Programming Software Systems

Introduction to Programming
for the Computer Engineering Track

Lecture 9 + Tutorial 9
An Introduction to Java

Eugene Zouev
Fall Semester 2020
Innopolis University

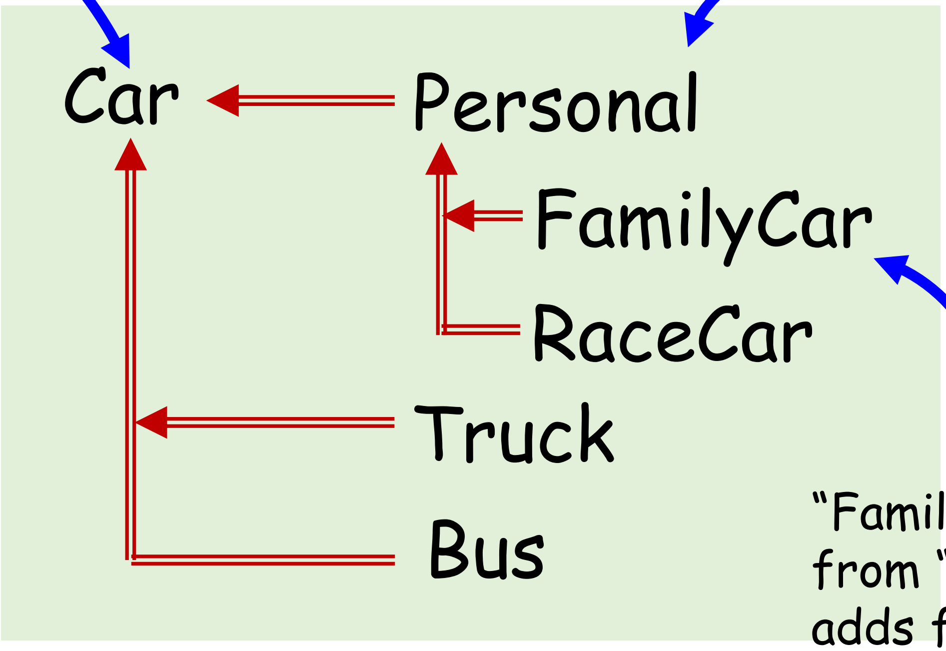
What We Have Learnt

- Classes, class instances
- Value types and reference types
- Encapsulation, Inheritance, Polymorphism

Inheritance

"Car" defines features **common** to all kinds of cars, e.g.:

- Max. speed
- Engine
- Capacity
- Acceleration
- Etc.



"Personal" **inherits** all features from Car and adds features specific for personal cars, e.g.:

- No. of passengers
- Kind of transmiss.
- Etc.

"FamilyCar" inherits all features from "Car" and "Personal" and adds features specific for family cars, e.g.:

- Seats for children
- Navigator
- Etc.

Single Inheritance

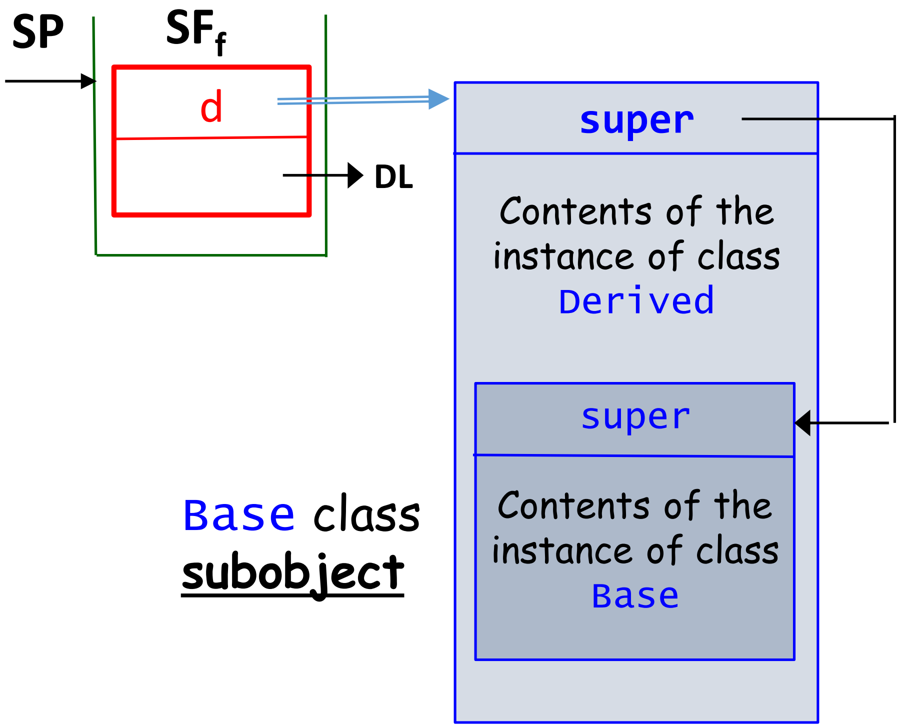
The "subobject" notion

```
class Base
{
    // Members
    // of class Base
}
```

```
class Derived extends Base
{
    // Members
    // of class Derived
}
```

```
class Other
{
    void f() {
      Derived d = new Derived()
    }
}
```

The structure of objects of class Derived:



Static & Dynamic Types

Static type of `figure` is `Shape`: it is specified statically, in the program text.

```
Circle circle = new Circle();
...
Shape figure = circle;
```

This is the conversion: **from derived type to base type**

After this assignment `figure` refers to an instance of class `Circle`. It's said, that the **dynamic type** of `figure` **now** is `Circle`.

Polymorphism

Small remark:
In Java, all methods are by default **virtual**.

The main rule of polymorphism

The interpretation of the call of a <u>virtual</u> method **depends on the type of the object for which it is called (the <u>dynamic type</u>)**,

whereas

the interpretation of a call of a non-virtual method function depends only on the type of the reference denoting that object (the **<u>static type</u>**).

Polymorphism

Late binding

```
class Base
{
  public int f(int p) { return x*x; }
}

class Derived extends Base
{
  public int f(int p) { return x*x*x; }
}
```

These two methods have the same signature

This method <u>overrides</u> the method with the same signature from the base class

```
class SomeOtherClass
{
  public void someOtherMethod()
  {
    int result;
    Base m = new Base(); result = m.f(3);
    m = new Derived();   result = m.f(3);
  }
}
```

Here, the dynamic type of m is Base. The method f from Base gets called

The static type of m is (always) Base

Here, the dynamic type of m is Derived. The method f from Derived gets called!

What's For Today

- Class `Object`
- Casts
- Abstract classes
- Interfaces

Universal Base Class Object 1

```
class Base { ... }

class Derived extends Base { ... }
```

Class Derived inherited from Base.
What about a superclass for Base???

C++ answer:
Base doesn't inherit from any other class

Universal Base Class `Object` 1

```
class Base { ... }

class Derived extends Base { ... }
```

Class `Derived` inherited from `Base`.
What about a superclass for `Base`???

C++ answer:
`Base` <u>doesn't inherit</u> from any other class

Java answer

- Each Java class always inherits (either directly or indirectly) the special **system-defined class** called `Object`.

  Even if a class doesn't contain **`extends`** clause it inherits `Object`. Therefore it's not needed to write the **`extends`** clause: the system does it for you.

  This means that any class hierarchy always has the single "root" class, and this class is `Object`.

Universal Base Class `Object` 2

- `Object` class contains a few methods that can be overridden by derived classes.

  Perhaps the most interesting methods are `toString()` and `equals()`.

  Even if a class doesn't contain **`extends`** clause it inherits `Object`.

Universal Base Class `Object` 2

- `Object` class contains a few methods that can be overridden by derived classes.

  Perhaps the most interesting methods are `toString()` and `equals()`.

  Even if a class doesn't contain **`extends`** clause it inherits `Object`.

- `toString()` converts the object it belongs to, to string representation.
- `equals()` performs comparison of two objects: the first is the object this method belongs to, and the second one is passed via parameter.

Downcasting & Type Checks 1

Upcasting:
Each Lion is an Animal
This relation is **always true** => conversion form Lion to Animal is always correct and safe.

