Inheritance, overriding & abstract type

Sub-typing / polymorphism

- The idea behind sub-typing and polymorphism is that:
  - A behavior (method) depends on the kind of object effectively contained in the receiver identifier
    - Printing an Object differs from printing a Pixel and differs from printing a Person
    - But all of these objects could be printed...
    - All of them provide the « method » toString()
    - It is the same with methods equals(), hashcode()...

Sub-typing

- Essentially, we want to have **types**
  - On which some **methods are available (functionalities)**
  - But whose **precise definition (behavior) depends on the sub-type**
  - The method finally executed will be as precise as possible
  - Example: all Shape has a surface, but computing surface of a square differs to computing surface of a circle...

How to define sub-types

- Some conversions are possible between **primitive type** values => **this is not sub-typing**
  - `byte < short < int < long < float < double`
    `char < int`
  - It’s a mater of data representation

- Sub-typing only relies on types (not on stored data)
- We already saw that any class A implicitly extends class Object, and thus defines a **type** A which is a **sub-type** of type Object
  - This **is** sub-typing, whatever the data stored in A

How to define sub-types

– **Inheritance** defines sub-types:
  - Either explicitly:
    `class Student extends Person { ... }`

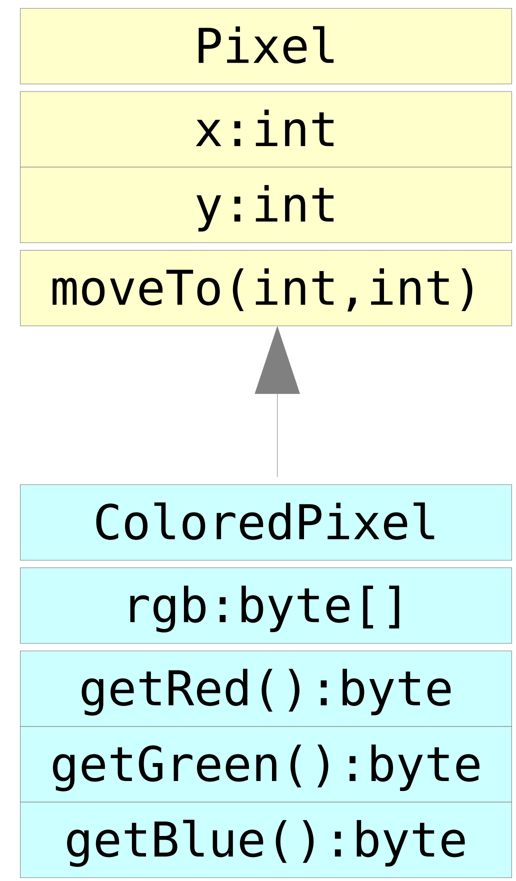
  - Or implicitly :
    `Pixel` or `int[]` extends `Object`

– **Implementation of interface** also defines sub-types
  - An interface **declares methods available** on objects of any class implementing it
  - A **class** implements an interface by **defining its methods**:

```
class Carre implements Mesurable { ... }
```

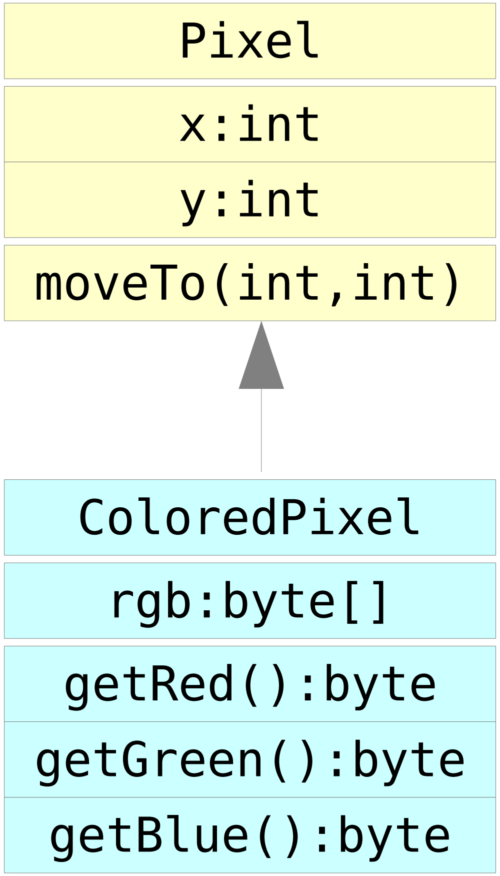
Inheritance

- Consists in defining a class,
  known as sub-class,
  from another class,
  known as super-class,
  by automatically retrieving
  in the sub-class
  all members
  of the super-class,
  potentially completed
  by new members



Inheritance

```java
public class Pixel {
    private int x;
    private int y;
    public void moveTo(int newX, int newY) {
        this.x = newX;
        this.y = newY;
    }
}
```




```java
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    public byte getRed() { return rgb[0]; }
    public byte getGreen() { return rgb[1]; }
    public byte getBlue() { return rgb[2]; }
}
```

What are the objects of a sub-class?

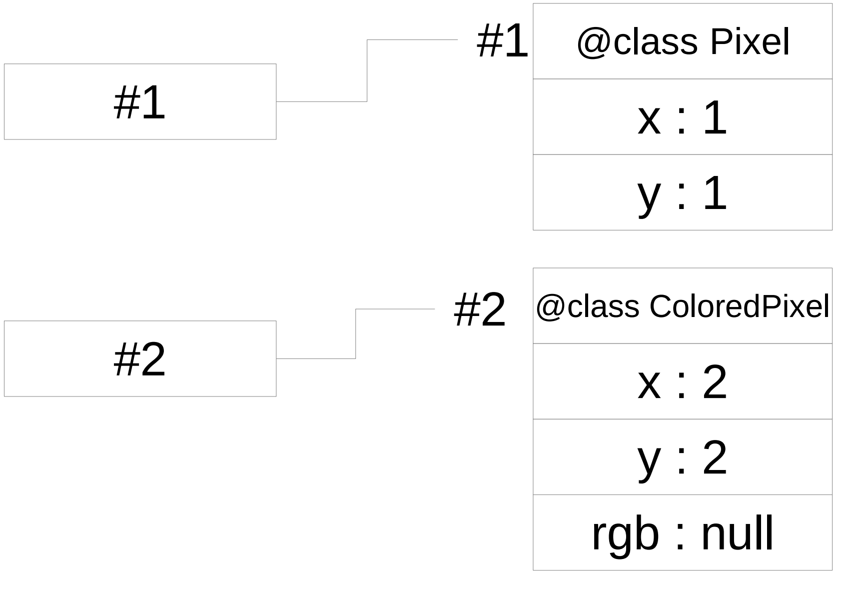
- All object of a sub-class is firstly considered as being an object of its super-class
  - A colored pixel « is » firstly a pixel
- All object of a sub-class « accumulates » the fields of the super-class with those defined in its own class
  - There is a int x and a int y in any object of class ColoredPixel

