Essential of Object Oriented Programming

Alexandre Bergel
http://bergel.eu
06-09-2021

Outline

1. Liskov principle
    1. theory
    2. concrete applications
2. Exercise: talking to the Suchai satellite

Outline

1. **Liskov principle**
    1. **theory**
    2. **concrete applications**
2. Exercise: talking to the Suchai satellite

Liskov substitution principle

Initially introduced in 1974 by Barbara Liskov

Formulated in 1994 with Jeannette Wing as follows:

Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T.

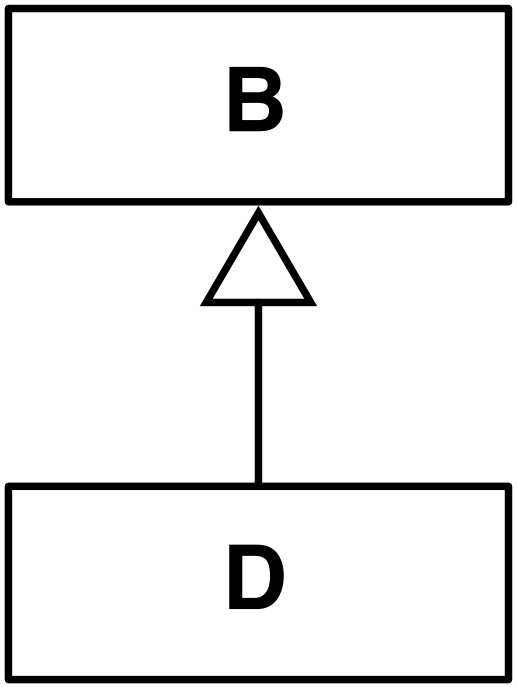
Barbara Liskov received the Turing Award in 2008

Liskov principle vulgarized

Subtypes must be substitutable for their base types

Liskov principle vulgarized

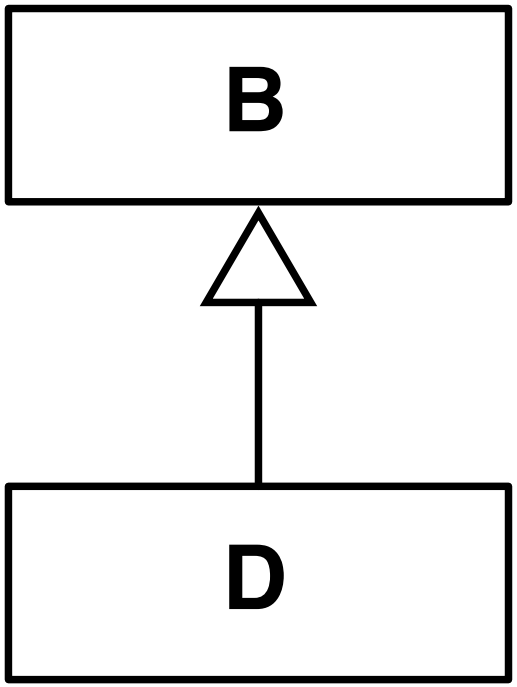
```
void f (B object) {
   ...
}
```



Liskov principle vulgarized

```
void f (B object) {
   ...
}
```

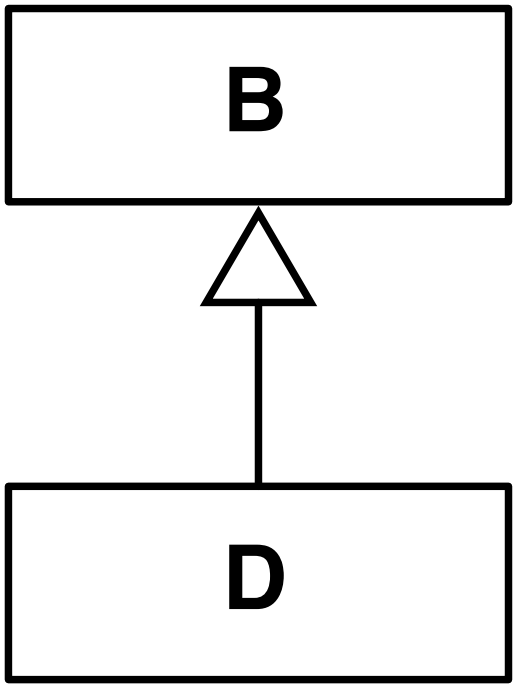
if `f(new B())` behaves correctly, `f(new D())` has to correctly behave as well