Downcasting:
If **this** particular Animal **is actually** a Lion (if we know this for sure ☺) then the cast to the derived class is correct and safe.

```
class Animal { ... }

class Lion extends Animal { ... }
class Frog extends Animal { ... }
 ...
Animal a = new Frog();
 ...
a = new Lion();
... (Lion)a ...
```

Downcasting & Type Checks 1

Upcasting:
Each Lion is an Animal
This relation is **always true** => conversion form Lion
to Animal is always correct and safe.

Downcasting:
If **this** particular Animal **is actually** a Lion (if we know this for sure ☺) then the cast to the derived class is correct and safe.

a can refer to an object of class Animal OR to an object of its derived class

```
class Animal { ... }

class Lion extends Animal { ... }
class Frog extends Animal { ... }
 ...
Animal a = new Frog();
 ...
a = new Lion();
... (Lion)a ...
```

Downcasting & Type Checks 1

Upcasting:
Each `Lion` is an `Animal`
This relation is **always true** => conversion form `Lion` to `Animal` is always correct and safe.

Downcasting:
If **this** particular `Animal` **is actually** a `Lion` (if we know this for sure ☺) then the cast to the derived class is correct and safe.

a can refer to an object of class `Animal` OR to an object of its derived class

Here we know for sure that a refers to the object of class `Lion` (i.e., the dynamic type of a is `Lion`) => the cast is safe

```
class Animal { ... }

class Lion extends Animal { ... }
class Frog extends Animal { ... }
  ...
Animal a = new Frog();
  ...
a = new Lion();
... (Lion)a ...
```

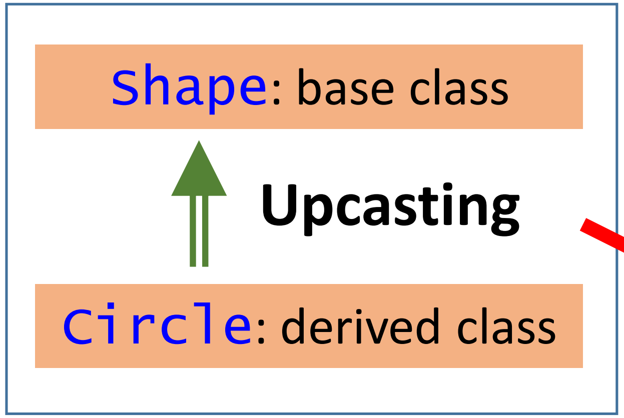
Downcasting & Type Checks 2

```
class Shape { ... }

class Circle extends Shape { ... }
```

```
Circle circle = new Circle();
...
Shape figure = circle;
...
Circle c2 = (Circle)figure;
```

Downcasting & Type Checks 2




```
class Shape { ... }

class Circle extends Shape { ... }
```

```
Circle circle = new Circle();
...
Shape figure = circle;
...
Circle c2 = (Circle)figure;
```

Basic OOP rule:

- **Object of the derived type <u>can be converted</u> to an object of the base type**

The rule is based on the relation "is a":

Circle **is a** Shape hence Circle can be treated as Shape.

Downcasting & Type Checks 2

```
class Shape { ... }

class Circle extends Shape { ... }
```

```
Circle circle = new Circle();
...
Shape figure = circle;
...
Circle c2 = (Circle)figure;
```

Shape: base class

Upcasting

Circle: derived class

Shape: base class

Downcasting

Circle: derived class

Basic OOP rule:

- **Object of the derived type <u>can be converted</u> to an object of the base type**

The rule is based on the relation “is a”: Circle **is a** Shape hence Circle can be treated as Shape.

<u>Upcasting</u>: always valid

<u>Downcasting</u>: valid only if the instance is actually of the target type

Downcasting & Type Checks 3

Type check operator: `instanceof`

```
obj instanceof Class
```

RTTI:
run-time type identification

Returns `true` if dynamic type of `obj` is `Class` OR **any of its derived classes**, and `false` otherwise

Downcasting & Type Checks 3

Type check operator: `instanceof`

```
obj instanceof Class
```

RTTI:
run-time type identification

Returns `true` if dynamic type of `obj` is `Class` OR **any of its derived classes**, and `false` otherwise

```
class Animal { ... }
class Lion extends Animal { ... }
...
Animal a = new Lion();
boolean r1 = a instanceof Animal;      // true
boolean r2 = a instanceof Lion;        // true
boolean r3 = a instanceof Car;         // false
```

C++: `typeid` operator (*not exactly the same*)
C#: `is` operator **(!!!)**

Downcasting & Type Checks 4

Static type of a is Animal.
a can refer to an object of types Animal, Lion, or Frog.

```
class Animal { public int f1; }
class Lion extends Animal { public int f2;}
class Frog extends Animal { public int f3;}

Animal a = new Lion();
...
a = new Frog();
...
if (a instanceof Lion)
    // Downcasting is safe here
    ...((Lion)a).f1...
else if (a instanceof Frog)
    ...((Frog)a).f3...
```

Downcasting & Type Checks 4

Static type of a is Animal.
a can refer to an object of types Animal, Lion, or Frog.

```
class Animal { public int f1; }
class Lion extends Animal { public int f2;}
class Frog extends Animal { public int f3;}

Animal a = new Lion();
...
a = new Frog();
...
if (a instanceof Lion)
   // Downcasting is safe here
   ...((Lion)a).f1...
else if (a instanceof Frog)
   ...((Frog)a).f3...
```

Here, a is treated as Lion. Therefore, features from Lion (and, of course, Animal) are accessible via a.

Static type of a is (still) Animal. Actually, a refers to the object of type Frog. The dynamic type of a is Frog.

Abstract Classes & Methods 1

An informal introduction from Prof Giancarlo Succi:

Sometimes, a class that you define represents an abstract concept and, as such, should not be instantiated.

Take, for example, **food** in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (our favorite) chocolate.

Food represents the abstract concept of things that we all can eat. It doesn't make sense for an instance of food to exist.

Abstract Classes & Methods 1

An informal introduction from Prof Giancarlo Succi:

Sometimes, a class that you define represents an abstract concept and, as such, should not be instantiated.

Take, for example, **food** in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (our favorite) chocolate.

Food represents the abstract concept of things that we all can eat. It doesn't make sense for an instance of food to exist.

(Zouev's addition ☺)
However we know for sure that each kind of food has some **common features**: attributes & behavior. For example, "caloricity", ingredients, the way of cooking etc. We know nothing about "caloricity of food" (it's just an *abstract* feature), but know caloricity of apple…

Abstract Classes & Methods 2

So the **conclusion** is:
If you are going to represent an abstract notion in your program, think about making the corresponding class **abstract**.

```
abstract class Vehicle
{
  // Features that are common
  // to all possible vehicles
  Color color;
  int numWheels;
  ...
  abstract void startEngine();
  ...
}
```

Abstract Classes & Methods 2

So the **conclusion** is:
If you are going to represent an abstract notion in your program, think about making the corresponding class **abstract**.

```
abstract class Vehicle
{
   // Features that are common
   // to all possible vehicles
   Color color;
   int numWheels;
   ...
   abstract void startEngine();
   ...
}
```

We **cannot create instances of abstract classes**: what does it mean "an instance of a vehicle"?

In the abstract class we can define behavior of each categories of vehicles – without any detalization (no body)
These are **abstract methods**.

Abstract Classes & Methods 2

So the **conclusion** is:
If you are going to represent an abstract notion in your program, think about making the corresponding class **abstract**.

```
abstract class Vehicle
{
  // Features that are common
  // to all possible vehicles
  Color color;
  int numWheels;
  ...
  abstract void startEngine();
  ...
}
```

```
abstract class Car extends Vehicle
{ ... }
```

We **cannot create instances of abstract classes**: what does it mean “an instance of a vehicle”?

In the abstract class we can define behavior of each categories of vehicles – without any detalization (no body)
These are **abstract methods**.

Classes representing “real” vehicles are declared as derived classes. They can be “usual” classes OR in turn abstract ones!

Abstract Classes & Methods 3

Some remarks & details

- One could correctly argue that deriving from **class Vehicle** is only a way to logically group objects of the derived classes.
- No "Vehicle" objects exist in real life: we have cars, planes, trains, bikes, etc., but no "generic" vehicles.
- Java, C#, C++: abstract classes;
  Eiffel: deferred classes.
- Java, C#: abstract methods;
  C++: ***pure virtual*** methods.
- A class that is declared abstract does not have to have abstract methods in it.
- A class containing an abstract method must be declared abstract.