What are the objects of a sub-class?

```java
public static void main(String[] args) {
    Pixel p = new Pixel();
    p.moveTo(1, 1);

    ColoredPixel cp = new ColoredPixel();
    cp.moveTo(2, 2);

    cp.getRed();  // compiles, but raise NullPointerException
                  // because the array has not been allocated
}
```



All the fields are inherited

- They could be handled if allowed by their accessibility
  - if x is not private in Pixel, we could use this.x in ColoredPixel
  - Usually, **we avoid non private fields**

```
public class Pixel {
    int x;
    private int y;
    // ...
}
```

```
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    void test() {
        System.out.println(this.x);  // 0
        System.out.println(this.y);  // did not compile !
                             // field Pixel.y is not visible
    }
}
```

All the fields are inherited

- They could be **hidden** by other field definition in the sub-class that have the same name
  - Warning : they are « hidden » and not « override » as for methods...

```java
public class Pixel {
    int x;
    private int y;
    // ...
}
```

```java
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    private String x;
    void test() {
        System.out.println(this.x);   // ??
    }
}
```

Same name (?!)
but two fields

All the fields are inherited

- if String x is declared in ColoredPixel, this field is concerned in this class when using this.x
- it is possible to manipulate the hidden field (if accessible) through the notation **super.x**
- **super** has the same value as **this** at run time but is has the **type** of the super-class (here Pixel)

```
public class Pixel {
    int x;
    private int y;
   // ...
}
```

```
public class ColoredPixel extends Pixel {
    private byte[] rgb;
    private String x;
    void test() {
        System.out.println(this.x);   // null
        System.out.println(super.x);  // 0
    }
}
```

Field resolution

- The « **resolution** » is the process of **identifing which field** to use, in order to know **where** finding its value at run time
- **Field** resolution is done by the compiler, based on the **declared type** of the identifier containing the reference

```
public static void main(String[] args) {
    ColoredPixel cp = new ColoredPixel();
   // declared type of cp is ColoredPixel
    System.out.println(cp.x);   // null

    Pixel p = cp;
    // declared type of p is Pixel, even if the reference
    // contained in p is those of a ColoredPixel
   System.out.println(p.x);    // 0
}
```

Hidden fields

- Usually, having a field with the same name as a field of a super-class is a **bad idea**
  - **super**, is **this** considered with the **type of the super-class**
  - **super.super.x** doesn't exist...
  - Neither ref.super nor ref.super.x...

```
class A {
  int x = 1;
}

class B extends A {
  String x = "zz";
}
```

- Nevertheless, ***cast*** allow to access any field by changing the declared type of the reference ref

```
class C extends B {
  boolean x = true;
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);      // true
    System.out.println(((B)c).x); // zz
    System.out.println(((A)c).x); // 1
  }
}
```

Constructors and inheritance

- construction (initialization) of any instance of any class always starts by construction (initialization) of an instance of Object
  - Indeed, all constructor starts by a call to the constructor of its super-class: **`super()`**

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

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  public ColoredPixel(int x, int y) {
    super(x, y); // note that x and y are private!
    rgb = new byte[3];
  }
}
```

Constructors and inheritance

- `super()`
  - Must be the first instruction of the constructor
  - The implicit constructor (generated by the compiler) call the constructor without argument of the super-class
- Constructors are not inherited

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

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  public ColoredPixel() { // Do not compile !
    // super(); // Constructor Pixel() is undefined
  }
}
```

Constructors and initializations

- A call to the constructor of a class is a step the initialization process of an object of this class:
  - It starts initializing the fields of the object « as an instance of the super-class »: this is the call to super()
  - Next it initializes its own fields (as an instance of the sub-class)
  - The call to super() cannot use fields whose existence or value would depend on the instance of the sub-class

```java
public class ColoredPixel extends Pixel {
  private int v = 0;
  private static int s = 0;
  public ColoredPixel() {
    // super(v,v);
    // error: cannot reference v before supertype constructor has been called
    super(s,s); // OK
  }
}
```

Inheritance of methods

- In addition to fields, as « members », the **sub-class inherits the methods of the super-class**
- Only **constructors are not inherited**
  - They stay local in their own class
- **Warning**: the **code** (semantics) of a super-class method **could become wrong** in the sub-class
  - Pixel::moveTo() is correct in ColoredPixel but Pixel::equals() or Pixel::toString() aren't!
- Often, it is necessary to give a **new definition** for the inherited method in the sub-class

Inheritance => sub-typing

- A ColoredPixel "*is a kind*" of Pixel
- Everywhere a Pixel is expected, it is possible to use a ColoredPixel
- What sense (semantics) methods must have?

```
public static void main(String[] args) {
   ColoredPixel cp = new ColoredPixel(1,2);
   cp.setRed((byte) 100);

   Pixel p = cp; // inheritance => sub-typing

   System.out.println(p);  // ?

   System.out.println(p.equals(new ColoredPixel(1,2))); // ?
}
```

Overriding methods

- Give a new definition for an inherited method:
- Same name, same parameters, distinct code
- Annotation **@Override** ask the compiler for verifying that we actually override an inherited method

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  // ...
  @Override
  public String toString() {
    return super.toString()+"["+rgb[0]+":"+rgb[1]+":"+rgb[2]+"]";
  }
  public static void main(String[] args) {
    ColoredPixel cp = new ColoredPixel(2,2);
    System.out.println(cp);  // (2,2)[0:0:0]
    Pixel p = new Pixel(5,5);
    System.out.println(p);   // (5,5)
    Object o =  new ColoredPixel(2,2);
    System.out.println(o);   // (2,2)[0:0:0]
  }
}
```

Inheritance is...

… three indivisible things:

- You **want** to get (inherit) **all members** (fields, methods) from the super-class (even private)
- You **must override** all methods that haven’t the correct semantics in the sub-class
- You want the sub-class **defining a sub-type** of the super-class

if you **don’t want** one of these three things, then then **you shouldn't use inheritance**.

Inheritance and Object class

- In Java, all classes extends Object
  - Either directly

    compiler add “**`extends java.lang.Object`**”
  - Or indirectly

    ColoredPixel extends Pixel, that extends Object

  => all class are sub-types of Object
- You have to override equals() / hashCode() / toString() if needed !

Overriding (methods) *versus* hiding (field)

- **All fields** defined in all super-classes are present in an object of a sub-class
  - Even with same name and same type
  - A field of the immediate super-class could be reached with super.x
  - **Field resolution** depends on the **declared type** of the identifier
  - This allows us to reach any field, through a type cast of the identifier
- For **methods**, only one remains in the sub-class!
  - We could reach those of the immediate super-class with super.m()
  - **Method resolution** is done in two steps
    - **Compile-time**: looking for a solution wrt declared identifier types
    - **Runtime**: looking for the most precise implementation of this solution, given the « actual » type of the receiver
  - Other methods (of super-classes) are no more reachable

Override vs Overload

- If the signature of the method differs between the super-class and the sub-class, this provides us with **overloading** rather than overriding:
  - In this case, both methods coexist in the sub-class

