Summer Semester 2015

# Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

## Liskov Substitution Principle

# ***L****iskov* ***S****ubstitution* ***P****rinciple*

*Subtypes must be behaviorally substitutable for their base types.*

–Barbara Liskov, 1988  (ACM Turing Award Receiver)

We identified class inheritance and subtype polymorphism as primary mechanisms for supporting the **open-closed principle** (OCP) in object-oriented designs.

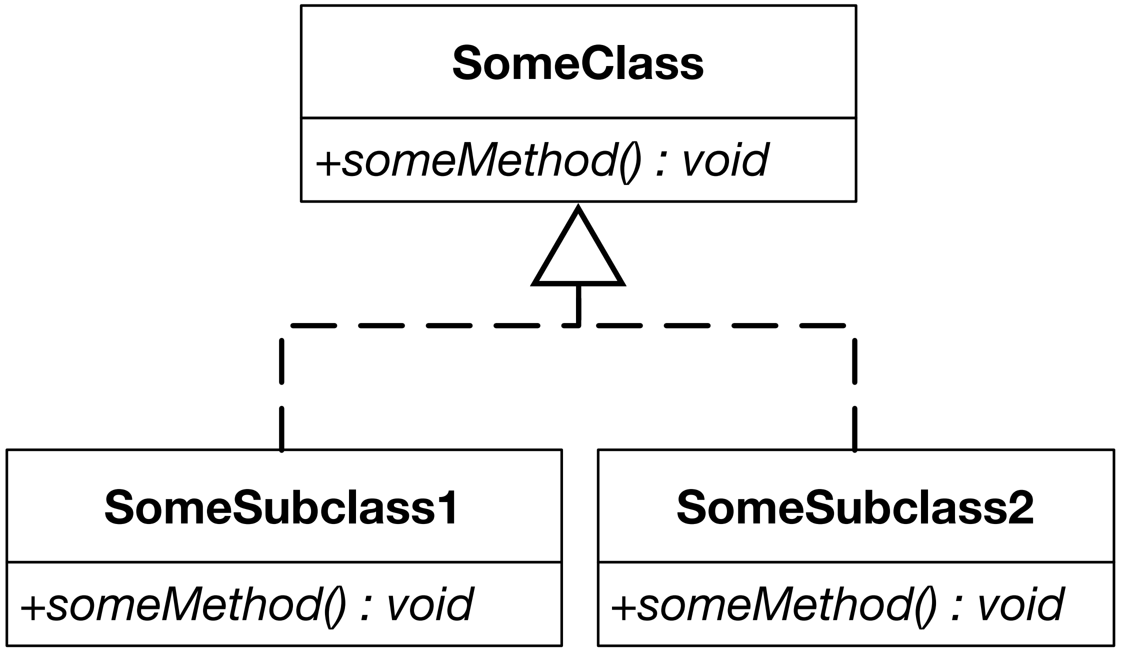
# The Liskov Substitution Principle

- … gives us a way to characterize good inheritance hierarchies.

- … increases our awareness about traps that will cause us to create hierarchies that do not conform to the open-closed principle.

# Substitutability in object-oriented programms

```
void clientMethod(SomeClass sc)
{
    …
    sc.someMethod();
    …
}
```



# Liskov Substitution Principle by Example

- Assume, we have implemented a class **Rectangle** in our system.

```
class Rectangle {
  public void setWidth(int width) { this.width = width; }
  public void setHeight(int height) { this.height = height;}
  public void area() { return height * width;}
}
```

- Let's now assume that we want to implement a class **Square** and want to maximize reuse.