Abstract Classes & Methods 4

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This “preliminary” declaration is only to tell the developer of derived classes that *the implementation of the method* ***is required*** *in all direct subclasses that want to become instantiable.*

Abstract Classes & Methods 4

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This "preliminary" declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

```
class Motobike extends Vehicle
{
    ...
    void startEngine()
    {
      // real algorithm
    }
}
```

If the derived class provides implementations **for all** abstract methods from its superclass then this derived class **is not considered abstract**. – It's a "real" class...

Abstract Classes & Methods 4

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This “preliminary” declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

```
class Motobike extends Vehicle
{
    ...
    void startEngine()
    {
      // real algorithm
    }
}
```

If the derived class provides implementations **for all** abstract methods from its superclass then this derived class **is not considered abstract**. – It’s a “real” class…

```
Motobike my = new Motobike();
```

…and we can create instances of this class.

Abstract Classes & Methods 5

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This "preliminary" declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

Abstract Classes & Methods 5

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This "preliminary" declaration is only to tell the developer of derived classes that *the implementation of the method **is required** in all direct subclasses that want to become instantiable.*

```
class FlyingVehicle extends Vehicle
{
    ...
// void startEngine()
// {
//    // real algorithm
// }
}
```

If the derived class **doesn't provide** implementations for some abstract methods from its superclass then this derived class **is still considered abstract**…

Abstract Classes & Methods 5

```
abstract class Vehicle
{
   ...
   abstract void startEngine();
   ...
}
```

This “preliminary” declaration is only to tell the developer of derived classes that *the implementation of the method* ***is required*** *in all direct subclasses that want to become instantiable.*

```
class FlyingVehicle extends Vehicle
{
    ...
// void startEngine()
// {
//    // real algorithm
// }
}
```

If the derived class **doesn’t provide** implementations for some abstract methods from its superclass then this derived class **is still considered abstract**…

```
FlyingVehicle my =
     new FlyingVehicle(); // ERROR
```

…and we **cannot** create instances of this class.

Interface & Implementation 1

What should be inherited:
interface and/or implementation?

```
class Airplane {
  public void fly()
  {
    // Standard flying algorithm
  }
}

class AirbusA extends Airplane {
  ...
  // here fly() is not overriden;
  // standard algorithm is used
}

class AirbusB extends Airplane {
  ...
  // fly() is not overriden;
  // standard algorithm is used
}
```

Interface & Implementation 1

What should be inherited:
interface and/or implementation?

```
class Airplane {
  public void fly()
  {
    // Standard flying algorithm
  }
}

class AirbusA extends Airplane {
  ...
  // here fly() is not overriden;
  // standard algorithm is used
}

class AirbusB extends Airplane {
  ...
  // fly() is not overriden;
  // standard algorithm is used
}
```

```
Airplane a = new AirbusA();
a.fly();  // Airplane‘s fly
...
Airplane b = new AirbusB();
b.fly();  // Airplane‘s fly
```

Here, the implementation of `fly()` is inherited

- Is it *always* good?

Interface & Implementation 2

```
class Airplane {
  public void fly()
  {
    // Standard flying algorithm
  }
}

class AirbusA extends Airplane
{ ... }

class AirbusB extends Airplane
{ ... }

class Boeing extends Airplane {
  ...
  // Standard fly() algorithm is
  // inherited!
  // - But here should be another
  // algorithm!
}
```

What should be inherited:
interface and/or implementation?

```
Airplane c = new Boeing();
c.fly();  // Airplane's fly
```

Here, Boeing has to fly by Airbus' algorithm!?..

Interface & Implementation 3

```
abstract class Airplane {
  public abstract void fly();
  protected void defaultFly()
  {
    // Standard flying algorithm
  }
}

class AirbusA extends Airplane {
  void fly() { defaultFly();
}

class AirbusB extends Airplane {
  void fly() { defaultFly(); };
}

class Boeing extends Airplane {
  public void fly() {
    // Boeing's own
    // flying algorithm
  }
}
```

Solution:
separate interface and implementation!

```
Airplane a = new AirbusA();
a.fly();  // Airplane's fly
...
Airplane b = new AirbusB();
b.fly();  // Airplane's fly
...
Airplane c = new Boeing();
c.fly();  // Boeing's fly
```

Here, Boeing has its own flying algorithm.

Interface & Implementation 4

The same solution in C#

```
abstract class Airplane {
  public abstract void fly();
  protected void defaultFly()
  {
    // Standard flying algorithm
  }
}

class AirbusA : Airplane {
  public override void fly() { defaultFly(); }
}

class AirbusB : Airplane {
  public override void fly() { defaultFly(); };
}

class Boeing : Airplane {
  public override void fly() { /* Boeing‘s flying alg. */ }
}
```

Interface & Implementation 5

Conclusions

When you design a base class, and…

- If you need to provide **only interface** - make the method abstract (or **pure virtual** in C++); hide or restrict its implementation (e.g., as a separate method).

- If you want to provide **both interface and implementation** for derived classes - make the method **virtual** (explicitly as in C++/C#, or implicitly as in Java).

- If you wouldn't like to allow derived classes to modify the behavior of the method – make this method **non-virtual** (impossible in Java: all methods are virtual).

Final Methods

- Method overriding is one of Java's most powerful features.
- However, dynamic calls are <u>a bit slower</u> than "usual" call when the method is selected statically.

Late binding:
The concrete method to be called depends on the dynamic type of the object

Final Methods

- Method overriding is one of Java's most powerful features.
- However, dynamic calls are <u>a bit slower</u> than "usual" call when the method is selected statically.
- Therefore, sometimes it might be reasonable to prevent late binding.
- For that, the `final` specifier is used. Methods declared as final cannot be overridden.

Late binding:
The concrete method to be called depends on the dynamic type of the object

Early binding:
The concrete method to be called is selected using <u>the static type</u> of the object

Final Methods

```java
class Base {
  public void meth() {
    System.out.println("Base‘s meth");
  }
}

class Derived extends Base {
  public void meth() {
    System.out.println("Derived‘s meth"); }
  }
}
```

```java
Base b = new Base();
b.meth();         // Base‘s meth
b = new Derived();
b.meth();         // Derived‘s meth
```

Final Methods

```
class Base {
  public void meth() {
    System.out.println("Base‘s meth");
  }
}

class Derived extends Base {
  public void meth() {
    System.out.println("Derived‘s meth"); }
  }
}
```

```
class Base {
  public final void meth() {
    System.out.println("Base‘s meth");
  }
}

class Derived extends Base {
  public void meth() { // ERROR! Can‘t override
    System.out.println("Derived‘s meth"); }
  }
}
```

```
Base b = new Base();
b.meth();         // Base‘s meth
b = new Derived();
b.meth();         // Derived‘s meth
```

Final Methods

Methods declared as `final` can sometimes provide a <u>performance enhancement</u>.

Why:

- The compiler is free <u>to inline calls</u> to final methods because it "knows" they will not be overridden by a subclass.

Early binding

When a small final method is called, the Java compiler can copy the bytecode of the method directly to the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.

Final Methods

```java
class Base {
  public final void meth() {
    System.out.println("Base‘s meth");
  }
}

class Derived extends Base {
  ...  // No overriden meth
}
```

```java
Base b = new Base();
b.meth();          // Base‘s meth
b = new Derived();
b.meth();          // Base‘s meth again!
```

Early binding:
Both calls refer to the same method. Therefore, the compiler knows the method statically, and can:

- Either generate more efficient code for the call
- Or replace the call for the body of the method `meth` “in place”: **inlining**.

Final Classes

Sometimes it's reasonable to prevent a class from being inherited.

```
final class Base
{
    ...
}

class Derived extends Base
{
   ...
}
```

ERROR: Can't subclass Base

- Declaring a class as `final` implicitly declares all of its methods as **finals**, too.
- It's illegal to declare a class as both `abstract` and `final`

  Why? - try to explain.

