Liskov Principle & Visibility

Nancy Hitschfeld
Matías Toro

(Diapositivas de Alex Bergel)

Outline

1. Liskov principle
2. Visibility
3. Exercise: talking to the Suchai satellite

Outline

1. **Liskov principle**
2. Visibility
3. 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.

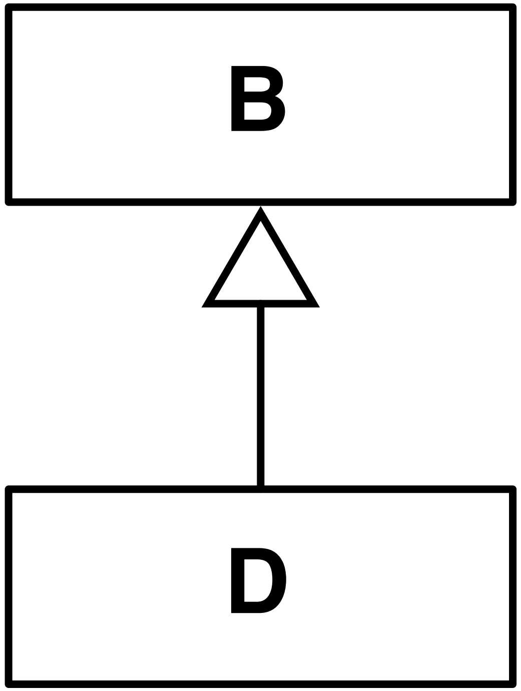
Barbara Liskov received the Turing Award in 2008

Liskov principle vulgarized

Subtypes must be substitutable for their base types

Liskov principle vulgarized

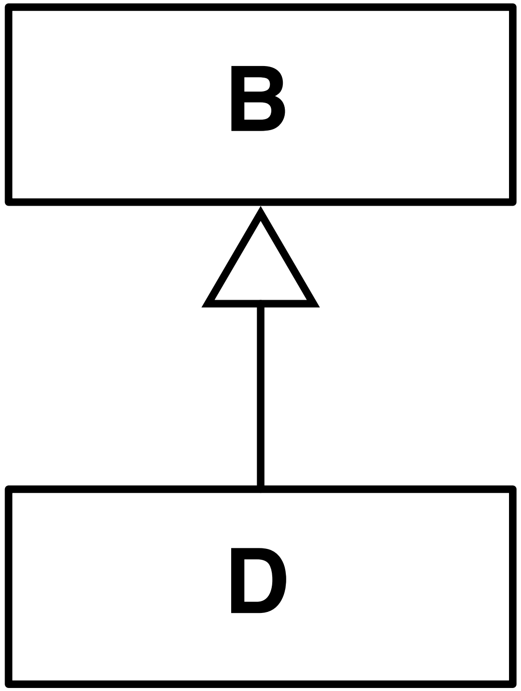
```
def f(o: B): Unit = {
  ...
}
```



Liskov principle vulgarized

```
def f(o: B): Unit = {
  ...
}
```

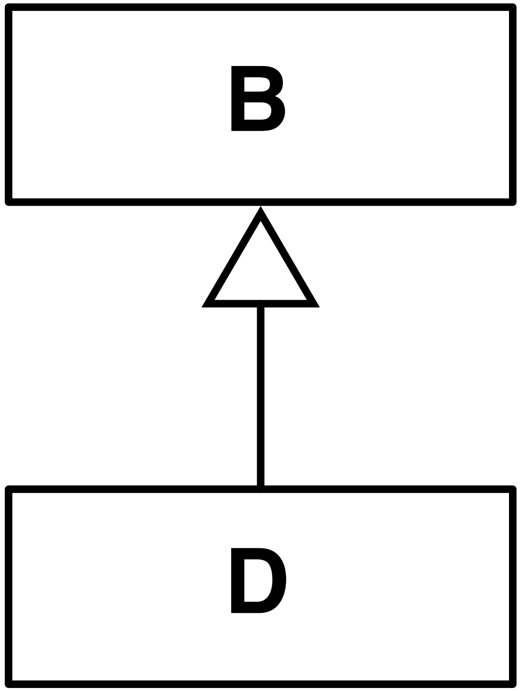
if `f(new B())` behaves correctly, `f(new D())` has to correctly behave as well



