Principles of Programming Languages

III Data Types and Type Systems

Dr. Herbert Prähofer
Institute for System Software
Johannes Kepler University Linz

III Data Types and Type Systems

- III.1 Types, Subtypes and Inheritance
- III.2 Algebraic Data Types and Type Classes
- III.3 Generic Types
- III.4 Type extensions

Principles of Programming Languages

III.1 Types, Subtypes and Inheritance

Dr. Herbert Prähofer
Institute for System Software
Johannes Kepler University Linz

III.1 Types, Subtypes and Inheritance

- Introduction
- Subtyping
- Liskov's substitution principle
- Multiple Inheritance
- Mixin Inheritance and Scala Traits
- Summary

DEFINITION TYPE

other definitions exist

"A type is a means for classification of values according to their properties, structure, and allowed operations"

- types are **assigned to values, variables, expressions, parameters, return values, functions**, …
- types define **subset of values**
  - where elements in the subset own equal properties, structure and operations
- types support **type checking**
  - operations are allowed
  - assignments are safe
  - ...

Types as Sets

- **Universal domain** of all computable/representable values of a programming language
- Types define **subsets of values** from the universal domain

Correspondence between types and sets

types		**interpretation in set theory**	
type	T	set_T	: set of values defined by T
is type	$x : T$	$x \in set_T$	: value of x is element of set of values of T
subtype	$S \leq B$	$set_S \subseteq set_B$	: set of subtype S is subset of set of type B

Type System

Type systems of programming languages define

- the **available elementary data types**
- the creation of new **data types**
- how the **allowed operations of types** are defined
- how **typing of values, expressions, and variables** is accomplished
- how the **type checking** and/or **type inference** works

Type Systems of Object-oriented Languages

Type systems of programming languages define

- the **available elementary data types**
- the creation of new **data types**
- how the **allowed operations of types** are defined
- how **typing of values, expressions, and variables** is accomplished
- how the **type checking** and/or **type inference** works

Object-oriented type systems do that by

- **built-in types**, e.g. int, Object, ...
- **classes, interfaces** with **inheritance** and **subtyping**
- **methods of classes** and **interfaces**
- **variable declarations** by programmer, **type inference** by compiler
- **static type checking** and **type inference** by compiler

TYPE VS. CLASS

- In object-oriented languages, types and classes are often used as synonyms
- We want to distinguish types and classes as follows:

Class

- implementation
  - of objects with same structure or interface
  - with fields and methods

Type

- set of objects
  - showing the same interface
  - having the same members

Subclassing

- inheritance of fields and method implementations
- <u>Goal</u>: implementation reuse

Subtyping

- logical relation corresponding to subset relation
- <u>Goal</u>: Guaranteeing compatibility

- A class introduces a type
- But types need not correspond to defined classes

Static typing vs. dynamic typing

- **Static typing**
  - ☐ Type checking is done by **compiler at compile time**
  - ➔ **Variables**, **expressions**, **functions** carry type information
  - ➔ Type errors are signaled as **syntax errors at compile time**

Examples:
- Haskell
- Java
- Scala
- C#
- C/C++
- Ada
- TypeScript
- Rust
- …

Some checks at run time, e.g. for downcasts

- **Dynamic typing**
  - ☐ **Type checking** is done at **run time**
  - ➔ **Values** carry type information
  - ➔ Type errors are signaled as **run-time errors**

Examples:
- Lisp/Scheme
- Smalltalk
- JavaScript
- Python
- Ruby
- …

Concepts and Terms

Strong typing

- No type errors are possible
  - prevented either by compiler or at run time
- Type system does not allow bypass typing rules, e.g., by
  - by unchecked typecasts
  - by pointer arithmetic

Examples:

Languages with strong typing:

- Haskell (by static checks only)
- Java, Scala, C# (by static and dynamic checks)

Languages without strong typing:

- C, C++

Type safety

- A language is called **type-safe** if it does not allow bypassing typing rules

Concepts and Terms

Predefined vs. user-defined types

- **Predefined types** are defined in the language specification
- **User-defined types** are defined in the application program
  - Language has to have constructors for defining new types

e.g. Fortran had no means for defining new types

Scalar vs. structured types

- Values of **scalar types** consist of a single coherent value without externally visible structure
- Values of **structured types** consists of several parts, individually accessible

Concepts and Terms

Explicit vs. implicit declaration

- **Explicit typing**
  - Types of program elements are **explicitly defined by the programmer**
- **Implicit typing**
  - Types of program elements are **inferred by the type system**

Named vs. unnamed types

- **Named types**
  - have an explicit name
    - e.g., classes in Java, e.g. `String`
- **Unnamed types**
  - Are constructed without giving an explicit name
    - e.g., arrays in Java line `String[]`
    - e.g. anonymous classes in Java

Monomorphic vs. polymorphic type systems

- **Monomorphic type systems**
  - A program element can only have a single type
  - A procedure/function can only work with elements of a single type

e.g. as in Algol 60 or Pascal

- **Polymorphic type systems**
  - A **program elements** can be of **different types**
  - A **procedure/function** can work with **elements of different types**

<u>from:</u>
Luca Cardelli, Peter Wegner: On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, Vol 17 N. 4, Dec 1985

- polymorphic type systems
  - universal quanitified
    - subtyping → oo type hierarchies
    - parametric → generics
  - ad hoc polymorph
    - overloading → e.g.
      ```
      1 + 2
      "a" + "B"
      ```
    - converting → e.g.
      ```
      int x = 1;
      double y = 1.0 + x
      ```
      x converted to double
    - **type classes**

see Section III.4

NOMINAL VS. STRUCTURAL TYPING

■ Nominal typing

- ☐ Types are equivalent if they have the **same name**

```
case class Person(
   name: String;
   age: Int;
)
```

```
case class Pet(
   name: String;
   age: Int;
)
```

```
val paul = Person("Paul", 5)
```

```
val son: Person = paul
```
✓

```
val dog: Pet = paul
```
✗

Person and Pet not compatible

■ Structural typing

- ☐ Types are equivalent if they have the **same structure**

```
class Person {
   name: string;
   age: number;
}
```

```
class Pet {
   name: string;
   age: number;
}
```

```
let paul : { name: "Paul", age: 5 }
```

```
let son: Person = paul
```
✓

```
let dog : Pet = paul
```
✓

compatible with both because same structure

Structural Typing in TypeScript

https://www.typescriptlang.org/

TypeScript is the statically-typed language layer for JavaScript

TypeScript is

- **statically** typed
- **structurally** typed

Structural equivalence

- on the level of objects

```
let son = { name: "Paul", age : 5 }

let cat = { name: "Susi", age : 12}

cat = son
```

- on the level of object types

```
type Pet = {
   name: string;
   age: number;
}
type Person = {
   name: string;
   age: number;
}

let son : Person = { name: "Paul", age : 5 }

let pet : Pet = son;
```

- on the level of classes

```
class Pet {
   constructor(name: string, age: number) {...}
   name: string;
   age: number;
}
class Person {
   constructor(name: string, age: number) {...}
   name: string;
   age: number;
}

let son : Person = new Pet("Paul", 5)
let dog : Pet = new Person("Paul", 5)
```

!?

III.1 Types, Subtypes and Inheritance

- Introduction
- Subtyping
- Liskov's substitution principle
- Multiple Inheritance
- Mixin Inheritance and Scala Traits
- Summary

SUBTYPING

$S \leq B$ ... relation meaning S is subtype or equal to B

Note: **smaller-equal** relation !

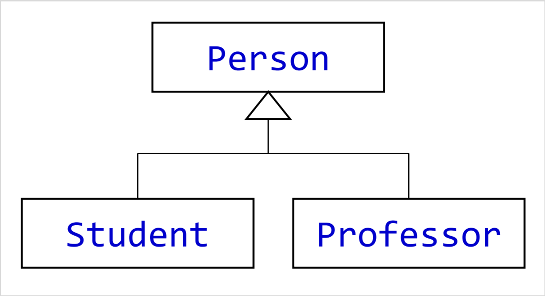
- **Subtyping** as **subset relations**

$S \leq B \Leftrightarrow set_S \subseteq set_B$ ... with set_T is set of all elements of type T

- **Type compatibility defined by subset relation**

$S \leq B \Rightarrow S \; is \; assignment \; compatible \; with \; B$

```
var x: Person = Student("Jim", 23)
```



Subtype Relation

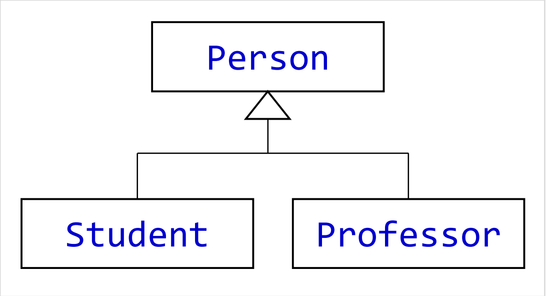
Subtype relations are defined by

nominal typing: explicitly defining subtype relations between data types
(e.g. by extends/implements relations of classes/interfaces)

```
class Person { ... }
```

```
class Student extends Person { ... }
```

```
class Professor extends Person { ... }
```



structural typing: structure ➔ type is a subtype of a supertype
if it has **at least the members** of the supertype

```
class Person {
    name: string;
    age: number;
}
```

```
class Student {
    name: string;
    age: number;
    study: string
}
```

```
class Professor {
    name: string;
    age: number;
    salary: number
}
```

```
let student: Student = {name: "Jim", age: 26, study: "CS"}
let professor: Professor = {name: "Bill", age: 62, salary: 100_000}
```

```
let person: Person
person = student
person = professor
```

both have **name** and **age**
➔ subtypes of **Person**

Student and **Professor**
compatible with **Person** !

JKU

Type Lattice

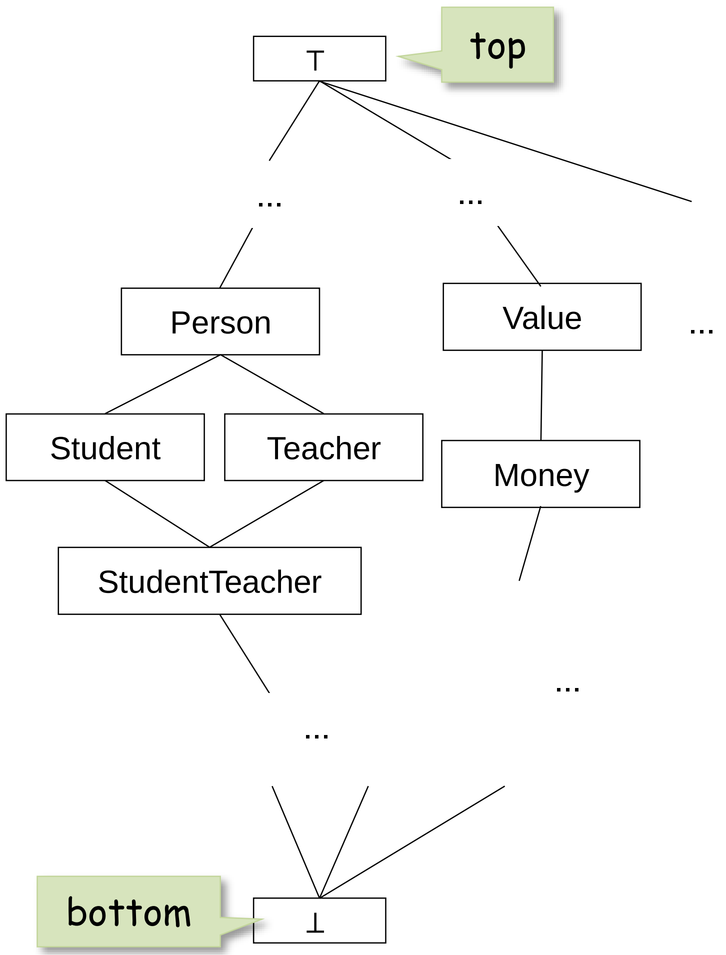
should

Types with subtype relation $\leq$ form a bounded lattice

$$TLattice = \langle Types, \leq, \sqcup, \sqcap, \top, \bot \rangle$$

with

- $\leq$ is a reflexive, antisymmetric, transitive **partial ordering relation** on ***Types***
- supremum $\boldsymbol{T_1} \sqcup \boldsymbol{T_2}$ is the **smallest common supertype** of T_1 and T_2
- infimum $\boldsymbol{T_1} \sqcap \boldsymbol{T_2}$ is the **largest common subtype** of T_1 and T_2
- $\top$ ist the **top element** with $\boldsymbol{S} \leq \top$ for all $S \in Types$
- $\bot$ is the **bottom element** with $\bot \leq \boldsymbol{S}$ for all $S \in Types$

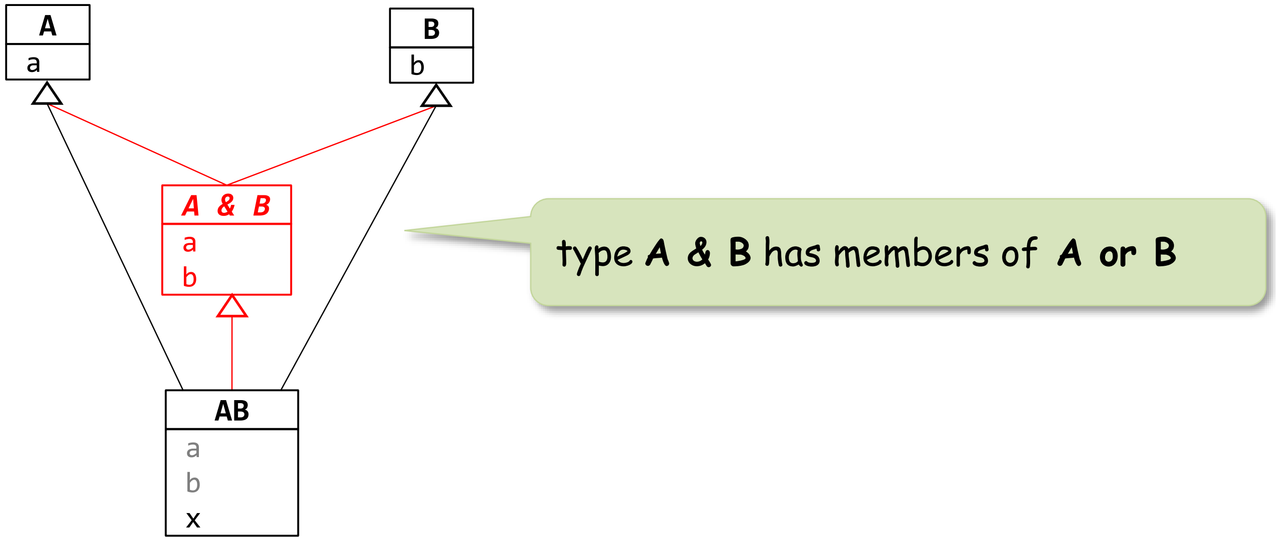


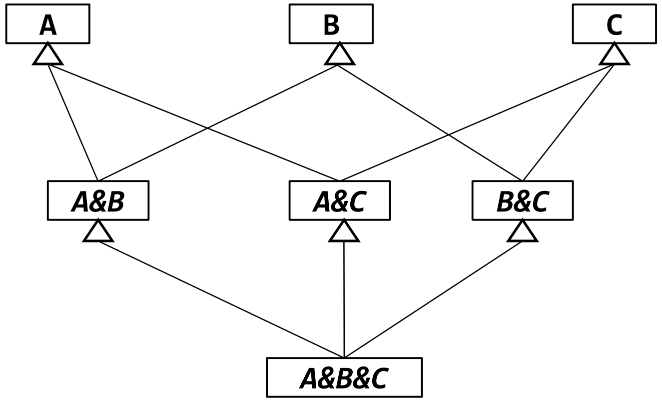
Type Intersection and Type Union

In type theory **supremum** and **infimum** are named **type union** and **type intersection**