```
class A {
  void m1() { ... }
  void m2() { ... }
  Pixel m3() { ... }
  void m4(Pixel p) { ... }
}
class B extends A {
  @Override void m1() { ... }                    // override
            void m2(int a) { ... }               // overload
  @Override ColoredPixel m3() { ... }            // override
  @Override void m4(Pixel p) { ... }             // override
            void m4(ColoredPixel p) { ... } // overload
}
```

Overriding principles

- Let B a sub-type of A and m() defined in A
- We override method m() in B in order to give a more precise definition (better suited for B)
- For a method call a.m()
  on an identifier a declared of type A,
  the compiler agrees since m() is defined in A
- We want the overridden version to be used at run-time if a actually contains an object of sub-type B

Overriding principles

- Compiler is supposed to avoid bad surprises (i.e. find out problems at run-time)
- This governs the main rules
  - An instance method cannot override a static method
  - Overriding cannot restrict accessibility
  - return type of a overridden method cannot be of a super-type
  - exceptions raised by an overridden method cannot be of a super-type of those thrown by the original method

Method equals()

- Just like method toString() of class Object, that any sub-class would override...
- … class Object provides a method equals(Object obj) whose « contract » is clearly established by the documentation
  - By default, it tests **primitive equality of references**
  - You must override it

```
public class Pixel {
  private int x, y;
  // ...
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof Pixel))
      return false;
    Pixel p = (Pixel) obj;
    return (x==p.x) && (y==p.y);
  }
}
```

```
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof ColoredPixel))
      return false;
    ColoredPixel cp = (ColoredPixel) obj;
    return super.equals(obj) &&
         rgb[0]==cp.rgb[0] &&
         rgb[1]==cp.rgb[1] &&
         rgb[2]==cp.rgb[2];
  }
}
```

Specification of method equals()

- Defines an equivalence relation on non-null object references
  - **reflexive**
    - for any non-null reference value x, x.equals(x) should return true
  - **symmetric**
    - for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
  - **transitive**
    - for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
  - **consistent**
    - for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
  - For any non-null reference value x, x.equals(null) should return false.
  - Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

Symmetric property...

- Ask a Pixel in (2,2) for being equals to a ColoredPixel in (2,2)… it will answer YES!
  - It only check coordinates...
- But ask a **ColoredPixel magenta** in (2,2) for being equals to Pixel en (2,2), it will answer NO!
  - It is supposed to check the color that a simple Pixel hasn't...
- You could find this code acceptable... or not

```
public class ColoredPixel extends Pixel {
  // ...
  public static void main(String[] args) {
    Object o1 = new Pixel(2,2);
    Object o2 = new ColoredPixel(2,2);
    System.out.println(o1.equals(o2)); // true
    System.out.println(o2.equals(o1)); // false
  }
}
```

Method hashCode()

- This method is used in hash tables, such as java.util.HashMap
- It specifies a « **contract** » (together with equals())

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

To be more strict...

- You must consider that two objects of distinct classes cannot be equals
  - instanceof is not sufficient
  - you need to know the « class » of the object (at runtime)
  - method Class getClass() of class Object

In Pixel :

```java
@Override
public boolean equals(Object obj) {
  if(obj == null) return false;
  if(obj.getClass() != getClass())
    return false;
  Pixel p = (Pixel) obj;
  return (x==p.x) && (y==p.y);
}
```

```java
@Override
public boolean equals(Object obj) {
  if(obj == null) return false;
  if(obj.getClass() != getClass())
    return false;
  ColoredPixel cp = (ColoredPixel) obj;
  return super.equals(obj) &&
    Arrays.equals(this.rgb, cp.rgb);
}
```

In ColoredPixel :

```java
public static void main(String[] args) {
  Object o1 = new Pixel(2,2);
  Object o2 = new ColoredPixel(2,2);
  System.out.println(o1.equals(o2)); // false
  System.out.println(o2.equals(o1)); // false
}
```

Warning: with this solution, two objects of two distinct classes that extends Pixel would no more be able to be equals… without completely overriding equals (without using super.equals)

hashCode() and equals()

- Sets and Maps use both hashCode() and equals()
- If equals() is overridden but hashCode is not, this is what happens:

```java
import java.util.HashSet;

public class Pixel {
  // ...
  public static void main(String[] args) {
    Pixel zero = new Pixel(0,0);
    Pixel def = new Pixel();
    HashSet<Pixel> set = new HashSet<>();
    set.add(zero);
    System.out.println(zero.equals(def));  // true
    System.out.println(set.contains(def)); // false
    System.out.println(zero.hashCode());   // 1808253012
    System.out.println(def.hashCode());    //  589431969
  }

}
```

inconsistency between equals() and hashCode()

hashCode() example for our pixels

```java
public class Pixel {
  // ...
  @Override
  public boolean equals(Object obj) {
    if(!(obj instanceof Pixel))
      return false;
    Pixel p = (Pixel) obj;
    return (x==p.x) && (y==p.y);
  }
  @Override
  public int hashCode() {
    return Integer.rotateLeft(x,16) ^ y;
  }
```

```java
public static void main(String[] a){
  Pixel zero = new Pixel(0,0);
  Pixel def = new Pixel();
  HashSet set = new HashSet();
  set.add(zero);
  set.contains(def); // true
  zero.hashCode();   // 0
  def.hashCode();    // 0
  zero.equals(def);  // true
}
```

```java
public class ColoredPixel extends Pixel {
  private byte[] rgb;
  // ...
  @Override
  public int hashCode() {
    // return super.hashCode() ^ Integer.rotateLeft(rgb[0],16)
    //        ^ Integer.rotateLeft(rgb[1],8) ^ rgb[0];
    return super.hashCode() ^ Arrays.hashCode(rgb);
  }
}
```

Classes and methods « final »

- The key-word **final** exists for methods:
  - It means that this method cannot be overridden in a sub-class
  - This could be useful to ensure that no other definition will replace the original one (security)
- The key-word **final** exists for classes:
  - It is then impossible to extends this class
  - Methods behave as if they were final

Interfaces

- A **class** defines:
  - A type
  - A data structure for its objects (their fields)
  - Some methods with their code (their definition)
- An **interface** defines:
  - A type
  - Some methods **without** their code
    (abstract methods) – but Java 8 : `default`
- => No fields, no object, no state

Interfaces

- An interface **cannot be instanciated**
- It is supposed to be « **implemented** » by classes
  - These classes will get the type of the interface
  - These classes will provide definitions (code) for each declared method of the interface
- The idea for an interface is a « promise » :
  - when declaring a identifier with the type of the interface, you can call on this identifier any method promised by (declared in) the interface
  - the compiler ensures that any reference contained in this identifier points to an object of a class providing an implementation for the method

The point of interfaces

- To give a common type to distinct classes I order to use them in a same way
- Example: handle arrays of « trucs », each of truc having a surface
  - Summing surfaces of trucs in this array