Final Classes

Sometimes it's reasonable <u>to prevent a class from being inherited</u>.

```
final class Base
{
    ...
}

class Derived extends Base
{
   ...
}
```

ERROR: Can't subclass Base

- Declaring a class as **final** implicitly declares all of its methods as **finals**, too.
- It's illegal to declare a class as both **abstract** and **final**

Why? – try to explain.

Since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Interfaces

(as a special language construct,
but not as a concept ☺)

Two Views at the World

- The world consists of (abstract and real) **objects**
- Objects have **state** (characteristics)
- Objects have **relationships** with other objects

Class-based approach

Two Views at the World

- The world consists of (abstract and real) **objects**
- Objects have **state** (characteristics)
- Objects have **relationships** with other objects

Class-based approach

- All entities in the world are **doing** something (are “active”).
- Therefore, the basic characteristics of an entity is its **behavior**.
- Various kinds of behavior are in some **relationships** with each other.

Interface-based approach

Two Views at the World

- The world consists of (abstract and real) **objects**
- Objects have **state** (characteristics)
- Objects have **relationships** with other objects

- All entities in the world are **doing** something (are “active”).
- Therefore, the basic characteristics of an entity is its **behavior**.
- Various kinds of behavior are in some **relationships** with each other.

Class-based approach → **OOP!** ← Interface-based approach

Interfaces 1

Interfaces is a good alternative to **multiple inheritance**

As a natural continuation
of the previous considerations:

Interface as a special language construct

```
interface Features
{
   int numOfLegs;
   bool canFly();
   bool canSwim();
   ...
}
```

Each class implementing this interface **must contain** methods with specified signature and corresponding implementation.

C++, **Eiffel**: no interfaces (abstr.classes or "deferred" classes)

C#, **Java**: **interfaces**

An interfaces is a **contract** between a class and the outside world.
When a class implements an interface, it promises to provide the behavior published by that interface.

Interfaces 2

```
interface Features
{
  int numOfLegs();
  bool canFly();
  bool canSwim();
  ...
}
```

Interfaces 2

```
interface Features
{
   int numOfLegs();
   bool canFly();
   bool canSwim();
   ...
}
```

- No bodies: **classes** should provide implementations
- No access specifiers: (obviously) **public** by default.
- Interface is not a class: no **new** operator, no interface **instances**.
- ~~Interfaces cannot have **data** – only function signatures.~~
- Interface is a **contract** of an implementing class
- Interface can be treated as (an abstract) **type**.

Interfaces 2

```
interface Features
{
   int numOfLegs();
   bool canFly();
   bool canSwim();
   ...
}
```

```
class Lion implements Features
{ ... }
...
Features f1 = new Features();
  // Error: cannot create
  // instances of interfaces
Features f2 = new Lion();
  // Correct
```

- No bodies: **classes** should provide implementations
- No access specifiers: (obviously) **public** by default.
- Interface is not a class: no **new** operator, no interface **instances**.
- ~~Interfaces cannot have **data** – only function signatures.~~
- Interface is a **contract** of an implementing class
- Interface can be treated as (an abstract) **type**.

Interfaces 3

```
interface Features
{
  int numOfLegs();
  bool canFly();
  bool canSwim();
}

class Lion implements Features
{
  int numOfLegs() { return 4; }
  bool canFly() { return false; }
  bool canSwim() { return true; }
}

Features f = new Lion(); // OK
...
if ( f.canFly() ) ...
```

If a class is declared as implementing an interface…

…This means that the class is responsible to (it must) provide implementations to all features declared in the interface it is implementing!

…And after that we can treat a lion as a ***set of its features*** ☺

Interfaces 4

A class can implement **several** interfaces:
a (kind of) easier and clearer replacement
for **multiple inheritance**

```
class Person implements iBodyParams, iSkills, iRelations, ...
{
  ...
}

Person john = new Person();
...
iSkills johnsSkills = john;
  // Consider “john“ as a set of his skills...
...
```

Interfaces Can Inherit

An interface can **inherit** from other interface(s):

```
interface speedFeatures {
  float maxSpeed();
  float maxAcceleration();
}
interface engineFeatures extends speedFeatures {
  float numOfCyls();
  float enginePower();
}
class Car implements engineFeatures
{
  ...
}
```

Must implement interfaces from both
`speedFeatures` & `engineFeatures`

Classes Inherit Interfaces

Interfaces are inherited (as classes):

```
interface HasLegs {
  int noLegs();
}
class Mammal implements hasLegs
{
  int noLegs() { return 4; }
}
class Lion extends Mammal
{
   ...
}
...
Lion a = new Lion();
int legs = a.noLegs();
```

`Lion` inherits interface's implementation from its base class

Interfaces With Inheritance

Interfaces can be used **together** with inheritance:

```
interface colorFeatures {
  Color color();
  Border border();
}
class Shape {
  abstract void Draw();
  ...
}
class Rectangle extends Shape
{
  ...
}
class ColoredRectangle extends Rectangle, implements colorFeatures
{
  // Inherits from Rectangle
  // and implements features from colorFeatures
}
```

In some sense, interfaces are **orthogonal** to inheritance mechanism...

Interfaces & Type Checks

Type check operators are applicable to interfaces as well:

Interfaces can be **empty**! Sometimes that's useful: they act like "tags".

```
interface Printable { void print(); }
interface Movable { void move(); }
interface Serializable { void serialize(); }

class Shape { ... }

class Rectangle extends Shape
                implements Printable, Movable, Serializable
{
  ...
}
...
Shape a = new Rectangle();
if ( a instanceof Printable )
   ((Printable)a).print(); // valid if a is really Pintable
if ( a instanceof Movable )
   ((Movable)a).move();    // valid if a is really Movable
```

Nested Interfaces

```
class SomeClass {
  public interface Nested
  {
    boolean isNotNegative(int x);
  }
}
class MyClass implements SomeClass.Nested
{
  boolean isNotNegative(int x)
  {
    return x<0 : false : true;
  }
}
class Demo
{
  public static void Main() {
    SomeClass.Nested obj = new MyClass();
    if ( obj.isNotNegative(10) )
      System.out.println("10 is not negative");
  }
}
```

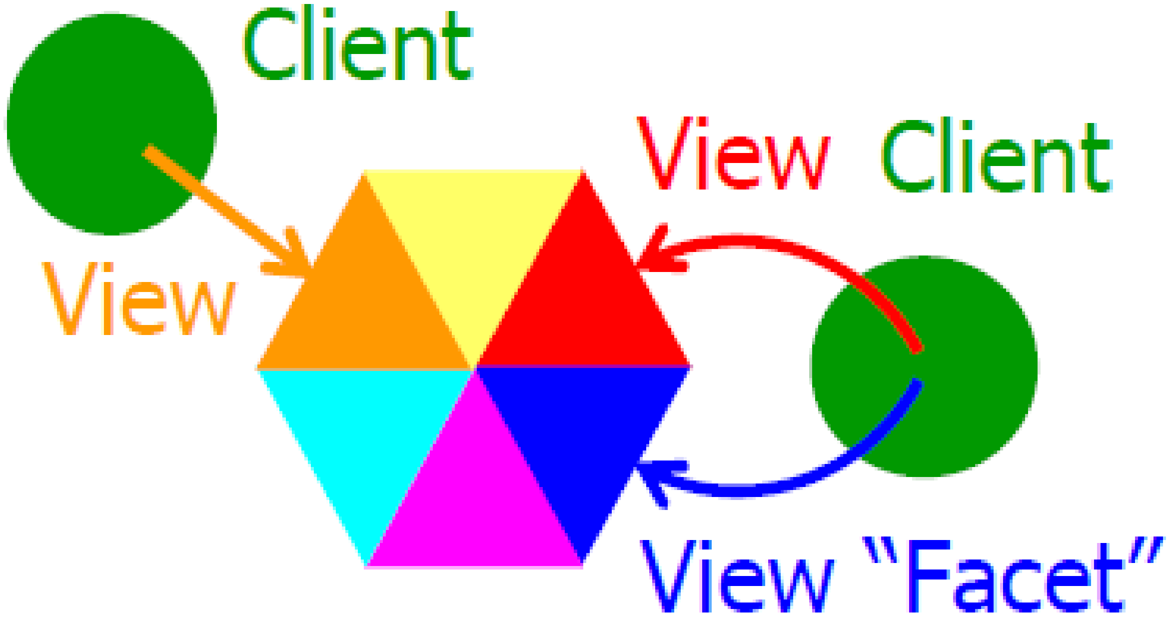
Nested Interfaces

```
class SomeClass {
  public interface Nested
  {
    boolean isNotNegative(int x);
  }
}
class MyClass implements SomeClass.Nested
{
  boolean isNotNegative(int x)
  {
    return x<0 : false : true;
  }
}
class Demo
{
  public static void Main() {
    SomeClass.Nested obj = new MyClass();
    if ( obj.isNotNegative(10) )
      System.out.println("10 is not negative");
  }
}
```

```
interface SharedConstants {
    int No    = 0;
    int Yes   = 1
    int MayBe = 2;
    int Later = 3;
    int Soon  = 4;
    int Never = 5;
}
```