Lattice		Type theory		
infimum	$T_1 \sqcap T_2$	type intersection	$T_1 \& T_2$	type which is subtype of T_1 and subtype of T_2 is type which has members present in T_1 **or** in T_2

Type intersection:



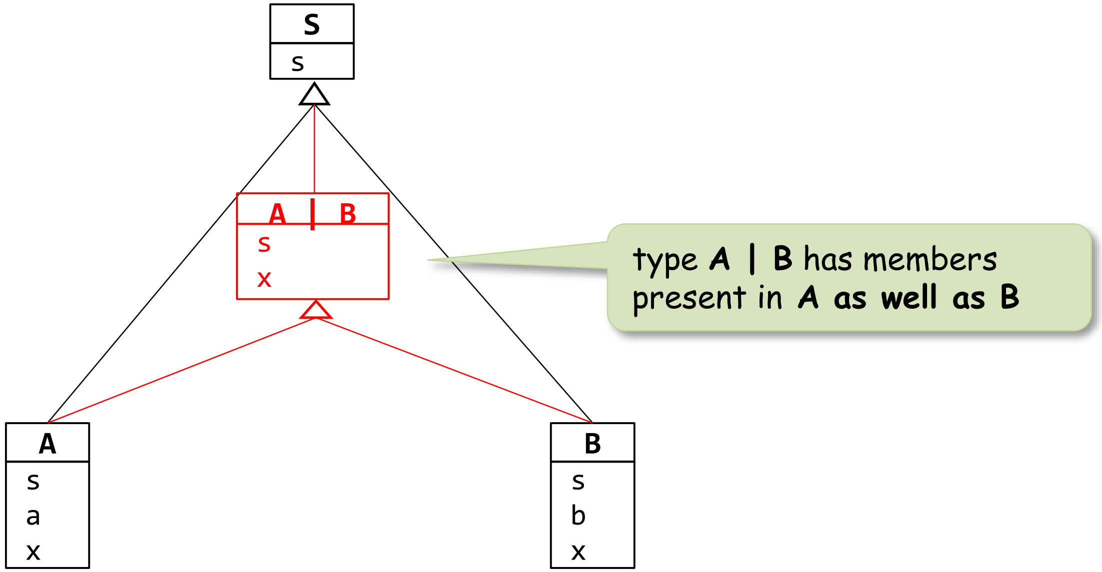


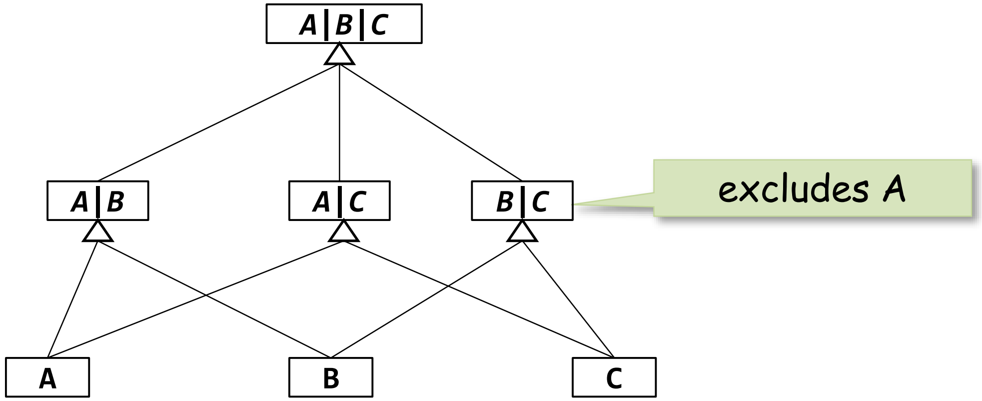
Type Intersection and Type Union

In type theory **supremum** and **infimum** are named **type union** and **type intersection**

Lattice		Type theory		
infimum	$T_1 \sqcap T_2$	type intersection	$T_1 \& T_2$	type which is subtype of T_1 and subtype of T_2 is type which has members present in T_1 **or** in T_2
supremum	$T_1 \sqcup T_2$	type union	$T_1 \mid T_2$	type which is supertype of T_1 and supertype of T_2 is type has members present in both T_1 **and** in T_2

Type union:





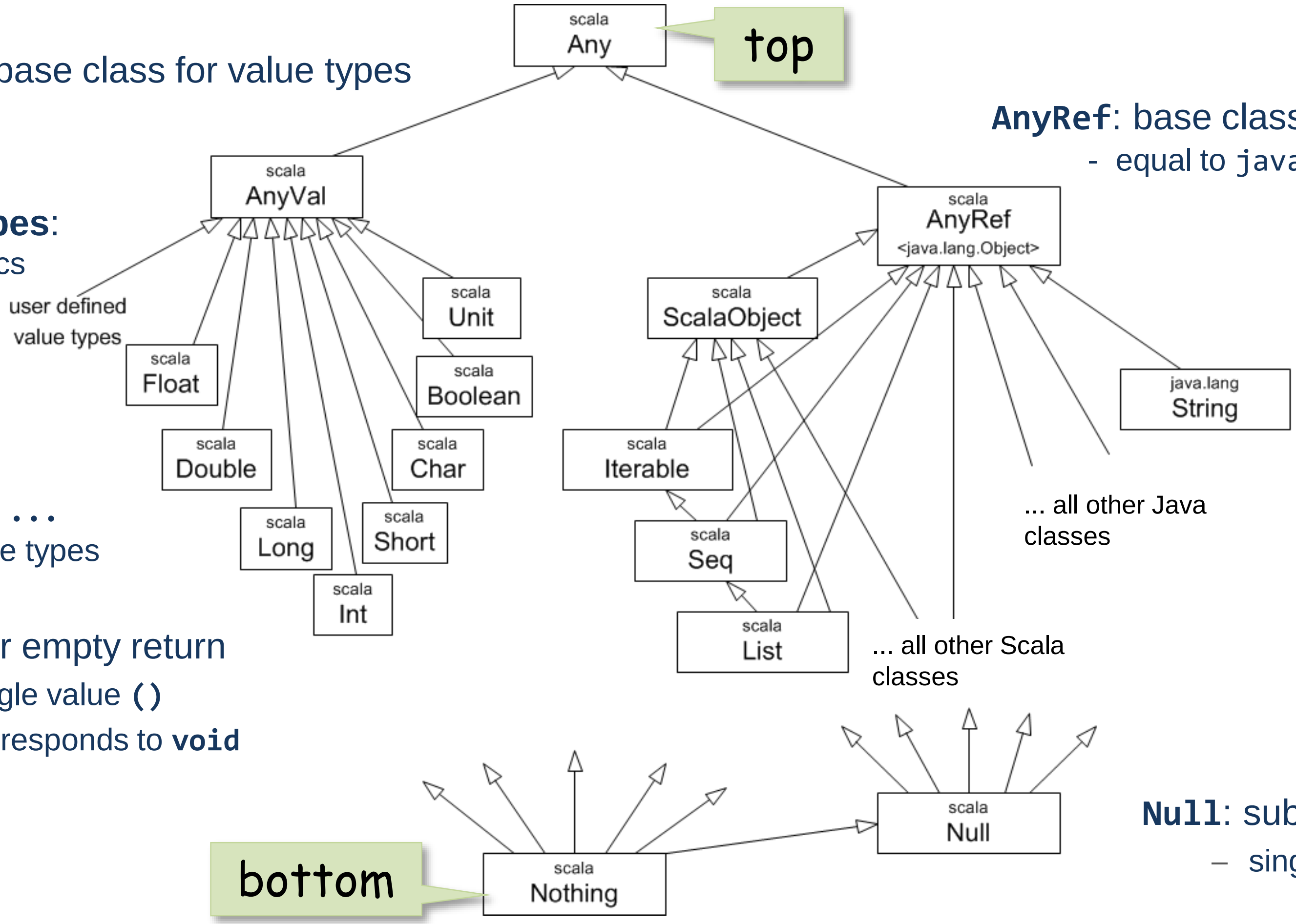
Type Lattice of Scala

`Any`: base class for all types

`AnyVal`: base class for value types

`AnyRef`: base class for reference types
- equal to `java.lang.Object`

user-defined value types:
classes with value semantics



`Int, Double, ...`
Java's built-in value types

`Unit`: for empty return
- single value **`()`**
- corresponds to **`void`**

`Null`: subclass of all reference types
- single value **`null`**

`Nothing`: subclass of all types
- **NO value!**

Type Null

Type **Null** has a single value **null**

Null is subtype of **all reference types**

- value **null** is compatible with all variables with reference type

```
var n : String = null
```

```
var p : Person = null
```

A value **null** has "all members all of reference types"

!?

- but throws an **NullPointerException**

```
p.toString
```

NullPointerException

Type Nothing

Nothing is subtype of **all types**

- type **Nothing** is **compatible with all types**
- does **not** have a **value**
- but important for **typing** and **type inference**

Example: Return type of methods with Exceptions

```
def error(message: String) : Nothing =
    throw new RuntimeException(message)
```

Result type **Nothing** because throws exception

```
def divide(x: Int, y: Int) : Int =
    if (y != 0) x / y
    else error("can't divide by zero!")
```

Result type **Int**

result of call to **error** has to be compatible with **Int** ➔ **Nothing** compatible with **Int**

Type Inference

Type inference based on supremum (type union) operation

```
val p = if (...) Student(...) else Professor(...)
```

➔ **p** is of type **Person** with **Student** ≤ **Person** and **Professor** ≤ **Person**

```
val x = if (...) 'y' else false
```

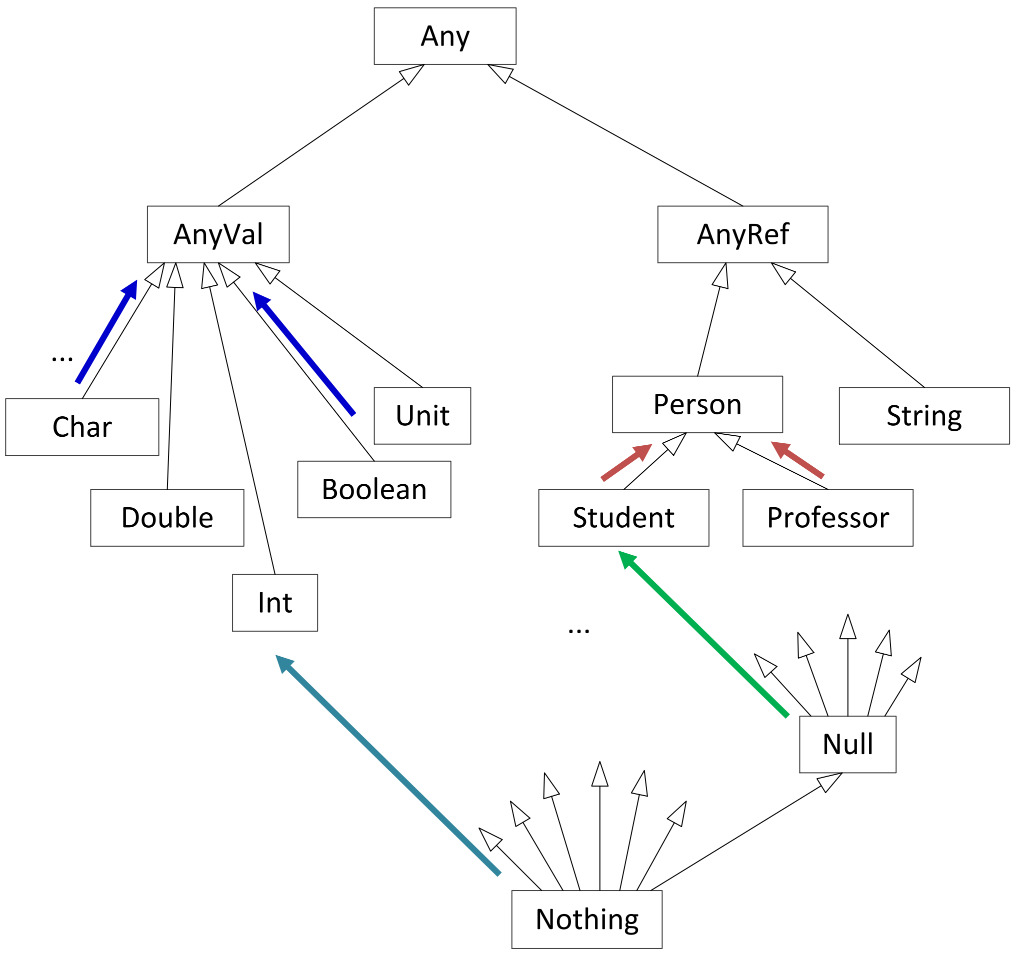
➔ **x** is of type **AnyVal** with **Char** ≤ **AnyVal** and **Boolean** ≤ **AnyVal**

```
val s = if (...) Student(...) else null
```

➔ **s** is of type **Student** because **Null** ≤ **Student**

```
val d = if (y != 0) x / y
        else throw Exception("divide by zero!")
```