```
public interface Surfaceable {
  public double surface();
}
```

```
public class AlgoOnTrucs {
  public static double totalSurface(Surfaceable[] array) {
    double total = 0.0;
    for(Surfaceable truc : array)
      total += truc.surface();
    return total;
  }
}
```

Using interfaces

- Two main advantages:
  - The algorithm for method totalSurface(Surfaceable[] array) is implemented independently of the real class of objects stored in array:
    this is provided by **sub-typing**
  - Each method surface() actually called on objects in the array will be most precise possible, with respect to the real type of each object:
    this is **polymorphism**

Using interfaces

```java
public class AlgoOnTrucs {

  public static double totalSurface(Surfaceable[] array) {
    ...
  }

  public static void main(String[] args) {
    Rectangle rectangle = new Rectangle(2,5);
    Square square = new Square(10);
    Circle circle = new Circle(1);
    Surfaceable[] t = {rectangle, square, circle};
    System.out.println(totalSurface(t));
                                 // 113.1415926535898
  }
}
```

Interface implementation

```java
public class Square implements Surfaceable {
  private final double side;
  public Square(double side) {
    this.side = side;
  }
  @Override
  public double surface() {
    return side * side;
  }
}
```

```java
public class Rectangle implements Surfaceable {
  private final double height;
  private final double width;
  public Rectangle(double height, double width) {
    this.height = height;
    this.width = width;
  }
  @Override
  public double surface() {
    return height * width;
  }
}
```

```java
public class Circle implements Surfaceable {
  private final double radius;
  public Circle(double radius) {
    this.radius = radius;
  }
  @Override
  public double surface() {
    return Math.PI * radius * radius;
  }
}
```

members of interfaces

- Public method declarations
  - All methods in interface are **`abstract public`**
    - Even if not specified, except `default` (see later)

```
public interface Surfaceable {
  double surface(); // equivalent to
  public abstract double surface();
}
```

members of interfaces

- Public constant fields
  - All fields in interface are **`public final static`**
    - Compiler adds these key-words

```
public interface I {
  int field = 10; // equivalent to
  public final static int field = 10;
}
```

Interface implementation and sub-typing

- A class can **implements** an interface
  - key-word **`implements`**

```
public class Rectangle implements Surfaceable {
  ...
}
```

  - Class `Rectangle` defines a **sub-type** of `Surfaceable`

```
Surfaceable s = null;

s = new Rectangle(2,5);
```

members of interfaces

- It is **not possible to instantiate** an interface, that is, impossible to create an object
  - You only could **declare identifiers** with its type
  - Such identifier will be able to **store references** to objects of **classes implementing the interface**

Interface implementation and sub-typing

- An interface cannot implement another interface
  - How to implement methods?
- **But** an interface can **extend** another interface
- Same `extends` key-word as for classes

```
public interface Paintable extends Surfaceable {
  double paint(byte[] color, int layers);
}
```

  - `Paintable` is a sub-type of `Surfaceable`

```
Surfaceable[] array = new Surfaceable[3]; // arrays!
Paintable p = null;
array[0] = p; // OK: Paintable < Surfaceable
p = array[1]; // Cannot convert from Surfaceable to Paintable
```

Sub-typing between interfaces

- An interface can **extends** de **several** other interfaces
  - Separate super-types with comas

```
public interface SurfaceableAndMoveable
               extends Surfaceable, Moveable {
   ...
}
```

  - Type SurfaceableAndMoveable define a **sub-type** of both types **Surfaceable** and **Moveable** (**multiple sub-typing**)
    - SurfaceableAndMoveable < Surfaceable et SurfaceableAndMoveable < Moveable
    - But Surfaceable and Moveable are not related

Class inheritance and interface implementation

- A class can both
- **extends** a single class
  - (single inheritance)
- and **implements** several interfaces
  - (multiple sub-typing)