Interfaces as Facets

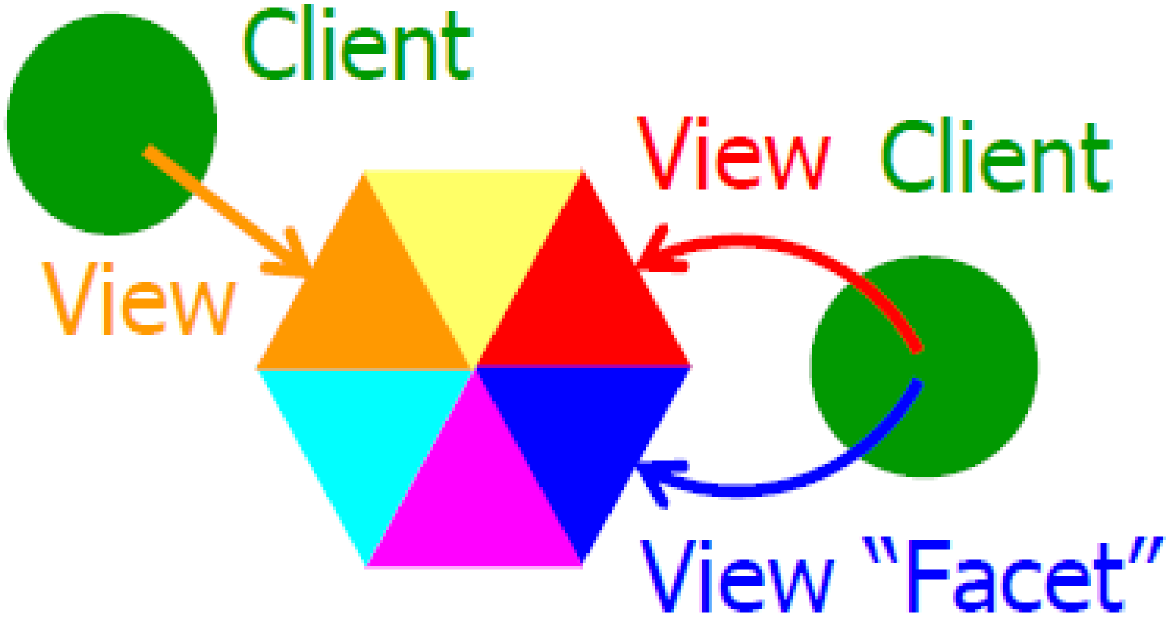
Interfaces can be treated as **various views** at an object (clients' points of views).

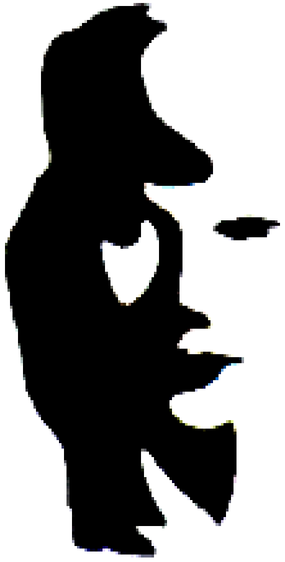


Pictures are taken from a lecture of Prof J.Gutknecht, ETH Zürich

Interfaces as Facets

Interfaces can be treated as **various views** at an object (clients' points of views).

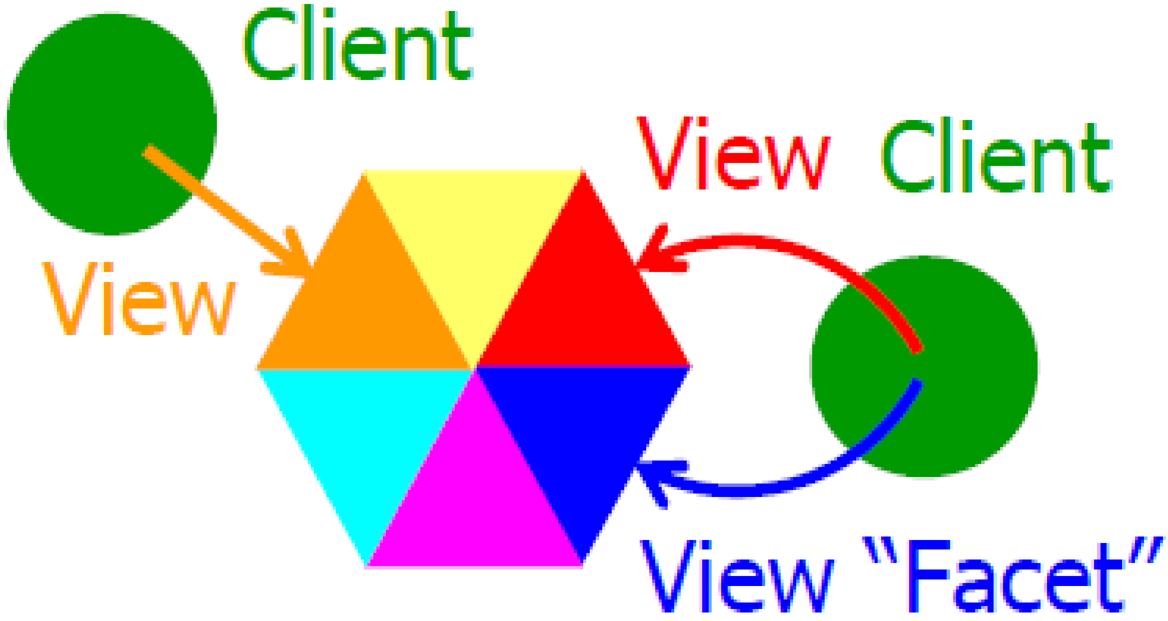


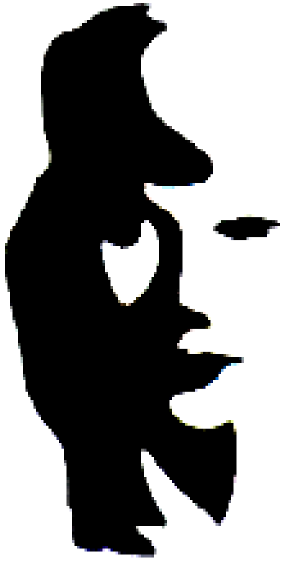


Pictures are taken from a lecture of Prof J.Gutknecht, ETH Zürich

Interfaces as Facets

Interfaces can be treated as **various views** at an object (clients' points of views).





Pictures are taken from a lecture of Prof J.Gutknecht, ETH Zürich

Interfaces vs Abstract Classes

The favorite question on many job interviews! ☺

Similarities:

- Both represent an abstraction.
- Cannot create instances of both.

Differences:

- Interface is a "pure" abstraction: i.e., only abstraction of behavior (can specify only functionality, but not the object state - *the latter is already not so!*)
- Abstract class can contain a) abstract specification of behavior, b) non-abstract functionality, and c) object state.

