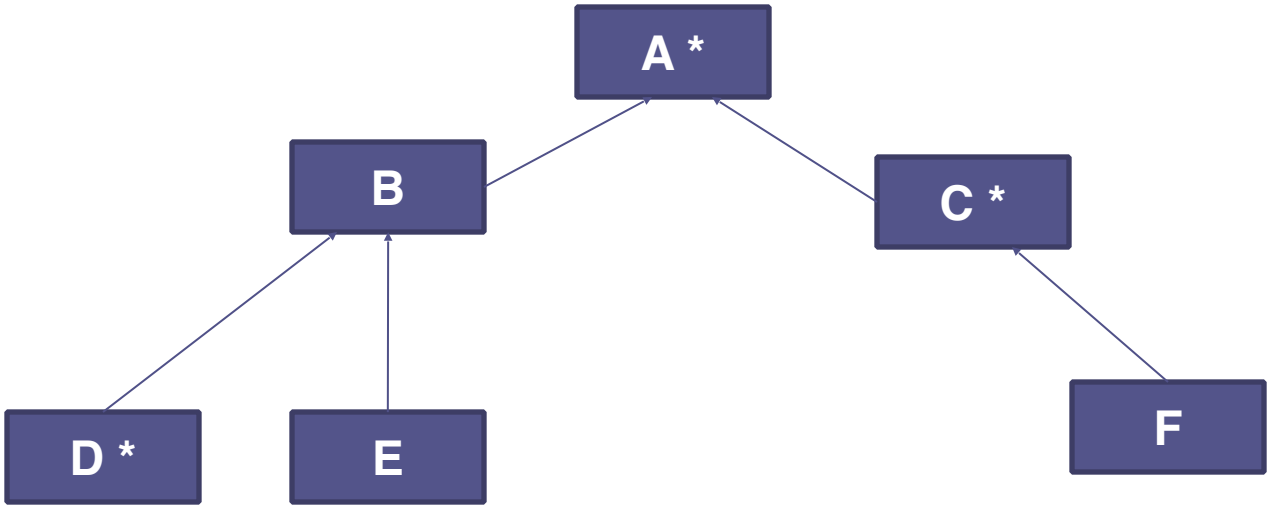
- Calling *f* in A invokes *f* from A
- Calling *f* in B invokes *f* from ...
- Calling *f* in C invokes *f* from ...
- Calling *f* in D invokes *f* from ...
- Calling *f* in E invokes *f* from ...
- Calling *f* in F invokes *f* from ...



Method overriding and successive derivations

Let *f* be a method defined in A and overridden in class C and D.

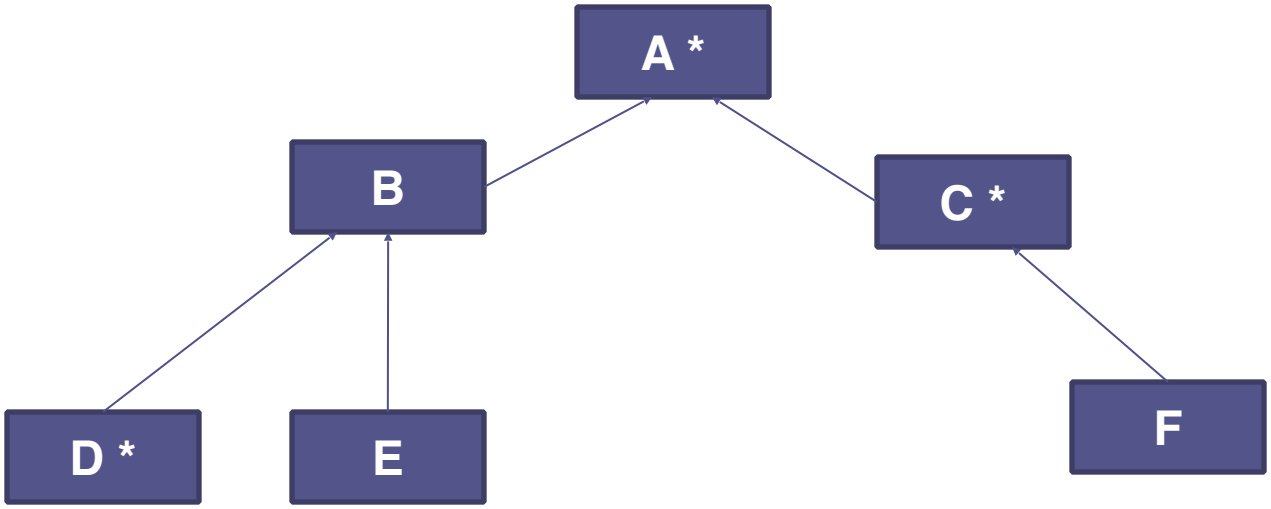
- Calling *f* in A invokes *f* from A
- Calling *f* in B invokes *f* from A
- Calling *f* in C invokes *f* from ...
- Calling *f* in D invokes *f* from ...
- Calling *f* in E invokes *f* from ...
- Calling *f* in F invokes *f* from ...



