Functional Programming

2 Introduction to Scala

2 Introduction to Scala

Installation and IDE

Language basics

Classes, objects, traits

Generics

Scala - A SCALABLE LANGUAGE

www.scala-lang.org

- Development
  - by Martin Odersky and his team at the EPFL Lausanne
  - 1st release 2004
  - current version 3.5, with Scala 3.3.4 as LTS
    - Scala 3 represent major step compared to previous Scala 2 versions

Scala 2.0 (2006) – Scala 2.13 (2021)

- Combines object-oriented and functional concepts
  - functional influencer: Haskell
  - Object-oriented influencers: Smalltalk, Self, Java, OCaml, ...

Also allows imperative programming

- Platform integration
  - Scala for Java VM
    - translates to Java-Bytecode
    - can use Java APIs
  - Scala on Android
  - Scala on JavaScript VM

Scala Community

`www.scala-lang.org`

Key Players

- Programming Methods Laboratory, EPFL Lausanne
  - Research and development
  - Theory, compiler, libraries, applications
- Lightbend Inc. (https://www.lightbend.com/)
  - Industrial application of Scala technology
  - Frameworks and tools, e.g., Akka

Scala Software

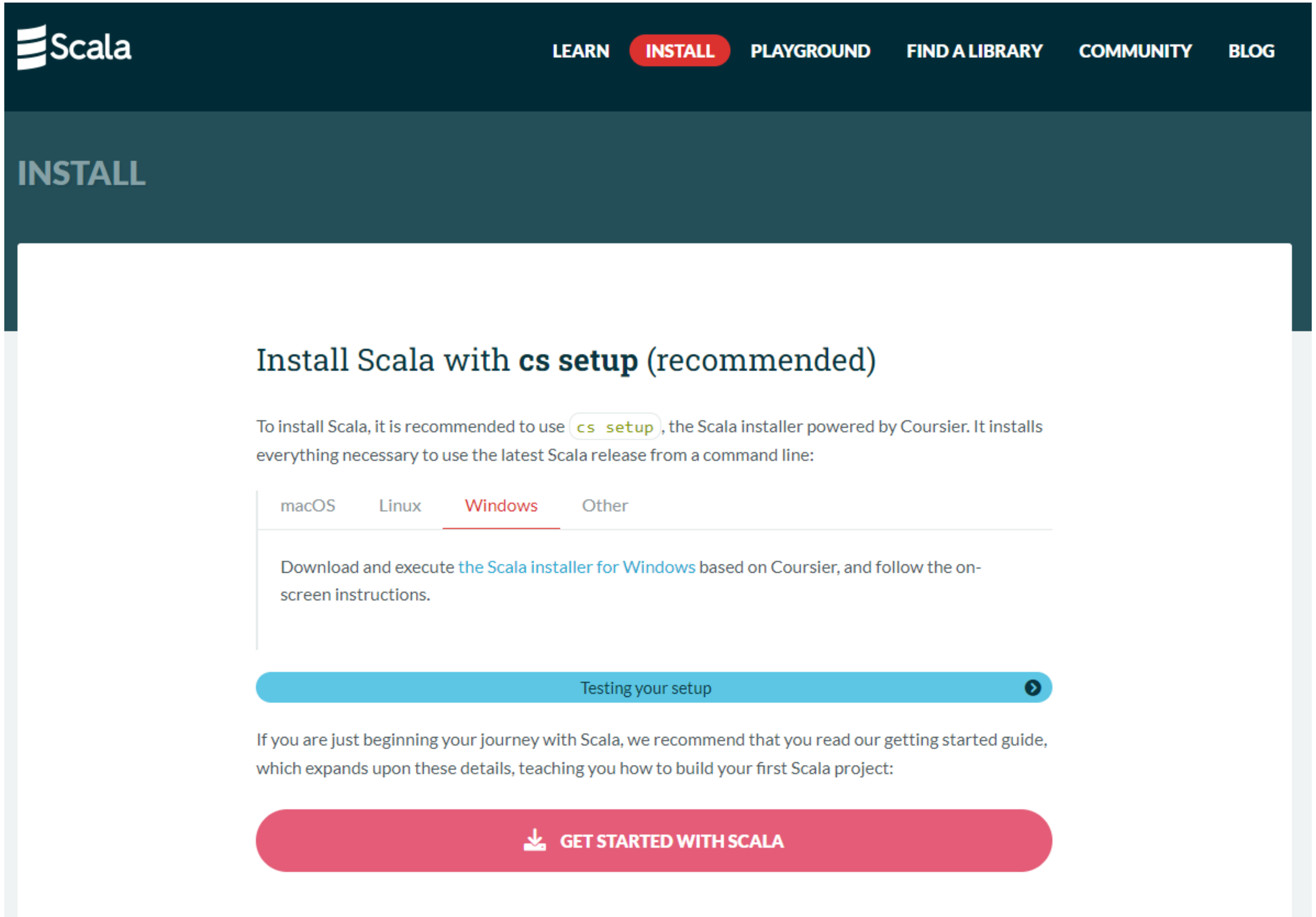
- Download from http://www.scala-lang.org/downloads

