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();
 }
}
```

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 ? **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 );
}
}
```

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.

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
}
```

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

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
}
```

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
}
```

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

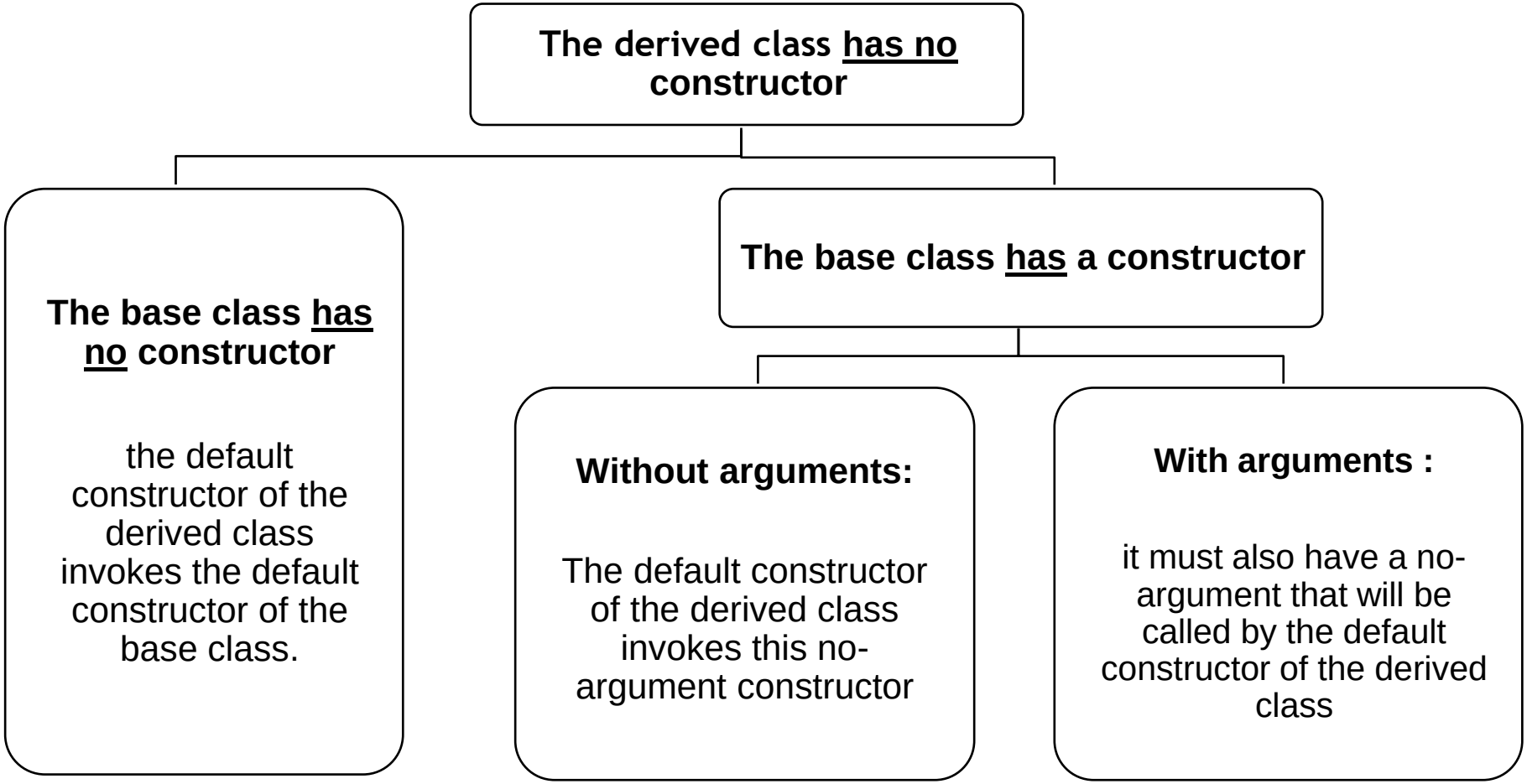
```
class Point{
private int x;
private int y;