Fragile class

```
def f(o: B): Unit = {
  ...
}
```

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



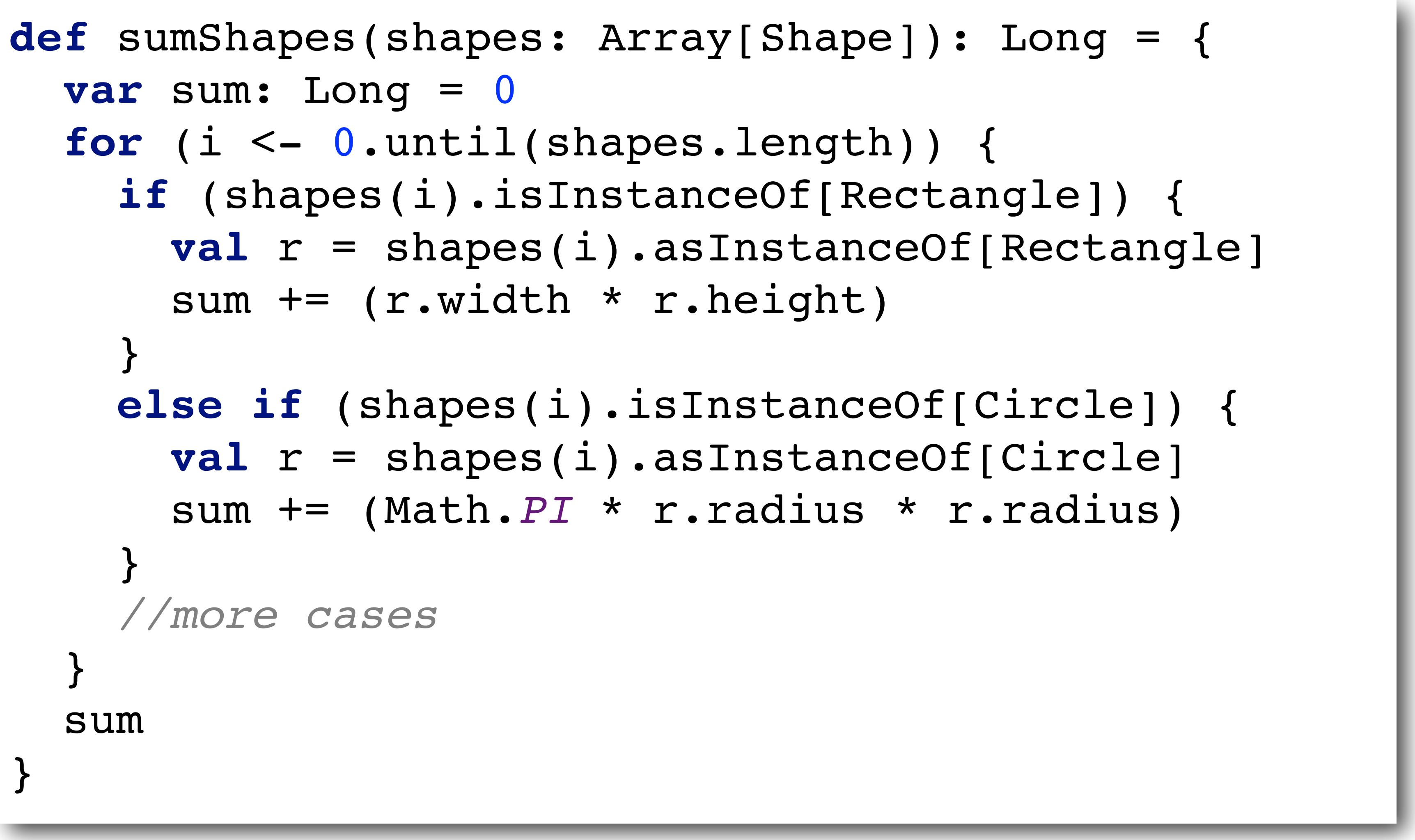
Some practical illustrations

Procedural coding style

Object initialization

Access privileges cannot be weakened

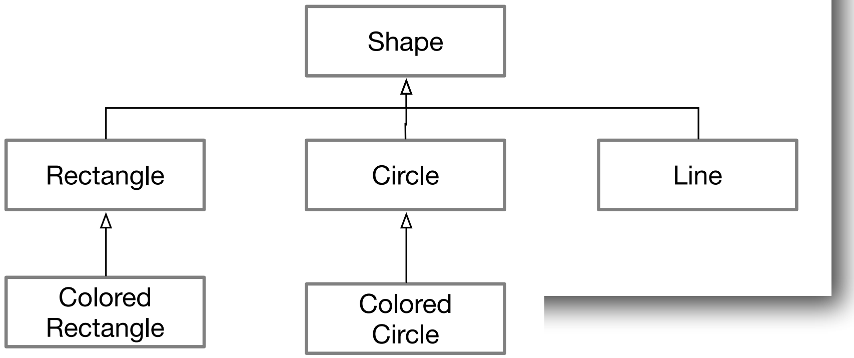
Procedural coding style



```
def sumShapes(shapes: Array[Shape]): Long = {
  var sum: Long = 0
  for (i <- 0.until(shapes.length)) {
    if (shapes(i).isInstanceOf[Rectangle]) {
      val r = shapes(i).asInstanceOf[Rectangle]
      sum += (r.width * r.height)
    }
    else if (shapes(i).isInstanceOf[Circle]) {
      val r = shapes(i).asInstanceOf[Circle]
      sum += (Math.PI * r.radius * r.radius)
    }
    //more cases
  }
  sum
}
```