```java
public class SolidCircle extends Circle implements Paintable, Moveable {
  private final Point center;
  public SolidCircle(Point center, double radius) {
    super(radius);
    this.center = center;
  }
  @Override  // To be able to implement Paintable
  public double paint(byte[] color, int layers) {
    // doThePaintingJob(color,layers);
    return layers * surface(); // SolidCircle < Circle < Surfaceable
  }
  @Override  // To be able to implement Moveable
  public void moveTo(int x, int y) {
    center.moveTo(x,y);
  }
  public static void main(String[] args) {
    SolidCircle sc = new SolidCircle(new Point(0,0), 3);
    Circle c = sc;      double d = c.surface();      // SolidCircle < Circle
    Paintable p = sc;   p.paint(new byte[]{0,0,0},2);// SolidCircle < Paintable
    Moveable m = sc;    m.moveTo(1, 1);              // SolidCircle < Moveable
  }
}
```

Compiler verifications

- **All** (abstract) **methods declared** in all implemented interfaces by a class **must be implemented** in the class
  - Defined with their code
- Accessibility modifier must be **`public`**
  - Even if we give default (package) accessibility to the interface, compiler adds **`public abstract`**

Compiler verifications

- What if several methods with same name and same signature from distinct interfaces have to be implemented in a single class?
  - They are « promises » (functionalities), and not implementations…
    - Thus, they are (syntactically) compatible
    - But it would be better if they were consistent!

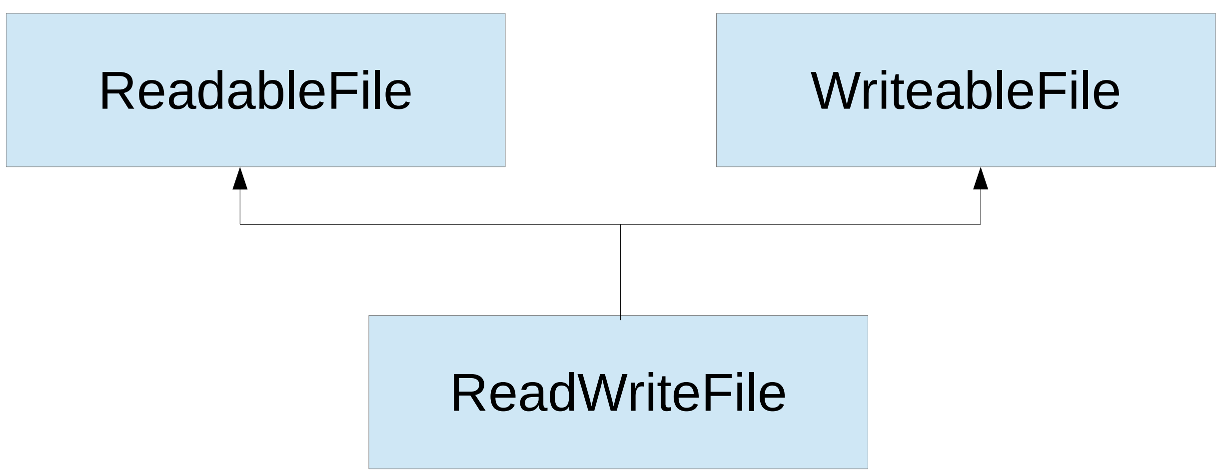
Interface inheritance another example

An interface extending other interfaces gather their promises:

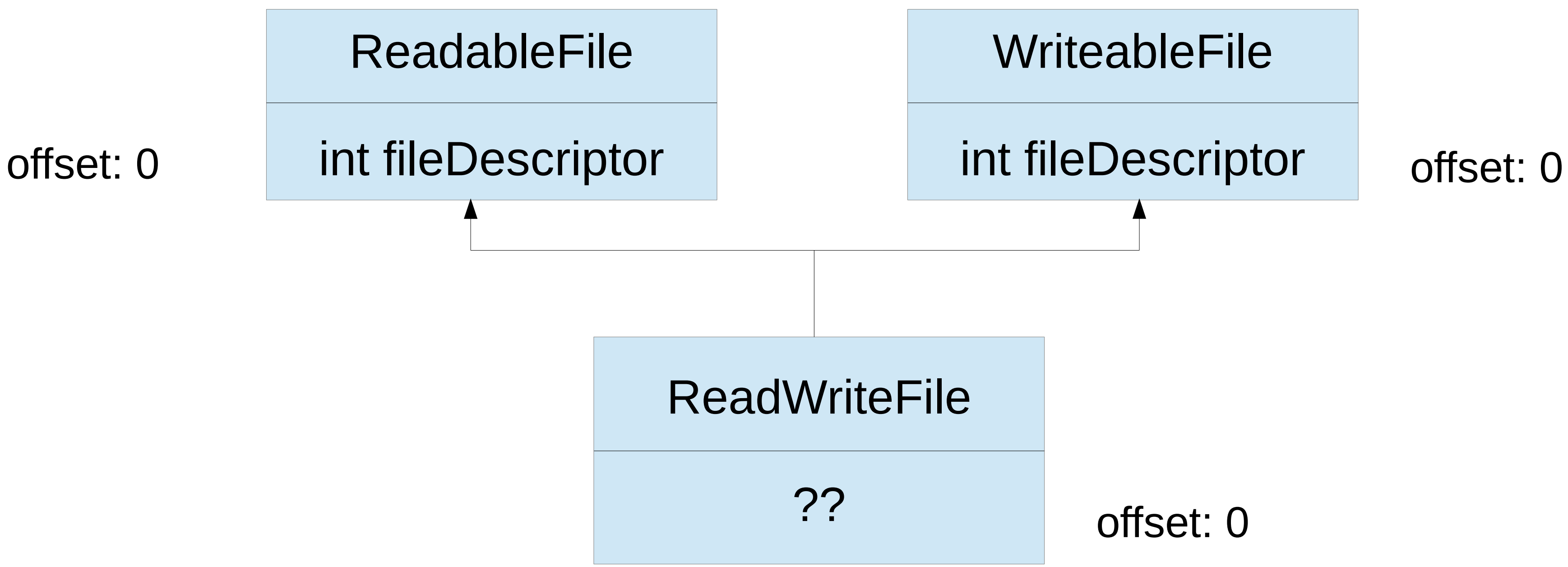
```
public interface ReadableIO {
  int length();
  int read(Buffer buffer);
}
public interface WritableIO {
  int length();
  int write(Buffer buffer);
}
public interface IO extends ReadableIO, WritableIO {
  // 3 methods: read, write et length
}
```

Single inheritance

- In Java (or C#), contrary to C++,
  it is only possible to extend a single class
  - There is no multiple inheritance of class
- And what if we want a class representing files that are both readable and writable?



Why not multiple inheritance?



Since fields are represented by an index, multiple inheritance would introduce conflicts between index!

In C++, this problem is handled by base address offset, beurk !

The problem of multiple inheritance

- It is not possible to have at the same time
  - Multiple inheritance of classes
  - A single header for an object (which is fairly important for the GC)
- There is no problem if there is no field!
- Solution comes with interfaces and multiple sub-typing

Summary : interface

An interface is a set of abstract methods (or not since Java 8) but **without fields!**

An interface is an **abstract type** allowing us to handle several distinct classes with a single common code

Interface agreement

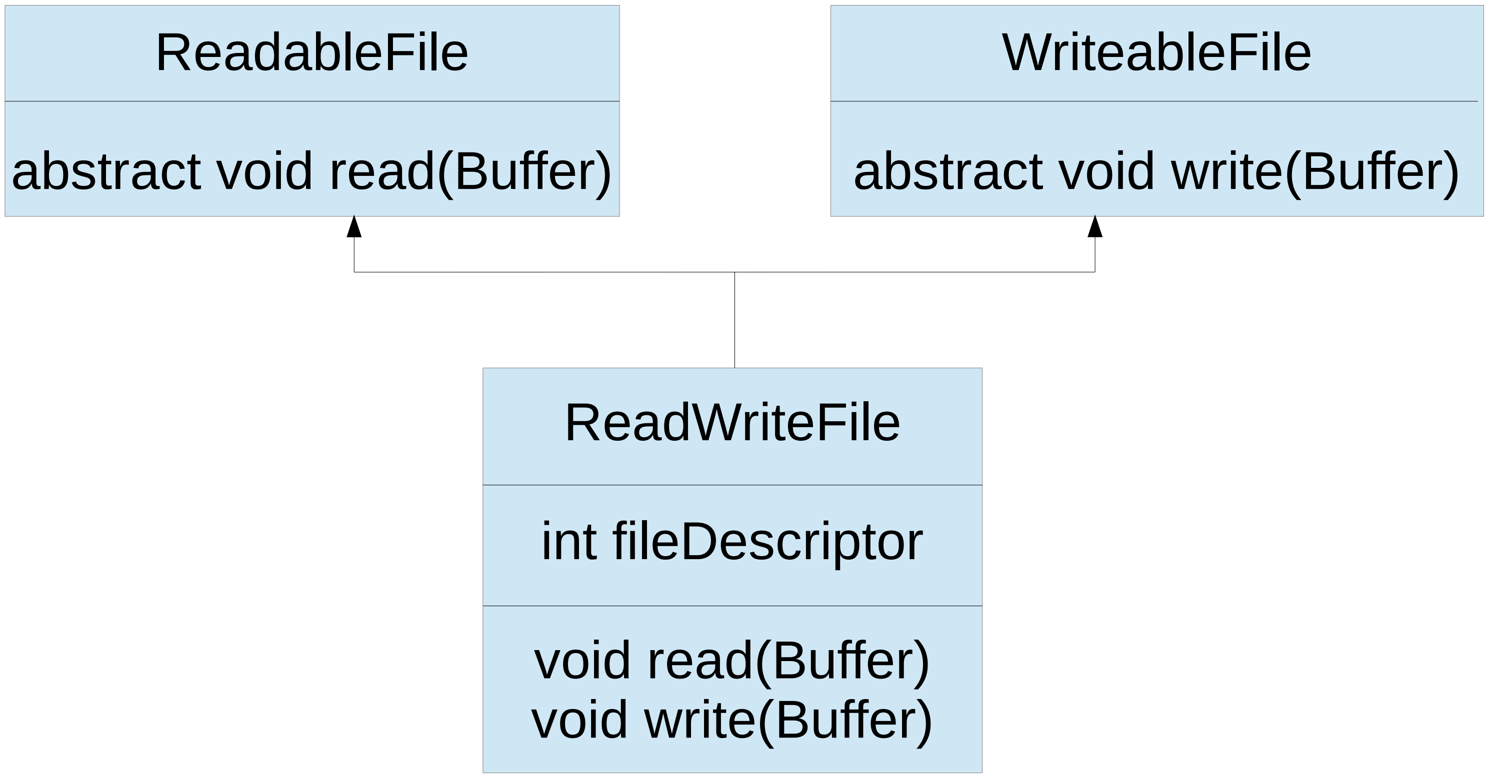
An interface specifies a **contract** that classes implementing it must respect

- Classes must implement all abstract methods

An interface allows us to get **sub-typing** and **polymorphism** without inheritance of fields and methods

- can be seen as a simplified form of inheritance

multiple sub-typing



A class can implements several interfaces

Default implementation of a method (Java 8)

A **default method** is a non-abstract method in an interface