# Liskov Substitution Principle by Example

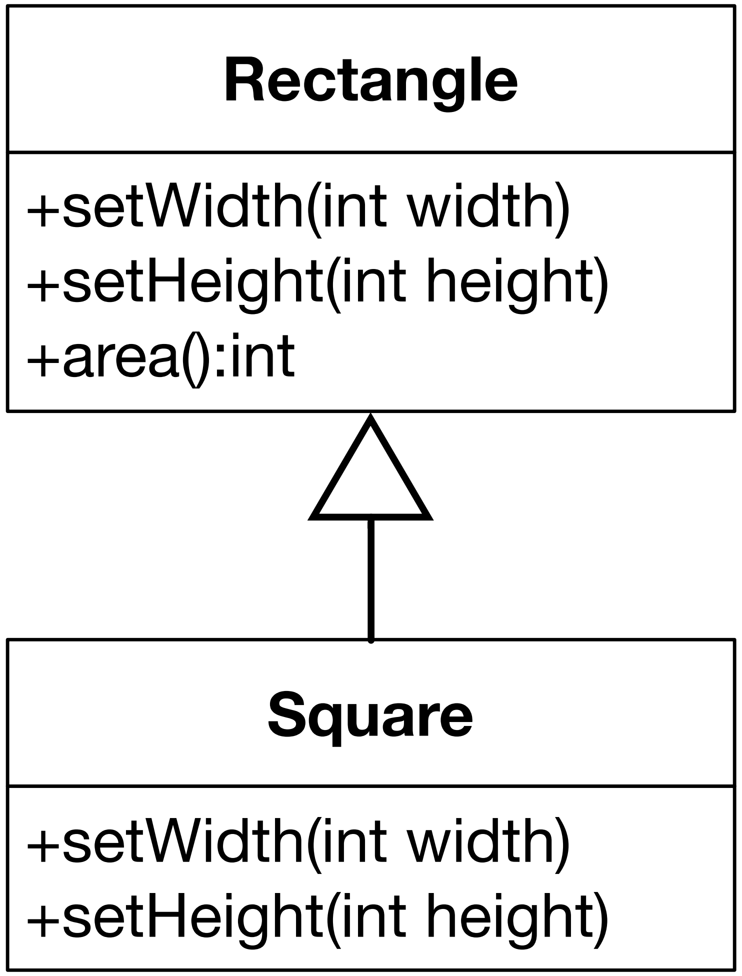
- Implementing `Square` as a subclass of `Rectangle`

```
class Square extends Rectangle {

  public void setWidth(int width) {
    super.setWidth(width);
    super.setHeight(width);
  }

  public void setHeight(int height) {
    super.setWidth(height);
    super.setHeight(height);
  }

}
```



**Rectangle**

+setWidth(int width)
+setHeight(int height)
+area():int

**Square**

+setWidth(int width)
+setHeight(int height)

- We can pass `Square` wherever `Rectangle` is expected.

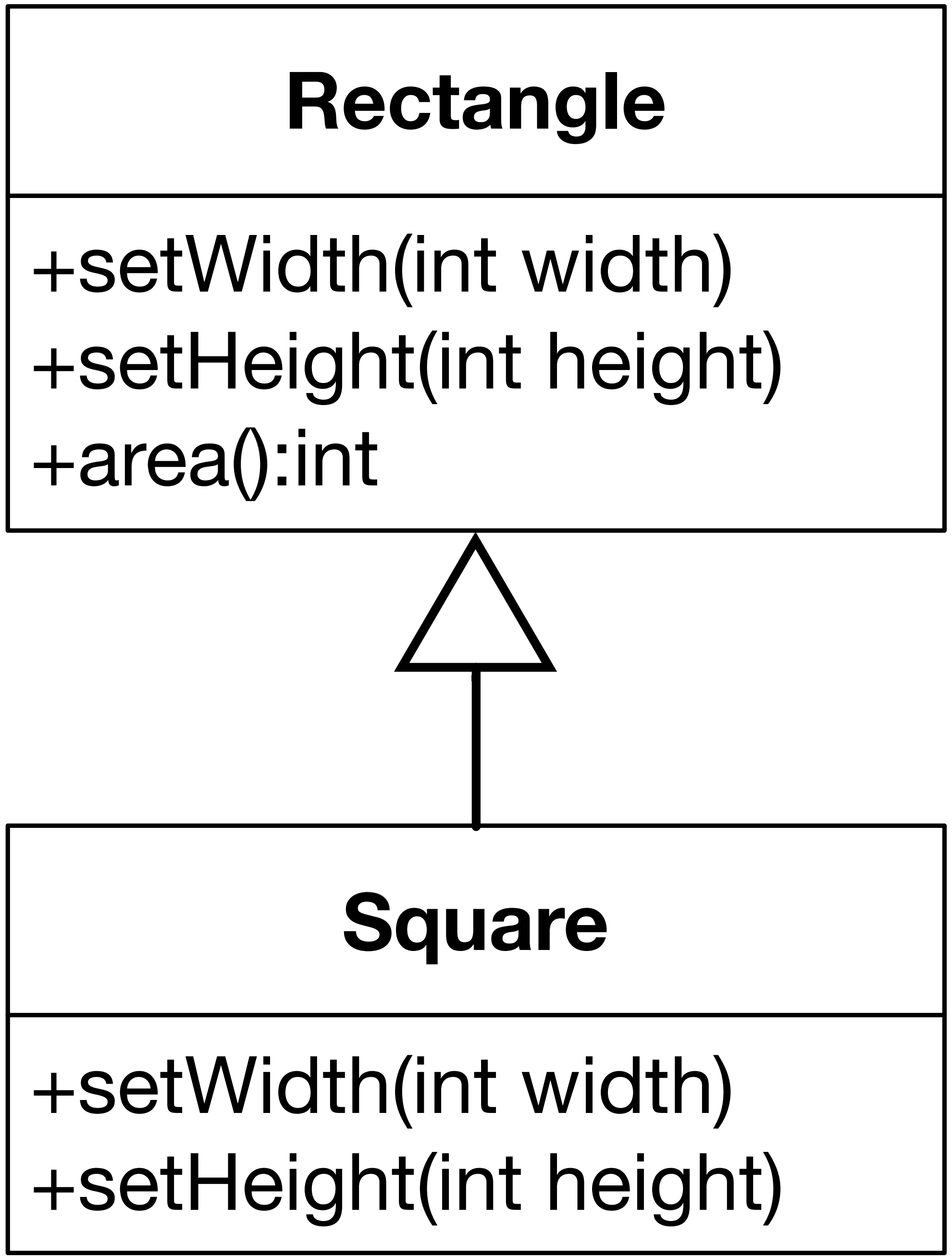
What do you think of this design?