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
}
```

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

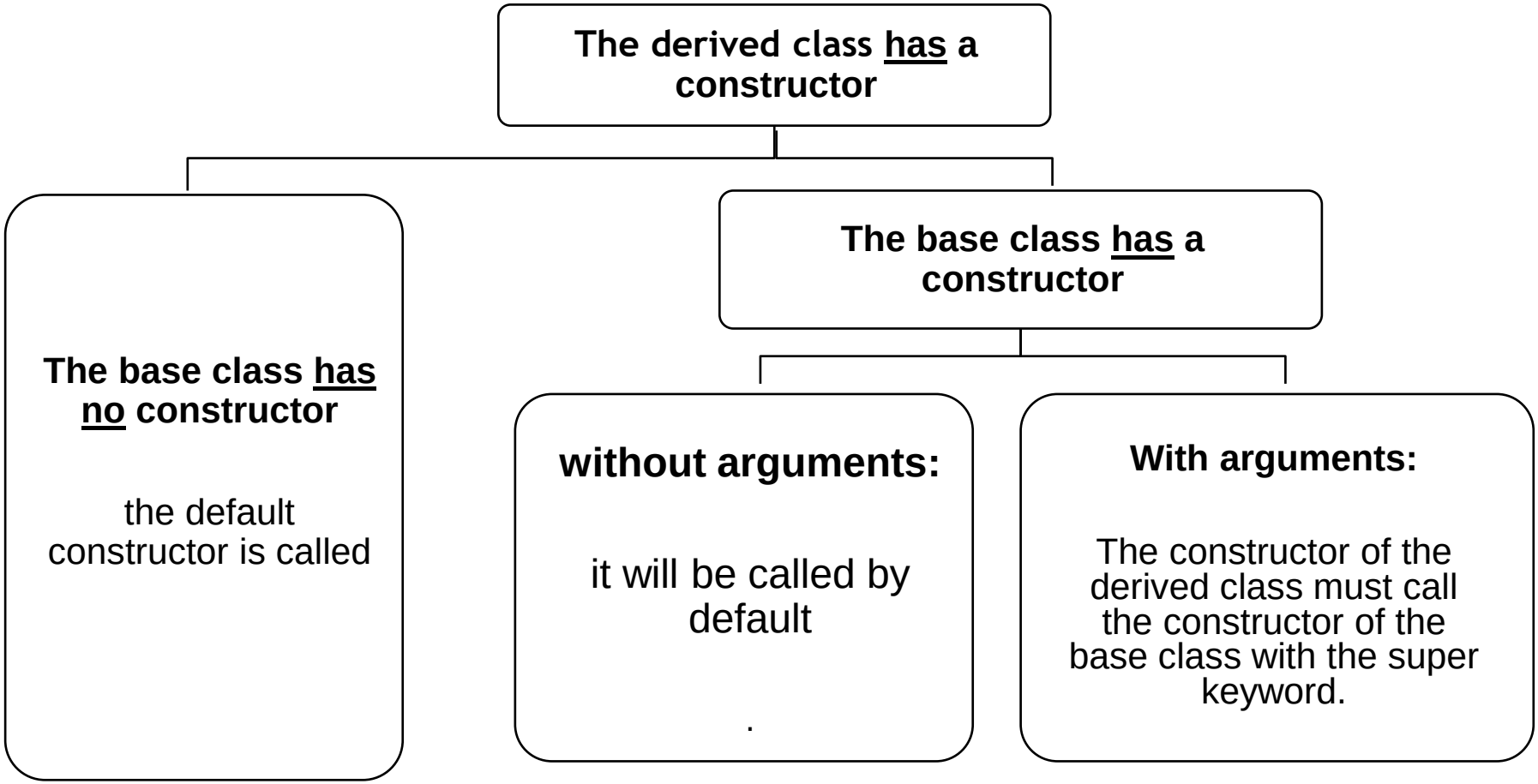
Construction of derived objects

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) ;}
}
```

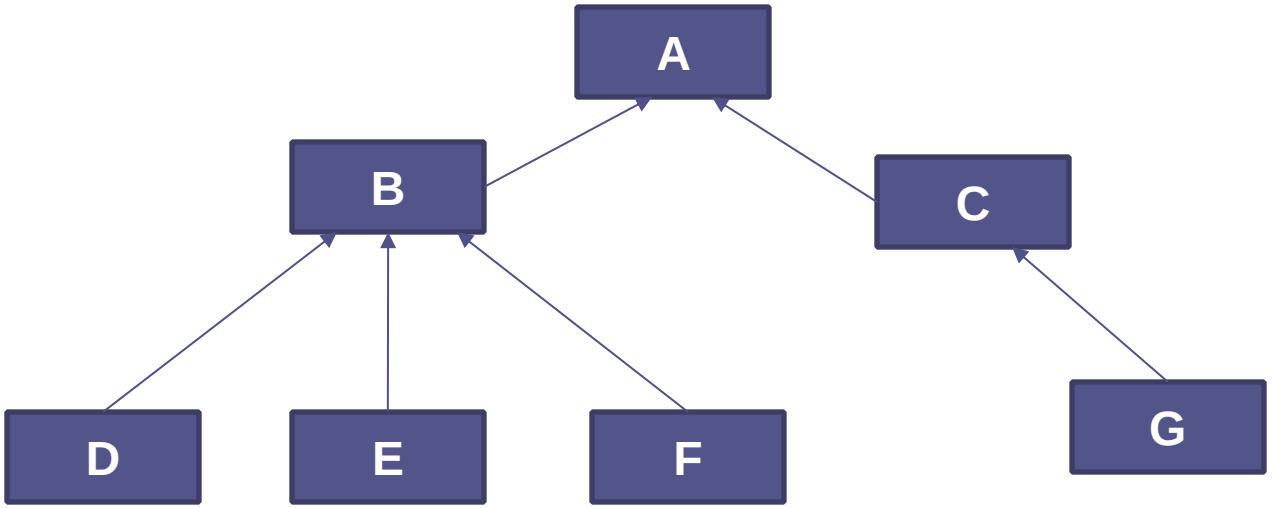
```
class B extends A {
private int q=25 ;
public B (int n, int pp){
super (n) ;
System.out.println (" Entry of constructor
B- value of attribute n=" + n) ;
System.out.println (" Entry of constructor
B- value of attribute p=" + p) ;
System.out.println (" Entry of constructor
B-  attribute value q=" + 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. How Are there any errors in this program? Which ones?
2. 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;
```

```
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 );
}

}
```