```
public interface Bag {
  public abstract int size();
  public default boolean isEmpty() {
    return size() == 0;
  }
}
```

Default implementation of a method (Java 8)

A defaut method implementation (code) is used when no other implantation is given

```
public class HashBag implements Bag {
  private int size;
  ...
  public int size() {
    return size;
  }
  // isEmpty default of Bag is used
}
```

default method and conflict

If two default methods are available, those of the sub-type – if any – is chosen; else, compiler warns you:

```
public interface Empty {
  public default boolean isEmpty() {
    return true;
  }
}
public class EmptyBag implements Bag, Empty {
  // problem: 2 default methods isEmpty() are available
}
```

Default methods in interface and toString, equals and hashCode

Since java.lang.Object always provides methods toString, equals and hashCode, its is useless to define en explicit default method toString, equals or hashCode in an interface.

The implementation of java.lang.Object will always be chosen instead of the interface one

Resolving conflict

It could be necessary to help the compiler resolving a ambiguity

```
public interface Empty {
  public default boolean isEmpty() {
    return true;
  }
}
public interface Bag {
  ...
  public default boolean isEmpty() {
    ...
  }
}
public class EmptyBag implements Bag, Empty {
  public boolean isEmpty() {
    return Empty.super.isEmpty();
  }
}
```

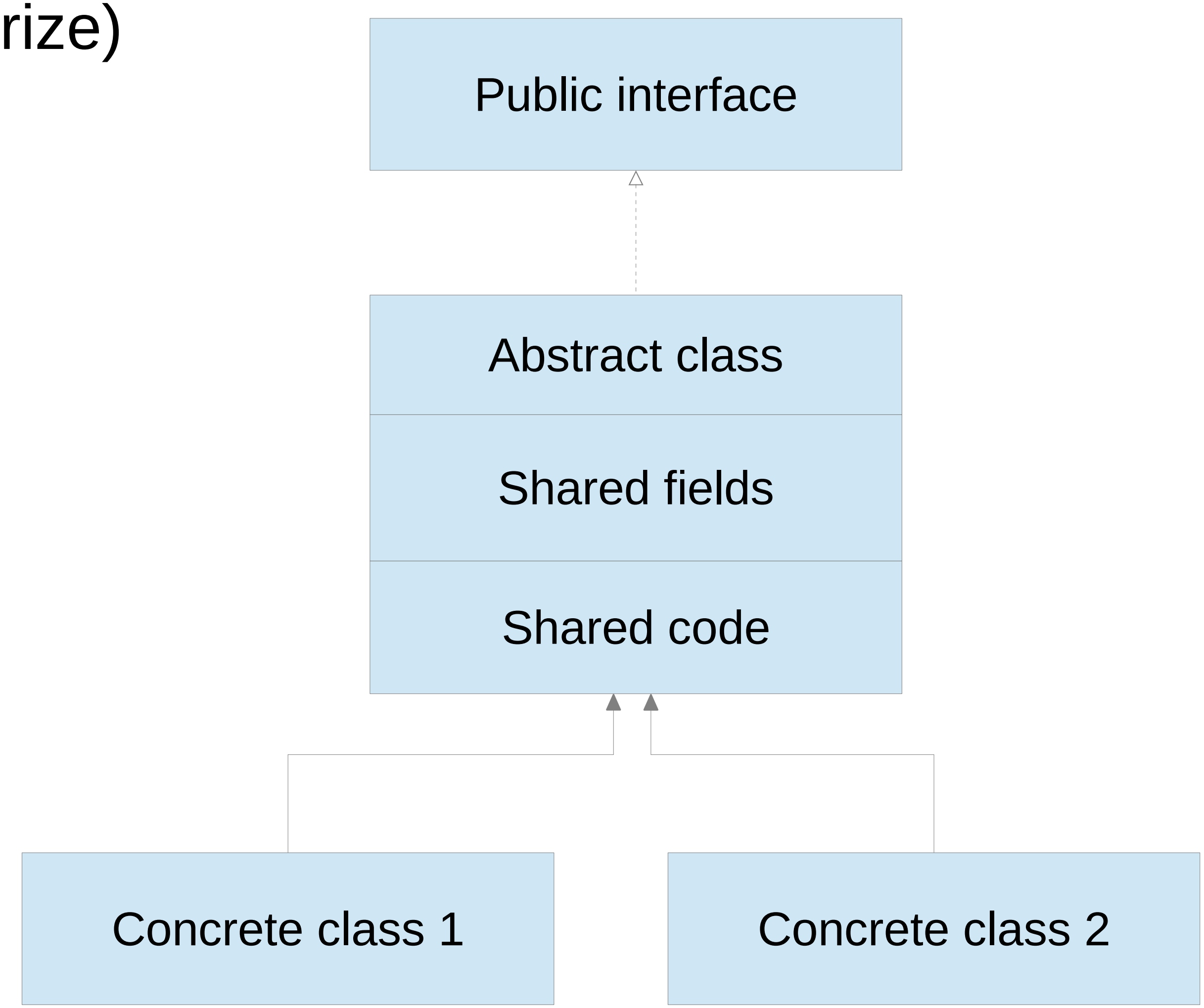
SuperInterface.super allows to points out a given default implementation in an interface

Design: interface or inheritance

- **We extend a class**
  - to create a new type that stands for « **a kind of** » super-class' type
- We **define an interface and implement it**
  - For a transversal functionality
    - Comparable, Closeable, *Mesurable, Movable...*
  - In order to gather a set of functionalities that could be implemented by classes that already implements other interfaces, or that extends another class

Abstract class

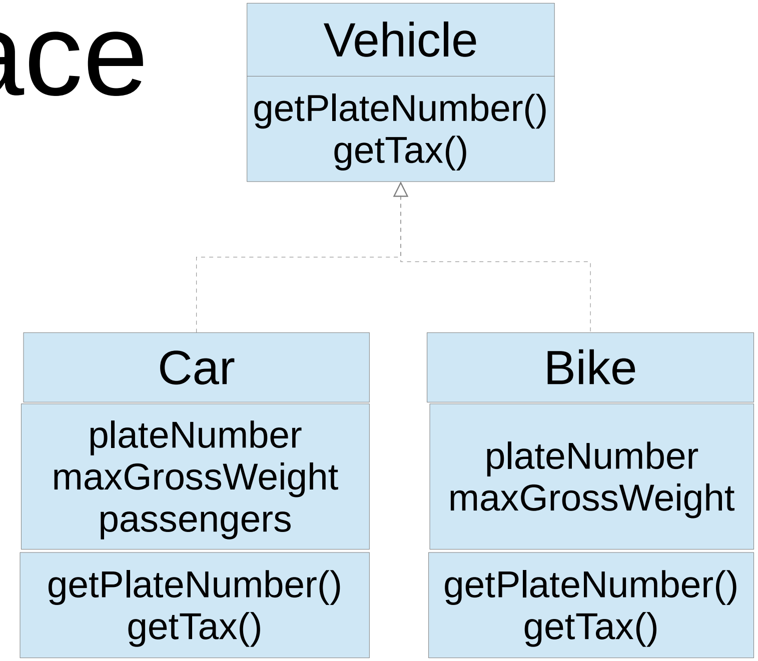
- You can define a class in which some methods are abstract
  - Useful to share (factorize) some fields an code
- Classical design :