Method overriding and successive derivations

Let *f* be a method defined in A and overridden in class C and D.

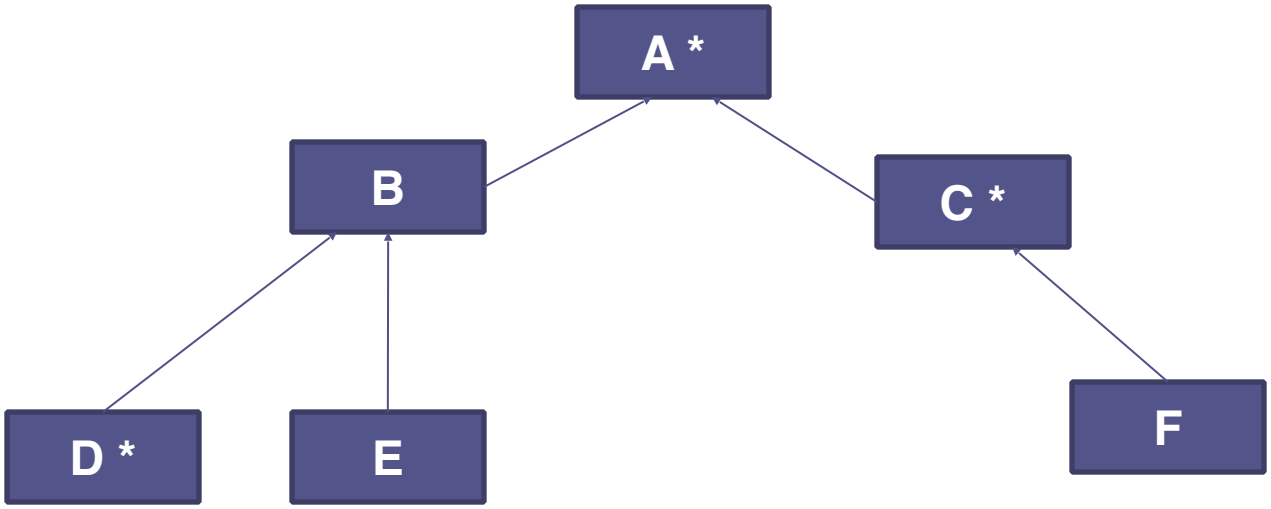
- Calling *f* in A invokes *f* from A
- Calling *f* in B invokes *f* from A
- Calling *f* in C invokes *f* from C
- Calling *f* in D invokes *f* from ...
- Calling *f* in E invokes *f* from ...
- Calling *f* in F invokes *f* from ...



Method overriding and successive derivations

Let *f* be a method defined in A and overridden in class C and D.