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);}
}
```

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

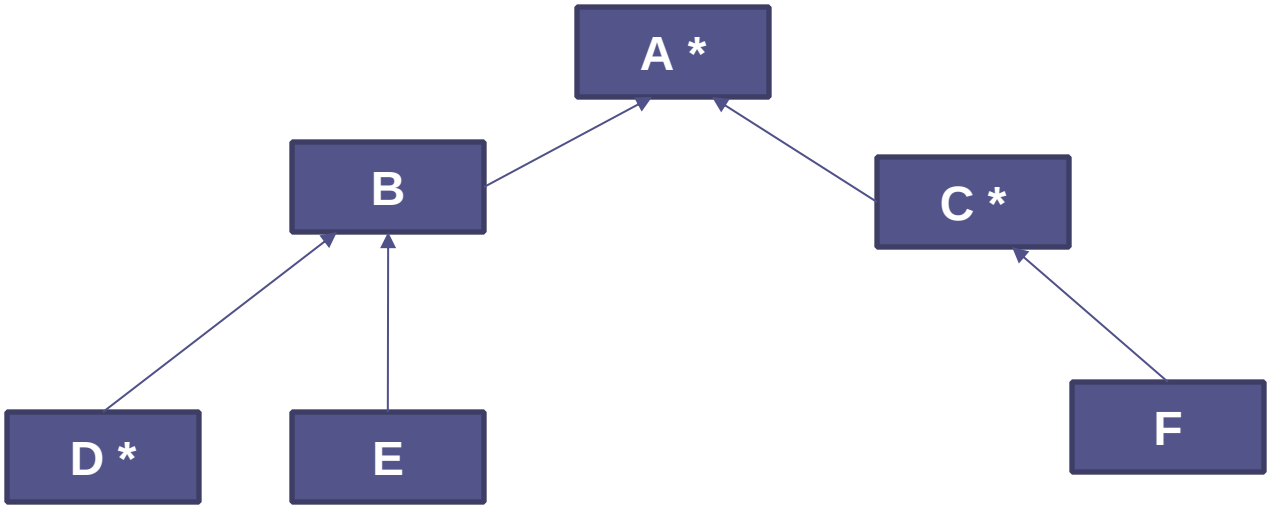
```
class ColPoint extends
Point{

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 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;
…
```

```
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); //	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

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 (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) {….}
…
}
```

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.

Exercice 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("-----");}
} }
```

Chapitre III

Part 2. Polymorphisme

Polymorphism

Introduction

- It's a powerful concept in OOP that complements inheritance.
- It allows to manipulate objects without knowing their exact type. For example, you can create an array of objects, some of type Point and others of type ColPoint, and call the display() method. Each object will respond according to its type.
- This illustrates the "is-a" relationship which states that a colored point is also a point and can therefore be treated as one. The converse, however, is false.

The basics of polymorphism (1/4)

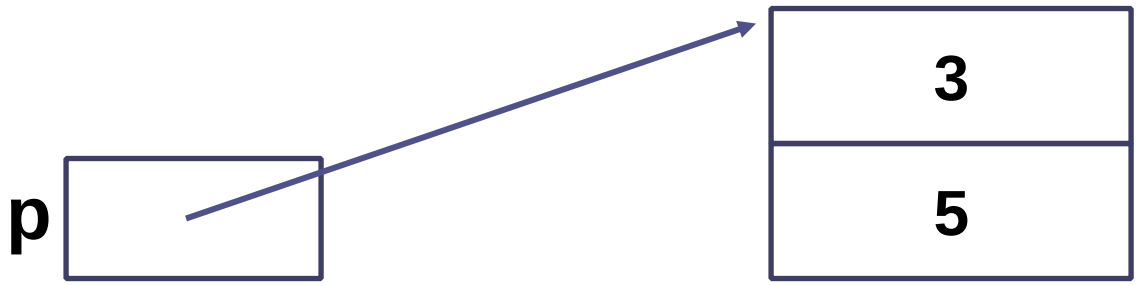
let's go back to the point and ColPoint classes

```
class Point{

private  int  x, y ;

public void display(){
//displays the coordinates
}
}
```

```
class ColPoint extends Point{

private String color;


public void display(){
// displays the coordinates
and the color

}
}
```

