

Introduction to Programming

Part I

Lecture 11

Introduction to Java

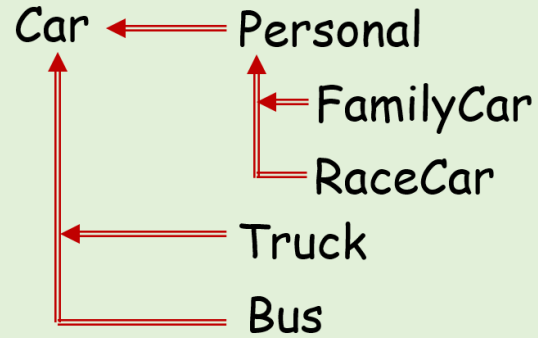
Final methods & classes; interfaces

Eugene Zouev
Fall Semester 2021
Innopolis University

What We Have Learnt

- Classes and class instances
- Value types and reference types
- Encapsulation, overloading
- Inheritance: single & multiple
- Static & dynamic types
- Method overriding
- Polymorphism
- Casts & type checks
- Abstract classes & methods
- Packages
- Exceptions

Inheritance 3



Inheritance can be treated as "**is a**" relation:

"Personal" is a "Car"
"FamilyCar" is "Personal"
"FamilyCar" is a "Car"

Another kind of relation is **delegation**: "**has a**" relation:

"Car" has an "engine". Therefore, "Personal" and "FamilyCar" also have an "Engine" - as all other kinds of "Cars".

Static & Dynamic Types 2

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 7

Late

```
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 overrides 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);
    }
}
```

The static type of `m` is (always) `Base`

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

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

Packages in Java 1

- Each class or group of classes can be made a member of a package:

```
package myPackage;
class C1 { ... }
class C2 { ... }
...
```

This is a kind of "header" of the package called `myPackage`.

All following classes within this file are treated as members of `myPackage` package.

Two parts of the same package

```
package myPackage;
class C10 { ... }
class C20 { ... }
...
```

Full names of the classes are `myPackage.C1`, `myPackage.C2` etc. ("Fully qualified names")

A package can be made up of **several files** (all residing in the same directory)

19/27

32/35

4/39

Today:

Interface & implementation (again)

Final methods & classes

Interfaces

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

What happens?

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

Is it correct?

Apparently not!

Will the compiler report a bug?

No!! - the code is formally
correct

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

```
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. */ }  
}
```

The same solution in **C#**

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 a bit slower than "usual" calls 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 the static type of the object

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 performance enhancement.

Why:

- The compiler is free to inline calls 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

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

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

```
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.

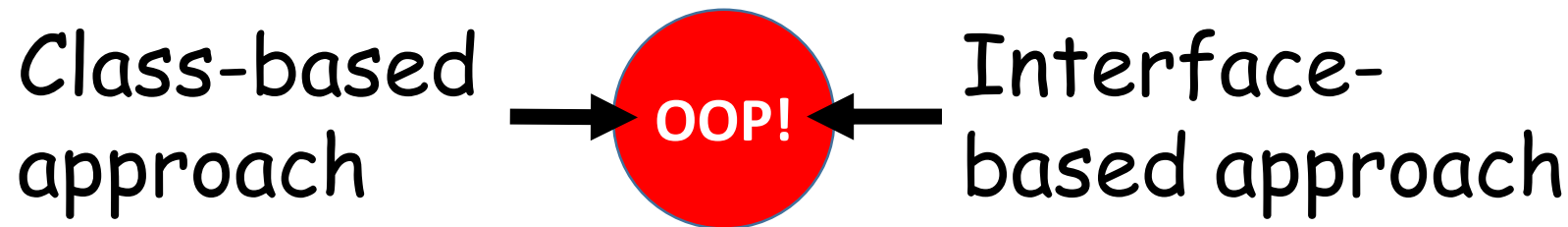
Since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

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
- 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.



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 (abstract 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();
    ...
}
```

```
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**

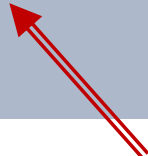
```
class Person implements iBodyParams, iSkills, iRelations, ...
{
    ...
}