➔ **d** is of type **Int** because **Nothing** ≤ **Int**



Intersection Types in Scala

Type declaration with several types

➔ guarantees that concrete object has members of all supertypes

```
val paula : Female & Professional = FemaleManager("Paula", ...)
```

guaranteed: **describe, name, work, sex = F** but not **manage**

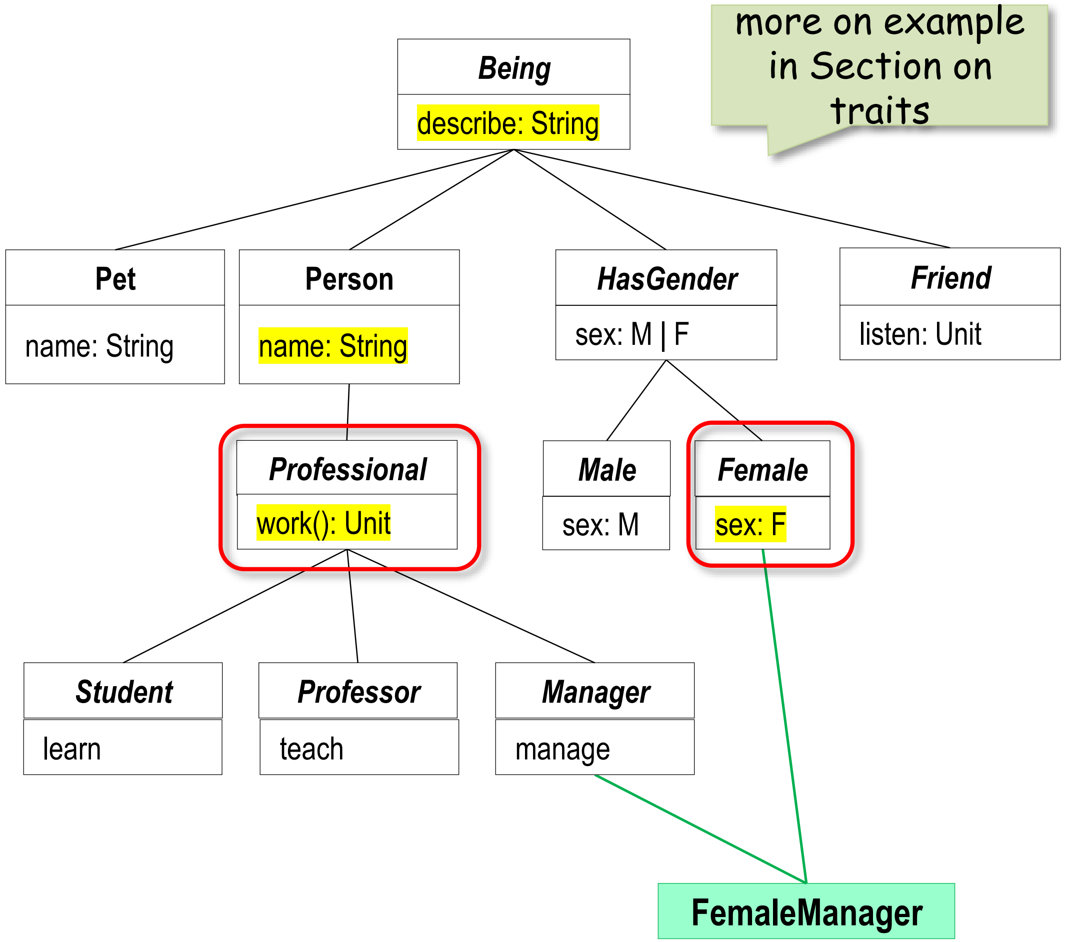
```
def writeMax[T <: Ordered[T] & Writeable](x: T, y : T): Unit = {
  if (x >= y) then x.write else y.write
}
```

guaranteed: **<, >, >=, <=, compare** and **write**

Java uses intersection types for type parameters

```
static <T extends  Comparable<T> & Writeable> void writeMax(T x, T y) {
    if (x.compareTo(y) >= 0) {
        x.write();
    } else {
        y.write();
    }
}
```

guaranteed: **compare** and **write**



Union Types in Scala

Declaration of elements having one of given types

➔ guarantees that concrete object has members of all supertypes

```
val personOrPet : Person | Pet = Dog("Lassy", ...)
```

```
val personOrPet : Person | Pet = FemaleManager("Paula", ...)
```

guaranteed: **describe** from superclass **Being** only

```
personOrPet.describe
```

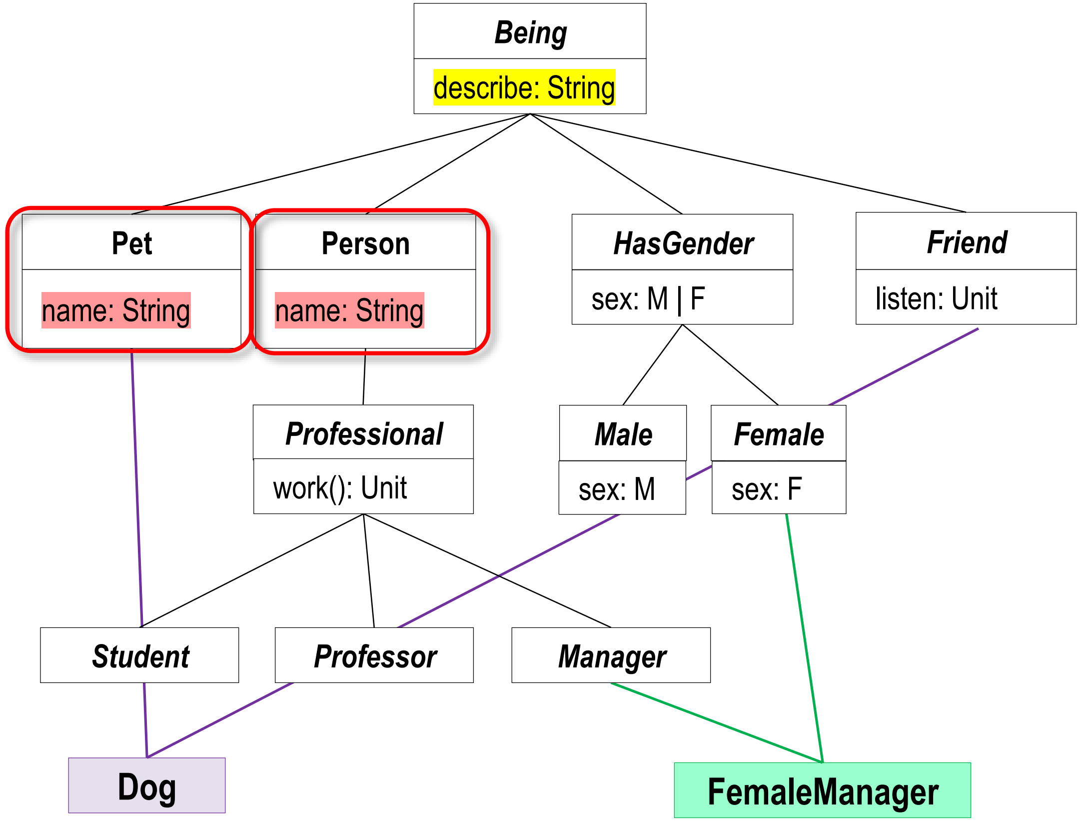
however: using features present in both but not in common superclass not supported

```
personOrPet.name  ✗
```

Would be a sort of structural typing!

but: can distinguish types in pattern match

```
personOrPet match {
  case person : Person => println("Go to bar with " +  person.name)
  case pet    : Pet    => println("Go to park with " + pet.name)
}
```



Union Types in Scala

Example

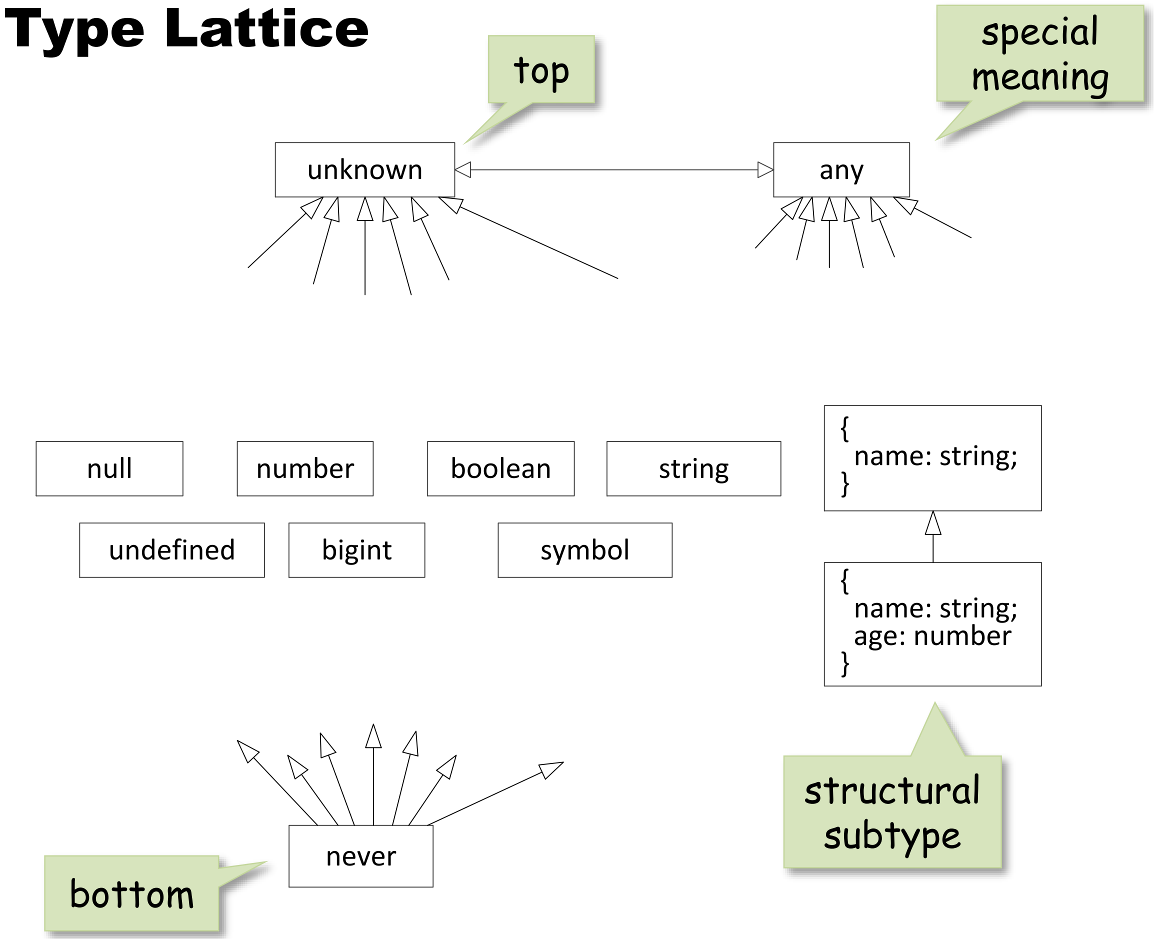
```
def errorMessage (msg : String | Int ) : String = {
  msg match {
    case code: Int    => "Error number: " + code
    case expl: String => "Error: " + expl
  }
}
```

```
println(errorMessage(1))
println(errorMessage("Fatal"))
```

An alternative to overloading!

TypeScript

Type Lattice



any:
- supertype of all types
- without type checks ➔ allows all operations

Type union

- in variable declarations

```
let personOrNull : Person | null
```

```
let studentOrProfessor : Student | Professor | undefined
```

- in type inference

```
let studentOrProfessor = (Math.random() > 0.5) ? student : professor
```

inferred type: **Student | Professor**

TypeScript: Structural Type Inference

Type inference based on structure

```
let studentOrProfessor = (Math.random() > 0.5) ? student : professor
```

inferred type: **Student | Professor**

Student | Professor equivalent to type with members present in both

```
{
   name: string;
   age: number;
}
```

which is structurally equivalent to class **Person**

```
class Person {
   name: string;
   age: number;
}
```

```
class Person {
   name: string;
   age: number;
}
```

```
class Student {
   name: string;
   age: number;
   study: string
}
```

```
class Professor {
   name: string;
   age: number;
   salary: number
}
```

TypeScript: Structural Type Inference

Type inference based on structure

```
let studentOrNull = (Math.random() > 0.5) ? student : null
```

inferred type: **Student | null**

Cannot access members present in **Student**

```
console.log(studentOrNull.name)
```

✗

Error: **name** not member of **null**

Guaranteeing special type within scope by ***narrowing***

```
if (studentOrNull != null) {
   console.log(studentOrNull.age)
} else {
   console.log("is null")
}
```

narrowing type in then-branch by if condition:
studentOrNull is type **Student**

```
class Person {
   name: string;
   age: number;
}
```

```
class Student {
   name: string;
   age: number;
   study: string
}
```

```
class Professor {
   name: string;
   age: number;
   salary: number
}
```

Structural Typing in Scala

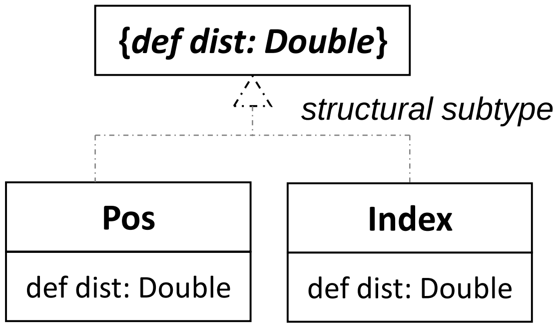
But rarely used

Scala allows **structural typing** as an alternative to nominal typing

Classes **Pos** and **Index** both with method **dist**

```
case class Pos(x: Double, y: Double) {
  def dist: Double = Math.sqrt(x*x + y*y)
}
```

```
case class Index(i: Int, j: Int) {
  def dist: Double = i + j
}
```



Method **printDist** with structural type requiring method **dist**

```
import
reflect.Selectable.reflectiveSelectable

def printDist(d : { def dist: Double }) = {
  println (d.dist)
}
```

import required

<u>Structural type</u>: Any object which implements function **dist()**

Calling printDist with **Pos** and **Index** objects

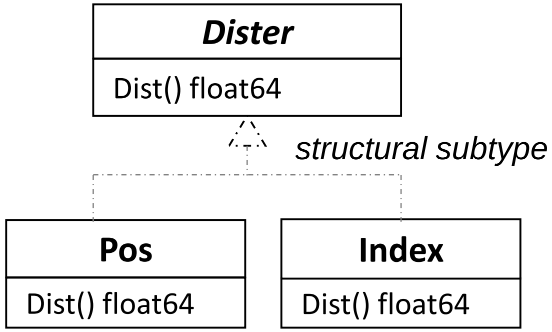
```
printDist(Pos(2, 3))
printDist(Index(2, 3))
```

Structural Typing in Go

Go is Google's system programming language
www.go-lang.org

Type system of **Go** is based on

- **interfaces**
- **structural typing:** check if object types have declared interface functions



Interface **Dister**
with abstract **Dist** function

```
type Dister interface {
    Dist() float64
}
```

Object types
Pos and **Index**

```
type Pos struct {
  X, Y float64
}
```

```
type Index struct {
  I, J int
}
```

No relation to interface **Dister**

Functions **Dist** for object types **Pos** and **Index**

```
func (p Pos) Dist() float64 {
  return math.Sqrt(p.X * p.X + p.Y * p.Y
}
```

```
func (i Index) Dist() float64 {
  return I + J
}
```

Function **PrintDist** with parameter of interface type **Dister**:

```
func PrintDist(d Dister) {
    fmt.Println(d.Dist())
}
```

Function call with **Pos** and **Index** objects

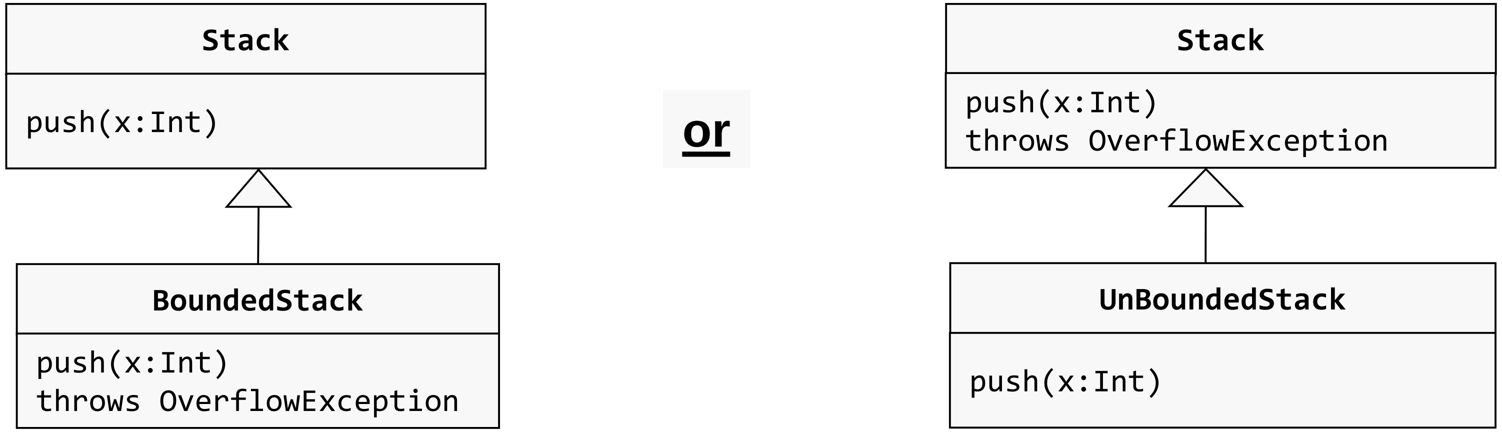
```
PrintDist( Pos(3, 4) )
PrintDist( Index(3, 4) )
```

III.1 Types, Subtypes and Inheritance

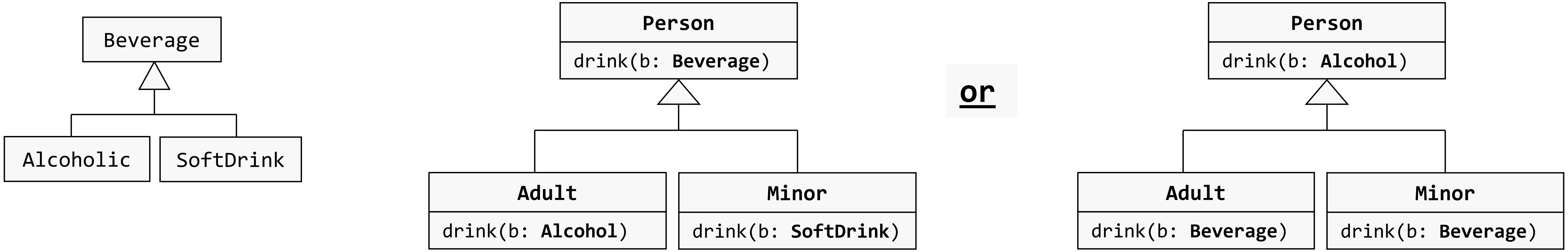
- Introduction
- Subtyping
- Liskov's substitution principle
- Multiple Inheritance
- Mixin Inheritance and Scala Traits
- Summary

Motivation

What is type-safe?



