UEF4.3 Object Oriented Programming

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

Abstract classes & interfaces

Problem 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 figure hierarchy will simply be defined as derived classes of Figure, and they must implement the methods. For example:

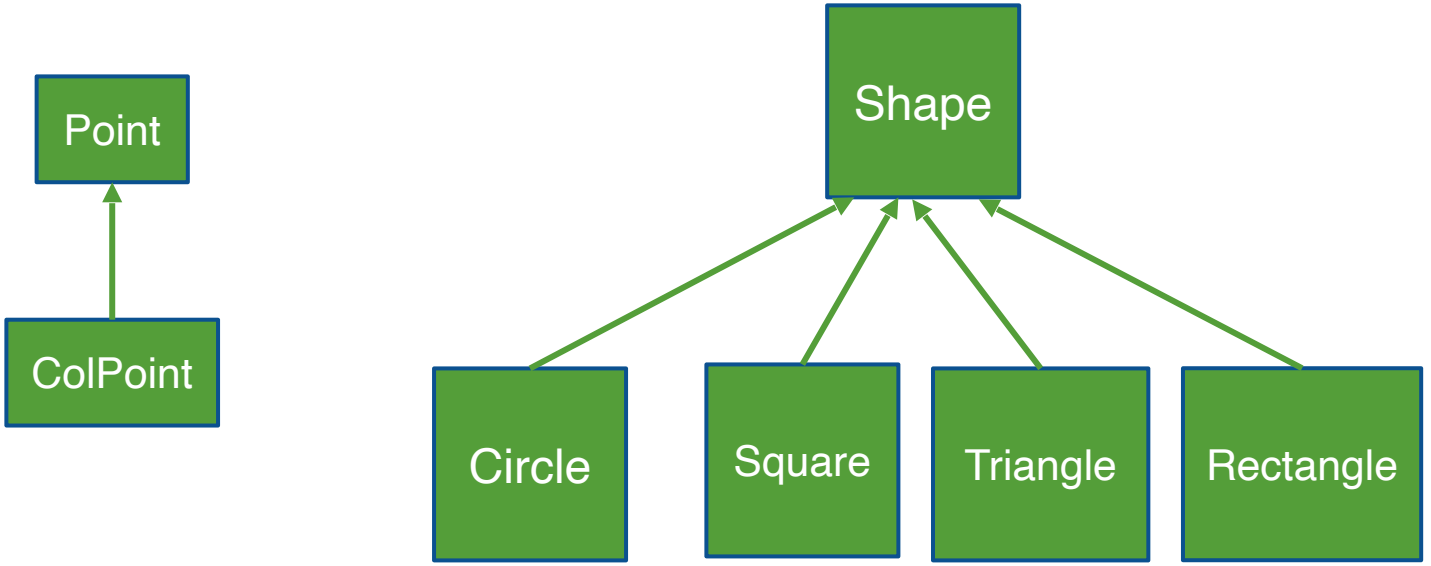
```
class Triangle extends Shape {
public void display() {.....}
public void calculerSuperficie(){....}
public void calculerPerimetre(){....}
}
```

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 figure 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?

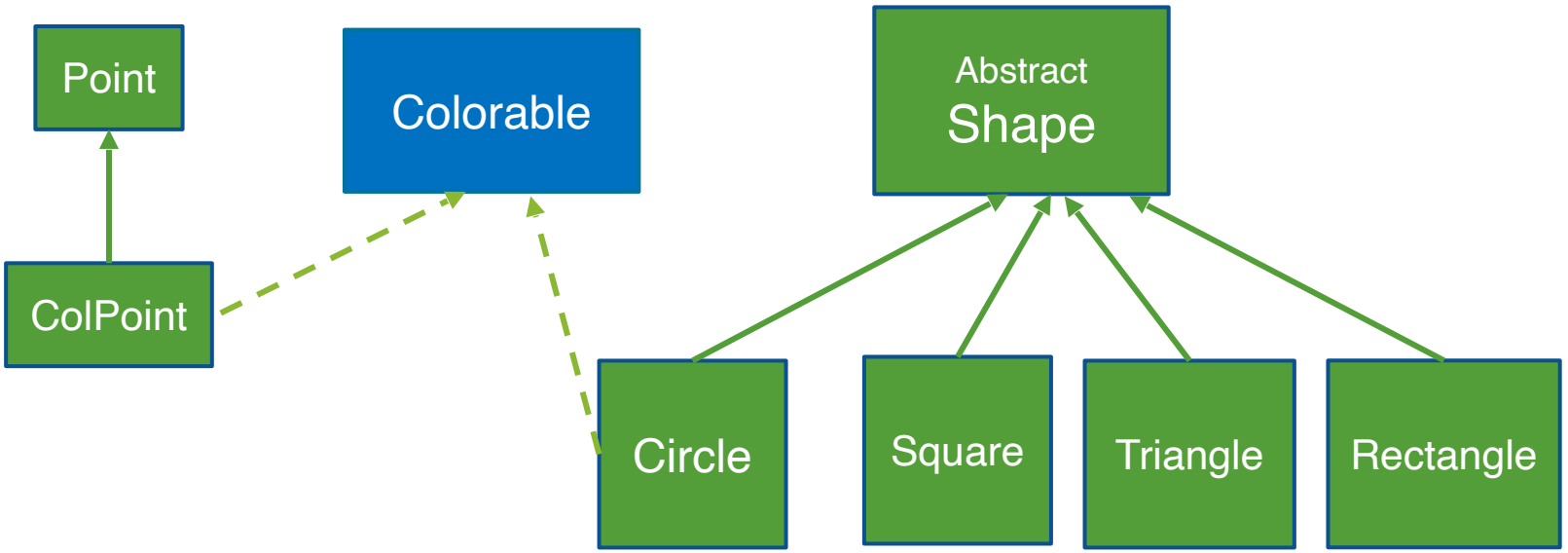


Interfaces

- 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 {

/* Pas d'implémentation */

public void m1();

/* Implémentation par défaut*/

default void m2() {
// corps de la méthode
}

}
```

“Default” methods
syntaxe

```
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?

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.

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;
}
}
```

Comparison between interfaces and abstract classes

Feature	Interface	Abstract Class
Definition	Defines a contract that implementing classes must follow.	Serves as a base class with shared behavior.
Methods	Can have abstract methods (without implementation) and, since Java 8, default/static methods with implementations.	Can have both abstract (without implementation) and concrete methods (with implementation).
Fields	Only public, static, and final (constants).	Can have instance variables with any access modifier.
Inheritance	A class can implement multiple interfaces.	A class can extend only one abstract class.
Constructor	No constructors (cannot instantiate an interface).	Can have constructors (called when a subclass is instantiated).
Access Modifiers	Methods are implicitly public (unless default or static).	Methods can have any access modifier.
Use Case	Used to enforce a common behavior across unrelated classes.	Used to create a base class with shared behavior.