Fragile class

```
void f (B object) {
   ...
}
```

if `f(new B())` behaves correctly and `f(new D())` not, then we say that `D` is fragile in the presence of `f`



Some practical illustrations

Procedural coding style

Object initialization

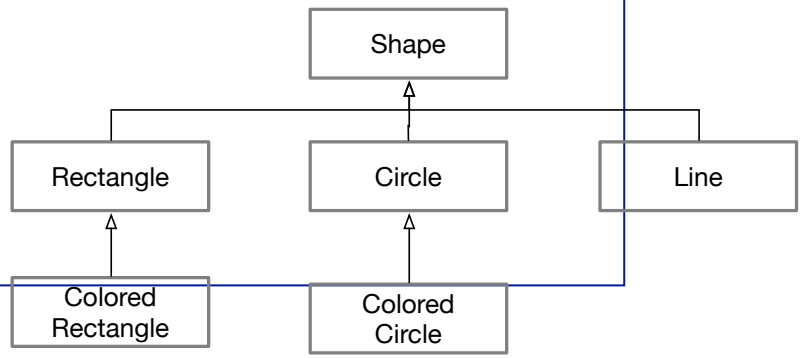
Access privileges cannot be weakened

Procedural coding style

```
public static long sumShapes(Shape[] shapes) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        if (shapes[i] instanceof Rectangle) {
            Rectangle r = (Rectangle)shapes[i];
            sum += (r.width * r.height);
            break;
        }
        if (shapes[i] instanceof Circle) {
            Circle r = (Circle)shapes[i];
            sum += (Math.PI * r.radius * r.radius);
            break;
        }
        // more cases
    }
    return sum;
}
```

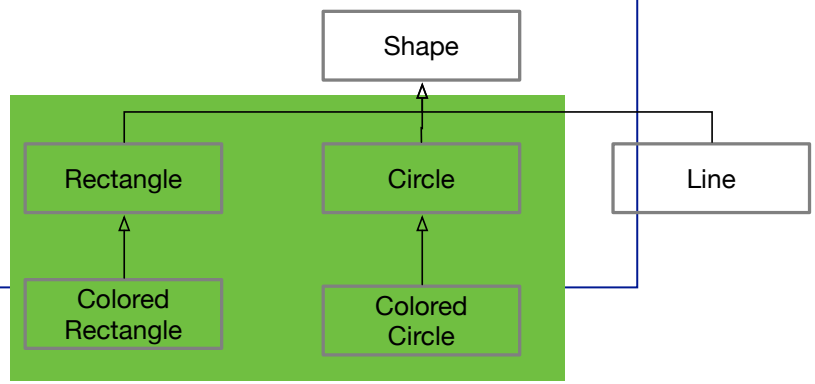
Procedural coding style

```
public static long sumShapes(Shape[] shapes) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        if (shapes[i] instanceof Rectangle) {
            Rectangle r = (Rectangle)shapes[i];
            sum += (r.width * r.height);
            break;
        }
        if (shapes[i] instanceof Circle) {
            Circle r = (Circle)shapes[i];
            sum += (Math.PI * r.radius * r.radius);
            break;
        }
        // more cases
    }
    return sum;
}
```