What is type-safe?

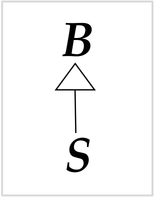


Liskov's Substitution Principle

B. Liskov and J.M. Wing: **A Behavioral Notion of Subtyping**. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 6, 1994.

Liskov's Substitution Principle defines **necessary conditions** so that a **type *S* is a valid subtype of a type *B***

w.r.t. type safety



A **type** S **is subtype of a type** B
if an **object of type** S **can be used** wherever **an object of type** B **can be used.**

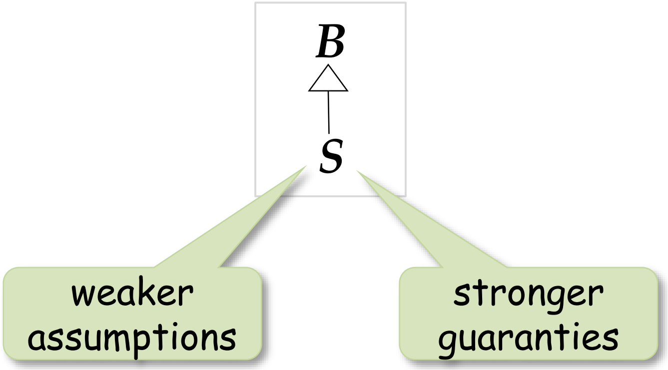
```
b : B = new S()
```

location for type B

allows subtype S

This can be guaranteed when

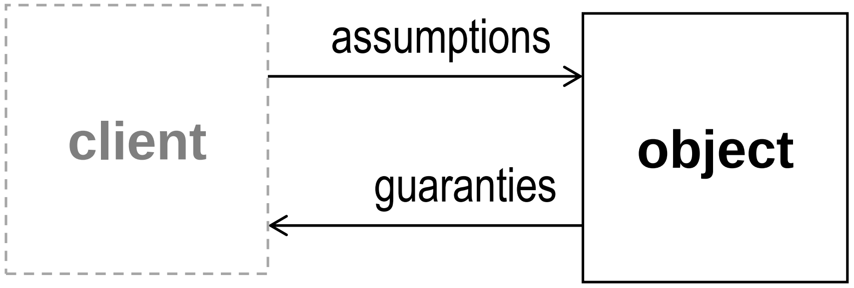
- all **assumptions** of S are **weaker** than those of B
- and all **guaranties** of S are **stronger** than those of B



WEAKER ASSUMPTIONS – STRONGER GUARANTIES

Type declaration represents a contract

- **assumptions** for **application** by client
- **guaranties** for the **results** returned to client

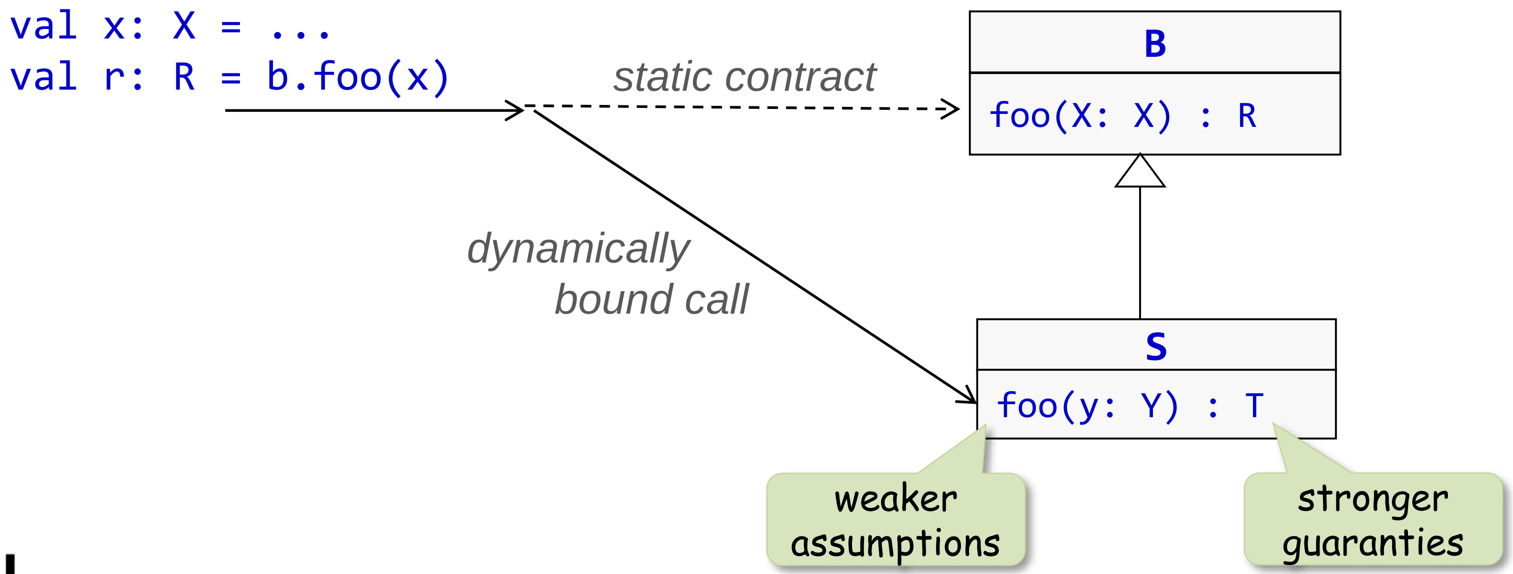


Static variable type **defines contract**

```
b : B
```

Concrete object must **fulfill contract**

```
b = new S()
```



assumptions of B ⇒ assumptions of S

- **stronger assumptions** of ***B*** guarantee that **assumptions of *S*** *are* ***fulfilled***

guaranties of S ⇒ guaranties of B

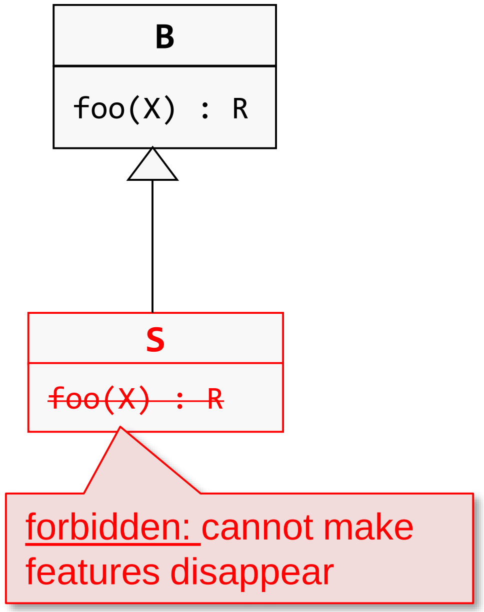
- **stronger guaranties** of ***S*** guarantee that **result of *S*** fulfill **guaranties of *B***

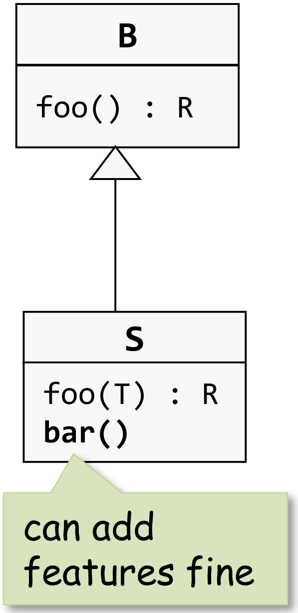
No Features can be Deleted in Subclass

Features *B* $\subseteq$ **Features *S***

- **No features** from ***B*** can be **deleted** in ***S***
- ***S*** can **add additional features**

➔ **Assumptions of subclass** are **weaker**





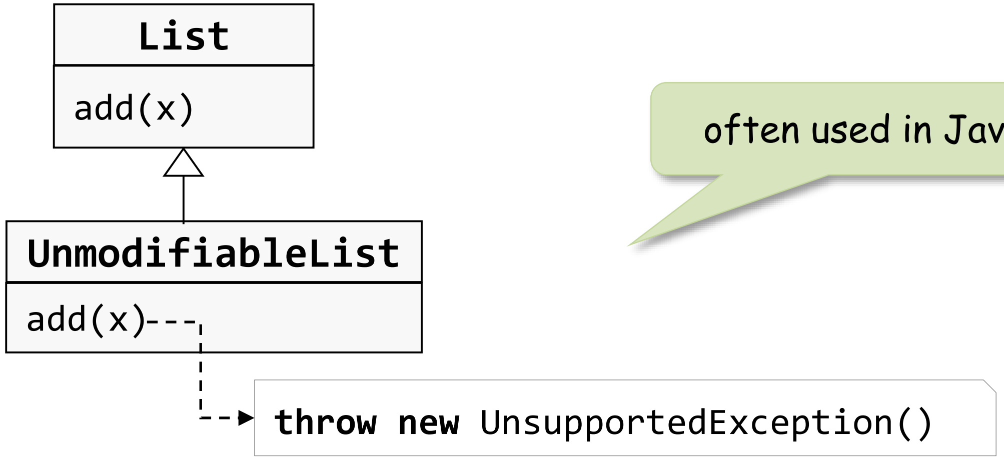
No Features can be Deleted in Subclass

Features *B* $\subseteq$ Features *S*

- **No features** from ***B*** can be **deleted** in ***S***
- ***S*** can **add additional features**

➔ **Assumptions of subclass** are **weaker**

Workaround: throw unchecked exception in subclass

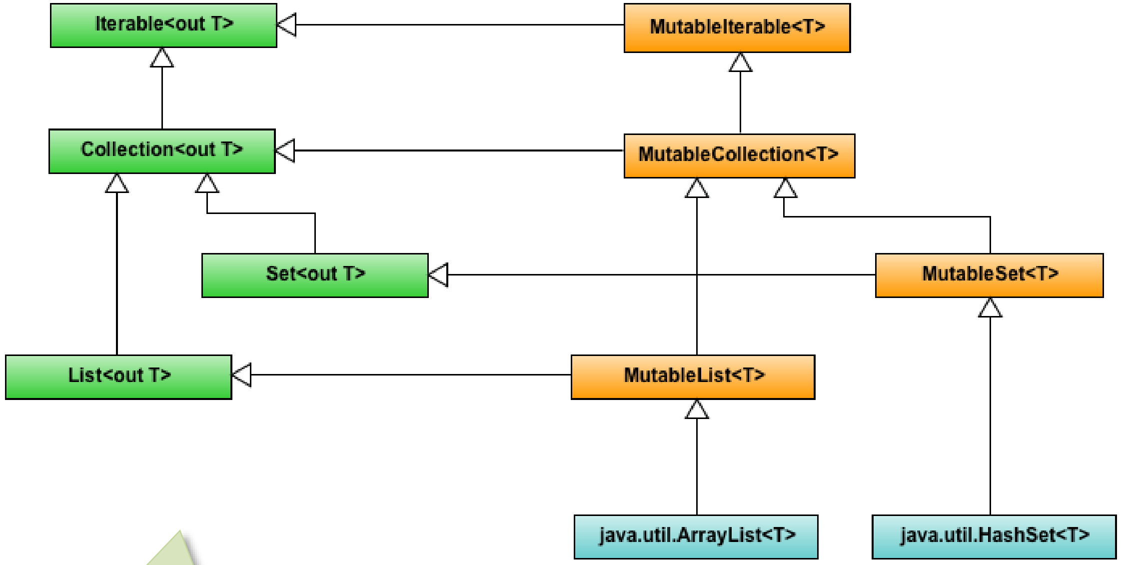


often used in Java

Approach rather questionable

- **base contract promises** to provide **add**!
- **subclass refuses** to implement **add** properly!?