This model is self-consistent!

# Liskov Substitution Principle by Example

- A client that works with instances of `Rectangle`, but breaks when instances of `Square` are passed to it

```
void clientMethod(Rectangle rec)
{
  rec.setWidth(5);
  rec.setHeight(4);
  assert(rec.area() == 20);
}
```



**Rectangle**

+setWidth(int width)
+setHeight(int height)
+area():int

**Square**

+setWidth(int width)
+setHeight(int height)

# Software Is All About Behavior

Programmers do not define entities that are something, but entities that behave somehow.

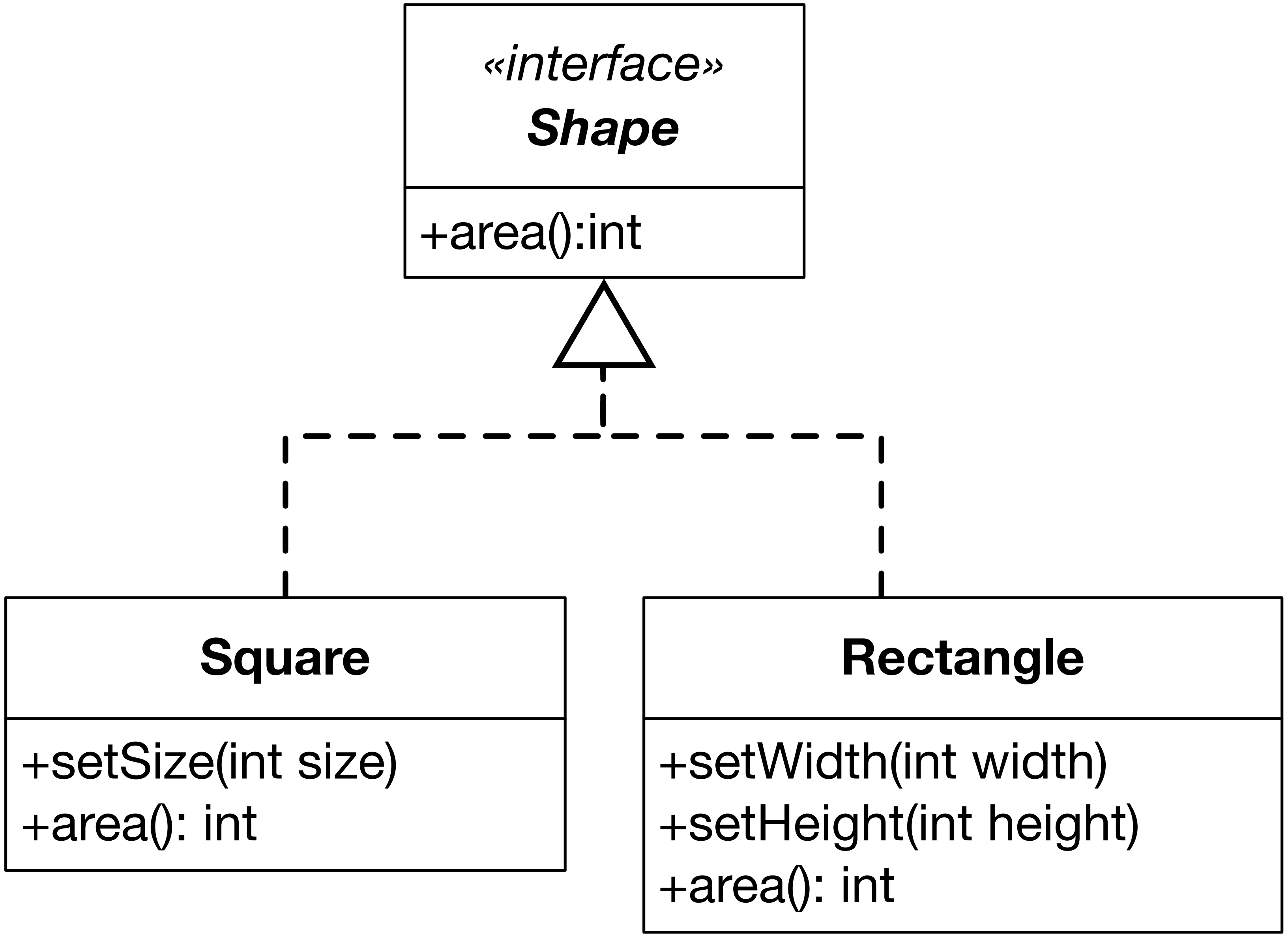
# Validity is not Intrinsic!