```java
public class Car {
  private final String plateNumber;
  private final int maxGrossWeight;
  private final int passengers;
  public Car(String plateNumber, int maxGrossWeight, int passengers) {
    this.plateNumber = plateNumber;
    this.maxGrossWeight = maxGrossWeight;
    this.passengers = passengers;
  }
  public String getPlateNumber() {
    return plateNumber;
  }
  public int getTax() {
    return passengers * maxGrossWeight / 10;
 }
}
```

```java
public class Bike {
  private final String plateNumber;
  private final int maxGrossWeight;
  public Bike(String plateNumber, int maxGrossWeight) {
    this.plateNumber = plateNumber;
    this.maxGrossWeight = maxGrossWeight;
  }
  public String getPlateNumber() {
    return plateNumber;
  }
  public int getTax() {
    return maxGrossWeight / 2;
  }
}
```

Common type => interface




```java
public interface Vehicle {
  String getPlateNumber();
  int getTax();
}
```

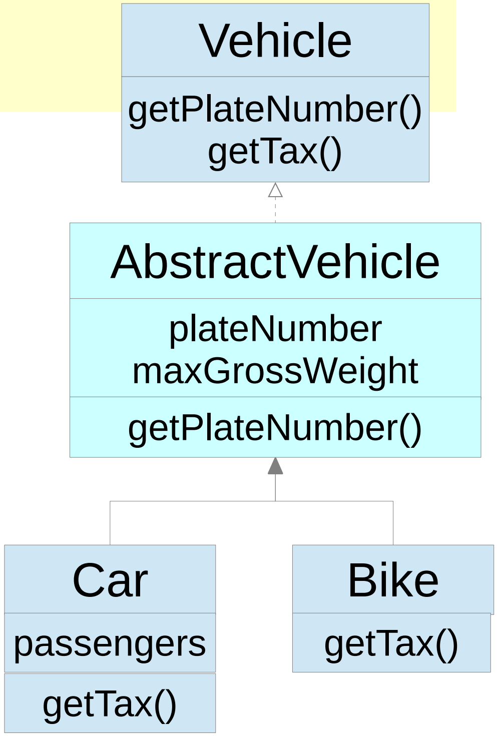
```java
public class Car implements Vehicle {
  ...
}
```

```java
public class Bike implements Vehicle {
  ...
}
```

```java
public static void main(String[] args) {
  Vehicle[] array = {
    new Car("AA-111-AA", 1850, 5),
    new Bike("BB-222-CC", 450),
    ... };
  for(Vehicle v : array) {
    System.out.println(v.getPlateNumber() + " must pay " + v.getTax());
  }
}
```

Factorization of fields / code => abstract class

```java
public interface Vehicle {
  String getPlateNumber();
  int getTax();
}
```




```java
abstract class AbstractVehicle implements Vehicle {
  private final String plateNumber;
  private final int maxGrossWeight;
  public AbstractVehicle(String plateNumber, int maxGrossWeight) {
    this.plateNumber = plateNumber;
    this.maxGrossWeight = maxGrossWeight;
  }
  public String getPlateNumber() {
    return plateNumber;
  }
  int getMaxGrossWeight() { // package accessibility
    return maxGrossWeight;
  }
}
```

```java
public class Car extends AbstractVehicle {
  private final int passengers;
  public Car(String plateNumber,
             int maxGrossWeight,
             int passengers) {
    super(plateNumber, maxGrossWeight);
    this.passengers = passengers;
  }
  public int getTax() {
    return passengers * getMaxGrossWeight() / 10;
  }
}
```