Kotlin's approach: interfaces for read-only and mutable collections



only read access operations

also changing operations

Access Modifiers can be Relaxed

Access modifiers can be more relaxed in subtypes

➔ more is visible in subclass

➔ **Assumptions** are **weaker**

For example in Java

package visibility

```
class A {

  void foo() = {...}
}
```

protected visibility

```
class B extends A {
  @Override
  protected void foo() = {...}
}
```

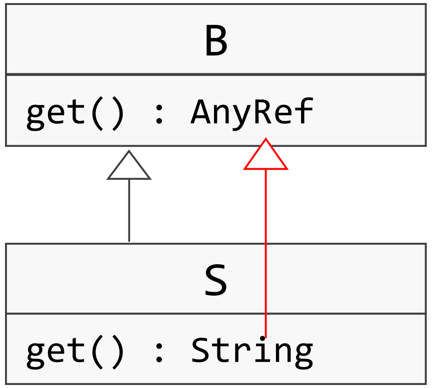
public visibility

```
class C extends B {
  @Override
  public void foo() = {...}
}
```

Variance of Inputs and Outputs

Co-variance of output parameters and return values (outputs can be more special in subclass)

➔ **Guaranties** of **outputs** are **stronger**



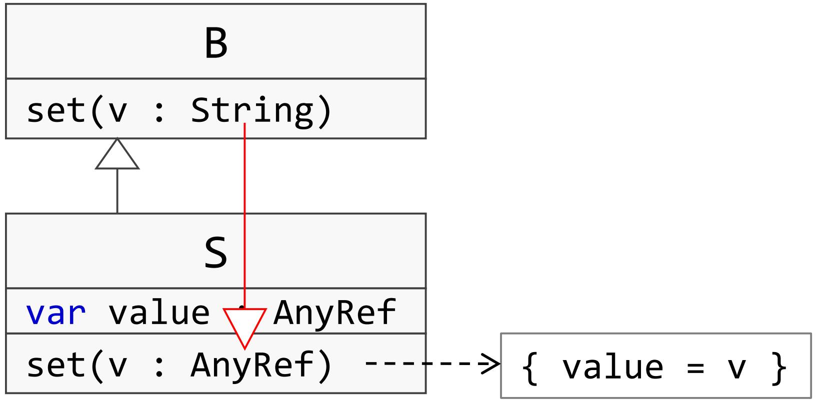

```
var b : B
b = new S()

val x : AnyRef = b.get()
```

B only guarantees **AnyRef** as result

Concrete object of subtype **S** returns **String** compatible with **AnyRef**

Contra-variance of input parameters (inputs can be more general in subclass)

➔ **Assumptions** of **inputs** are **weaker**



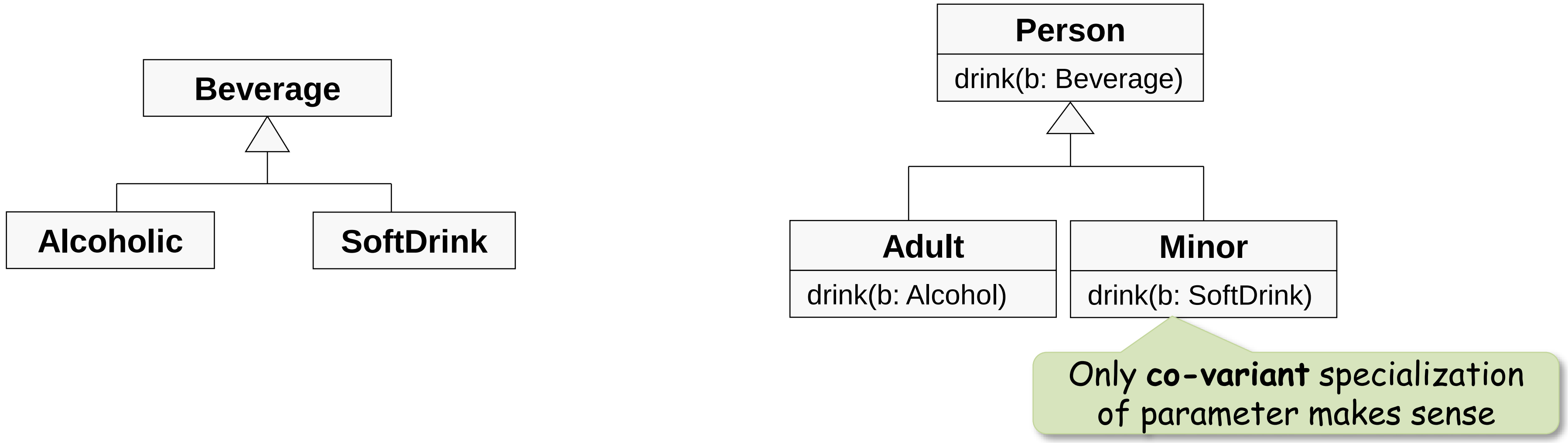

```
var b : B
b = new S()

b.set("String")
```

Type-safe because **B** only allows **Strings** but **S** accepts **AnyRef**

Co-Variance vs. Contra-Variance of Inputs

- **Contra-variance** of inputs is **type-safe**
- however, is sometimes **counter-intuitive**

Example: **Co-variant** parameter in method **drink** in classes **Adult** and **Minor**



➔ For that reason, most languages including Java, Scala and C#
do not allow contra-variant but only **invariant** input parameters

➔ **Eiffel** is a language with **co-variant parameters** and thus is **not type-safe**

Eiffel

https://www.eiffel.com https://www.eiffel.org/

Eiffel is an object-oriented language developed in the mid 1980s by Bertrand Meyer

Bertrand Meyer

- **Design-by-contract**
  - **pre-conditions** and **post-conditions** for methods and **class invariants**
- **Uniform-access-principle**
  - Function with **no argument** same as **field access operations**
  - **Fields** are **dynamically bound** and can be **overridden**

same in Scala

- **Command-query separation**
  - either **function returning value**
  - or **command changing state**

Co-Variant Parameters in Eiffel

Example: MINOR drinking ALCOHOL

```
class BEVERAGE ... end
class SOFT_DRINK inherit BEVERAGE ... end
class ALCOHOL    inherit BEVERAGE ... end

class PERSON
feature
     beverage: BEVERAGE
     drink(b: BEVERAGE) do
          beverage := b
     end
end

class MINOR inherit PERSON
redefine drink end
feature
     drink(soft: SOFT_DRINK) do
          beverage := soft
     end
end

-- main program --
little_willy: MINOR
beer: ALCOHOL
c: PERSON
c := little_willy

c.drink(beer)
```

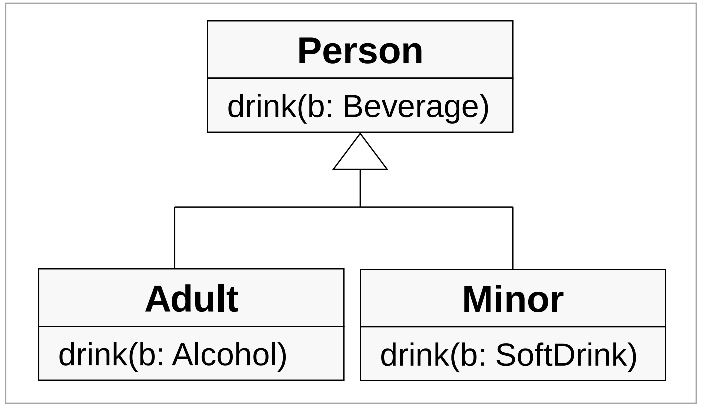
method drink

redefine ~ override annotation

co-variant override

polymorphic assignment

type error: little_willy is not allowed to drink beer!!



➔ Methods calls in Eiffel are **not type-safe**

TypeScript: Bivariance of Parameters

Parameters in TypeScript are **bivariant**

➔ can be more general or more special

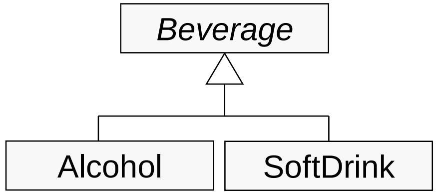
```
class Beverage {...}
class Alcohol extends Beverage {...}
class SoftDrink extends Beverage {...}
```

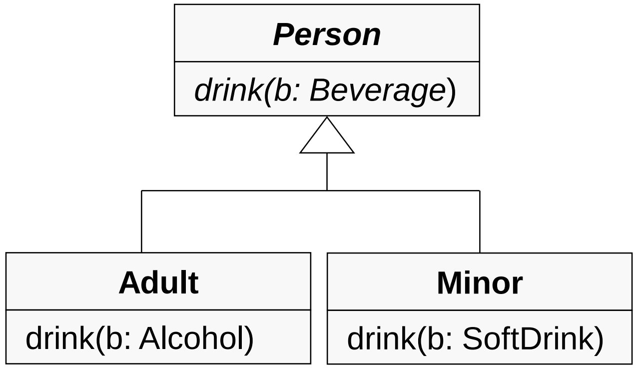
```
abstract class Person {
  abstract drink(b : Beverage) : void
}
class Adult extends Person {
    drink(b : Alcohol) : void {
        console.log("Drinking " + b)
    }
}
class Minor extends Person {
    drink(b : SoftDrink) : void {
        console.log("Drinking " + b)
    }
}
```

parameter more special
➔ **not type-safe**

```
let person : Person = new Minor()
person.drink(new Alcohol("beer", 5))
```

Drinker drink allows **Beverage**
➔ **Minor** can drink beer





TypeScript: Bivariance of Parameters

Parameters in TypeScript are **bivariant**
➔ can be more general or more special

```
interface Checker {
    check(x : string | number): boolean
}
```

```
class NullChecker implements Checker {
    check(x: string | number | null): boolean {
        return x == null ? true : ....
    }
}
```

parameter more general ➔ **type-safe**

```
class StringChecker implements Checker {
    check(x: string): boolean {
        return x.length > 0
    }
}
```

parameter more special ➔ **erroneous**

```
let checker : Checker = new StringChecker()
checker.check(3)
```

Error: **Checker** allows **number** but **StringChecker** only works for have **string** !!

Co-Variance of Exceptions

Exception thrown by method of **subtype *S*** must be **a subset of the exceptions** thrown by method of **supertype *B***

➔ **Guaranties** are **stronger**

```java
public class B {

  void foo(...) throws Exception {
    ...
  }
}
```

```java
public class S extends B {

  @Override
  void foo(...) throws IOException  {
    ...
  }
}
```

```java
B b = new S();

try {
  b.foo(...);
  ...
} catch (Exception e) {
   ...
}
```

must care for **Exception** with **IOException** possibly thrown by **S** being compatible

```java
class B {

  void foo(..) throws Exception {
    ...
  }
}
```

```java
class S extends B {

  @Override
  void foo(..)    {
    ...
  }
}
```

Subclass can decide to throw **no exception**

Co-Variance of Exceptions

Example: OverflowException in Stack classes

```
class StackException extends Exception {...}
class OverflowException extends StackException {...}
class UnderflowException extends StackException {...}
```

```
public class Stack {
  int limit = 16;

  void push(int x) throws OverflowException {
    ...
  }
  ...
}
```

```
public class UnboundedStack extends Stack {

  @Override
  void push(int x)  {
    ...
  }
  ...
}
```

UnboundedStack decides not to throw ***OverflowException***

```
Stack stack = new UnboundedStack();

try {
  stack.push(1);
  ...
} catch (OverflowException ofe) {
   System.out.println(ofe)
}
```

must handle ***OverflowException*** but ***UnboundedStack*** will never throw one

Co-variant Overrides of Members in Scala

Overrides between variables and methods without parameters

- a **mutable field** can override a **method**
- an **immutable field** can override a **mutable field**
- an **immutable field** can override a **method**

<u>Recall</u>: also fields are dynamically bound in Scala

Specialization hierarchy of member types

- **methods** with **no parameter** ➔ **return value** dependent on **object state**
- **getter** of mutable **var** fields ➔ **return value** dependent on **mutable variable**
- immutable **val** fields ➔ **return value** is **constant**

more special

➔ **Guarantees** are **stronger**

Co-variant Overrides of Members in Scala

Example: Method **isFull** in **Stack** overridden by **val** in **UnboundedStack**

```scala
class Stack(limit : Int) {
  protected var elems : List[Int] = List()
  ...
  def isFull : Boolean = elems.length == limit
}
```

Stack:
isFull is computed from **elems.length**

```scala
class UnboundedStack extends Stack(Int.MaxValue) {
  ...
  override val isFull : Boolean = false
}
```

UnboundedStack:
isFull is constant **false**

Co-variant Overrides of Members in Scala

Example: co-variant overrides of **width** and **height** in **Shape** classes

```
abstract class Shape {
  abstract def width : Int
  abstract def height : Int
  ...
}
```

in **Shape** width and height are **abstract methods** !

```
class Group extends Shape {
  var elems : List[Shape] = List()
  override def width : Int = computeWidth(elems)
  override def height : Int = computeHeight(elems)
  ...
}
```

in **Group** width and height are **computed** from mutable elements!

```
class Rect(w : Int, h : Int) extends Shape {
  override var width = w
  override var height = h
  ...
}
```

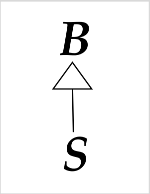
in **Rect** width and height are defined by **mutable fields** !

```
class Circle(r : Int) extends Shape {
  override val width = 2 * r * PI
  override val height = 2 * r * PI
  ...
}
```

in **Circle** width and height are **constant vals**!

JKU

Variance of Assertions



Preconditions have to be weaker

- Preconditions of ***B*** must imply precondition of ***S***

$$\boldsymbol{pre_B \Rightarrow pre_S}$$

Postconditions have to be stronger

- Postconditions of ***S*** must imply postconditions of ***B***

$$\boldsymbol{post_S \Rightarrow post_B}$$

Invariants have to be stronger

- Invariants of ***S*** must imply invariants of ***B***

$$\boldsymbol{inv_S \Rightarrow inv_B}$$

Variance of Assertions

Example: Postcondition

```
class Collection {
  ...
  def add(x : Int) = {

  }

  ...
}
```

Postcondition: `this.contains(x)`

```
class Stack extends Collection {
  ...
  def add(x : Int) = {
    push(x)
  }

  def push(x : Int) = {
    ...
  }

}
```

Postcondition: `this.top == x`

$post_S \Rightarrow post_B$: `this.top == x` $\Rightarrow$ `this.contains(x)`

Eiffel: Design-By-Contract

Eiffel supports **preconditions, postconditions of methods** and **class invariants**

```
class Collection
feature
  count: integer;
  capacity: integer is 100;
  items : array[string]

  add (x: string) is
    require
      count < capacity
    do
      --some add operation
    ensure
      contains(x)
      count = old count + 1
    end   -- add
  contains(x: string) is ...
  ...
invariant
  0 <= count
  count <= capacity
end -- Collection
```

Precondition
- Condition on parameter and fields
- must be assured by caller

Postcondition
- Condition of parameters and fields (incl. **old** field values)
- must be guaranteed by method

Class invariants
- Condition on field values
- must hold between method calls

Eiffel: Design-By-Contract

```
class Collection
feature
  ...
  add (x: integer) is
    require
      count <= capacity
    do
      --some add operation
    ensure
      contains (x)
      count = old count + 1
    end  -- add
invariant
  0 <= count
  count <= capacity
end -- Collection
```

```
class Stack inherit Collection
redefine add end
feature
  top : integer
  add (x: integer) is
    do
      push(x)
  ensure
     item[top] = x
  end; -- add
  ...
end; -- Stack
```

`item[top] = x` ⇒ `contains(x)`

Eiffel: Inheritance of Assertions

```
class B
feature
  m(...) is
  require preB
  do ...
  ensure postB
  end;
end;
```

```
class S inherit B
redefine M end
feature
  m(...) is
  require else preS
  do ...
  ensure then postS
  end;
end;
```

Logical combination of assertions from **superclass** and **subclass with**

require else : Preconditions combined by **or** (||)

➔ **Preconditions** get **weaker**

ensure then: Postconditions combined by **and** (&&)

➔ **Postcondition** get **stronger**

pre_B

B.m()

$post_B$

$pre_B \;||\; pre_S$

S.m()

$post_B \;\&\&\; post_S$

$pre_B \Rightarrow pre_B \;||\; pre_S$

Precondition of ***B*** implies preconditon of ***S***

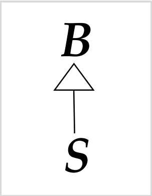
$post_B \;\&\&\; post_S \Rightarrow post_B$

Postcondition of ***S*** implies postconditon of ***B***

LISKOV'S SUBSTITUTION PRINCIPLE (FORMAL)

B. Liskov and J.M. Wing: A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No 6, 1994

Liskov's substitution principle defines necessary conditions for subtyping



Let $S, B \in Types$ and let m_B be method defined in B and method m_S in S overrides m_B
then S is a proper subtype of B, i.e., $S \leq B$, if the following conditions hold:

Overriding subtype methods $\boldsymbol{m_S}$ preserve supertype methods' $\boldsymbol{m_B}$ behavior:

1) **Contra-variance of arguments**: for all input parameters $p : TI_B$ of method m_B and corresponding input parameters $p_S : TI_S$ of method m_S it follows $TI_B \leq TI_S$

2) Co-variance of results: for all results and output parameters of type TO_B of methods m_B and corresponding results and output parameters of type TO_S of methods m_S it follows $TO_S \leq TO_B$

3) Exceptions: execptions $excpt_S$ thrown by method m_S must be a subset in exceptions $excpt_B$ thrown by method m_B

4) Preconditions: a precondition pre_B for method m_B must imply the precondition pre_S for method m_S

5) Postconditions: a postcondition $post_S$ for method m_S must imply the postcondition $post_B$ for method m_B

Subtype constraints ensure supertype constraints:

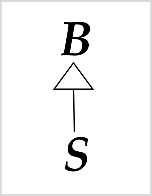
6) Preservation of invariants: Let Inv_B be an invariant defined for B and Inv_S be an invariant defined for S then Inv_S must imply Inv_B

7) History constraint $HistConstr_S$ for subtype S must imply history constraints $HistConstr_B$ of supertype B

history constraints are about state changes

LISKOV'S SUBSTITUTION PRINCIPLE (SHORT FORM)

$\boldsymbol{S \leq B}$:



Overriding subtype methods $\boldsymbol{m_S}$ preserve supertype methods $\boldsymbol{m_B}$ behavior:

1) **Contra-variance of input parameter types $\boldsymbol{TI_B}$** and $\boldsymbol{TI_S}$: $\boldsymbol{TI_B \leq TI_S}$

2) **Co-variance of output types $\boldsymbol{TO_B}$** and $\boldsymbol{TO_S}$: $\boldsymbol{TO_S \leq TO_B}$

3) **Co-variance of Exceptions $\boldsymbol{Excpt_B}$** and $\boldsymbol{Excpt_S}$: $\boldsymbol{Excpt_S \subseteq Except_B}$

4) **Contravariance of Preconditions $\boldsymbol{Pre_B}$** and $\boldsymbol{Pre_S}$: $\boldsymbol{Pre_B \Rightarrow Pre_S}$

5) **Co-variance of Postconditions $\boldsymbol{Post_B}$** and $\boldsymbol{Post_S}$**:** $\boldsymbol{Post_S \Rightarrow Post_B}$

Subtype constraints ensure supertype constraints:

6) **Preservation of invariants**: $\boldsymbol{Inv_S \Rightarrow Inv_B}$

7) **Subtype history constraints** ensure supertypes history constraints: $\boldsymbol{HistConstr_S \Rightarrow HistConstr_B}$

III.1 Types, Subtypes and Inheritance

- Introduction
- Subtyping
- Liskov's substitution principle
- Multiple Inheritance
- Mixin Inheritance and Scala Traits
- Summary

Multiple Inheritance

Class derived from more than one superclass

Motivation

- factor out common behavior
- improve code reuse

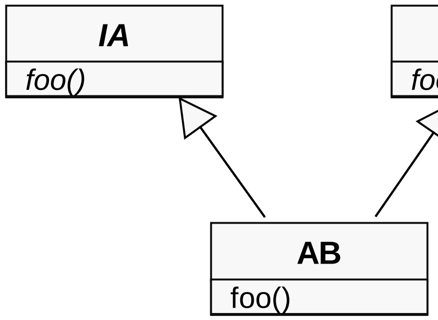
Approaches

- multiple class inheritance
- multiple interface inheritance
- mixin inheritance
- Scala traits

MULTIPLE INTERFACE VS. IMPLEMENTATION INHERITANCE

Interface inheritance

- abstract methods represent requirement that method has to be implemented



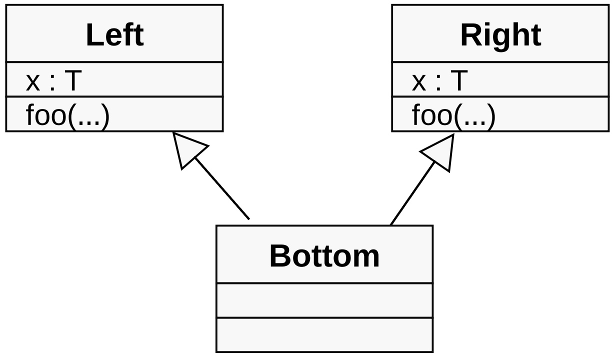
IA and *IB* require method *foo()*

class **AB** implements **foo()** and fullfills both required method **foo** from **IA** and **IB**

➔ **No conflicts**

Implementation inheritance

- inheritance of concrete method implementations and data fields

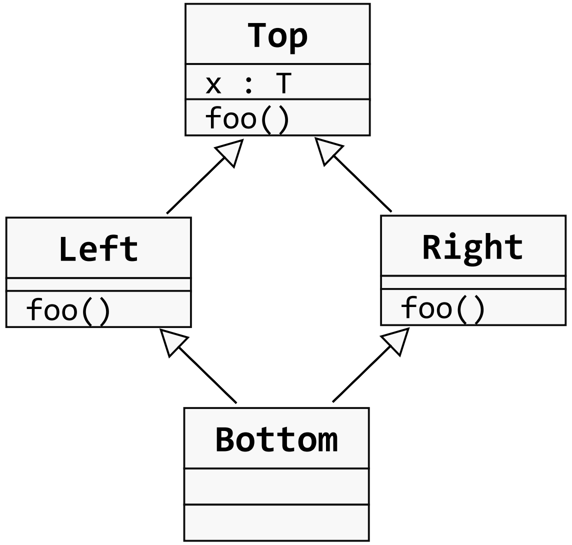


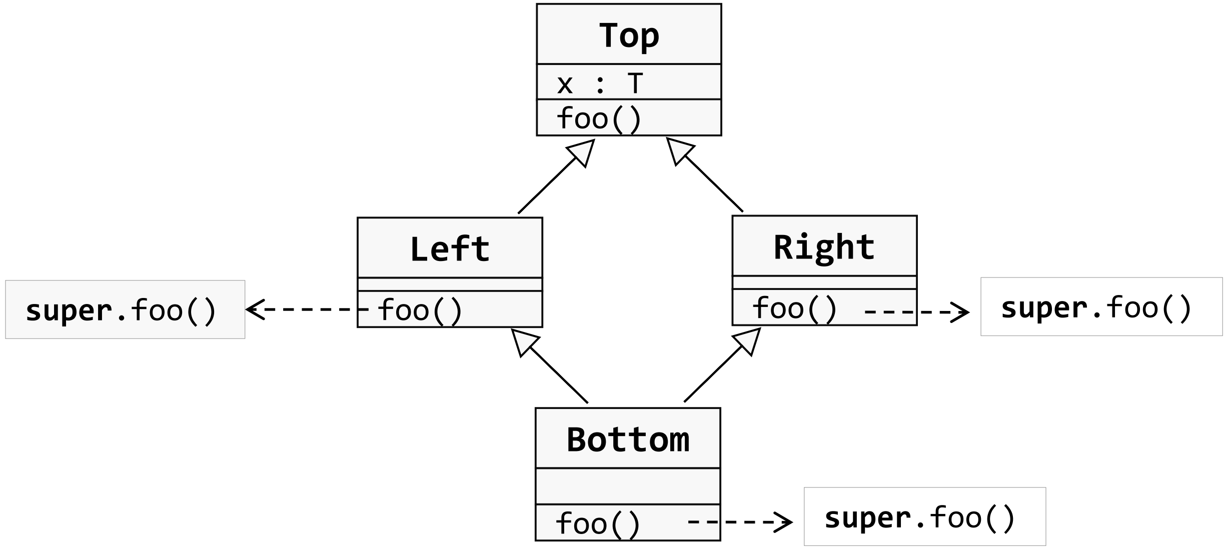
➔ **Conflicts:**

- Inheritance of instance variables? Once? Twice?
- Which method to inherit ?
- Order of super calls ?

Diamond Problem

Inherit from different paths





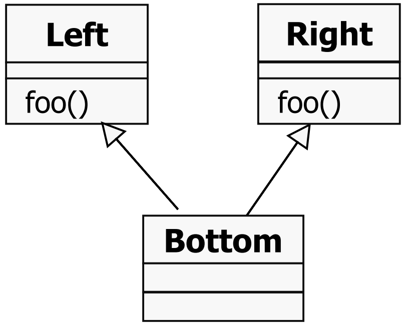

```
Bottom b = new Bottom();
b.x
b.foo()
```

Questions:

- Does Bottom have one x or two?
- Which method foo is inherited? from Left, Right, or both?
- Which method foo() is called?
- In which order are super calls resolved?

Conflict Resolution in C++

- C++ supports multiple implementation inheritance
- Conflict resolution by developer




```
Bottom* bottom = new Bottom();
```

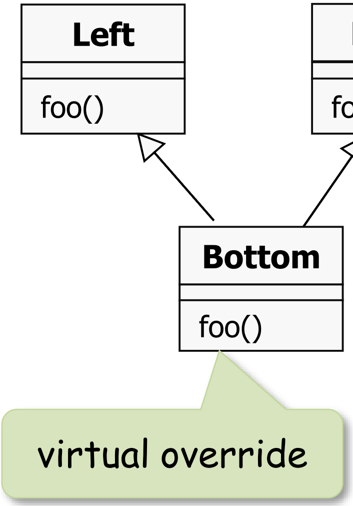
```
bottom.foo();
```
ambiguous ➔ forbidden

```
bottom.Left::foo();
```
Left's implementation

```
bottom.Right::foo();
```
Right's implementation

Conflict Resolution in C++

■ Overriding method **foo()** in **Bottom**




```
Bottom* bottom = new Bottom();
```

```
bottom.foo();
```

Bottom's implementation

```
bottom.Left::foo();
```

Left's implementation

```
bottom.Right::foo();
```

Right's implementation

But:

```
Left* left = bottom;
left.foo();
```

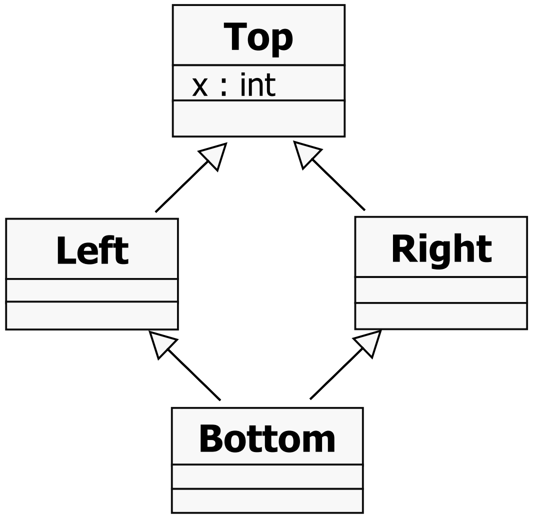
Bottom's implementation

```
Right* right = bottom;
right.foo();
```

Bottom's implementation

Diamond Problem in C++

Inheriting fields: Two versions



- virtual inheritance
  ➔ inherit once

```
class Top {
   int x;
};

class Left: virtual Top {...};

class Right: virtual Top {...};

class Bottom: Left, Right {...};
```

- normal inheritance
  ➔ inherit twice

```
class Top {
   int x;
};

class Left: Top {...};

class Right: Top {...};

class Bottom: Left, Right {...}
```

Memory layout:

Bottom	x

x only once

Bottom	Left	x	Right	x

x twice

INHERITANCE OF ABSTRACT MEMBERS

Multiple inheritance of abstract members does not represent conflict

- abstract declarations only state that **methods must be present**
- no inheritance but **subtyping**

Example: Old Java interfaces (without default methods)

```java
public interface ILeft {
  public void foo();
}
```

```java
public interface IRight {
  public void foo();
}
```

foo() required twice

```java
public class CBottom implements ILeft, IRight {

  public void foo() {
    ...
  }
}
```

foo() implemented once

Override-Equivalent Methods Signatures

Override-equivalent methods are

- methods where **one would override** the other
- override-equivalent methods **have to be implemented once**

```java
public interface ILeft {
  public Object get();
}
```

```java
public interface IRight {
  public String get();
}
```

override-equivalent methods get!

```java
public class CBottom implements ILeft, IRight {
   public String get() { ... }
}
```

must implement more specific

III.1 Types, Subtypes and Inheritance

Mixins

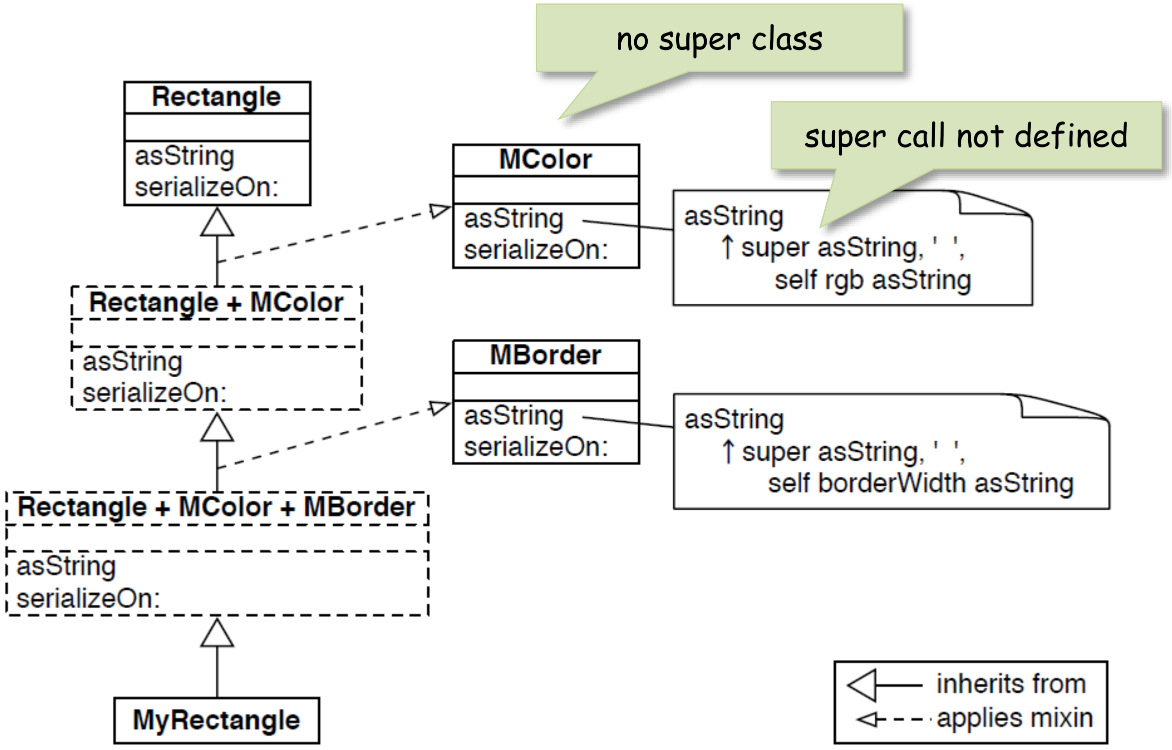
- **Mixin** = independent piece of behavior
  - for **extending** other classes
  - at **composition time**

time when mixin is mixed to other class

➔ **mixin class** has no **defined superclass**

➔ **super calls not determined** in **class definition**

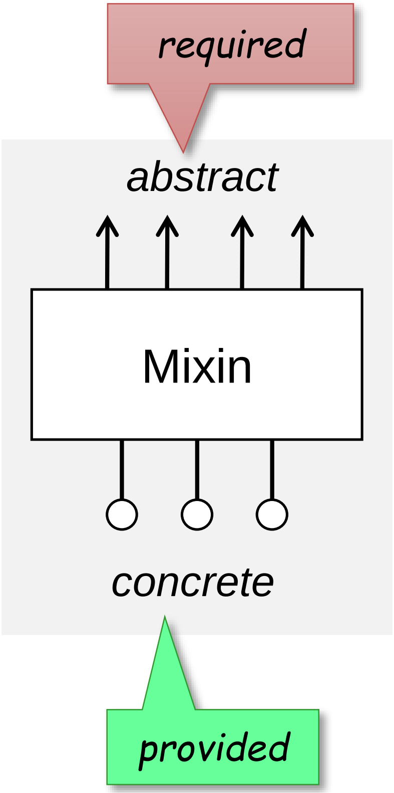
<u>Example</u>: Smalltalk with Mixins



MIXIN CLASSES

Mixins with concrete and abstract features

- abstract features ➔ ***required by mixin class***
- concrete features ➔ ***provided by mixin class***