Inspecting the `Square/Rectangle` hierarchy in isolation did not show any problems. In fact it even seemed like a self-consistent design.

**We had to inspect the clients to identify problems.**

# Liskov Substitution Principle by Example

Rectangles and Square - LSP Compliant Solution



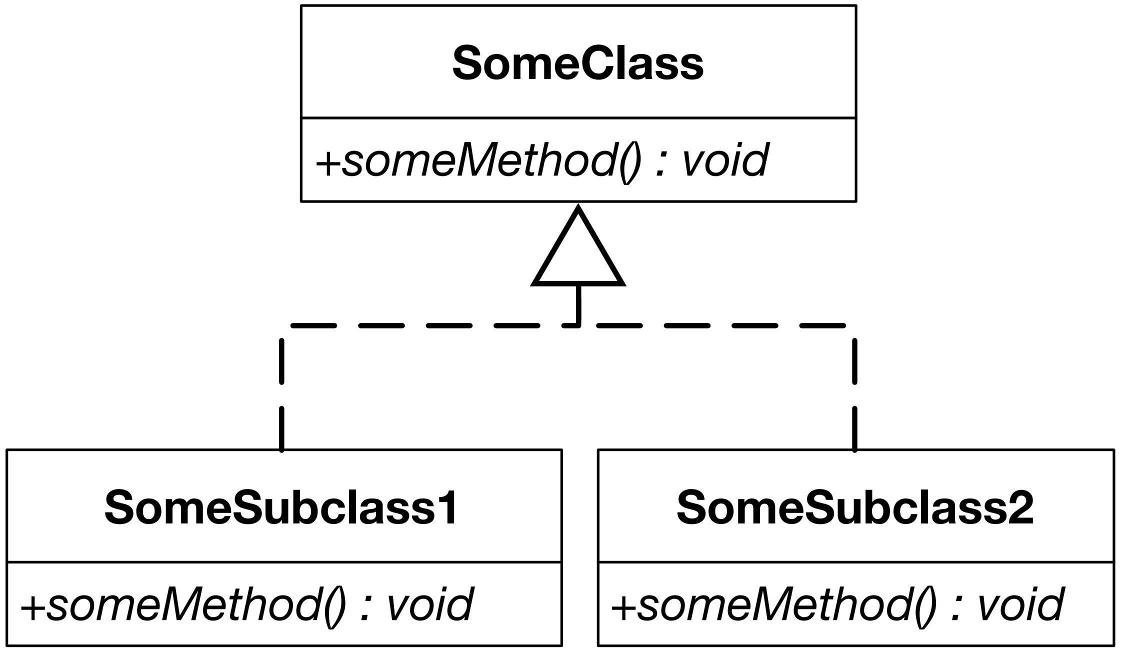
So what does the Liskov Substitution Principle add to the common object-oriented subtyping rules?

The Liskov Substitution Principle additionally requires behavioral substitutability

# Behavioral Substitutability

- It's not enough that instances of `SomeSubclass1` and `SomeSubclass2` provide all methods declared in `SomeClass`.
  These methods should also behave like their heirs!

- A client method should not be able to distinguish the behavior of objects of `SomeSubclass1` and `SomeSubclass2` from that of objects of `SomeClass`.



# Behavioral Subtyping

S is a behavioral subtype of T, if objects of type T in a program P may be replaced by objects of type S without altering any of the properties of P.

# The Relation between LSP and OCP

- Consider a function `f` parameterized over type `T`
  - `S` is a derivate of `T`.
  - when passed to `f` in the guise of objects of type `T`, objects of type `S` cause `f` to misbehave.
  - `S` violates the Liskov Substitution Principle.

`f` is fragile in the presence of `S`

Can you think of straightforward examples of violations of the *__L__iskov __S__ubstitution __P__rinciple?*

# Properties extends Hashtable

LSP Violation in the JDK

Because `Properties` inherits from `Hashtable`, the `put` and `putAll` methods can be applied to a `Properties` object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not `Strings`. The `setProperty` method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail.

# What mechanisms can we use to support LSP?

Recall:
A model viewed in isolation cannot be meaningfully validated with respect to LSP.
Validity must be judged from the perspective of possible usages of the model.