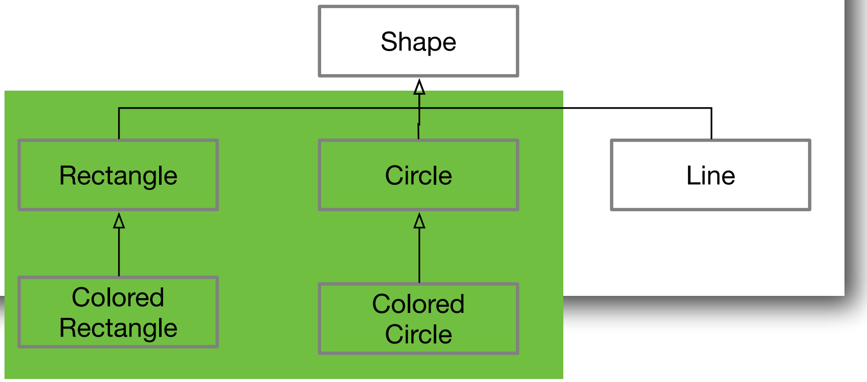
Procedural coding style

```
def sumShapes(shapes: Array[Shape]): Long = {
  var sum: Long = 0
  for (i <- 0.until(shapes.length)) {
    if (shapes(i).isInstanceOf[Rectangle]) {
      val r = shapes(i).asInstanceOf[Rectangle]
      sum += (r.width * r.height)
    }
    else if (shapes(i).isInstanceOf[Circle]) {
      val r = shapes(i).asInstanceOf[Circle]
      sum += (Math.PI * r.radius * r.radius)
    }
    //more cases
  }
  sum
}
```



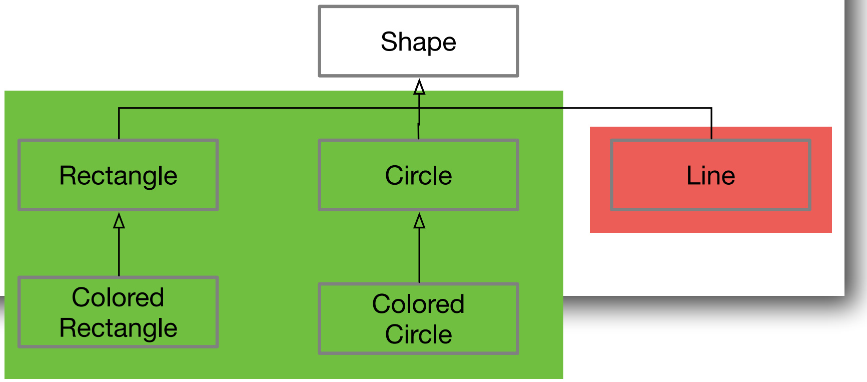
Procedural coding style

```
def sumShapes(shapes: Array[Shape]): Long = {
  var sum: Long = 0
  for (i <- 0.until(shapes.length)) {
    if (shapes(i).isInstanceOf[Rectangle]) {
      val r = shapes(i).asInstanceOf[Rectangle]
      sum += (r.width * r.height)
    }
    else if (shapes(i).isInstanceOf[Circle]) {
      val r = shapes(i).asInstanceOf[Circle]
      sum += (Math.PI * r.radius * r.radius)
    }
    //more cases
  }
  sum
}
```



Procedural coding style

```scala
def sumShapes(shapes: Array[Shape]): Long = {
  var sum: Long = 0
  for (i <- 0.until(shapes.length)) {
    if (shapes(i).isInstanceOf[Rectangle]) {
      val r = shapes(i).asInstanceOf[Rectangle]
      sum += (r.width * r.height)
    }
    else if (shapes(i).isInstanceOf[Circle]) {
      val r = shapes(i).asInstanceOf[Circle]
      sum += (Math.PI * r.radius * r.radius)
    }
    //more cases
  }
  sum
}
```