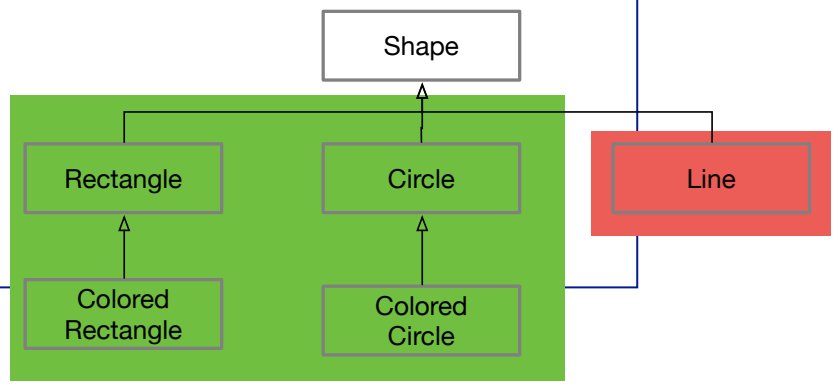
Procedural coding style

```
public static long sumShapes(Shape[] shapes) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        if (shapes[i] instanceof Rectangle) {
            Rectangle r = (Rectangle)shapes[i];
            sum += (r.width * r.height);
            break;
        }
        if (shapes[i] instanceof Circle) {
            Circle r = (Circle)shapes[i];
            sum += (Math.PI * r.radius * r.radius);
            break;
        }
        // more cases
    }
    return sum;
}
```



Procedural coding style

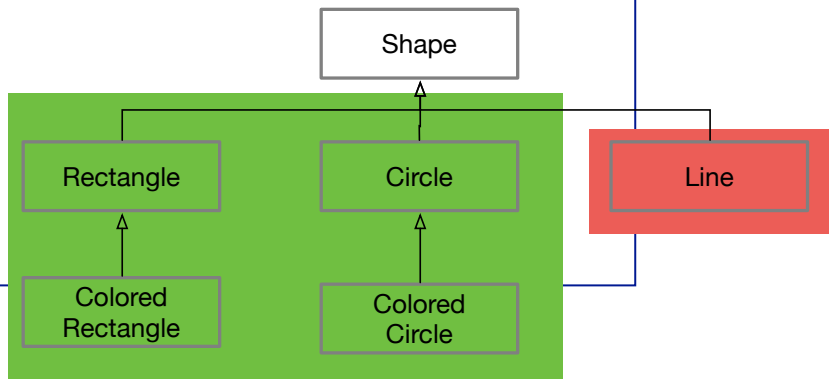
```
public static long sumShapes(Shape[] shapes) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        if (shapes[i] instanceof Rectangle) {
            Rectangle r = (Rectangle)shapes[i];
            sum += (r.width * r.height);
            break;
        }
        if (shapes[i] instanceof Circle) {
            Circle r = (Circle)shapes[i];
            sum += (Math.PI * r.radius * r.radius);
            break;
        }
        // more cases
    }
    return sum;
}
```



Procedural coding style

```
public static long sumShapes(Shape[] shape
    long sum = 0;
    for (int i=0; i<shapes.length; i+
        if (shapes[i] instanceof Rec
            Rectangle r = (Rectang
            sum += (r.width * r.heigh
            break;
        }
        if (shapes[i] instanceof Circle)
            Circle r = (Circle)shapes[i];
            sum += (Math.PI * r.radius * r.radius);
            break;
        }
        // more cases
    }
    return sum;
}
```