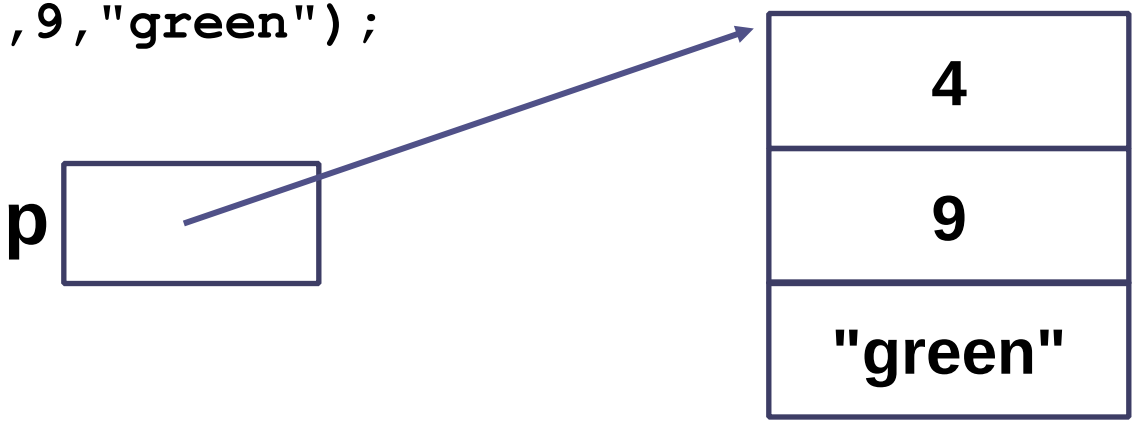
The basics of polymorphism (2/4)

```
Point p;
p = new Point(3,5)
```



This assignment is possible

```
p = new ColPoint (4,9,"green");
```



An object variable can reference an object of a derived type

The basics of polymorphism(3/4)

```
Point p = new Point (3,5);
p.display(); // invokes the display method of the Point class
p = new ColPoint (4,9,"Green");
p.display(); // invokes the display method of the ColPoint class
```

The second statement *p.display()* is based, not on the type of the variable **p** but on the actual type of the object referenced by **p** at the time of the call, as this type can change over time.

This choice, made at runtime rather than at compile time, is called dynamic binding (or late binding).

the method call is resolved at runtime based on the actual type of the object

The basics of polymorphism (4/4)

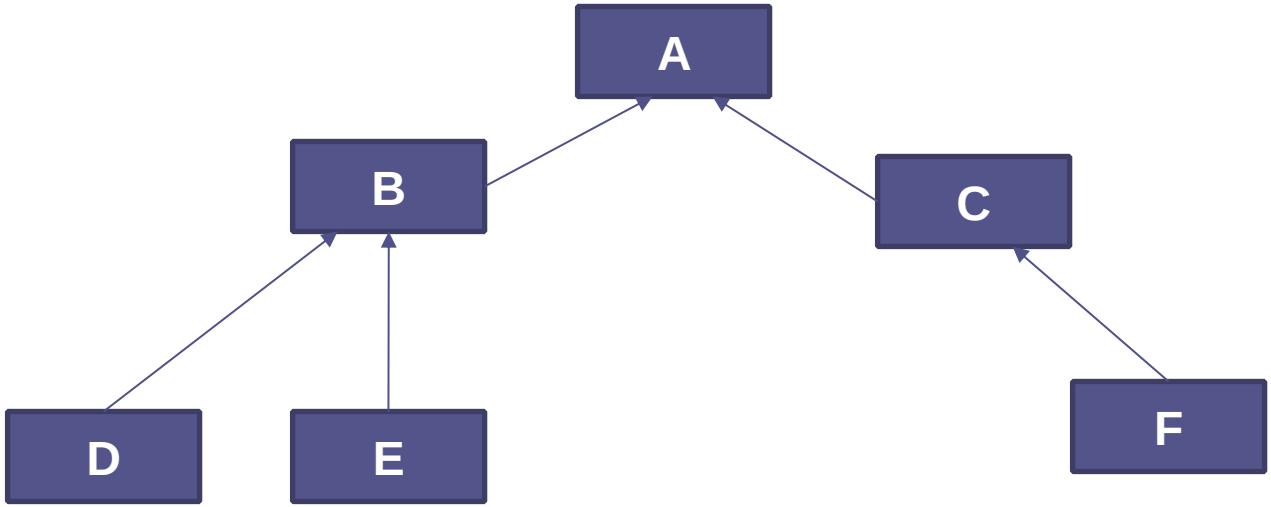
In summary:

Polymorphism is characterized by:

- Assignment compatibility between a class type and an ancestor type
- Dynamic binding of methods

Polymorphism and Successive Derivations

- A a; B b; C c; D d; E e; F f;



a = b; OK

a = c; OK

c = d; NO

a = d; OK

a = e; OK

b = a; NO

a = f; OK

b = d; OK

b = e; OK

c = f; OK

d = c; NO

Polymorphism, overriding and overloading

```
class A {
public void f(float x) {…}
}
class B extends A {
public void f(float x) {…} //overriding of f
public void f (int n) {…} //overloading de f
…….
}
A a = new A(); B b = new B(); int n;
```

`a.f(n);`	calls f (float) of A
`b.f(n);`	calls f (int) of B
`a = b;`	a contains a reference to an object of type B
`a.f(n);`	calls f(float) of B

Polymorphism, overriding and overloading

Explanation :

Although **a.f(n)** and **b.f(n)** both apply a method f to an object of type **B**, they do not call the same method.