```java
public class Bike extends AbstractVehicle {
  public Bike(String plateNumber,
              int maxGrossWeight) {
    super(plateNumber, maxGrossWeight);
  }
  public int getTax() {
    return getMaxGrossWeight() / 2;
  }
}
```

Abstract class and instantiation

Just like an interface, an abstract class cannot be instantiated
AbstractVehicle v = new AbstractVehicle();
doesn't compile

An class can be declared abstract without abstract method => this forbids its instantiation

If a method in a class is abstract, then the class must be declared abstract

Abstract method and...

A method cannot be both abstract and static: nonsense

- abstract: must be overridden
- static: impossible to override

A method cannot be both abstract and private: nonsense

- abstract: must be overridden in sub-class
- private: not accessible outside (including sub-class)

Sub-class and protected

- Accessibility protected means accessible
  - Either from classes in same package
  - Or by (extending) sub-classes in other packages
- This allows some methods to be accessible by all sub-classes, but not public…
- You **should not** declare a field as **protected**, because sub-classes could use it and then avoid any (intern) modifications of your class
- Usually you must avoid abstract classes to be public => only (intern) implementation purpose

Refinement of abstraction

From pure abstraction to implementation

- Interface

  Only abstract methods (public)

- Interface with default methods
  - Abstract and implemented methods (public)
- Abstract class
  - Fields + abstract and implemented methods
- Class
  - Fields + implemented methods

static methods can be defined anywhere

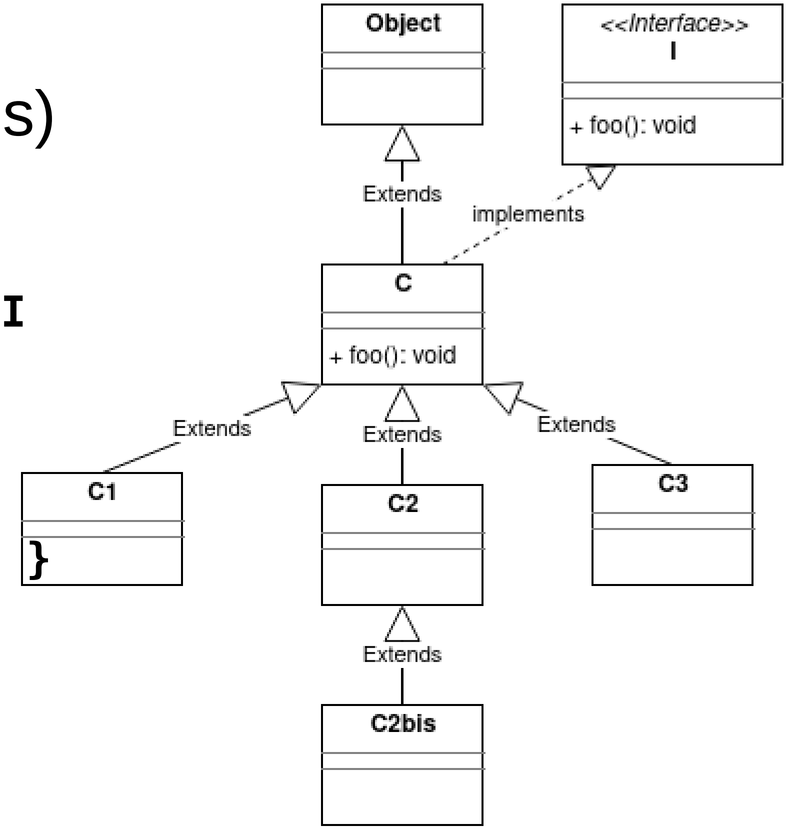
Restriction of sub-types : **`sealed`**

/!\ Preview feature version 15

- classes or interfaces **`sealed`** restrict/limit the set of classes or interfaces allowed to extends them or implement them
- Goal : control/limit the amount of code to « manage », i.e. set of sub-types intended to respect the « contract » of the super-type…
- Clause **`permits`** lists all sub-types « authorized », and forbids all other unknown sub-types...

Each sub-type listed by `permits`

- Must directly extends/implements the sealed type
- Must have explicitly one of the 3 modifiers
  - `final` (one cannot extend it)
  - `sealed` (specified – again – authorized sub-types)
  - `non-sealed` (relax all constraints for sub-types)




```java
public interface I { ... }

public sealed class C extends Object implements I
    permits C1, C2, C3 { ... }

public final class C1 extends C { }

public sealed class C2 extends C permits C2bis { }

public non-sealed class C3 extends C { }

public final class C2bis extends C2 { }
```

Sealed / permits (preview feature Java15)

When key-word **`sealed`** is used to declare a class or an interface

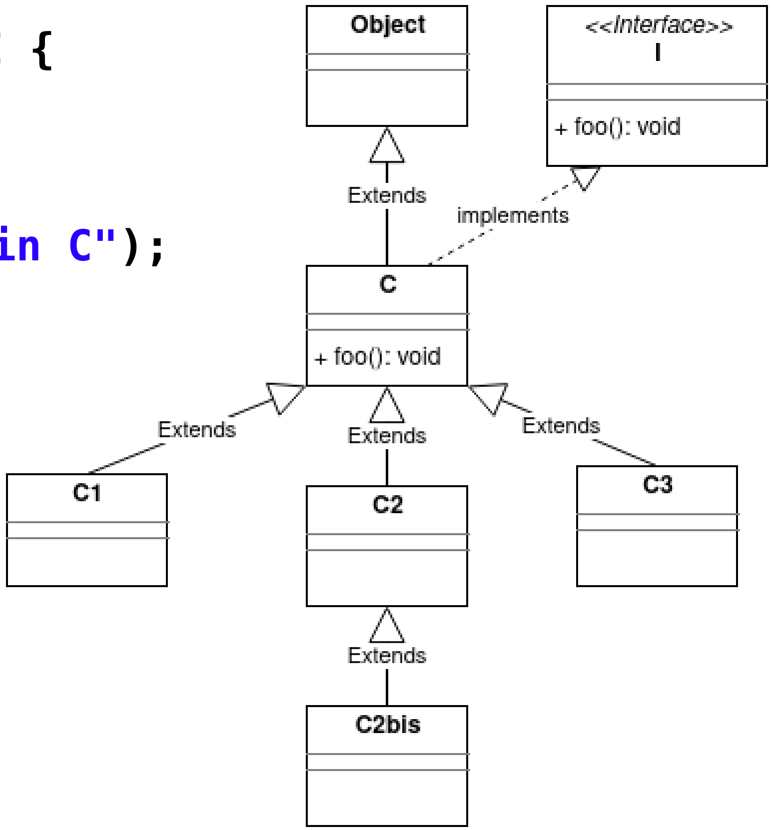
- You must add the key-word **`permits`**
  - After clauses `extends` and `implements`
  - With all authorized sub-types
  - All authorized sub-types must be known at compile-time

```
public interface I {
    void foo();
}
public sealed class C extends Object implements I permits C1, C2, C3 {
    @Override
    public void foo() {
        System.out.println("foo() implementation in C");
    }
}
```

Inference of “permits”

– Compiler can “infer” (guess) permits as long as all of them are declared in the same file

```java
public sealed class C extends Object implements I {
    // permits C1, C2, C3 {
    @Override
    public void foo() {
        System.out.println("foo() implementation in C");
    }
}
final class C1 extends C { }
sealed class C2 extends C permits C2bis { }
final class C2bis extends C2 { }
non-sealed class C3 extends C { }
```



Same principle for interfaces

limit authorized sub-interfaces / sub-classes

A record can be part of the permits

- In this case, final key-word is not mandatory

```java
public sealed interface I permits C, Rhi, J {
     void foo();
}
record Rhi(int v1, int v2) implements I {
     @Override
     public void foo() {
          System.out.println("foo() implementation in Rhi");
     }
}


public sealed interface J extends I permits Rjay {
     void bar();
}
record Rjay(String name, int value) implements J {
     @Override
     public void foo() {
          System.out.println("foo() implementation in Rjay");
     }
    @Override
     public void bar() {
          System.out.println("bar() implementation in Rjay");
     }
}
```