Violation of the Liskov principle



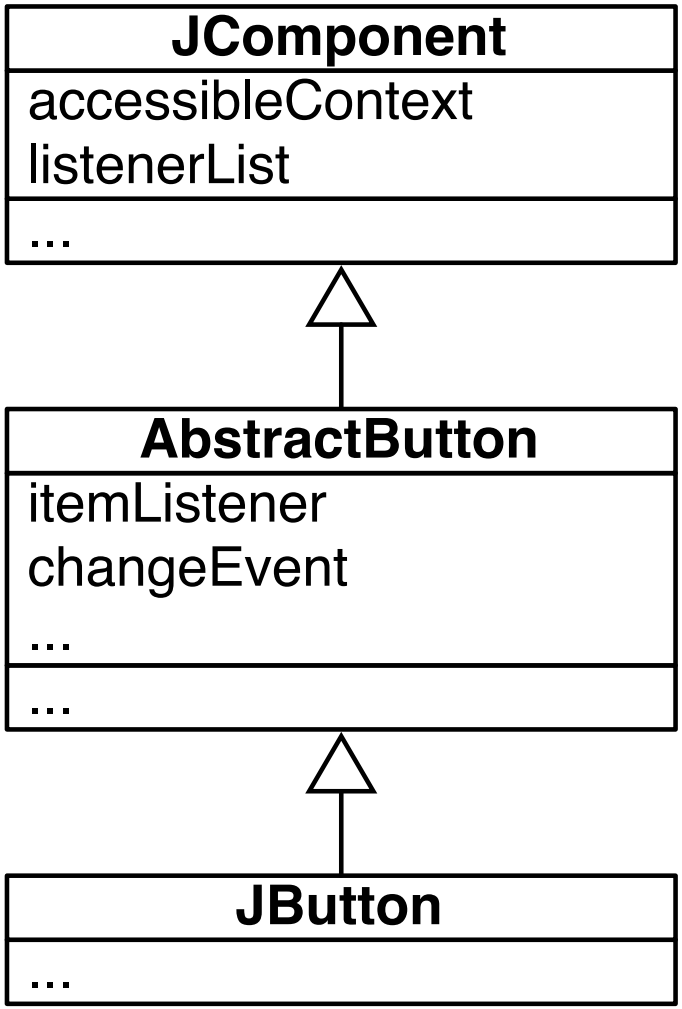
Procedural coding style

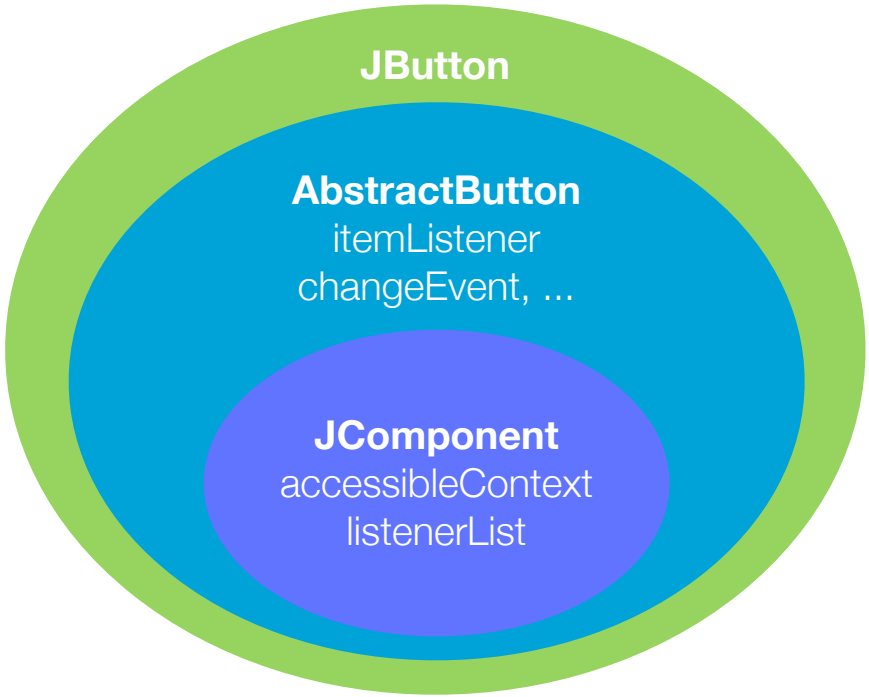
In general, procedural coding style (e.g., programming in plain C) makes difficult to extend a software

Software extension comes at a high cost:

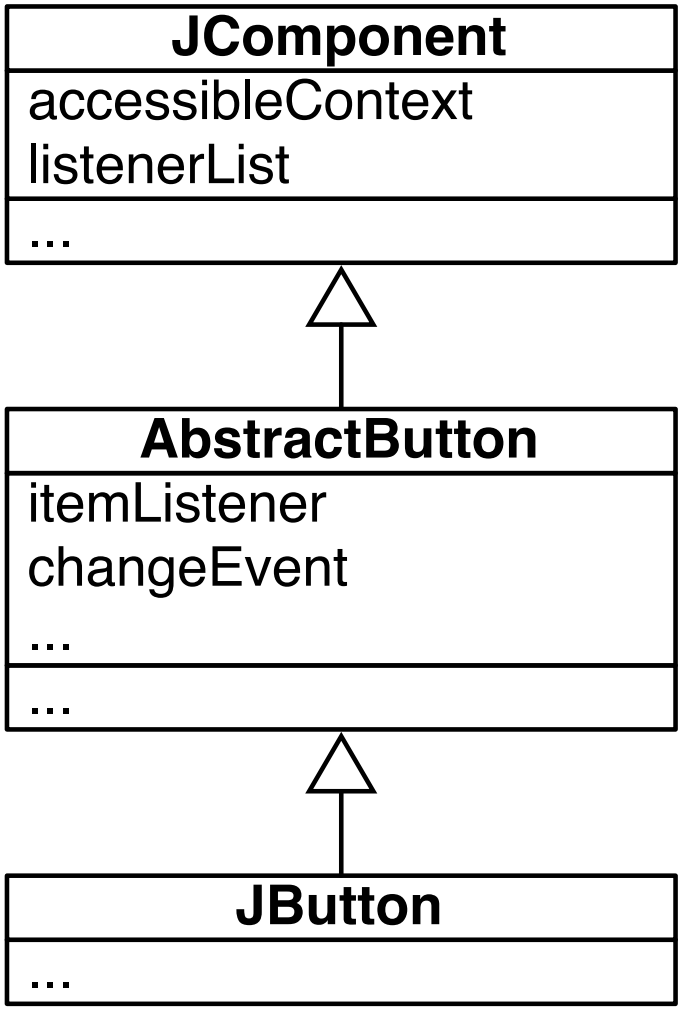
E.g., existing code, which has nothing to do with the extension, may have to be modified