For the statement **a.f(n)**, the compiler looks for the best matching method (according to method overriding rules) among all methods in class **A** or its ancestors. Here, it selects the method void f(float x) from **A**.

Once the appropriate method is determined, its signature and return type are fixed at compile time,

At runtime, the method resolution is based on the actual type of the object referenced by **a**, ensuring that a method with the expected signature and return type is called. As a result, the method **f(float)** from **B** is executed, despite the existence of a more suitable method in **B** for the given argument type.

Thus, despite dynamic binding, polymorphism is still constrained by a method signature and return type determined at compile time, which remain unchanged during execution.

Polymorphism, overriding and overloading

The simultaneous use of overloading and overriding can lead to complex situations.

It is therefore advisable to use them carefully and avoid overusing them

Conversion of actual arguments (1/4)

Case of Object-Type Arguments

Case 1: Method neither overloaded nor overridden

```
class A {
public void indentity(){
System.out.println("Object of type A");}
}

class B extends A{
// no overriding of the identity method
}

class Util {
static void f(A a){ a.identity();}
}

A a = new A(…); B b = new B(…);

Util.f(a); // displays "Object of type A"
Util.f(b);// displays "Object of type A"
```

Conversion of actual arguments (2/4)

Case of Object-Type Arguments

Case 2: Method overridden but not overloaded

```
class A {
public void indentity(){
System.out.println(" Object of type A");}
}

class B extends A {
public void indentity () {
   System.out.println("Object of type B");}
}
class Util{

static void f(A a) {
  a.identite();}
}
B b =  new B();
Util.f(b);// Displays "Object of type B"
```

Conversion of actual arguments(3/4)

Case of Object-Type Arguments

Case 3: Overloaded Method (Simple Example)

```
class A {…….}
class B extends A {……}
class Util{
static void f(int n, B b) {….}
static void f(float x, A a){….}
}
…
A a = new A(); B b = new B(); int n; float x;

Util.f(n, b); //without conversion: Call of f(int, B)
Util.f(x, a); //without conversion: Call of f(float,A)
Util.f(n, a); //convert n to float: Call of f(float,A)
Util.f(x, b); //convert b to A: Call of f(float,A)
```

Conversion of actual arguments(3/4)

Case of Object-Type Arguments

Case 3: Overloaded Method (Less Trivial Example)

```
class A {……}
class B extends A {……}
class Util{
static void f(int p, A a) {……}
static void f(float x, B b){……}
}
…
A a = new A(); B b = new B(); int n; float x;

Util.f(n, a);   //Without Conversion: Call to f(int, A)
Util.f(x, b);   //Without Conversion: Call to f(float,B)
Util.f(n, b);   // Compilation Error: Ambiguity due to two
                   possible matches
Util.f(x, a);   // Compilation Error: No suitable method
found, cannot convert A to B nor float to int
```

Polymorphism Rules in Java

As we have seen, excessive use of method overriding and overloading can result in complex situations. Here is a summary of the key rules:

1. Compatibility

There is an implicit conversion from a reference to an object of class C to a reference to an object of a superclass (both in assignments and in actual method arguments).

2. Dynamic Binding

In a method call of the form ***x.f(...)***, where ***x*** is declared of type ***C***, the selection of ***f*** follows these steps:

- **At Compilation:** The best matching method signature is determined within class ***C*** or its superclasses. This also defines the return type.
- **At Runtime:** The actual method ***f*** with the determined signature and return type is searched for, starting from the runtime class of the object referenced by ***x***. This class can be ***C*** or a subclass. If the class does not contain an appropriate method, the search moves up the inheritance hierarchy until a matching method is found.

Explicit reference conversion(1/2)

It is not possible to assign to a reference of an object of type T a reference to an object of a superclass type.

Exemple:

```
class Point{…}
class ColPoint extends Point {…}
…
ColPoint cp;
cp = new Point (…)
```

```
// compilation error
```

Now, consider this situation

```
Point p; ColPoint cp1, cp2;
cp1 = new ColPoint (…);
…
p= cp1; // p contains a reference to an object of type ColPoint
```

```
cp2 = p;
```

```
// Compilation error even if p contains a reference
        to an object of type ColPoint
```

Explicit reference conversion (2/2)

A conversion must be done using the cast operator, just as for primitive types.

Solutionn

```
Point p; ColPoint cp1, cp2;
cp1 = new ColPoint (…);
…
p= cp1; //p contains a reference to an object of type ColPoint
…
cp2 = (ColPoint) p;
```

```
//Accepted by the compiler but checked at runtime. If p
does not store a reference to an object of type
ColPoint or a derived type, execution will be stopped."
```

Note:
To check type compatibility, the instanceof operator can be used.
To determine the actual type of an object, the getClass() method can be used.

The superclass *Object*

- The *Object* class is the top of the hierarchy for all Java classes.
- Every class is derived from *Object*, whether it is predefined in Java or user-defined.

When we define a class

```
class Point { … }
```

This implicitly means

```
class Point extends Object{… }
```

Using a reference of type *Object*

Given the possibilities offered by polymorphism, a variable of type *Object* can be used to reference an object of any type. This can be useful for handling objects whose exact type is unknown at the time of use.