Shape

Rectangle

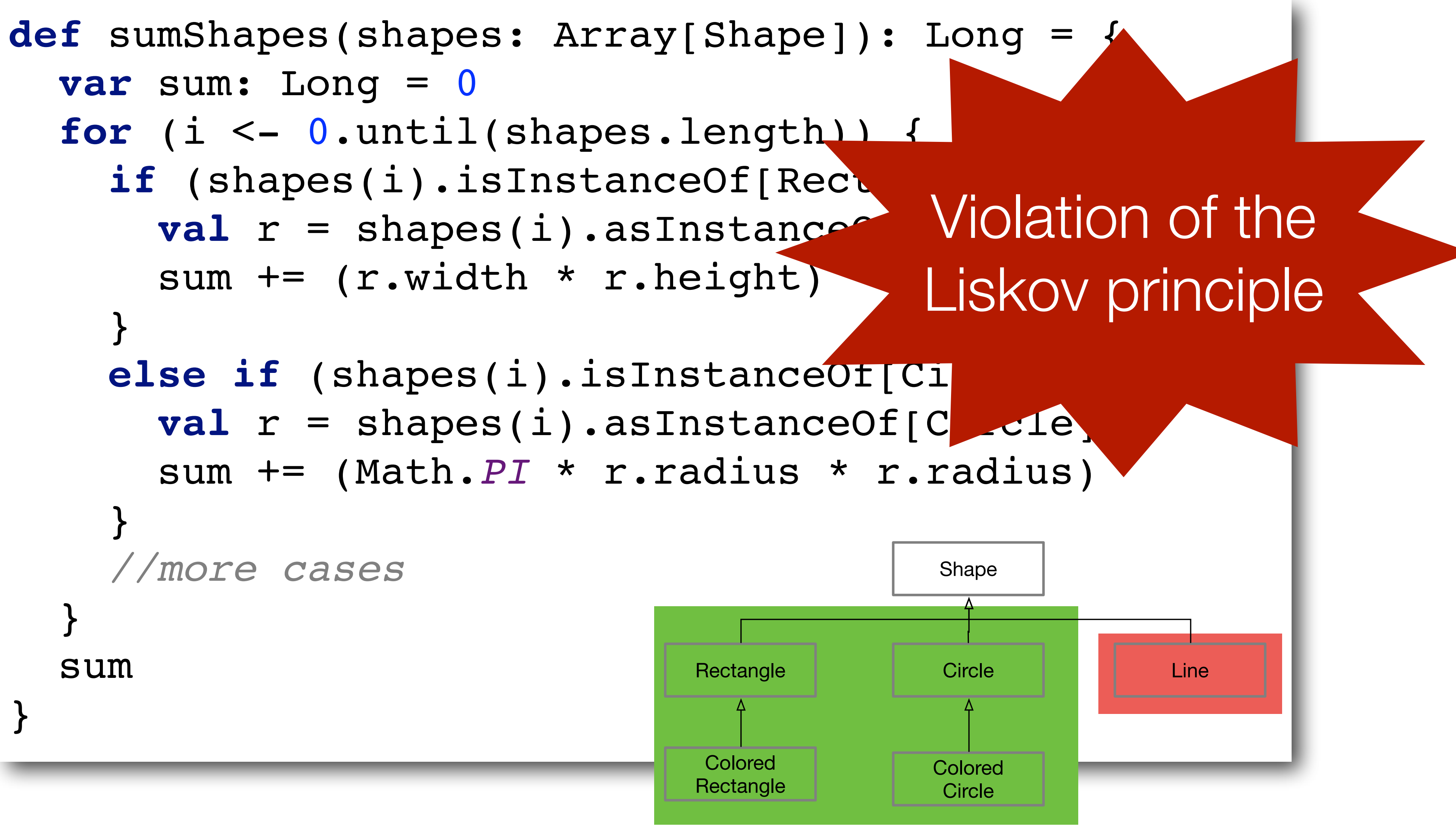
Circle

Line

Colored Rectangle

Colored Circle

Procedural coding style



```
def sumShapes(shapes: Array[Shape]): Long = {
  var sum: Long = 0
  for (i <- 0.until(shapes.length)) {
    if (shapes(i).isInstanceOf[Rect
      val r = shapes(i).asInstance
      sum += (r.width * r.height)
    }
    else if (shapes(i).isInstanceOf[Ci
      val r = shapes(i).asInstanceOf[C  cle]
      sum += (Math.PI * r.radius * r.radius)
    }
    //more cases
  }
  sum
}
```

Violation of the Liskov principle

Procedural coding style

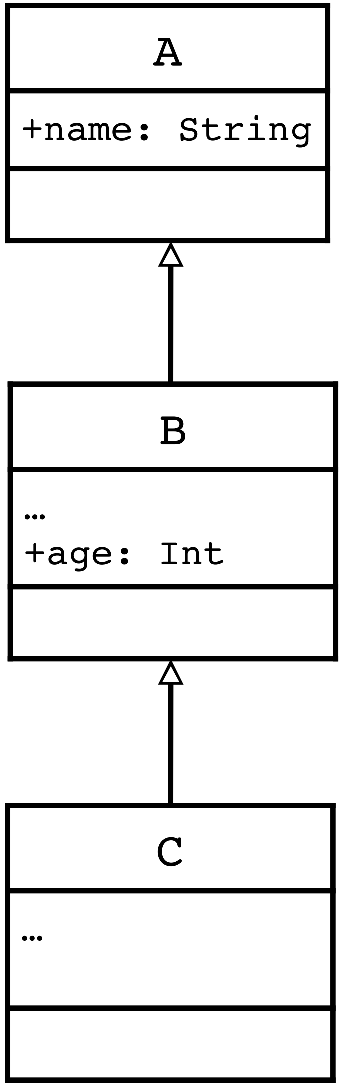
In general, procedural coding style (e.g., programming in plain C) makes difficult to extend a software