Interfaces: Ad-hoc Polymorphism

```
interface Frog {
  boolean isGreen();
  boolean canJump();
  boolean canSwim();
  boolean likesToQuack();
}

class Somebody // NO interfaces
{
  boolean isGreen() { return true; }
  boolean canJump() { return true; }
  boolean canSwim() { return true; }
  boolean likesToQuack() { return true; }
}

Somebody likeAFrog = new Somebody();

Frog frog = likeAFrog; // ????
```

Interfaces: Ad-hoc Polymorphism

```
interface Frog {
  boolean isGreen();
  boolean canJump();
  boolean canSwim();
  boolean likesToQuack();
}

class Somebody // NO interfaces
{
  boolean isGreen() { return true; }
  boolean canJump() { return true; }
  boolean canSwim() { return true; }
  boolean likesToQuack() { return true; }
}

Somebody likeAFrog = new Somebody();

Frog frog = likeAFrog; // ????
```

If the last conversion is allowed in a language, then this is so called **ad-hoc polymorphism**.

Interfaces: Ad-hoc Polymorphism

```
interface Frog {
  boolean isGreen();
  boolean canJump();
  boolean canSwim();
  boolean likesToQuack();
}

class Somebody // NO interfaces
{
  boolean isGreen() { return true; }
  boolean canJump() { return true; }
  boolean canSwim() { return true; }
  boolean likesToQuack() { return true; }
}

Somebody likeAFrog = new Somebody();

Frog frog = likeAFrog; // ????
```

If the last conversion is allowed in a language, then this is so called **ad-hoc polymorphism**.

Or… “**duck typing**”: «Если нечто ходит как утка, плавает как утка и крякает как утка, то это, скорее всего, утка и есть». ☺

Tutorial:
Some Useful Idioms and Patterns

The Single Instance?

How to prohibit any creation, except the very first one?
Or, simply speaking, how to provide
creation of exactly one instance of a class?

Why have such an exotic class?

- Cash file
- File with virtual memory pages in OS or VM
- Some kinds of dialogue windows in UI
- Device drivers
- etc.

The Single Instance?

How to prohibit any creation, except the very first one?
Or, simply speaking, how to provide
creation of exactly one instance of a class?

Why have such an exotic class?

- Cash file
- File with virtual memory pages in OS or VM
- Some kinds of dialogue windows in UI
- Device drivers
- etc.

Why not to use just a global variable? (or a static variable)

- Uncontrolled access
- Cannot control creation time

Some attempts

(Yes, this is a very simple task but let's attack it stadially).

- Suppose we have a class: `class myClass { ... }`
- How to create an instance? `new myClass()`
- Can we create many instances? Of course!

Some attempts

(Yes, this is a very simple task but let's attack it stadially).

- Suppose we have a class: `class myClass { ... }`
- How to create an instance? `new myClass()`
- Can we create many instances? Of course!
- How to prevent creation?

Some attempts

(Yes, this is a very simple task but let's attack it stadially).

- Suppose we have a class: `class myClass { ... }`
- How to create an instance? `new myClass()`
- Can we create many instances? Of course!
- How to prevent creation?

```
class myClass
{
    private myClass() { }
}
```

Some attempts

(Yes, this is a very simple task but let's attack it stadially).

- Suppose we have a class: `class myClass { ... }`
- How to create an instance? `new myClass()`
- Can we create many instances? Of course!
- How to prevent creation?

```
class myClass
{
    private myClass() { }
}
```

- Does this solution really prevent creation?

Some attempts

(Yes, this is a very simple task but let's attack it stadially).

- Suppose we have a class:

```
class myClass { ... }
```

- How to create an instance?

```
new myClass()
```

- Can we create many instances?

Of course!

- How to prevent creation?

```
class myClass
{
    private myClass() { }
}
```

- Does this solution really prevent creation? - No:

```
class myClass
{
    private myClass() { }
    public static myClass getInstance() {
        return new myClass();
    }
}
```

What should we add to this code to make the instance created unique?

The Solution: Singleton Pattern

This static member keeps the reference to the single instance of the class. It is initialized by **null** at the very beginning of the program, and gets the reference to the instance after the very first call to getInstance.

```
public class Singleton
{
    private static Singleton unique;

    private Singleton() { }

    public static Singleton getInstance()
    {
        if ( unique == null )
            unique = new myClass();
        return unique;
    }
}
```

Private constructor: only class itself can create instances of the class

The first call to the method creates the unique instance of the class. The following calls just return the same instance

There is no other way to get access to unique except via call to getInstance.

Pattern Bridge: The Problem

The usual way in structuring OOP code:
class with interface and implementation

```
class Class {
  public ...
    // Interface
    ...
  private ...
    // Implementation
   ...
}
```

The problem here is that each class derived from `Class` inherits <u>both interface and implementation</u>.
It's not flexible and in some cases it might cause problems.

Pattern Bridge: The Problem

The usual way in structuring OOP code:
class with interface and implementation

```
class Class {
  public ...
    // Interface
    ...
  private ...
    // Implementation
   ...
}
```

The problem here is that each class derived from `Class` inherits both interface and implementation.
It's not flexible and in some cases it might cause problems.

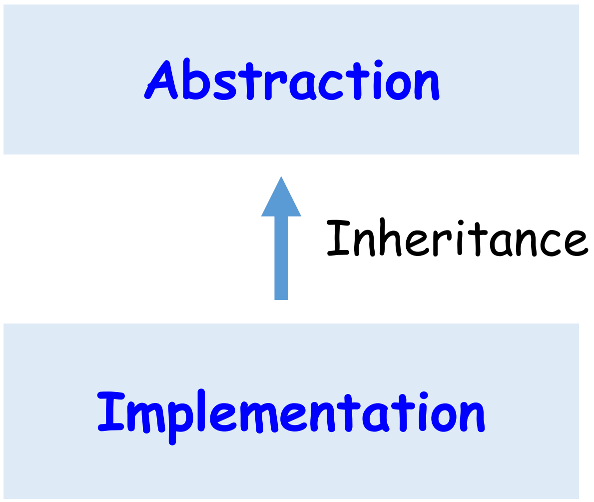
What do we do to make this construct more flexible and reliable: we **separate** interface & implementation introducing abstract class and move implementation to the derived class:

```
class Class {
  public ...
    // Abstract interface
    ...
  private ...
    // No implementation
}
```

```
class ClassImpl extends Class {
  public ...
    // Interface inherited
    // from its base class
    ...
  private ...
    // Implementation
}
```

Pattern Bridge: The Problem

The common scheme



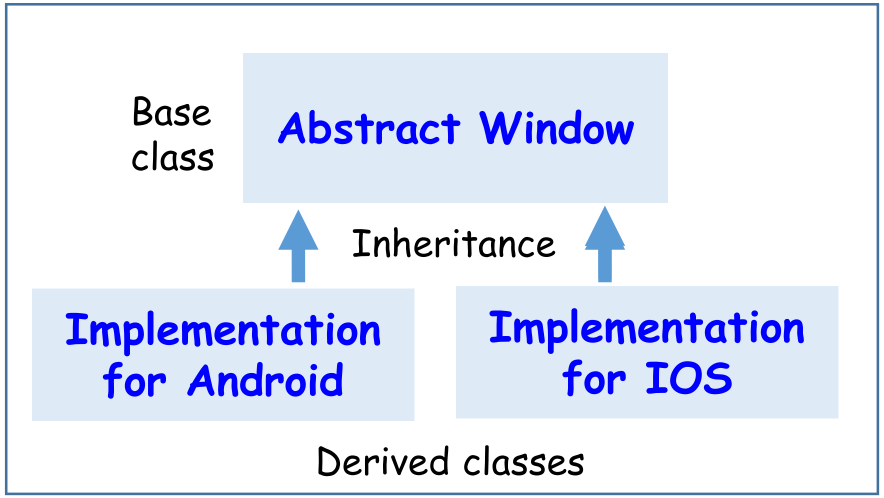
The problem is that such a configuration is not flexible enough:

- The implementation depends on its abstraction because **the relation is set on compile time**!

So, how to decouple an abstraction from its implementation so that the two can vary independently?