# Design by Contract

- Solution to the validation problem: A technique for **explicitly stating what may be assumed**.

- Two main aspects of design-by-contract:

  - We can specify **contracts** using Pre-, Post-Conditions and Invariants.
    They must be respected by subclasses and clients can rely on them.

  - **Contract enforcement** (behavioral subtyping).
    Tools to check the implementation of subclasses against contracts of superclasses.

# Contract for `Rectangle.setWidth(int)`

Design by Contract

```java
public class Rectangle implements Shape {
  private int width;
  private int height;

  public void setWidth(int w) {
    this.width = w;
  }
}
```

# Contract Enforcement

## Subclasses must conform to the contract of their base class!

- This is called behavioral subtyping.
- It ensures that clients won't break when instances of subclasses are used in the guise of instances of their heirs!

What would the subtyping rules look like?

What does it mean for a subclass to conform to the contract of the base class?

# Behavioral Subtyping

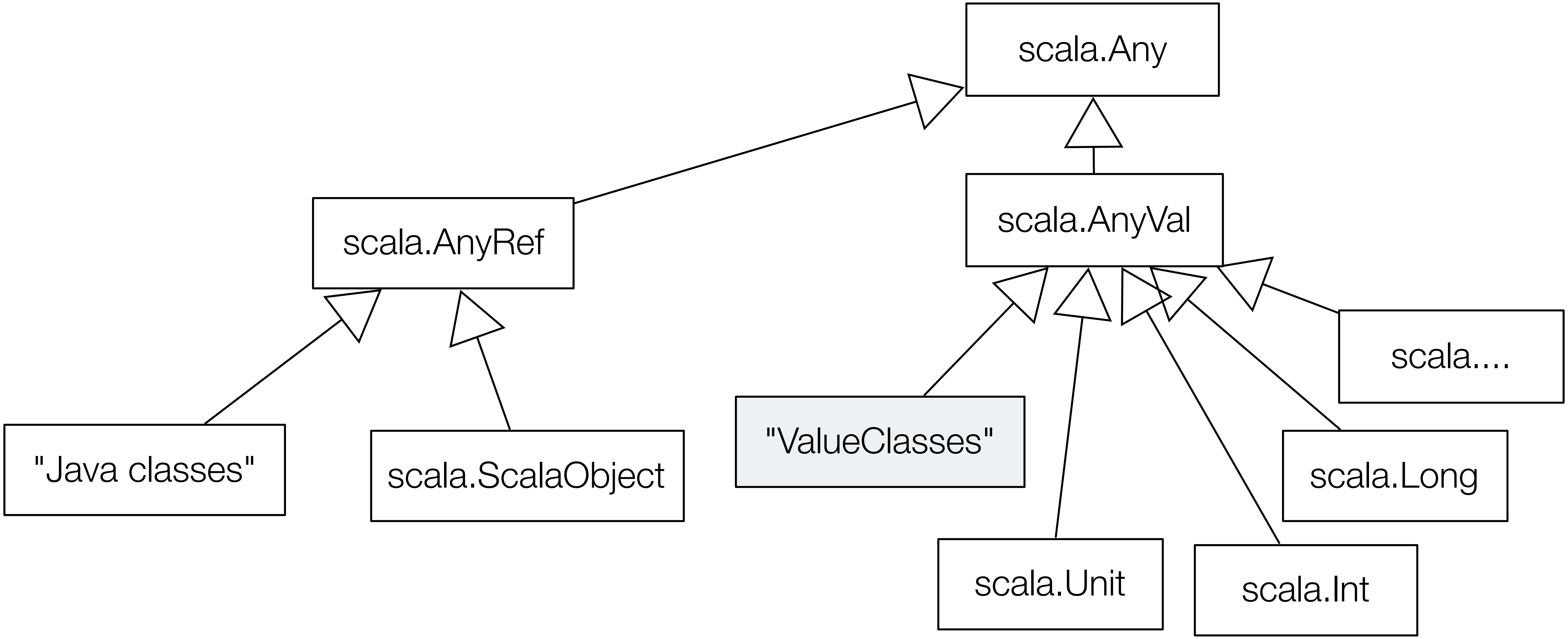
- Rule for Preconditions:
  - Preconditions of a class imply preconditions of its subclasses.
  - Preconditions may be replaced by **(equal or) weaker ones**.
- Rule for Postconditions:
  - Postconditions of a class are implied by those of its subclasses.
  - Postconditions may be replaced **by equal or stronger ones.**