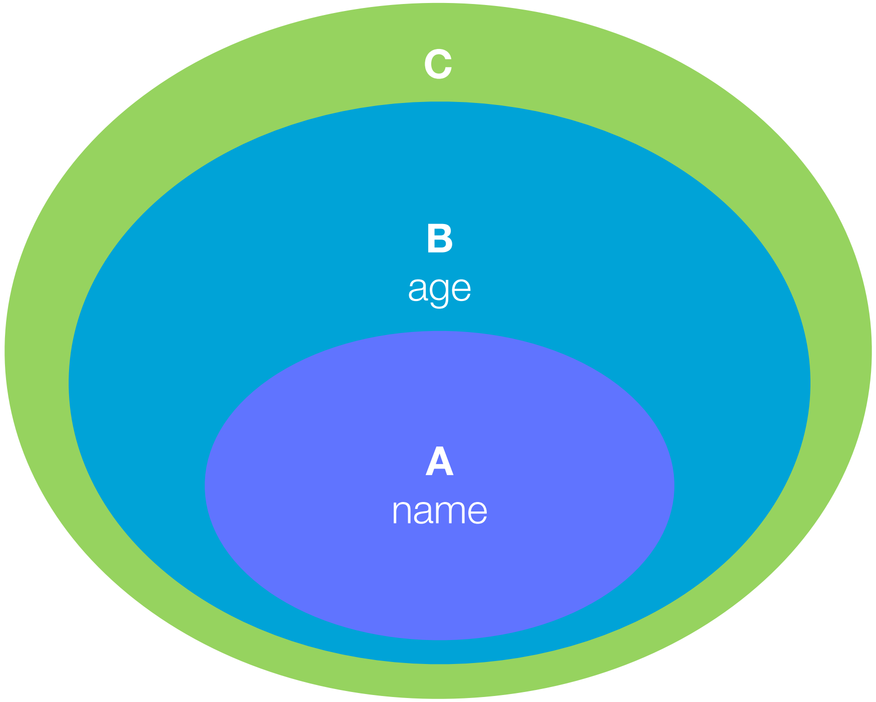
Software extension comes at a high cost:

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

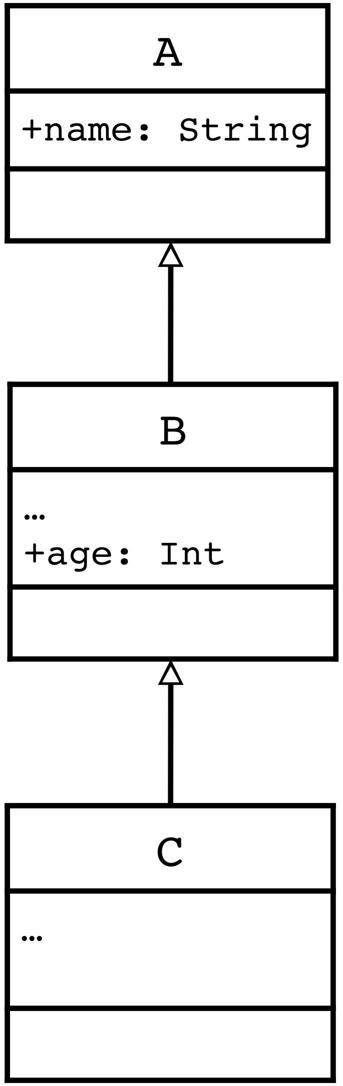
Object initialization

```
abstract class A(val name: String)
class B(name: String, val age: Int) extends A(name)
class C extends B("Foo", 0)
```