- Current long-term support release: 3.3.4 => major update to Scala 2 (partly not compatible)

- Packages
  - scala-devel The Scala compiler
  - scala-library The Scala library
  - scala-tool-support Tool support files for various text editors like emacs, vim or gedit
  - scala-documentation PDF documentation on the Scala programming language
  - scala-devel-docs Contains the Scala API and code examples
  - scala-test Test Suite we use to test the compiler and library
  - scala-msil Tools required to develop Scala programs for .NET

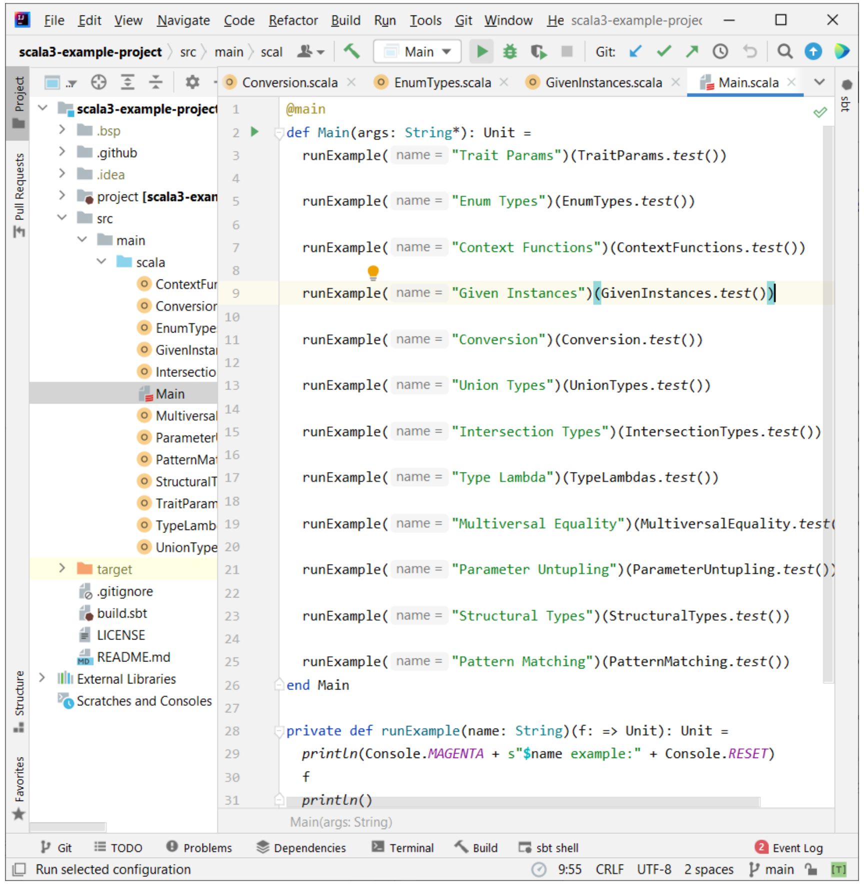
Scala Installation

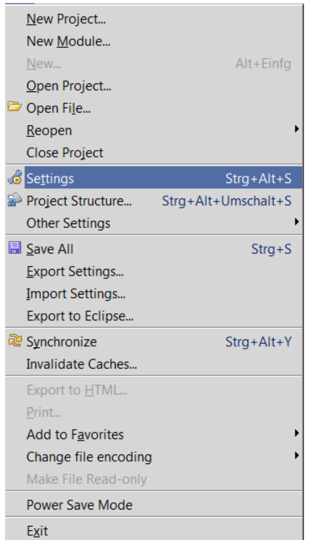
- Download from http://www.scala-lang.org/downloads