Example:

```
Point p =  new Point(1,2);
Object o;
...
o= p;

o.move(3 , 5); // compile error
```

```
((Point ) o). move(3 , 5); //OK
```

Using methods of the Object class

- The ***Object*** class includes several methods that can either be used as they are or overridden. The most important ones are:
  - ***toString:*** ***Returns a string containing the name of the class to which the object belongs and the object's address in hexadecimal.***

```
Point p = new Point (3,5);
System.out.println(" p =  " + p.toString() );
// displays, for example a = Point@fc17aedf
```

  - ***equals:*** **compares the addresses of two objects.**

```
Point p1 =  new Point(1,2);
Point p2 =  new Point(1,2);
…
p1.equals(p2) returns the value ....?
```

false

Inheritance and arrays

Even though arrays are considered objects, it is not possible to define their class, and therefore, they only benefit from some of the properties of objects.

1. An array can be considered as belonging to a subclass of *Object* class:

```
Object o;
o = new int[5]; // correct

o = new float [3]; // correct
```

2. It is not possible to derive a class from an array

```
class Test extends int []; // error
```

Polymorphism and arrays

- Polymorphism can be applied to arrays of objects.
- If ***B*** extends ***A***, an array of objects of type ***B*** is compatible with an array of objects of type ***A***.

```
class B extends A {…….}
A ta[];
B tb[];
……
ta = tb;        // OK because B derives from A
tb = ta;        // error
```

But this property cannot be applied to primitive types

```
int ti [], float tf[];
….
ti = tf; //error(obvious because float is not compatible with int)
tf = ti; //error although int is compatible with float
```

Final classes and methods

When the final keyword is applied to a variable or fields in a class, it prevents their values from being modified.

```
final int n = 23;
...
n=n+5; // error
```

The final keyword can be applied to a class or a method, but with a completely different meaning

- **A method declared final cannot be overridden in a derived class**
- **A class declared final cannot be derived**

class abstraites et interfaces

Problème 1

- We want to have a class hierarchy to manipulate geometric shapes.
- It should always be possible to extend the hierarchy by deriving new classes, but we want to enforce that these new classes always implement the following methods:
  - void calculateArea()
  - void calculatePerimeter()
  - void display()

What solution do you propose?

Abstract classes

- An abstract class is a class that cannot be instantiated. It can only be used as a base class for derivation.

```
abstract class A {......}
```

- An abstract class contains not only methods and attributes, but also so-called abstract methods. In other words, methods for which only the signature and type of the return value are provided.

```
abstract class A
{ public void f(){....}
  public abstract void g (int n);
}
```

Abstract classes

A few rules

1. A class with one or more abstract methods is abstract. As such, **cannot be instantiated.**

2. An abstract method **cannot be private** (since it is intended to be overriden in a derived class).

3. A class derived from an abstract class is not obliged to override all the abstract methods of its base class, and can even override none, remaining abstract itself.

4. A class derived from a non-abstract class can be declared abstract and/or contain abstract methods

Solution to the problem

- We simply need to create an abstract class *Shape* that contains the method signatures we want to enforce in derived classes, using the

abstract keyword.

```
abstract class Shape {
abstract public void display() ;
abstract public void calculerSuperficie();
abstract public void calculerPerimetre();
…
}
```

- The derived classes in the hierarchy will simply be defined as derived classes of Shape, and they must implement the methods. For example:

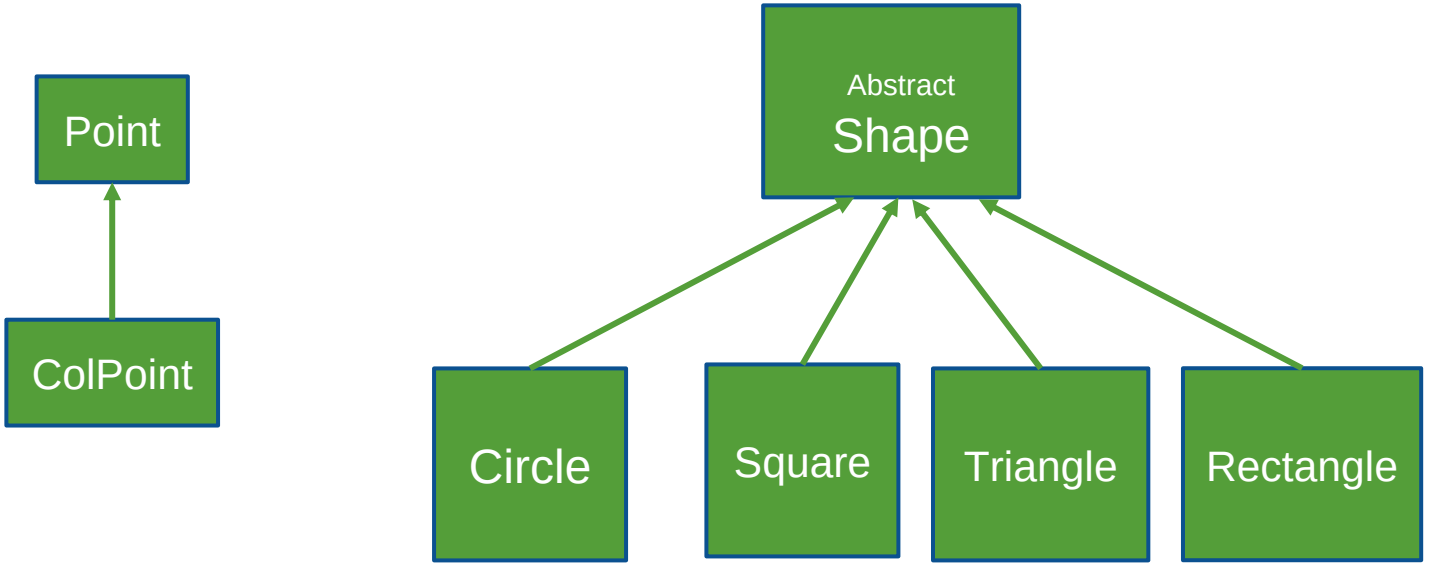
```
class Triangle extends Shape {
public void display() {…..}
public void calculerSuperficie(){….}
public void calculerPerimetre(){….}
}
```