Object initialization


A

+name: String

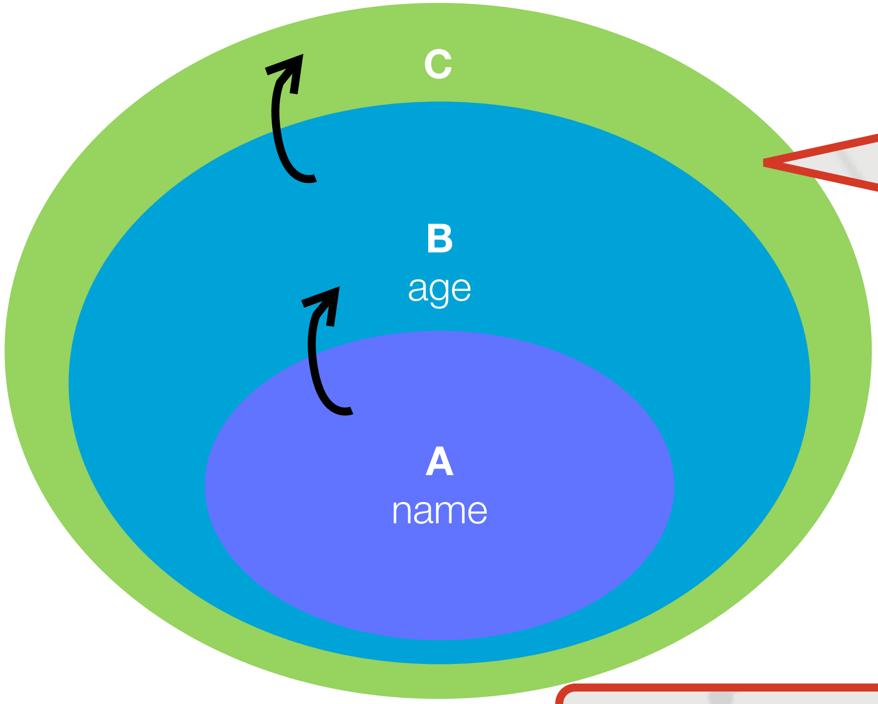
B

…
+age: Int

C

…

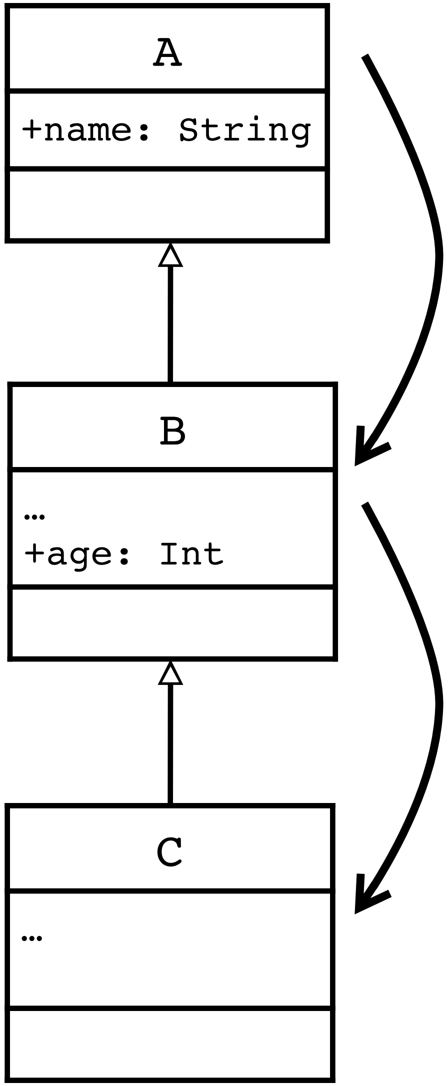
```
abstract class A(val name: String)
class B(name: String, val age: Int) extends A(name)
class C extends B("Foo", 0)
```



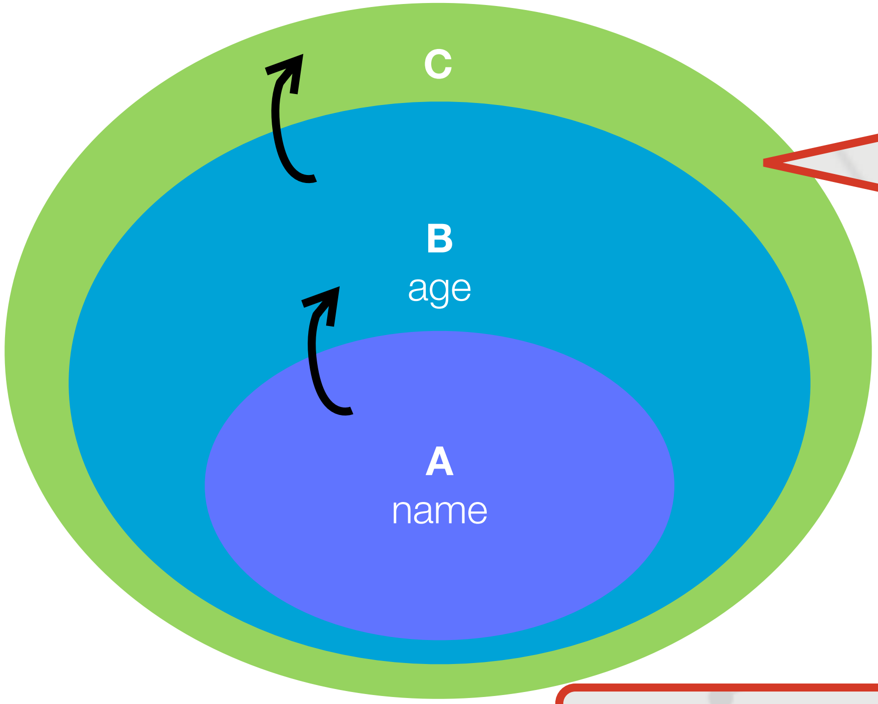
Order of object initialization, enforced by the call to the super class constructor in the primary constructor of each class

In other languages this is done by calling the super constructor.

Object initialization



```
abstract class A(val name: String)
class B(name: String, val age: Int) extends A(name)
class C extends B("Foo", 0)
```



Order of object initialization, enforced by the call to the super class constructor in the primary constructor of each class

In other languages this is done by calling the super constructor.