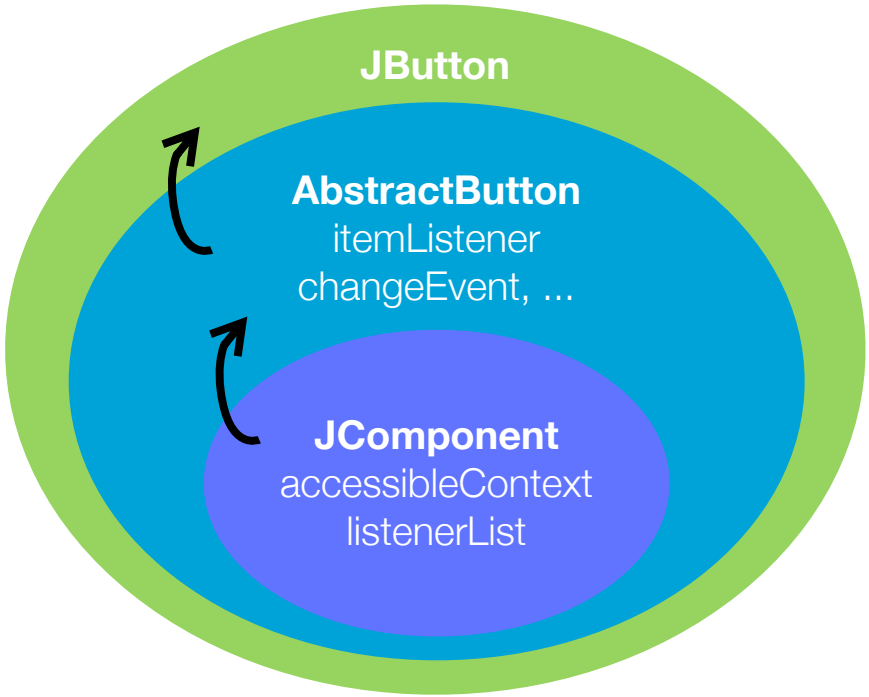
Object initialization





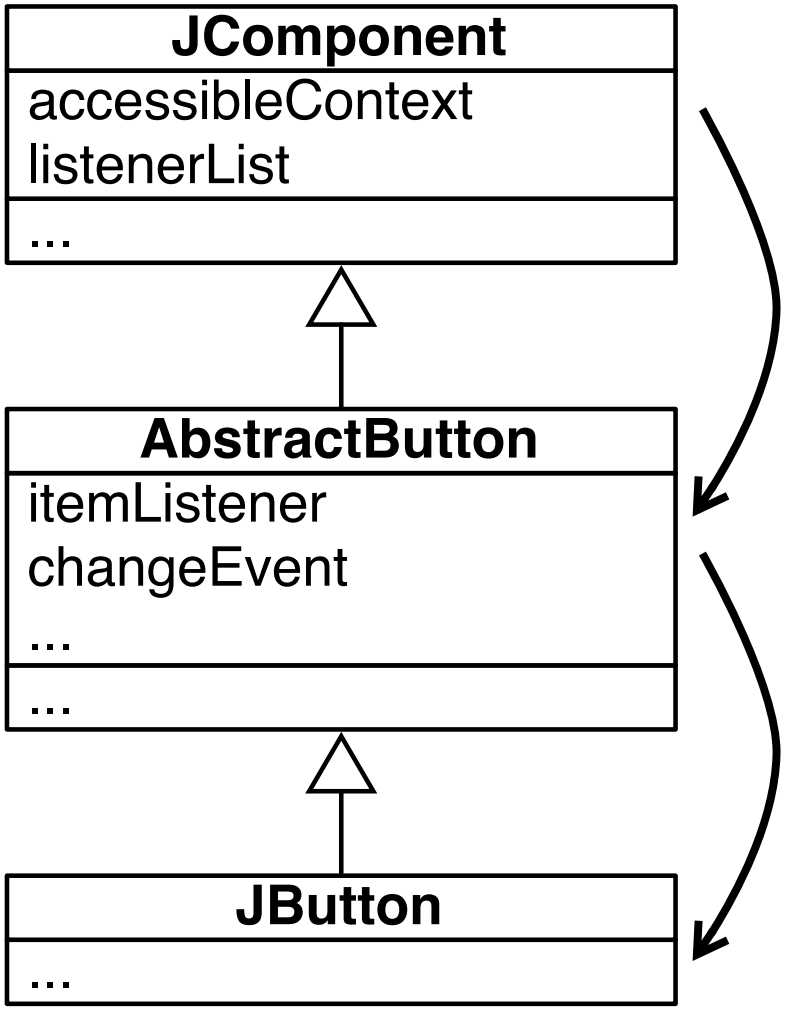
Object initialization

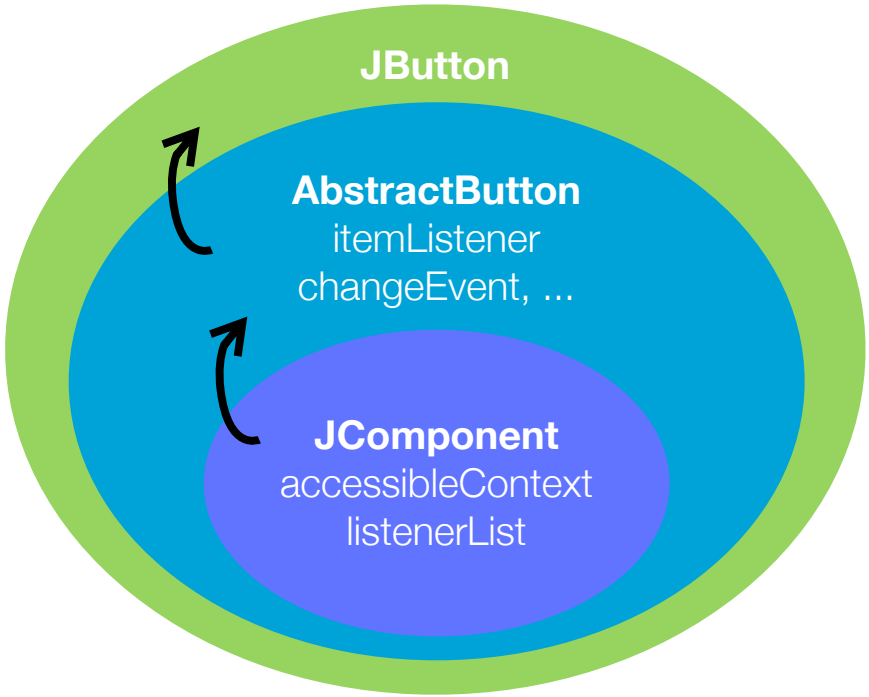




→ order of object initialization, enforced by the super(...) at the beginning of each constructor