# "Standard" Subtyping

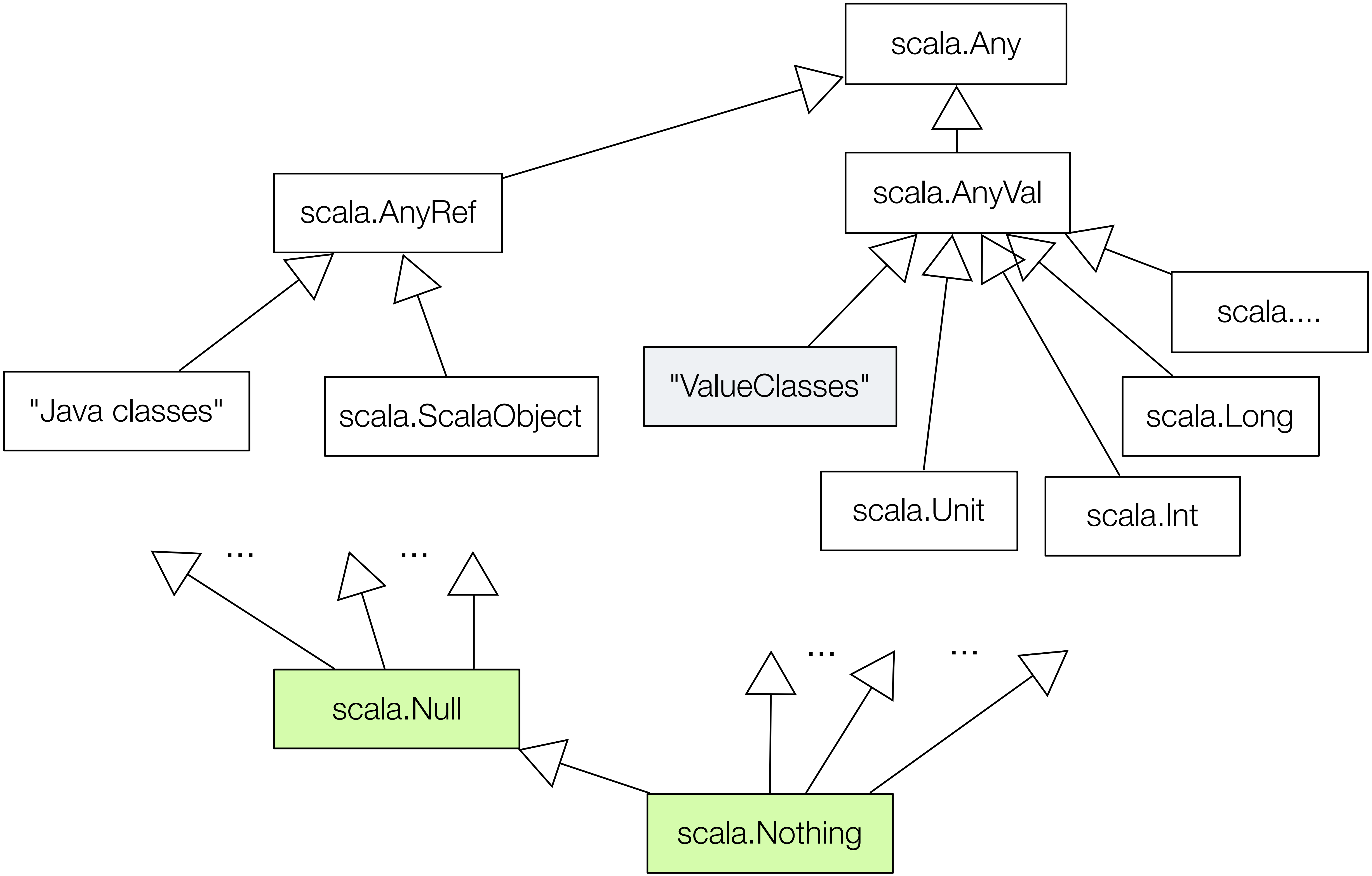
“Standard” subtyping relies on contra-variance of the argument types and covariance of the return type for enforcing “pre- and post-conditions on signatures”.

- f:        T1 → T2
- f’:       T1’ → T2’
- f’ <: f <=> T1 <: T1’ and T2’ <: T2 (f' is a subtype of f)

# Scala's Type Hierarchy



# Scala's Type Hierarchy



# Using Least Types

```scala
trait Store[+A] {
    def +[B >: A](b: B): Store[B]
    def contains(a: Any): Boolean
}
object EmptyStore extends Store[Nothing] {
    def +[B](b: B): Store[B] = new LinkedListStore(b, this)
    def contains(b: Any) = false
}
class LinkedListStore[+A](val v: A, val rest: Store[A]) extends Store[A] {
    def +[B >: A](b: B): LinkedListStore[B] = new LinkedListStore(b, this)
    def contains(a: Any): Boolean = this.v == a || (rest contains a)
}

object Main extends App {
    val a: Store[Int] = EmptyStore + 1 + 2
    val b: Store[Any] = a
    println(b contains 1)
    println(b contains 3)
}
```

# "Standard" Subtyping in Scala

```
val f: (Seq[_]) ⇒ Boolean = (s) ⇒ { s eq null }

val af1: (Object) ⇒ Boolean = null // = f ?

val af2: (List[_]) ⇒ Boolean = null // = f ?

val af3: (Seq[_]) ⇒ Any = null // = f ?

val af4: (Seq[_]) ⇒ Nothing = null // = f ?
```

Is it possible to assign a value of type `f` to the variable: `af1`, `af2`, `af3` or `af4`?