The benefits of abstract classes

- Abstract classes facilitate object-oriented design by allowing the placement of all functionalities that must be available to all their descendants within a single class:
  - Either in the complete form of methods (non-abstract) and fields common to all descendant classes.
  - Or as an interface of abstract methods that are guaranteed to exist in every derived class.

Problem 2

- Consider the hierarchy of geometric shape classes and point classes. Suppose that, for the needs of an application, we want to enforce the rule that only circles and colored points can be colorable. What solutions would you propose?

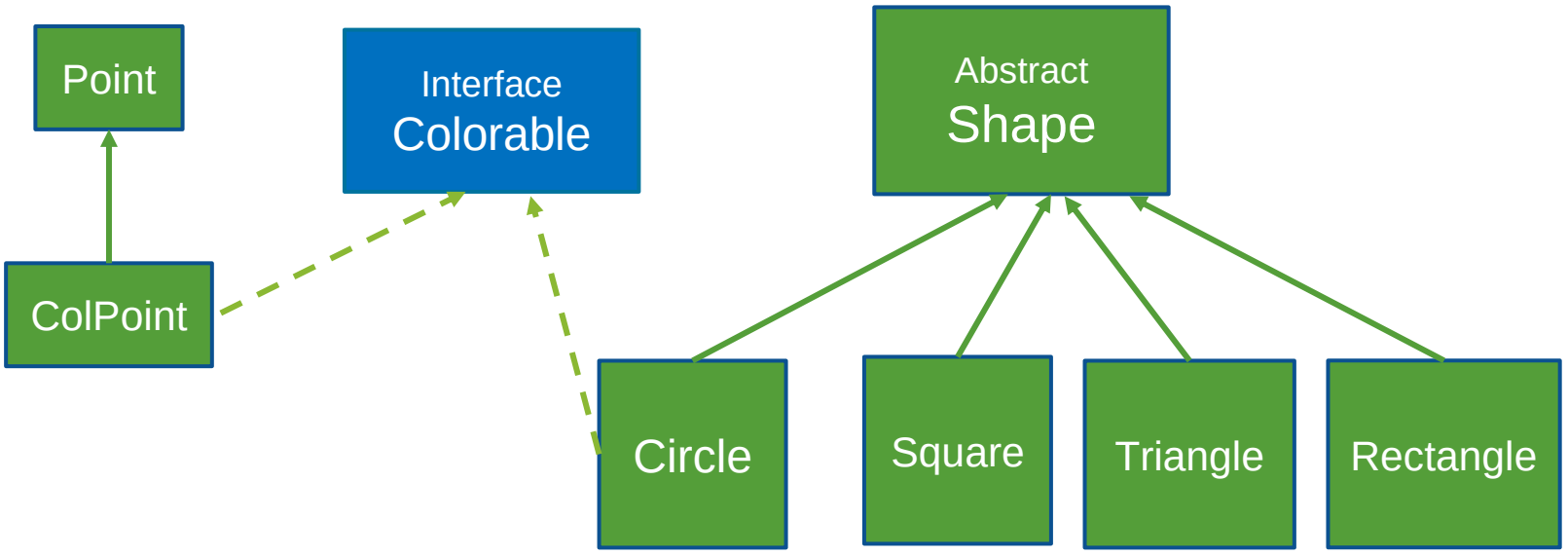


Interfaces

- If we consider an abstract class that does not implement any methods or define any fields (except for constants), we arrive at the concept of an interface.
- An interface defines the signatures of a set of methods, as well as constants.
- If a class implements an interface, it commits to providing an implementation for all its methods, even if the implementation is empty.

Solution to the problem

- The solution is to create an interface called ***colorable*** containing the abstract method ***setColor (String) and getColor()*** which only the ColPoint and Circle classes will implement.



Interfaces

Définition of an interface

An interface is defined in the same way as a class, except that the keyword class is replaced with interface,

```
public interface I{
final int MAX = 100;
void f(int n);// public and abstract are optional
void g();
}
```

The methods of an interface are implicitly abstract and public, so there's no need to specify these modifiers.