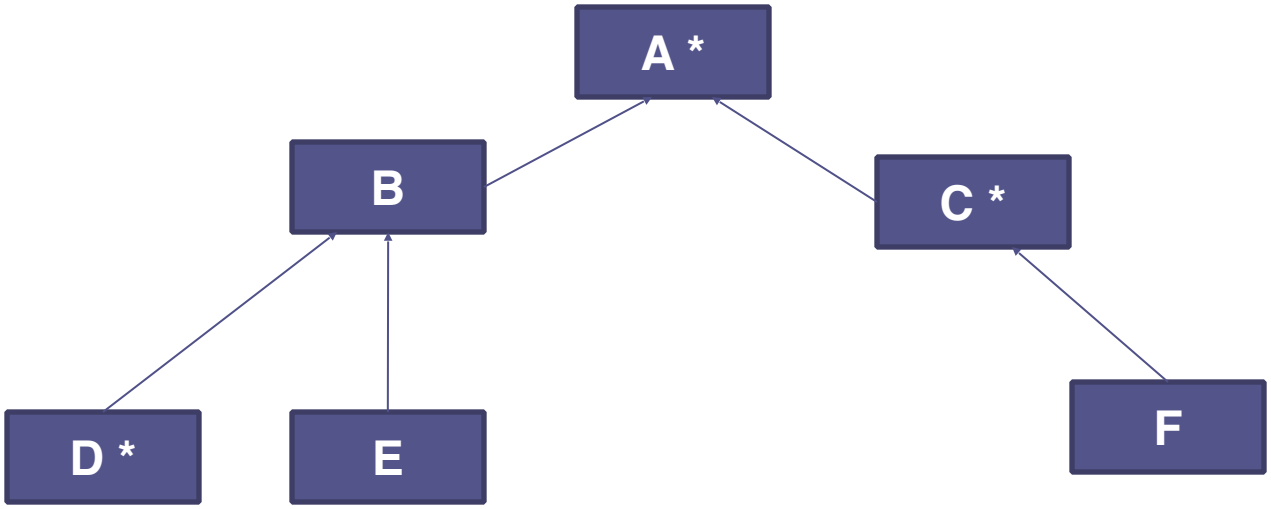
- Calling *f* in A invokes *f* from A
- Calling *f* in B invokes *f* from A
- Calling *f* in C invokes *f* from C
- Calling *f* in D invokes *f* from D
- Calling *f* in E invokes *f* from ...
- Calling *f* in F invokes *f* from ...



Method overriding and successive derivations

Let *f* be a method defined in A and overridden in class C and D.

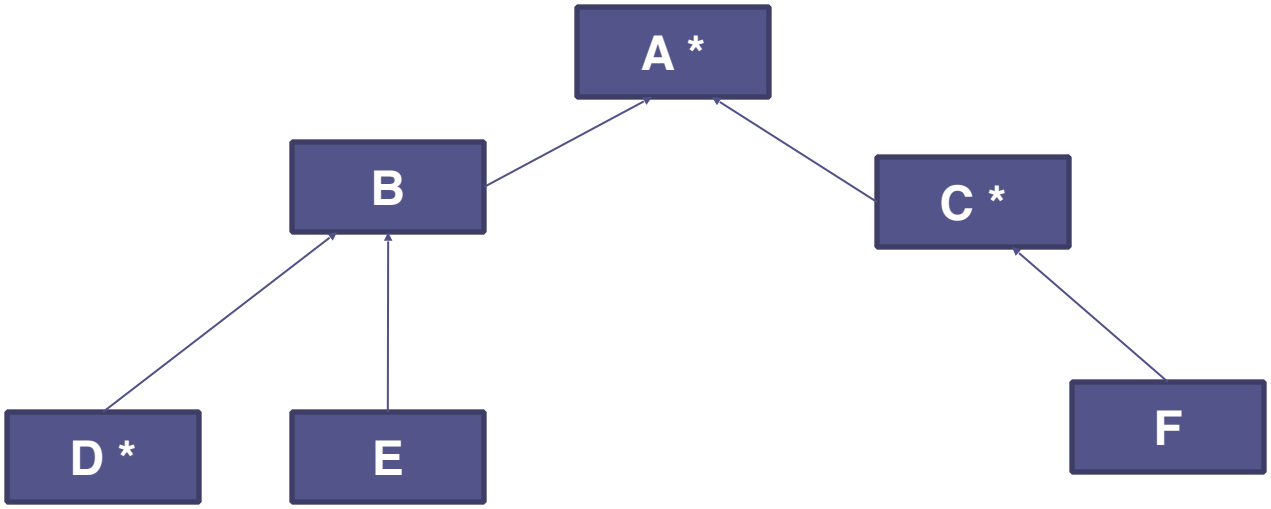
- Calling *f* in A invokes *f* from A
- Calling *f* in B invokes *f* from A
- Calling *f* in C invokes *f* from C
- Calling *f* in D invokes *f* from D
- Calling *f* in E invokes *f* from A
- Calling *f* in F invokes *f* from …