Object initialization





→ order of object initialization, enforced by the super(...) at the beginning of each constructor

Privilege access

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Access privileges apply to class definition
and class members (e.g., field, method, inner class)

More on http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Privilege access

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

```
package human;
public class Tank {
    private int healthPoint = 175;

    void receiveDamage(int amount) {
        this.healthPoint -= amount;
    }
}
```

```
package ai;
public class ArtificialHumanPlayer {
    public void doAction(Tank tank) {
        tank.receiveDamage(50);
    }
}
```

Privilege access

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Different package

```
package human;
public class Tank {
    private int healthPoint = 175;

    void receiveDamage(int amount) {
        this.healthPoint -= amount;
    }
}
```

```
package ai;
public class ArtificialHumanPlayer {
    public void doAction(Tank tank) {
        tank.receiveDamage(50);
    }
}
```

Privilege access

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Different package

```
package human;
public class Tank {
    private int healthPoint = 175;

    void receiveDamage(int amount) {
        this.healthPoint -= amount;
    }
}
```

```
package ai;
public class ArtificialHumanPlayer {
    public void doAction(Tank tank) {
        tank.receiveDamage(50);
    }
}
```

receiveDamage(…) is visible only in the package human

This code is incorrect and does not compile

Privilege access

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

```java
package human;
public class Tank {
    private int healthPoint = 175;

    public void receiveDamage(int amount) {
        this.healthPoint -= amount;
    }
}
```

```java
package ai;
public class ArtificialHumanPlayer {
    public void doAction(Tank tank) {
        tank.receiveDamage(50);
    }
}
```

This version of the code is correct

Why not having all methods public?

```
package database;
public class Account {
    private String user, password;

    public String getPassword() {
        return password;
    }
}
```

```
package database;
public class CheckLogin {
     public boolean canLogin(Account a,
                             String pass) {
         return a.getPassword().equals(pass);
     }
}
```

This version has a security vulnerability

Why not having all methods public?

```
package database;
public class Account {
    private String user, password;

    public String getPassword() {
        return password;
    }
}
```

```
package database;
public class CheckLogin {
     public boolean canLogin(Account a,
                             String pass) {
         return a.getPassword.equal(pass);
     }
}
```

```
package attacker;
public class Virus {
     public boolean getPassword(Account a) {
        System.out.println(a.getPassword());
     }
}
```

As soon as someone get an instance of Account, password can be accessed

Access privileges can only be widened

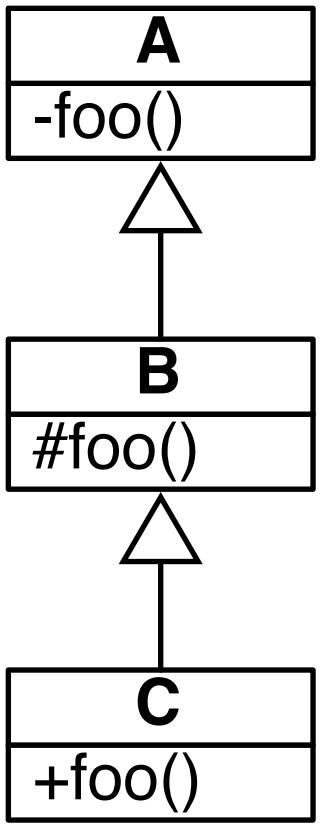
```
class A {
    private void foo () {
    }
}

class B extends A {
    protected void foo () {

    }
}

class C extends B {
    public void foo () {

    }
}
```



Would it be okay to have this?

```
class A {
    public void foo () {
    }
}

class B extends A {
    protected void foo () {

    }
}

class C extends B {
    private void foo () {

    }
}
```

Would it be okay to have this?

```
class A {
    public void foo () {
    }
}

class B extends A {
    protected void foo () {

    }
}

class C extends B {
    private void foo () {

    }
}
```