Interfaces

Implementing an interface

A class can implement one or multiple interfaces.

```
public interface I1 {
 void f();
}

public interface I2{
 void g()
}

class A implements I1{// A must override method f of I1}

class B implements I1, I2{// A must override method f of
I1 and g of I2 }
```

Interface-type variables and polymorphism

It is possible to define variables of an interface type.

```
public interface I { …. }
…
I i;   //i is a reference to an object of a class
implementing the interface I

class A implements I {….}
…
I i = new A(…); // ok
```

Derivation of an interface

An interface can be defined as derived from another interface using the extends keyword

```
interface I1{
static final MAXI =
100;
void f (int n);
}

interface I2 extends
I1{
static final MINI = 10;
void g();
}
```

```
interface I2 {
static final MINI = 10;
static final MAXI = 100;
void f (int n);
void g();
}
```

Derivation of an interface

An interface can inherit from multiple interfaces. Multiple inheritance is allowed for interfaces.

```
interface I1 {
void f();
}
interface I2 {
void f1();
}
interface I3 extends I1,I2 {
void f2();
}
```

Name conflicts

Exemple 1

```
interface I1{
  void f (int n);
  void g();
}

interface I2{
 void f(float x);
 void g();
}

class A implements I1,I2{
/*To satisfy I1 and I2, A must define two
methods f: void f(int) and void f(float),
but only one method g: void g() */
}
```

Name conflicts

Exemple 2

```
interface I1{
 void f (int n);
 void g();
}
interface I2{
 void f(float x);
 int g();
}
class A implements I1,I2{ // compilation error
/* To satisfy I1 and I2, A must contain a method
void g() and a method int g(), which is
impossible according to the overriding rules.
   A class cannot implement interfaces I1 and I2
as defined in this example. */
}
```

"Default » methods

Since Java 8, it is allowed to provide an ***implementation*** for methods in interfaces.

```
public interface I {

/* No implementation */

public void m1();

/* default implementation*/

default void m2() {
// body of the method
}

}
```

Default methods

Why Use Default Methods?

- Default methods allow an interface to evolve without causing issues for the classes that implement it.
- Example: the case of Java API interfaces.
- Classes that share the same implementation of an interface method no longer need to duplicate the same code at their level.

Default methods

Possible uses

- A class implementing an interface with a default method can:
  - Use this method directly
  - Override this method

- An interface ***I2*** deriving from an interface ***I1*** can:
  - Inherit the default method implementation
  - Override a default method
  - Declare it again, making it abstract

Default methods

Usage rules

- Si deux interfaces I1 et I2 déclarent la même méthode m () mais proposent des implémentations incompatibles, que se passe-t-il pour la class implémentant ces deux interfaces?

- La class doit redéfinir la méthode pour résoudre le conflit.

- Pour appeler la méthode par défaut d'une des deux interfaces, une syntaxe particulière a été ajoutée:

I2.**super**.m();

Default methods

Usage rules

- If two interfaces, I1 and I2, declare the same method ***m()***, but provide different implementations, what happens to the class implementing both interfaces?

The class must override the method to resolve the conflict.

To call the default method from one of the two interfaces, a specific syntax has been introduced:

```
I2.super.m();
```

Static methods in interfaces

- This is another new feature in Java 8.
- Interfaces can contain static methods, and these are used in exactly the same way as class methods

Example of the ***Comparable*** interface

- A class implements the `Comparable` interface if its objects can be ordered according to a specific order.
  - For example,
    - the ***String*** class implements ***Comparable*** because character strings can be ordered alphabetically.
    - Numeric classes like ***Integer*** and ***Double*** implement ***Comparable*** because numbers can be ordered numerically.

Example of the *Comparable* interface

- The ***Comparable*** interface consists of a single method (and no constants):
- ***int compareTo(T obj),*** which compares the current object to an object of type ***T***.

- A.compareTo(B) returns
  - a negative integer if object A is smaller than B,
  - The value 0 if both objects are equal,
  - a positive integer if object A is greater than B.

Example:

- “Bonjour”.compareTo("Bonsoir") returns a negative integer.

Exercise

Create a BankAccount class with a single balance attribute, and define the equality of two bank account objects, so that it's verified as soon as the two balances are equal.

Solution

```
class BankAccount implements Comparable{

    private float solde;

    public float getSolde(){ return solde;}
    public void setSolde(float s){ solde=s;}

    public int compareTo(Object cmp) {

    float solde2 = ((CompteEnBanque)cmp).getSolde();

     if(this.solde>solde2)
          return 1;
    else if(this.solde <solde2)
            return -1;
          else
            return 0;
}
}
```

Solution (another version)

```
class BankAccount implements Comparable<CompteEnBanque>{

    private float solde;

    public float getSolde(){ return solde;}
    public void setSolde(float s){ solde=s;}

    public int compareTo(CompteEnBanque cmp) {

    float solde2 = cmp.getSolde();

 if(this.solde>solde2)
        return 1;
    else if(this.solde <solde2)
            return -1;
    else
            return 0;
}
}
```