Method overriding and successive derivations

Let *f* be a method defined in A and overridden in class C and D.

- Calling *f* in A invokes *f* from A
- Calling *f* in B invokes *f* from A
- Calling *f* in C invokes *f* from C
- Calling *f* in D invokes *f* from D
- Calling *f* in E invokes *f* from A
- Calling *f* in F invokes *f* from C



Simultaneous use of overloading and overriding

```
class A
{public void f (int n) {…}
 public void f (float x){…}
….
}
Class B extends A
{public void f (int n){….}
 public void f (double y){….}
…
}
A a; B b;
int n; float x; double y;
…
```

Simultaneous use of overloading and overriding

```
class A
{public void f (int n) {…}
 public void f (float x){…}
….
}
Class B extends A
{public void f (int n){….}
 public void f (double y){….}
…
}
A a; B b;
int n; float x; double y;
…
```

```
a.f(n);  //
a.f(x); //
a.f(y); //
b.f(n); //
b.f(x); //
b.f(y);//
```

Simultaneous use of overloading and overriding

```
class A
{public void f (int n) {…}
 public void f (float x){…}
….
}
Class B extends A
{public void f (int n){….}
 public void f (double y){….}
…
}
A a; B b;
int n; float x; double y;
…
```

```
a.f(n);  //
a.f(x); //
a.f(y); //
b.f(n); //
b.f(x); //
b.f(y);//
```

invokes f (int) of A

Simultaneous use of overloading and overriding

```
class A
{public void f (int n) {…}
 public void f (float x){…}
….
}
Class B extends A
{public void f (int n){….}
 public void f (double y){….}
…
}
A a; B b;
int n; float x; double y;
…
```

```
a.f(n);  //
a.f(x); //
a.f(y); //
b.f(n); //
b.f(x); //
b.f(y);//
```

invokes f (int) of A

invokes f (float) of A

Simultaneous use of overloading and overriding

```
class A
{public void f (int n) {…}
 public void f (float x){…}
….
}
Class B extends A
{public void f (int n){….}
 public void f (double y){….}
…
}
A a; B b;
int n; float x; double y;
…
```

```
a.f(n);  //
a.f(x); //
a.f(y); //
b.f(n); //
b.f(x); //
b.f(y);//
```

invokes f (int) of A

invokes f (float) of A

Compilation error!

Simultaneous use of overloading and overriding

```
class A
{public void f (int n) {…}
 public void f (float x){…}
….
}
Class B extends A
{public void f (int n){….}
 public void f (double y){….}
…
}
A a; B b;
int n; float x; double y;
…
```

```
a.f(n);  //
a.f(x); //
a.f(y); //
b.f(n); //
b.f(x); //
b.f(y);//
```

invokes f (int) of A

invokes f (float) of A

Compilation error!

invokes f (int) of B

Simultaneous use of overloading and overriding

```
class A
{public void f (int n) {…}
 public void f (float x){…}
….
}
Class B extends A
{public void f (int n){….}
 public void f (double y){….}
…
}
A a; B b;
int n; float x; double y;
…
```

```
a.f(n);  //
a.f(x); //
a.f(y); //
b.f(n); //
b.f(x); //
b.f(y);//
```