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
```

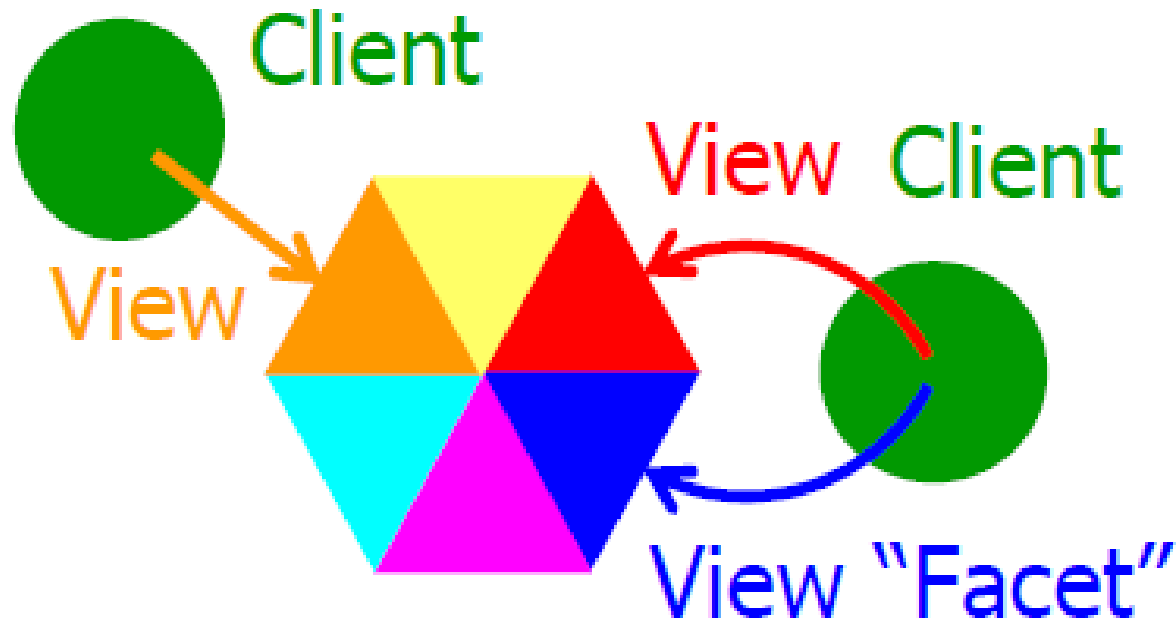
Nested Interfaces

```
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");
    }
}
```

```
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 vs Abstract Classes

Similarities:

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

The favorite question on many job interviews! 😊

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; // ????
```

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». ☺

Addendum: 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.

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!

- How to prevent creation?

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

- Does this solution really prevent creation? - No:

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

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


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 both interface and implementation. It's not flexible and in some cases it might cause problems.

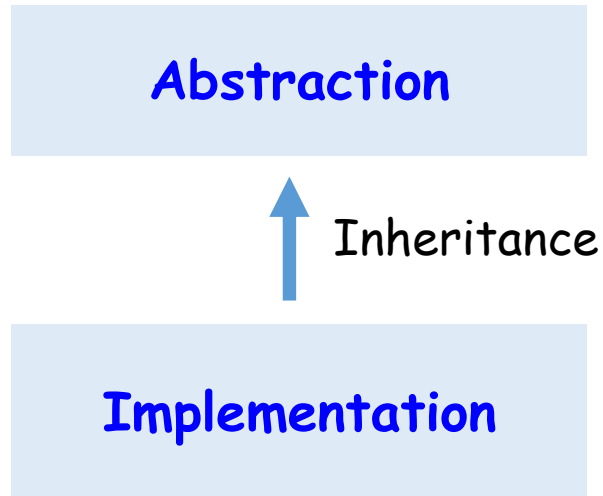
What do we do to make this construct more flexible and reliable: we **separate** interface & implementation introducing abstract class and move implementation to the derived class:

```
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



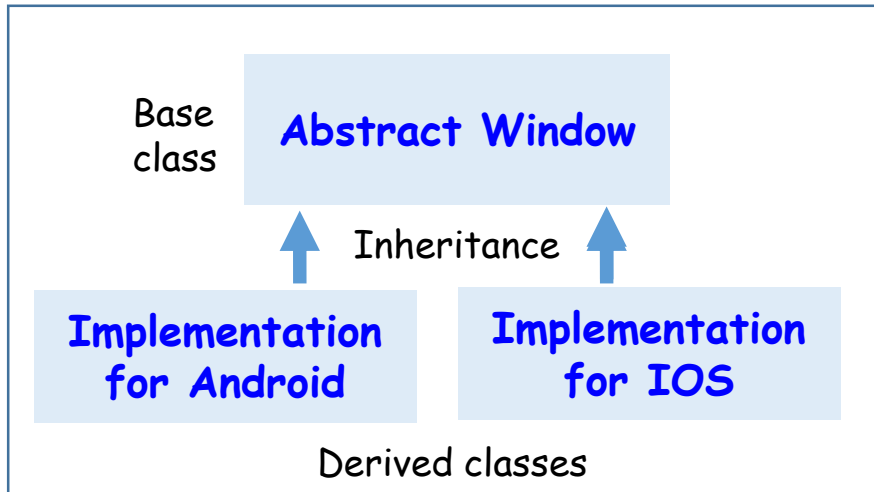
The problem is that such a configuration is not flexible enough:

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

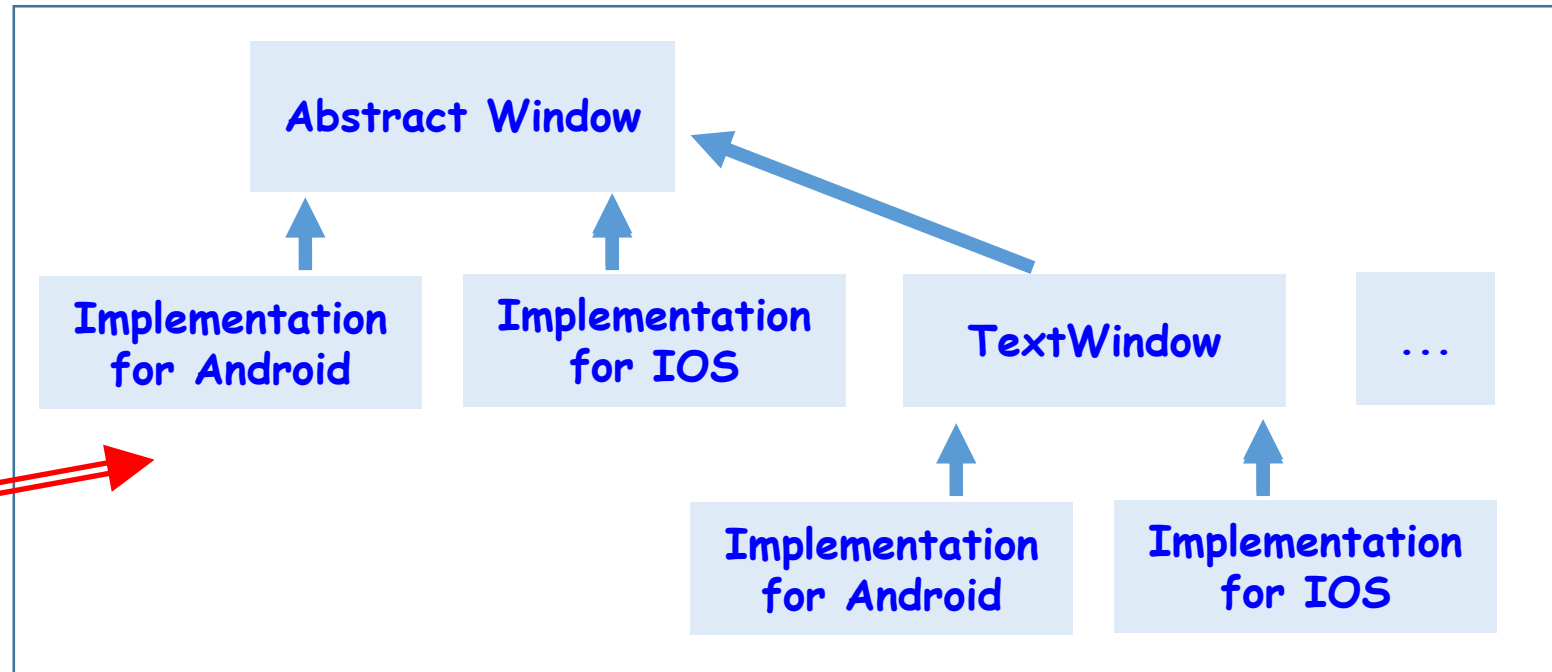
So, how to decouple an abstraction from its implementation so that the two can vary independently?