# Behavioral and Standard Subtyping in OO

**Behavioral subtyping** is a stronger notion than subtyping of functions defined in type theory.

- LSP imposes some standard requirements on signatures that have been adopted in OO languages:
  - contra-variance/covariance of method argument/return types.
  - no new (checked) exceptions should be thrown by methods of the subtype, except for those exceptions that are subtypes of exceptions thrown by the methods of the super-type.
- In addition, there are a number of conditions that behavioral subtypes must meet concerning values (rather than types) of input and output.
- Behavioral subtyping is undecidable in general.

# Languages and Tools for Design-by-Contract

- Contracts as comments in code or in documentation.
- Unit-tests as contracts.
- Formalisms and tools for specifying contracts in a declarative way and enforcing them.

# Java Modeling Language

- A behavioral interface specification language that can be used to specify the behavior of Java modules.

```java
public class Rectangle implements Shape {

  private int width;
  private int height;

  /*@
    @ requires w > 0;
    @ ensures height = \old(height) && width = w;
    @*/
  public void setWidth(double w) {
    this.width = w;
  }
}
```

# Contracts in Documentation

One should document any restrictions on how a method may be overridden in subclasses.

# The Contract of `Object.equals(...)`

```
public boolean equals(Object obj)
```

*Indicates whether some other object is "equal to" this one.*

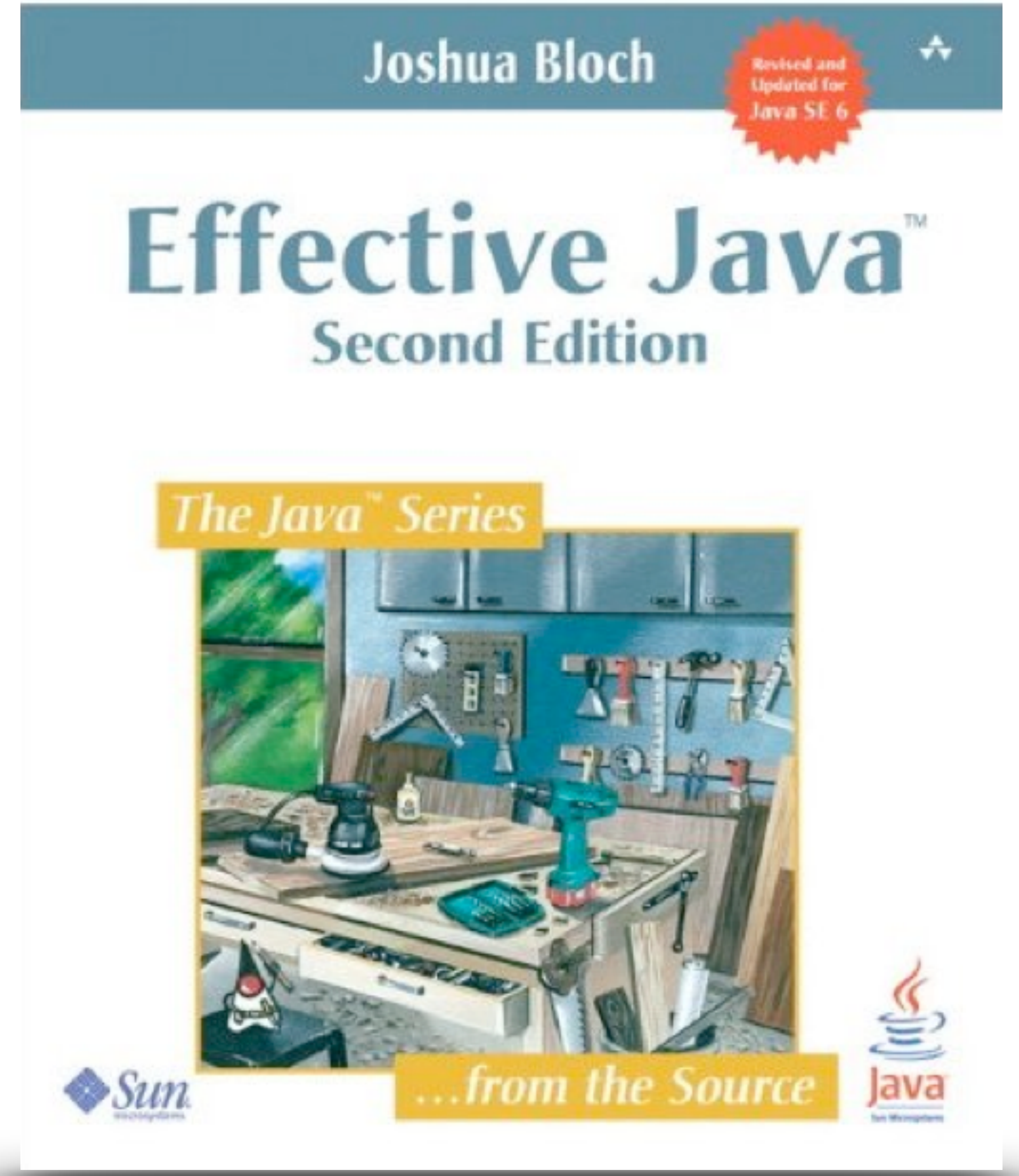
*The equals method implements an equivalence relation on non-null object references:*