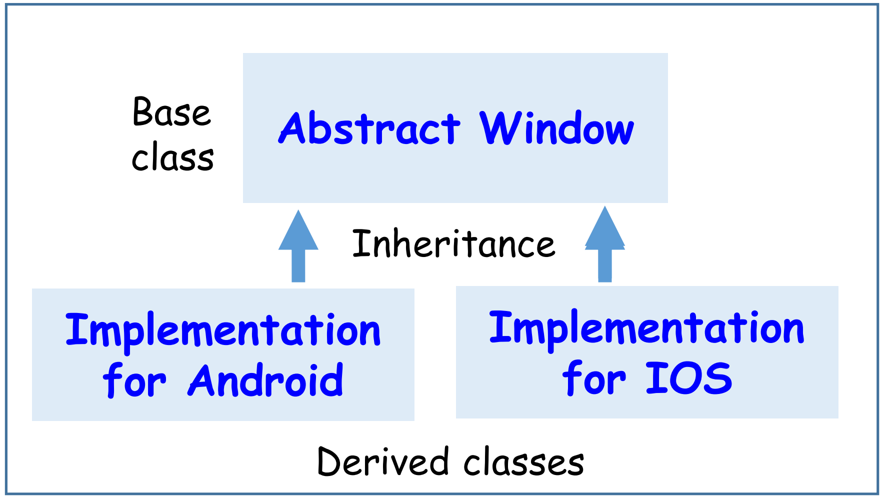
Pattern Bridge: The Problem

An example of the problem (the idea was taken from the E. Gamma's book): a **portable GUI hierarchy**



Pattern Bridge: The Problem

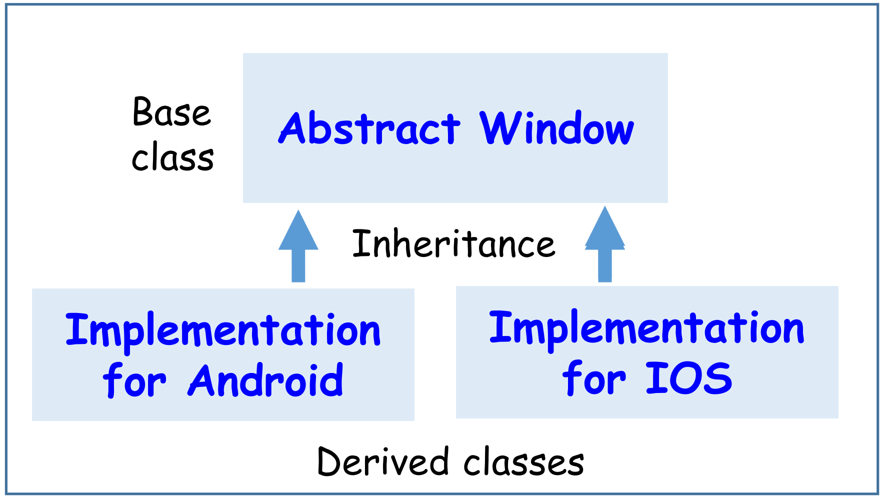
An example of the problem (the idea was taken from the E. Gamma's book): a **portable GUI hierarchy**



Suppose we need to provide **other kinds** of windows: text window, icon window etc. How to update the hierarchy?

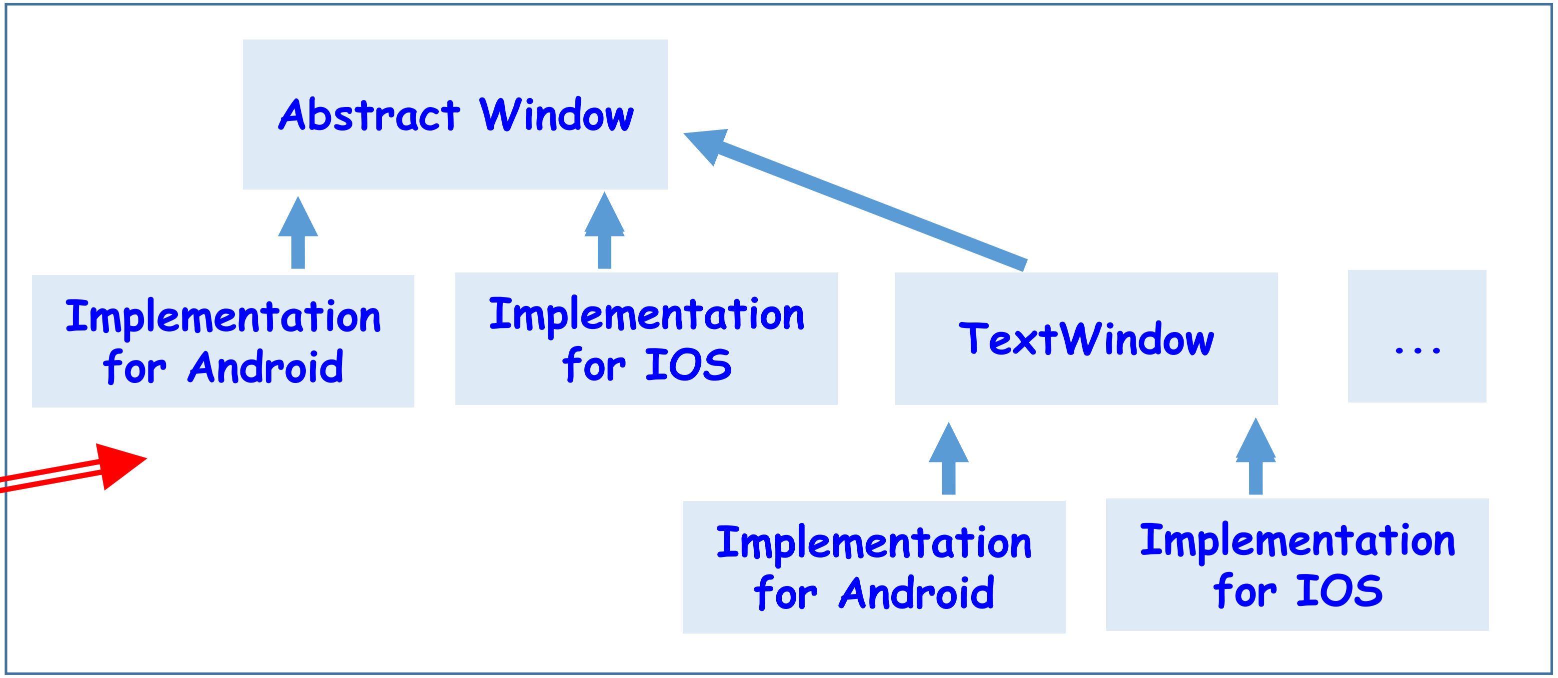
Pattern Bridge: The Problem

An example of the problem (the idea was taken from the E. Gamma's book): a **portable GUI hierarchy**



Suppose we need to provide **other kinds** of windows: text window, icon window etc. How to update the hierarchy?

- Hard to promote Window abstraction for new kinds of windows and for new planforms
- Client code becomes platform-dependent



The Solution: Pattern Bridge

Make two hierarchies instead on one:

- **Hierarchy of abstractions.**
- **Hierarchy of implementations.**

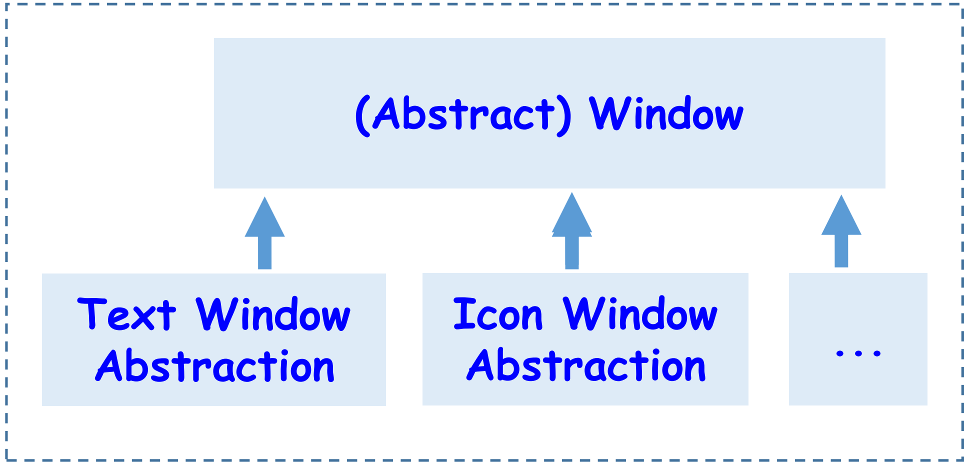
Use delegation to set up the communication between the two.