Pattern Bridge: The Problem

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



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



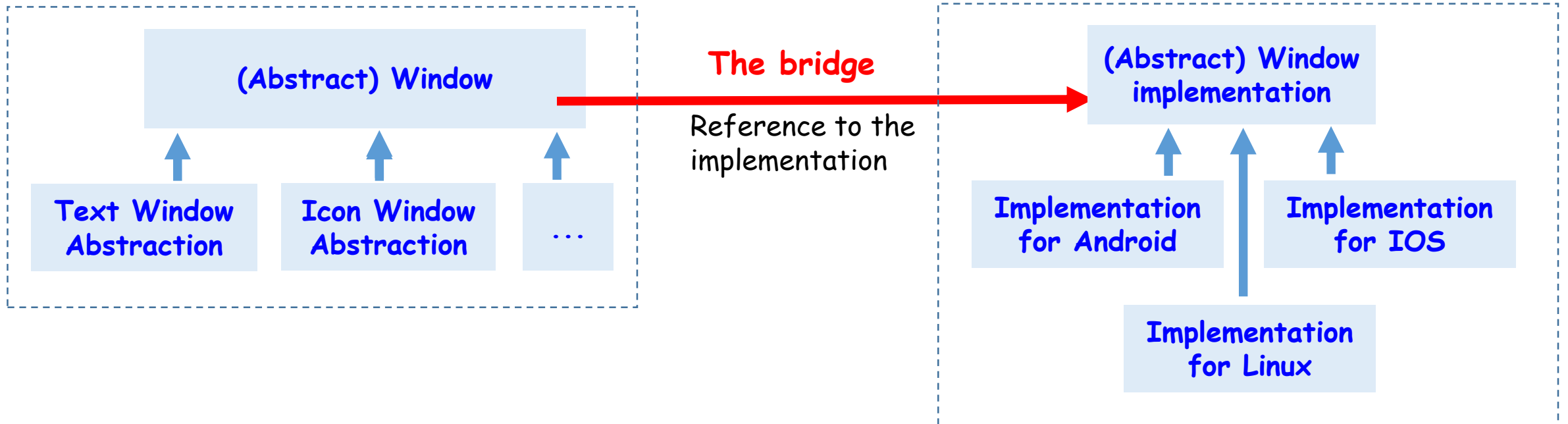
- Hard to promote Window abstraction for new kinds of windows and for new platforms
- Client code becomes platform-dependent

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.



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.