Outline

1. Liskov principle
    1. theory
    2. concrete applications
2. **Exercise: talking to the Suchai satellite**

The Suchai Nano-satellite

Nano-satellite (1000 cm^3 = 10cm × 10 cm × 10cm) built in 🇨🇱

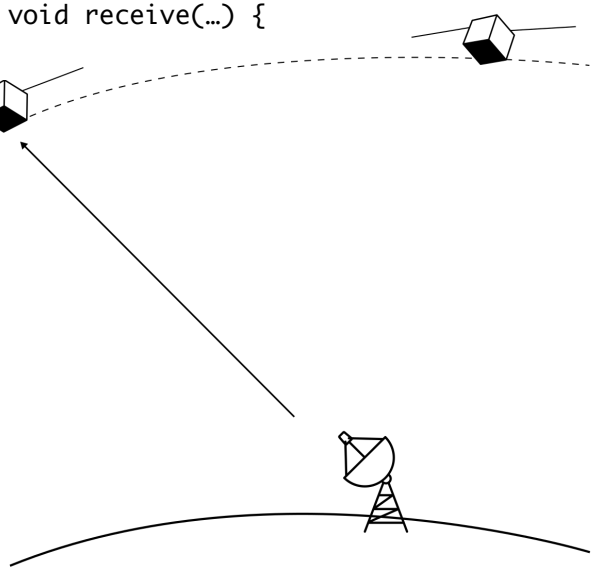
Low orbit (505km), but still above the international space station

Orbit in 90 minutes

Flight software is about > 25 000 KLOC

The Suchai Nano-satellite

```
class Suchai {
    public void receive(…) {
    …
}
```



How would you implement the class Suchai able to receive two commands? e.g., `Rotate` and `TakePicture`

Your design should be *easy* to extend (i.e., at a low cost)

A possible implementation

The key aspect is to make the Suchai open for extension

Adding a new command should be at a very lost cost

I.e., low cost = adding code, moderate/high cost = modifying code

```java
package suchai;

import java.util.ArrayList;
import java.util.List;

public class Suchai {
    private int angle;
    private List<Picture> pictures;

    public Suchai() {
        angle = 0;
        pictures = new ArrayList<>();
    }

    public void setAngle(int newAngle) {
        angle = newAngle;
    }

    public int getAngle() {
        return angle;
    }

    public int numberOfPictures() {
        return pictures.size();
    }

    public void receive(Command c) {
        c.doExecute(this);
    }

    public void addPicture(Picture pictur
        pictures.add(picture);
    }
}
```

```
package suchai;

public class GroundStation {
    public static void main(String[] args) {
        Suchai s = new Suchai();

        System.out.println("Angle = " + s.getAngle());
        System.out.println("Number of pictures = " + s.numberOfPictures());

        s.receive(new RotateCommand());
        s.receive(new TakePictureCommand());

        System.out.println("Angle = " + s.getAngle());
        System.out.println("Number of pictures = " + s.numberOfPictures());
    }
}
```

```java
package suchai;

public interface Command {
    void doExecute(Suchai suchai);
}
```

```java
package suchai;

public class RotateCommand implements Command {
    public void doExecute(Suchai suchai) {
        suchai.setAngle(suchai.getAngle() + 10);
    }
}
```

```java
package suchai;

public class TakePictureCommand implements Command {
    public void doExecute(Suchai suchai) {
        suchai.addPicture(new Picture());
    }
}
```

```java
package suchai;

public class Picture {
}
```

```java
package suchai;

import java.util.ArrayList;
import java.util.List;

public class Suchai {
    private int angle;
    private List<Picture> pictures;

    public Suchai() {
        angle = 0;
        pictures = new ArrayList<>();
    }

    public void setAngle(int newAngle) {
        angle = newAngle;
    }

    public int getAngle() {
        return angle;
    }

    public int numberOfPictures() {
        return pictures.size();
    }

    public void receive(Command c) {
        c.doExecute(this);
    }

    public void addPicture(Picture pictur
        pictures.add(picture);
    }
}
```

Double dispatch, which we will see next week

What you should know!

What is the Liskov principle?

How the Liskov principle affects the design of a programming language