"***When somebody provides my required features***
I can provide my concrete features"

Java Interfaces with Default Methods

Java interfaces with **default methods** are **mixins**

- **provided** (concrete) and **required** (abstract) methods
- **composition by inheritance**

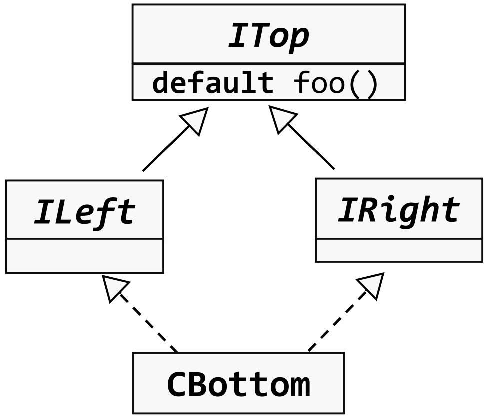
```
public interface Collection<E> extends Iterable<E> {
  …
  boolean remove(Object o);
  …
  default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
      if (filter.test(each.next())) {
        remove(each.next);
        removed = true;
      }
    }
    return removed;
  }

  ...
}
```

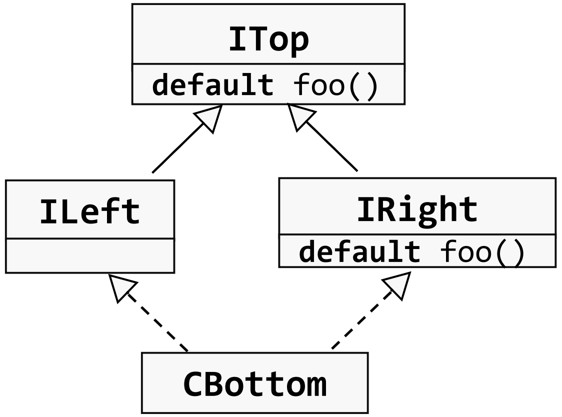
required

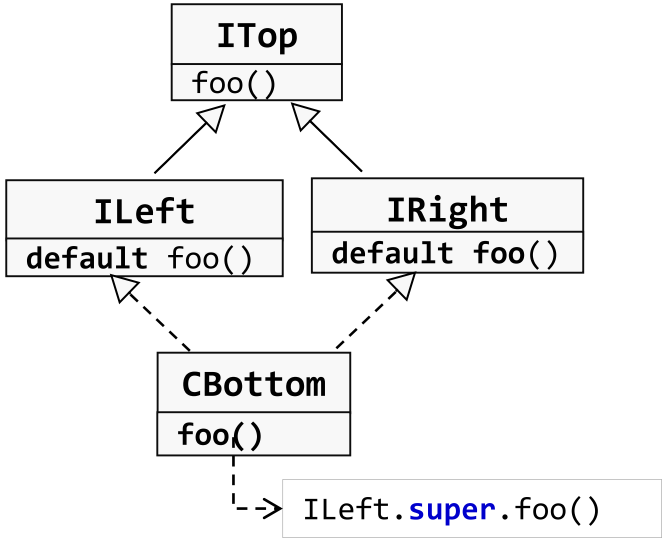
provided

call to abstract method **remove**!

Default Methods: Conflict Resolution








```java
CBottom c = new CBottom();
b.foo();
```

no conflict:
`ITop.foo()` called

no conflict:
`IRight.foo()` called

Conflict:
must be resolved by implementing method in class

```java
class CBottom implements ILeft, IRight {
  void foo() {
     ILeft.super.foo();
  }
}
```

Traits in Scala

Term **trait** not used uniformly
➔ compare traits in Rust

Traits in Scala are a sort of **mixins**

- Definition of **required features**
  - **abstract members:** must be implemented by concrete class
  - **deriving from classes**: concrete class must be derived from that class
  - **Self-type annotations:** requires itself to be of that type (➔ see later)
- Implementation of **concrete features**
  - **concrete methods** and **variables**

required class

```scala
trait SomeTrait extends SomeClass with SomeOtherTrait {

  => name : SomeType

  val abstractValField : Type

  def abstractMethod(...) : ReturnType

  val concreteValField = ...

  def concreteMethod(...) = {
    ... abstractMethod ... abstractValField ..
  }
}
```

Self-type annotation = requires itself to be of that type

abstract = required

concrete = provided

Prototypical Usage Scenarios of Traits

Rich interfaces

- Implementation of abstract trait with broad interface
- Class has to implement only some elementary operations
- broad interface mixed in

Orthogonal features

- Type system with traits providing orthogonal features
- Combination in class by mixing together different traits

Stackable modifications

- Auxiliary features implemented in traits
- Additions by traits
- Building chain of calls by super calls

not shown in the following!

Example Rich Interfaces: Ordered

- **Ordered** is trait extending **java.lang.Comparable**
  - with one abstract method **compare**
  - Implementation of **relational operators** based on **compare**

```
trait Ordered[A] extends Any with java.lang.Comparable[A] {
  def compare(that: A): Int
  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
}
```

single abstract

several concrete

```
case class Fract(nom: Int, denom: Int) extends Ordered[Fract] {
  ...
  def compare(that: Fract): Int = ...
}
```

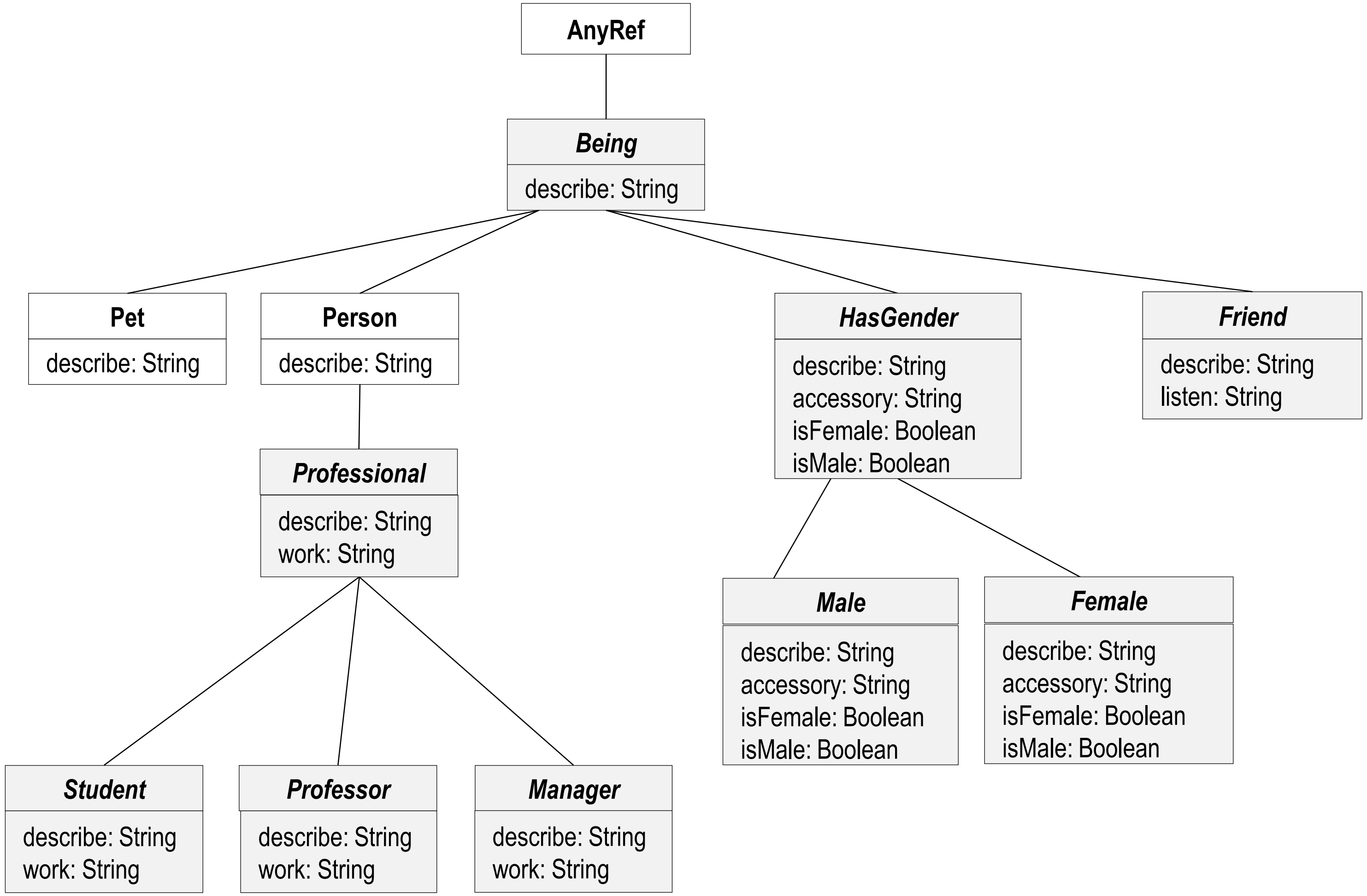
implement abstract method

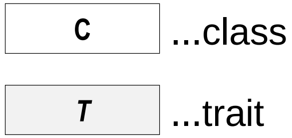
```
val f1 = Fract(4, 3)
val f2 = Fract(3, 2)

if (f1 < f2) "smaller"
else if (f1 > 0) "greater"
else "equal"
```

inherit concrete

Example Orthogonal Features: Beings

- Implementing orthogonal features by traits





Example Orthogonal Features: Beings

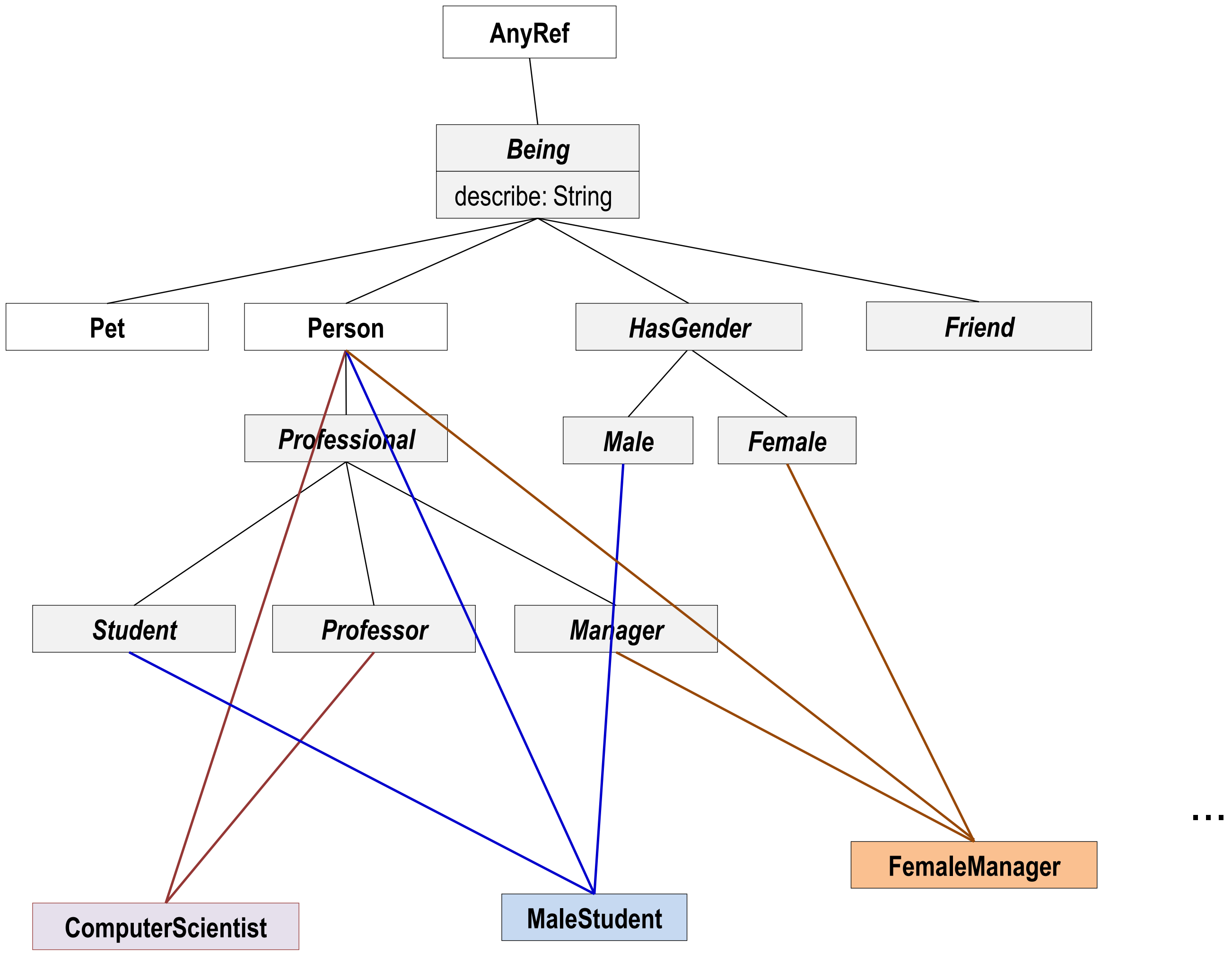
- Implementing orthogonal features by traits

```
trait Being :
  def describe: String

class Person(val name: String) extends Being :
  def describe = s"Person(name = $name)"

class Pet(val species: String, val name: String) extends Being :
  def describe = s"$species(name = $name)"

trait HasGender extends Being :
  val accessory: String
  def isFemale: Boolean
  def isMale: Boolean

trait Female extends HasGender :
  override val isFemale = true
  override val isMale = false
  abstract override def describe = super.describe + s"is female, has $accessory"

trait Male extends HasGender :
  override val isFemale = false
  override val isMale = true
  abstract override def describe = super.describe + s"is male, has $accessory"
```

```
trait Friend(val friendOf: Person) extends Being :
  abstract override def describe: String = super.describe + listen
  def listen: String = " listens to " + friendOf.name

trait Professional extends Person :
  def works: String

trait Professor(val research: String) extends Professional :
  def works = " thinks about " + research

trait Manager extends Professional :
  val works = " makes money "

trait Student(val study: String) extends Professional :
  def works = " learns " + study
```