- *It is reflexive: for any non-null reference value x, x.equals(x) should return true.*
- *It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.*
- *It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.*
- *It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.*
- *For any non-null reference value x, x.equals(null) should return false.*

*The equals method for class Object implements the most discriminating possible equivalence relation on objects...*

The documentation consists almost entirely of restrictions on how it may be overridden.

# The Contract of `Object.equals(...)`



# Object.equals(Object o)

## Example Implementation

```java
/**
 * Case-insensitive string. Case of the original string is preserved by toString, but
 * ignored in comparisons.
 */
public final class CaseInsensitiveString {

  private String s;

  public CaseInsensitiveString(String s) {
    if (s == null)
      throw new NullPointerException();
    this.s = s;
  }

  public boolean equals(Object o) {
    if (o instanceof CaseInsensitiveString)
      return s.equalsIgnoreCase(((CaseInsensitiveString) o).s);
    if (o instanceof String)
      return s.equalsIgnoreCase((String) o);
    return false;
  }
  // Remainder omitted
}
```

# Example Usage of CaseInsensitiveString

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
List list = new ArrayList();
list.add(cis);

return list.contains("polish"); // true or false ?
```

Once you have violated `equals`'s contract, you simply don’t know how other objects will behave when confronted with your object.

# The Implementation of `java.net.URL.equals`

```
public boolean equals(Object obj)
```

- *Compares this URL for equality with another object.*
- *If the given object is not a URL then this method immediately returns false.*
- *Two URL objects are equal if they have the same protocol, reference equivalent hosts, have the same port number on the host, and the same file and fragment of the file.*
- *Two hosts are considered equivalent if both host names can be resolved into the same IP addresses; else if either host name can't be resolved, the host names must be equal without regard to case; or both host names equal to null.*
- *Since hosts comparison requires name resolution, this operation is a blocking operation.*

# Enforcing Documented Contracts

- Maybe hard when done manually …
- May require very powerful tooling (theorem proving) …
- Is un-decidable in general.

# The Imperative of Documenting Contracts

It is necessary to carefully and precisely document methods that may be overridden because one cannot deduce the intended specification from the code.

```
package java.lang;

class Object {
  public boolean equals(Object ob ) { return this == ob; }
}
```

# The Imperative of Documenting Contracts

RFC (Request for Comments) 2119 defines keywords - may, should, must, etc. – which can be used to express so-called „subclassing directives“.

```
/**
 * Subclasses should override...
 * Subclasses may call super...
 * New implementation should call addPage...
 */
 public void addPages() {...}
```

# On the Quality of the Documentation

When documenting methods that may be overridden, one must be careful to document the method in a way that will **make sense for all potential overrides** of the function.

# Generating API Documentation with JAutoDoc

The complete documentation was auto-generated.

```
/**
 * The number of questions.
 */
private int numberOfQuestions;


/**
 * Sets the number of questions.
 *
 * @param numberOfQuestions the number of questions
 * @throws IllegalArgumentException the illegal argument exception
 */
public void setNumberOfQuestions(int numberOfQuestions)
      throws IllegalArgumentException {
    if (numberOfQuestions < 0) {
        throw new IllegalArgumentException("numberOfQuestions < 0");
    }
    this.numberOfQuestions = numberOfQuestions;
}
```

Takeaway

Subtypes must be behaviorally substitutable for their base types.

Takeaway

- Behavioral subtyping extends “standard” OO subtyping.
  Additionally ensures that assumptions of clients about the behavior of a base class are not broken by subclasses.

- Behavioral subtyping helps with supporting OCP.
  Only behavioral subclassing (subtyping) truly supports open-closed designs.

- Design-by-Contract is a technique for supporting LSP.
  Makes the contract of a class to be assumed by the clients and respected by subclasses explicit (and checkable).