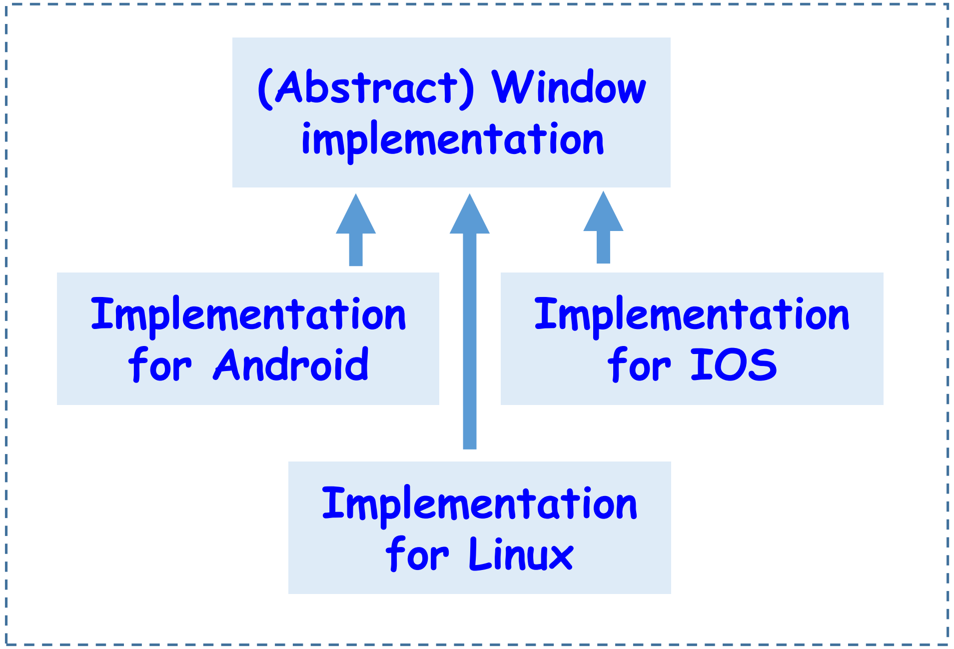
The Solution: Pattern **Bridge**

Make two hierarchies instead on one:

- **Hierarchy of abstractions.**
- **Hierarchy of implementations.**

Use delegation to set up the communication between the two.



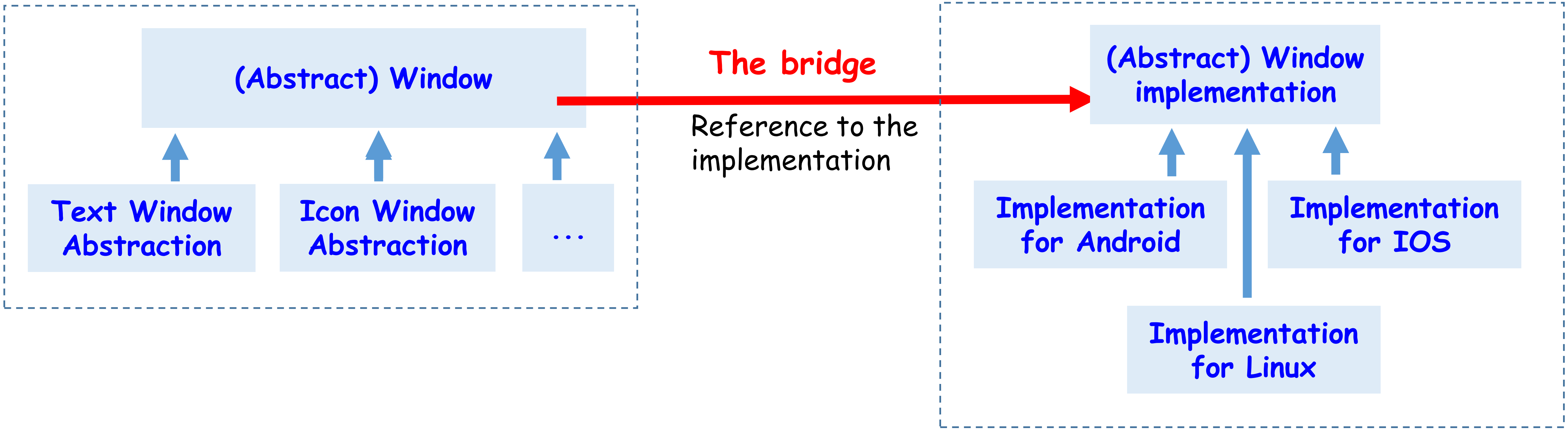


The Solution: Pattern Bridge

Make two hierarchies instead on one:

- **Hierarchy of abstractions.**
- **Hierarchy of implementations.**

Use delegation to set up the communication between the two.



Pattern Bridge: The Code Example

```
class Window {
   public Window(WindowImpl i) { impl = I; }
   // Own methods
   public void Open();
   public void Close();
   ...
   // Redirecting methods
   public void DrawLine(coords)
   { impl.DrawLine(cords); }
   public void DrawRect(coords) { ... }
   public void DrawText(String t,coords);
   ...
   private WindowImpl impl;
        // reference to the implementation!!
   ...
}
```

Pattern Bridge: The Code Example

```
class Window {
   public Window(WindowImpl i) { impl = I; }
   // Own methods
   public void Open();
   public void Close();
   ...
   // Redirecting methods
   public void DrawLine(coords)
   { impl.DrawLine(cords); }
   public void DrawRect(coords) { ... }
   public void DrawText(String t,coords);
   ...
   private WindowImpl impl;
        // reference to the implementation!!
   ...
}
```

```
class WindowImpl {
   public abstract void DrawLine(coords);
   public abstract void DrawRect(coords);
   public abstract void DrawText(String t,coords);
   ...
}
```

```
class iosWindow extends WindowImpl {
  public void DrawLine(coords) { implementation }
  public void DrawRect(coords) { implementation }
  public void DrawText(String t,coords)
                             { implementation }
   ...
}
```

Pattern Bridge: The Code Example

```
class Window {
   public Window(WindowImpl i) { impl = I; }
   // Own methods
   public void Open();
   public void Close();
   ...
   // Redirecting methods
   public void DrawLine(coords)
   { impl.DrawLine(cords); }
   public void DrawRect(coords) { ... }
   public void DrawText(String t,coords);
   ...
   private WindowImpl impl;
        // reference to the implementation!!
   ...
}
```

```
class WindowImpl {
   public abstract void DrawLine(coords);
   public abstract void DrawRect(coords);
   public abstract void DrawText(String t,coords);
   ...
}
```

```
class iosWindow extends WindowImpl {
  public void DrawLine(coords) { implementation }
  public void DrawRect(coords) { implementation }
  public void DrawText(String t,coords)
                              { implementation }
   ...
}
```

```
...
Window w = new Window(new iosWindow());
...
w.drawLine(coords);
```

- The client code doesn't depend on implementation details; it uses only `Window`'s interface.
- Implementation hierarchy is evolving independently from abstract Window interface.

Pattern Bridge: The Code Example

```
class Window {
   public Window(WindowImpl i) { impl = I; }
   // Own methods
   public void Open();
   public void Close();
   ...
   // Redirecting methods
   public void DrawLine(coords)
   { impl.DrawLine(cords); }
   public void DrawRect(coords) { ... }
   public void DrawText(String t,coords);
   ...
   private WindowImpl impl;
        // reference to the implementation!!
   ...
}
```

```
class WindowImpl {
   public abstract void DrawLine(coords);
   public abstract void DrawRect(coords);
   public abstract void DrawText(String t,coords);
   ...
}
```

```
class iosWindow extends WindowImpl {
  public void DrawLine(coords) { implementation }
  public void DrawRect(coords) { implementation }
  public void DrawText(String t,coords)
                             { implementation }
   ...
}
```

```
...
Window w = new Window(new iosWindow());
...
w.drawLine(coords);
```

- The client code doesn't depend on implementation details; it uses only Window's interface.
- Implementation hierarchy is evolving independently from abstract Window interface.

Bridge Pattern: Exercise

1. Suppose there are two implementations of **list**: one based on array and the second using pointers. Write the configuration of abstract list interface (independent from the implementation) and two implementations using the Bridge pattern.
2. Add new class for **stack** making it derived from abstract list interface – again, using the Bridge pattern approach.