- a.f(n); // — invokes f (int) of A
- a.f(x); // — invokes f (float) of A
- a.f(y); // — Compilation error!
- b.f(n); // — invokes f (int) of B
- b.f(x); // — invokes f (float) of A

Simultaneous use of overloading and overriding

```
class A
{public void f (int n) {…}
 public void f (float x){…}
….
}
Class B extends A
{public void f (int n){….}
 public void f (double y){….}
…
}
A a; B b;
int n; float x; double y;
…
```

a.f(n); //	invokes f (int) of A
a.f(x); //	invokes f (float) of A
a.f(y); //	Compilation error!
b.f(n); //	invokes f (int) of B
b.f(x); //	invokes f (float) of A
b.f(y);//	invokes f (double) of B

Simultaneous use of overloading and overriding

Simultaneous use of overloading and overriding

The coexistence of overloading and overriding can lead to complex situations, which would be avoided through careful class design.

Overriding constraints (1/3)

Signature

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public float f (float x) {….}
…
}
```

Overriding or overloading ?

Overriding constraints (1/3)

Signature

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public float f (float x) {….}
…
}
```

Overriding or overloading ?

Overriding constraints (2/3)

Return value

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public float f (int n)  {….}
…
}
```

Overriding or overloading ?

Overriding constraints (2/3)

Return value

```
class A{
public void f (int n) {…}
…..
}
Class B extends A{
public float f (int n)  {….}
…
}
Overriding or overloading ?
```

Neither one nor the other!
It's not an overloading of the f method, since the signature is the same.
Nor is it an overriding because the return types are different
compilation error!

Overriding constraints (3/3)

Access rights

```
class A{
public void f (int n) {…}
…
}
```

```
class B extends A{
private void f (int n) {….}
…
}
```

Overriding constraints (3/3)

Access rights

```
class A{
public void f (int n) {…}
…
}
```

```
class B extends A{
private void f (int n) {….}
…
}
```

Compilation error !

If this was accepted, an object of class A would have access to the method f, while an object of class B would no longer have access to it.Class B would break the Contract established by class A

Overriding constraints (3/3)

Access rights

```
class A{
public void f (int n) {…}
…
}
```

```
class B extends A{
private void f (int n) {….}
…
}
```

Compilation error !

If this was accepted, an object of class A would have access to the method f, while an object of class B would no longer have access to it.Class B would break the Contract established by class A

Overriding a method must not reduce its accessibility. On the other hand, it can increase it.

Duplicate attributes

- A derived class can declare an attribute with the same name as one in an ancestor class.
- The new attribute coexists with the original one, and both can be accessed. The *super* keyword is used to refer to the ancestor class's attribute.

Exercise 2

```
class A {
protected int a = 5;
public A(int a) { this.a = a;}
public void displayClass() {System.out.println("class A");}
public void displayVariables() {
System.out.println("a = " + a);}
 }
class B extends A {
protected int b = 6;
public B(int b){super(2 * b);a = b;}
public void displayClass() {
super.displayClass();
System.out.println("class B");}
public void displayVariables()
{ super.displayVariables();
System.out.println("b = " + b);}
}
```

```
class C extends B {
protected int b = 7; protected int c = 8;
public C(int c) { super(3 * c);b = c;}
public void displayClass() {
super.displayClass(); System.out.println("classC");}
public void displayVariables()
{ super.displayVariables();
System.out.println("c = " + c);}
}
class Alphabet {
public static void main(String args[]) {
 A[] as = new A[3];
as[0] = new A(1);
as[1] = new B(2);
as[2] = new C(3);
for (int i = 0; i < as.length; i++) { as[i].displayClass();
System.out.println("-----");
}
for (int i = 0; i < as.length; i++) { as[i].displayVariables();
System.out.println("-----");}
} }
```