Example Orthogonal Features: Beings

- Implementing orthogonal features by traits



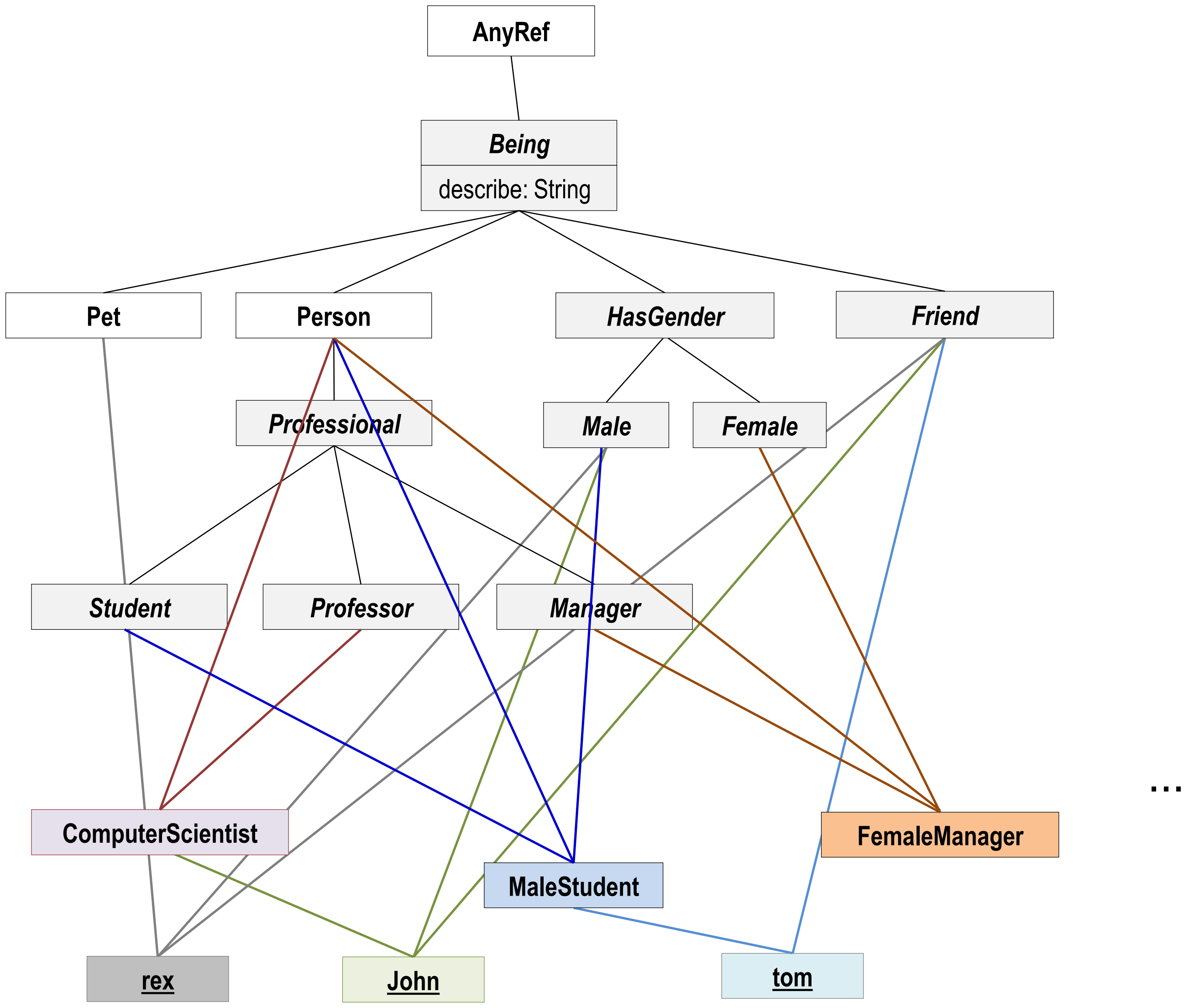
Example Orthogonal Features: Beings

- Mixing together multiple traits

```scala
class ComputerScientist(name: String, research: String) extends Person(name) with Professor(research)

class MaleStudent(name: String, acc: String, study: String) extends Person(name) with Male with Student(study) :
  override val accessory = acc

class FemaleManager(name : String, acc: String) extends Person(name) with Female with Manager :
  override val accessory = acc
```

Example Orthogonal Features: Beings

- Implementing orthogonal features by traits



Example Orthogonal Features: Beings

- Mixing together multiple traits

```scala
class ComputerScientist(name: String, research: String) extends Person(name) with Professor(research)

class MaleStudent(name: String, acc: String, study: String) extends Person(name) with Male with Student(study) :
  override val accessory = acc

class FemaleManager(name : String, acc: String) extends Person(name) with Female with Manager :
  override val accessory = acc
```

```scala
val tom = new MaleStudent("Tom", "mp3 player", "philosophy") with Friend(John)

object John extends ComputerScientist("John", "languages") with Male with Friend(tom) :
  override val accessory = "laptop"

val rex = new Pet("Dog", "Rex") with Male with Friend(John) :
  val accessory = "neckband"

val rita = new FemaleManager("Rita", "iPhone") with Friend(John)

…
```

Example Orthogonal Features: Beings

Typing

- Intersection types

petPrint requires Pet and Friend

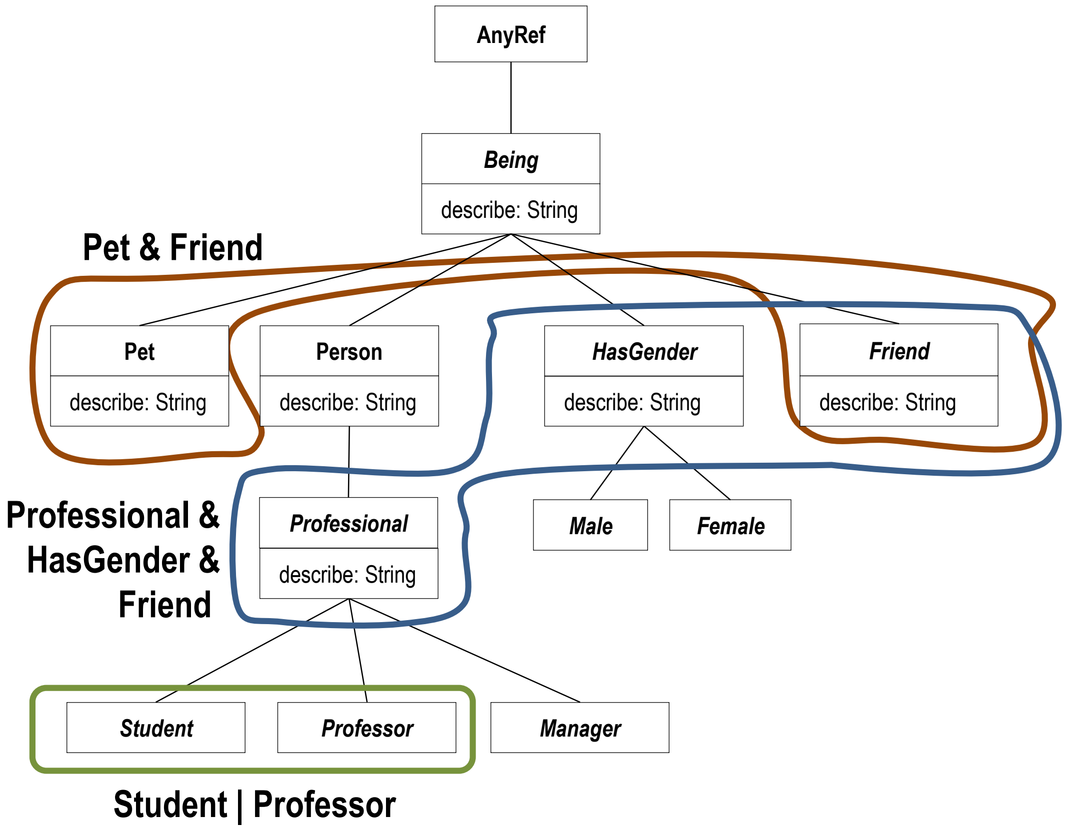
```
def petFriend(p: Pet & Friend) = {
  println("Pet " + p.name + " " + p.listen)
}
```

```
def profFriendPrint(p: Professional & Friend & HasGender) = {
  if (p.isMale) then print("Mr ")
  else  print("Mrs ")
  println(p.name + + " " p.works + "and" + p.listen)
}
```

doWork allows Student or Professor but excludes Manager

- Union types

```
def doWork(p : Student | Professor) = {
  println(p.name + p.works)
}
```



Example Orthogonal Features: Collections

(too)

- Concrete classes as mixture of many traits

```
sealed abstract class List[+A]
  extends AbstractSeq[A]
    with LinearSeq[A]
    with LinearSeqOps[A, List, List[A]]
    with StrictOptimizedLinearSeqOps[A, List, List[A]]
    with StrictOptimizedSeqOps[A, List, List[A]]
    with IterableFactoryDefaults[A, List]
    with DefaultSerializable {

abstract class AbstractSeq[+A] extends scala.collection.AbstractSeq[A] with Seq[A]

trait LinearSeq[+A]
  extends Seq[A]
    with collection.LinearSeq[A]
    with LinearSeqOps[A, LinearSeq, LinearSeq[A]]
    with IterableFactoryDefaults[A, LinearSeq] {

trait LinearSeqOps[+A, +CC[X] <: LinearSeq[X], +C <: LinearSeq[A] with LinearSeqOps[A, CC, C]]
  extends Any with SeqOps[A, CC, C]
    with collection.LinearSeqOps[A, CC, C]
```

…

List is composed of many traits providing orthogonal features to the implementation

Linearization of Inheritance Hierarchy

For **resolving conflicts** and **determining the order of super calls**

Scala builds a **linear sequence of classes/traits** for an **inheritance hierarchy** as follows
- **class** itself is **most special**
- **right** superclass/supertrait **more special than left** superclass/supertrait
- a **class/trait** always **before its declared superclass/supertrait**

The approach is a variant of the **C3 superclass linearization** algorithm also used e.g. by
- Dylan (https://opendylan.org/)
- Python
- and others

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A monotonic superclass linearization for Dylan. In P*roceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (OOPSLA '96). ACM, New York, NY, USA, 69–82. DOI:https://doi.org/10.1145/236337.236343

Linearization of Inheritance Hierarchy

Specification

Definition 5.1.2 Let C be a class with template C_1 **`with`** ... **`with`** C_n { *stats* }. The *linearization* of C, $\mathcal{L}(C)$ is defined as follows:

$$\mathcal{L}(C) \quad = \quad C\,,\, \mathcal{L}(C_n) \vec{+} \ldots \vec{+} \mathcal{L}(C_1)$$

Here $\vec{+}$ denotes concatenation where elements of the right operand replace identical elements of the left operand:

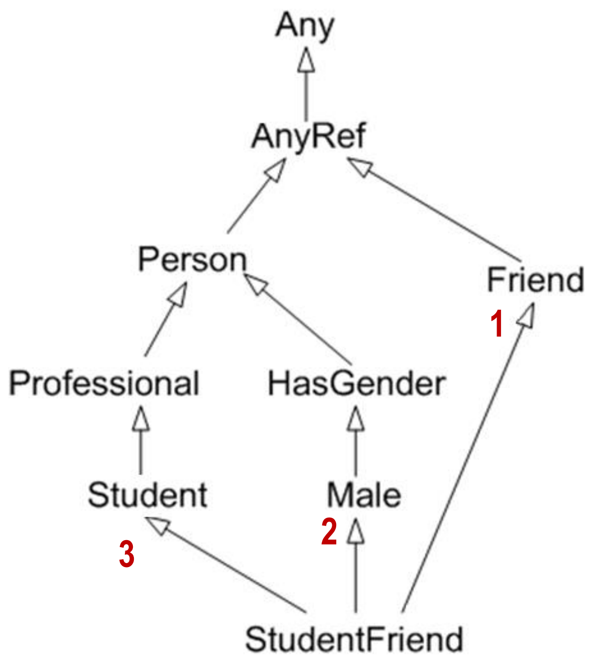
$$\begin{array}{lcll} \{a, A\} \vec{+} B & = & a, (A \vec{+} B) & \textbf{if}\, a \notin B \\ & = & A \vec{+} B & \textbf{if}\, a \in B \end{array}$$

when class **a** occurs multiple times
the last occurrence ist taken

LINEARIZATION OF INHERITANCE HIERARCHY

Example:

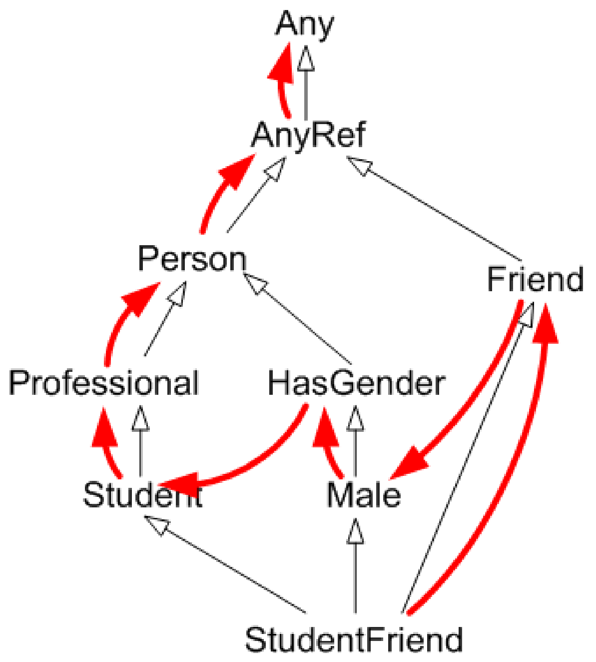
```
trait Person extends AnyRef
trait Friend extends AnyRef
trait Professional extends Person
trait Student extends Professional
trait HasGender extends Person
trait Male extends HasGender
class StudentFriend extends Student with Male with Friend
```



Linearization of Inheritance Hierarchy

Example:

```
trait Person extends AnyRef
trait Friend extends AnyRef
trait Professional extends Person
trait Student extends Professional
trait HasGender extends Person
trait Male extends HasGender
class StudentFriend extends Student with Male with Friend
```



$\mathscr{L}$(StudentFriend) =
StudentFriend,
Friend,
Male,
HasGender,
Student,
Professional,
Person,
AnyRef,
Any

III.1 Types, Subtypes and Inheritance

- Introduction
- Subtyping
- Liskov's substitution principle
- Multiple Inheritance
- Mixin Inheritance and Scala Traits
- Summary

Summary

- Types
  - Types define subsets of values which share same properties and members
- Subtyping
  - types with subtype relation from a lattice
  - supremum and infimum in lattice correspond to type union und type intersection operation
- Liskov's Substitution Principle states necessary conditions for subtyping to be type-safe
  - assumptions in subtypes must be weaker
  - guaranties of subtypes must be stronger
- Different forms of multiple inheritance
  - multiple implementation inheritance (like in C++)
  - multiple interface inheritance (like in Java)
  - mixins
- Traits in Scala
  - a form of mixin inheritance
  - with linearization of inheritance hierarchy (a variant of the C3 superclass linearization algorithm)