Outline

1. Liskov principle
2. **Visibility**
3. Exercise: talking to the Suchai satellite

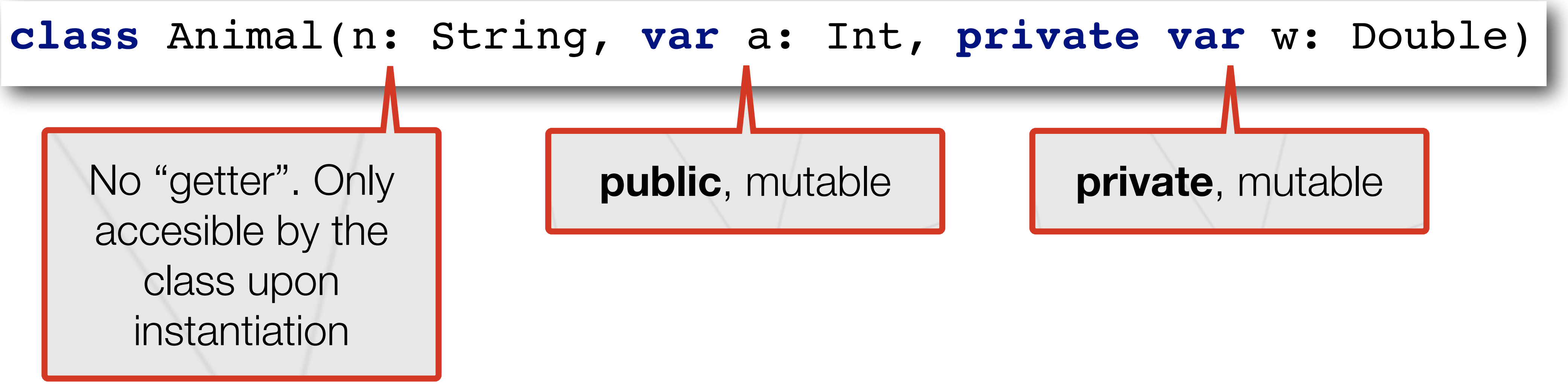
Visibility modifiers

Modifier	Class	Package	Subclass	World
`No modifier`	Y	Y	Y	Y
`protected`	Y	N	Y	N
`private`	Y	N	N	N

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

Visibility modifiers

Modifier	Class	Package	Subclass	World
`No modifier`	Y	Y	Y	Y
`protected`	Y	N	Y	N
`private`	Y	N	N	N



```
class Animal(n: String, var a: Int, private var w: Double)
```

No “getter”. Only accesible by the class upon instantiation

public, mutable

private, mutable

Visibility modifiers

Modifier	Class	Package	Subclass	World
No modifier	Y	Y	Y	Y
protected	Y	N	Y	N
private	Y	N	N	N

```
class Foo {
  private def isFoo = true
  def doFoo(other: Foo) {
    if (other.isFoo) {
      // ...
    }
  }
}
```

Does this compiles?

Yes!

Visibility modifiers

Modifier	Class	Package	Subclass	World
No modifier	Y	Y	Y	Y
protected	Y	N	Y	N
private	Y	N	N	N

```
package uchile

class A {
  private def foo() = {}
}
```

```
package uchile

class B extends A {
  def bar() = {
    foo()
  }
}
```

Does this compiles?

No!

Visibility modifiers

Modifier	Class	Package	Subclass	World
No modifier	Y	Y	Y	Y
protected	Y	N	Y	N
private	Y	N	N	N

```
package uchile

class A {
  protected def foo() = {}
}
```

```
package uchile

class B extends A {
  def bar() = {
    foo()
  }
}
```

Does this compiles?

Yes!

Visibility modifiers

Modifier	Class	Package	Subclass	World
No modifier	Y	Y	Y	Y
protected	Y	N	Y	N
private	Y	N	N	N

```
package uchile

class A {
  protected def foo() = {}
}
```

```
package uchile

class B {
  def bar() = {
    (new A()).foo()
  }
}
```

Does this compiles?

No!

Visibility modifiers

Modifier	Class	Package	Subclass	World
No modifier	Y	Y	Y	Y
protected	Y	N	Y	N
private	Y	N	N	N

```
package uchile

class A {
  protected def foo() = {}
}
```

```
package suchai

class B extends A {
  def bar() = {
    foo()
  }
}
```

Does this compiles?

Yes!

Visibility modifiers (refined - Scala exclusive)

Modifier	Class	Package	Subclass	World
No modifier	Y	Y	Y	Y
protected	Y	N	Y	N
private	Y	N	N	N

```
package uchile

class A {
  private[uchile] def foo() = {}
}
```

Private for members of the uchile package

```
package uchile

class B {
  def bar() = {
    (new A()).foo()
  }
}
```

Does this compiles?

Yes!

Visibility modifiers (refined - Scala exclusive)

Modifier	Class	Package	Subclass	World
`No modifier`	Y	Y	Y	Y
`protected`	Y	N	Y	N
`private`	Y	N	N	N

The strongest form of privacy: only visible by the same instance

```
class A {
  private[this] def foo() = {}
  def bar(a: A) = {
    a.foo()
  }
}
```

Does this compiles?

No!

Why not having all methods public?

```
class Account(var user: String, var password: String) {
  def getPassword(): String = password
}
```

```
class CheckLogin {
  def canLogin(a: Account, pass: String): Boolean = {
    a.getPassword == pass
  }
}
```

This version has a security vulnerability

Why not having all methods public?

```
class Account(var user: String, var password: String) {
  def getPassword(): String = password
}
```

```
class CheckLogin {
  def canLogin(a: Account, pass: String): Boolean = {
    a.getPassword == pass
  }
}
```

```
class Virus {
  def getPassword(a: Account): Unit = {
    println(a.getPassword)
  }
}
```

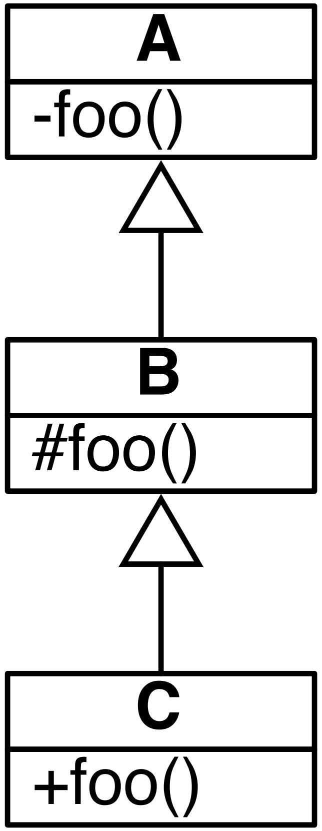
As soon as someone get an instance of Account, password can be accessed

Access privileges can only be widened

```
class A {
   private def foo(): Unit = {
   }
}

class B extends A {
   protected def foo(): Unit = {
   }
}

class C extends B {
   override def foo(): Unit = {
   }
}
```



Would it be okay to have this?

```
class A {
  def foo(): Unit = {
  }
}

class B extends A {
  override protected def foo(): Unit = {
  }
}

class C extends B {
  override private def foo(): Unit = {
  }
}
```

Would it be okay to have this?

```
class A {
  def foo(): Unit = {
  }
}

class B extends A {
  override protected def foo(): Unit = {
  }
}

class C extends B {
  override private def foo(): Unit = {
  }
}
```

Violation of the Liskov principle

Outline

1. Liskov principle

2. Visibility

3. **Exercise: talking to the Suchai satellite**

The Suchai Nano-satellite

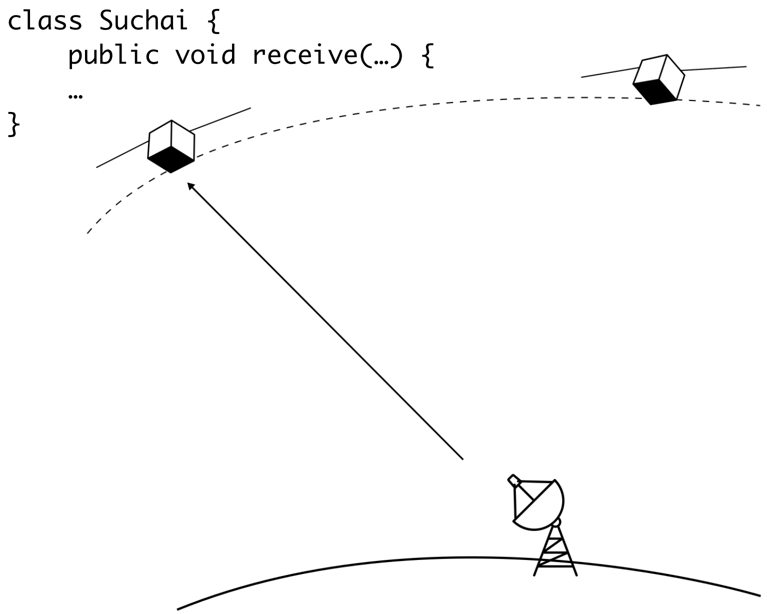
Nano-satellite (1000 cm^3 = 10cm × 10 cm × 10cm) built in 🇨🇱

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

```
package suchai;
import scala.collection.mutable.ListBuffer

class Suchai {
  private var angle: Int = 0
  private var pictures: ListBuffer[Picture] = ListBuffer()

  def setAngle(newAngle: Int): Unit = {
    angle = newAngle
  }

  def getAngle(): Int = angle

  def numberOfPictures(): Int = pictures.size

  def receive(c: Command): Unit = {
    c.doExecute(this)
  }

  def addPicture(p: Picture): Unit = {
    pictures += p
  }
}
```

```
package suchai;

object GroundStation {
    def main(args: Array[String]): Unit = {
      val s = new Suchai()

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

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

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

```
package suchai;

trait Command {
  def doExecute(suchai: Suchai): Unit
}

class RotateCommand extends Command{
  override def doExecute(suchai: Suchai): Unit = {
    suchai.setAngle(suchai.getAngle()+10)
  }
}

class TakePictureCommand extends Command {
  def doExecute(suchai: Suchai): Unit = {
    suchai.addPicture(new Picture())
  }
}
```

```
package suchai;

class Picture
```

```
package suchai;
import scala.collection.mutable.ListBuffer

class Suchai {
  private var angle: Int = 0
  private var pictures: ListBuffer[Picture] = ListBuffer()

  def setAngle(newAngle: Int): Unit = {
    angle = newAngle
  }

  def getAngle(): Int = angle

  def numberOfPictures(): Int = pictures.size

  def receive(c: Command): Unit = {
    c.doExecute(this)
  }

  def addPicture(p: Picture): Unit = {
    pictures += p
  }
}
```

Double dispatch

What you should know!

What is the Liskov principle?

How the Liskov principle affects the design of a programming language

License

dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

/ DCCUCHILE