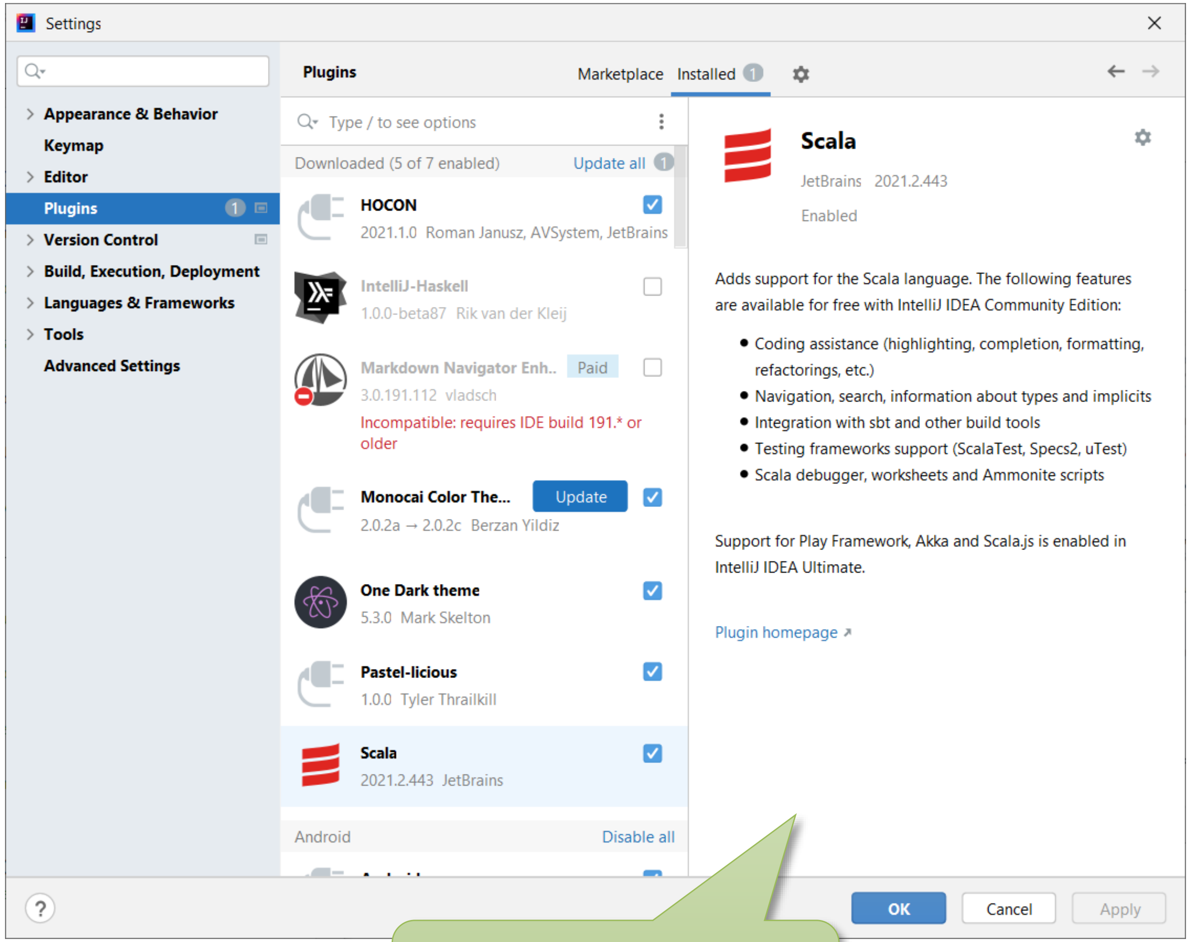


IntelliJ IDEA Plugin for Scala

- IntelliJ IDEA (http://www.jetbrains.com/idea/) + Scala Plugin





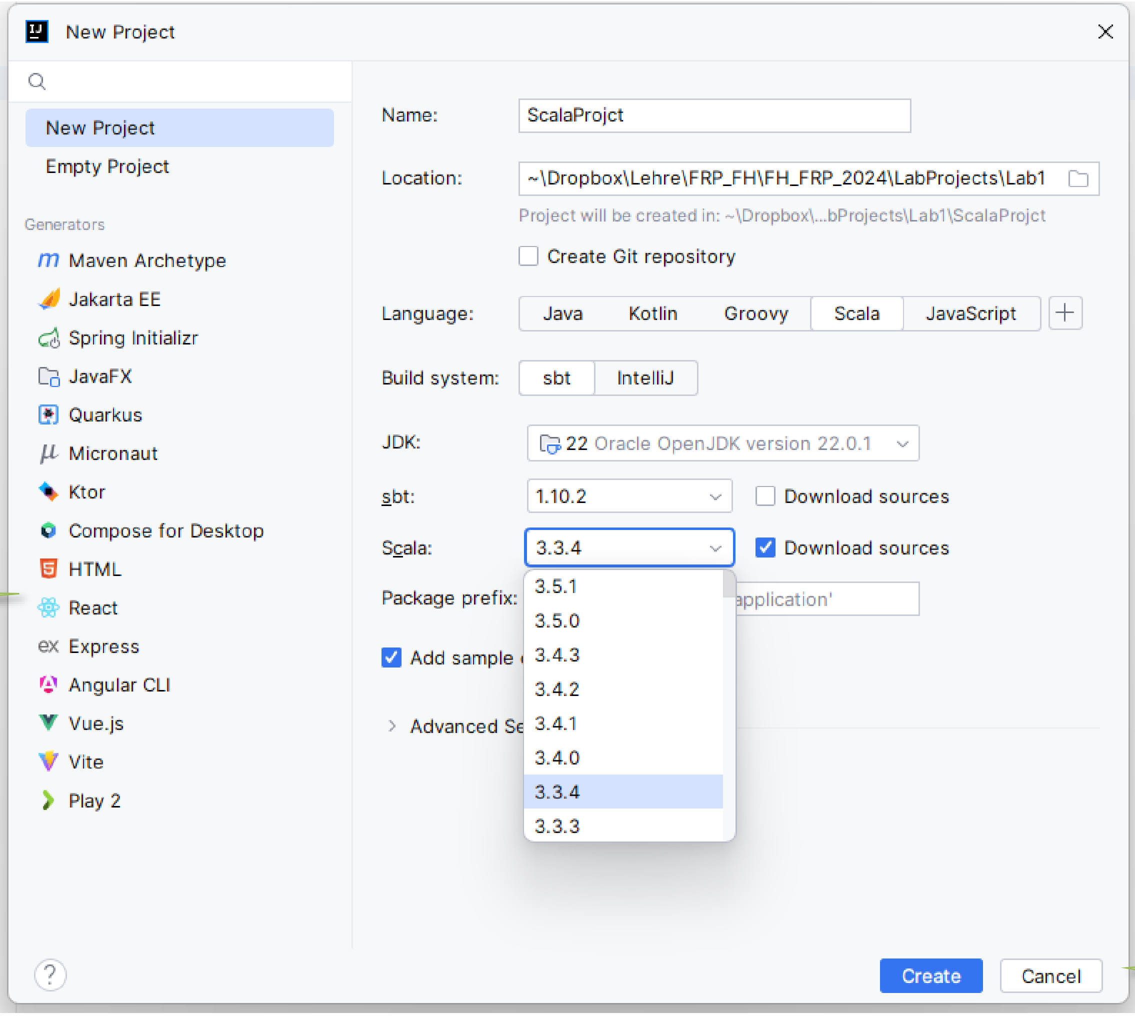


With plugin, no Scala installation needed !!

INTELLIJ IDEA PLUGIN FOR SCALA

Create Project

- Language: **Scala**
- Build system: **sbt**
- Scala version: **3.3.4**

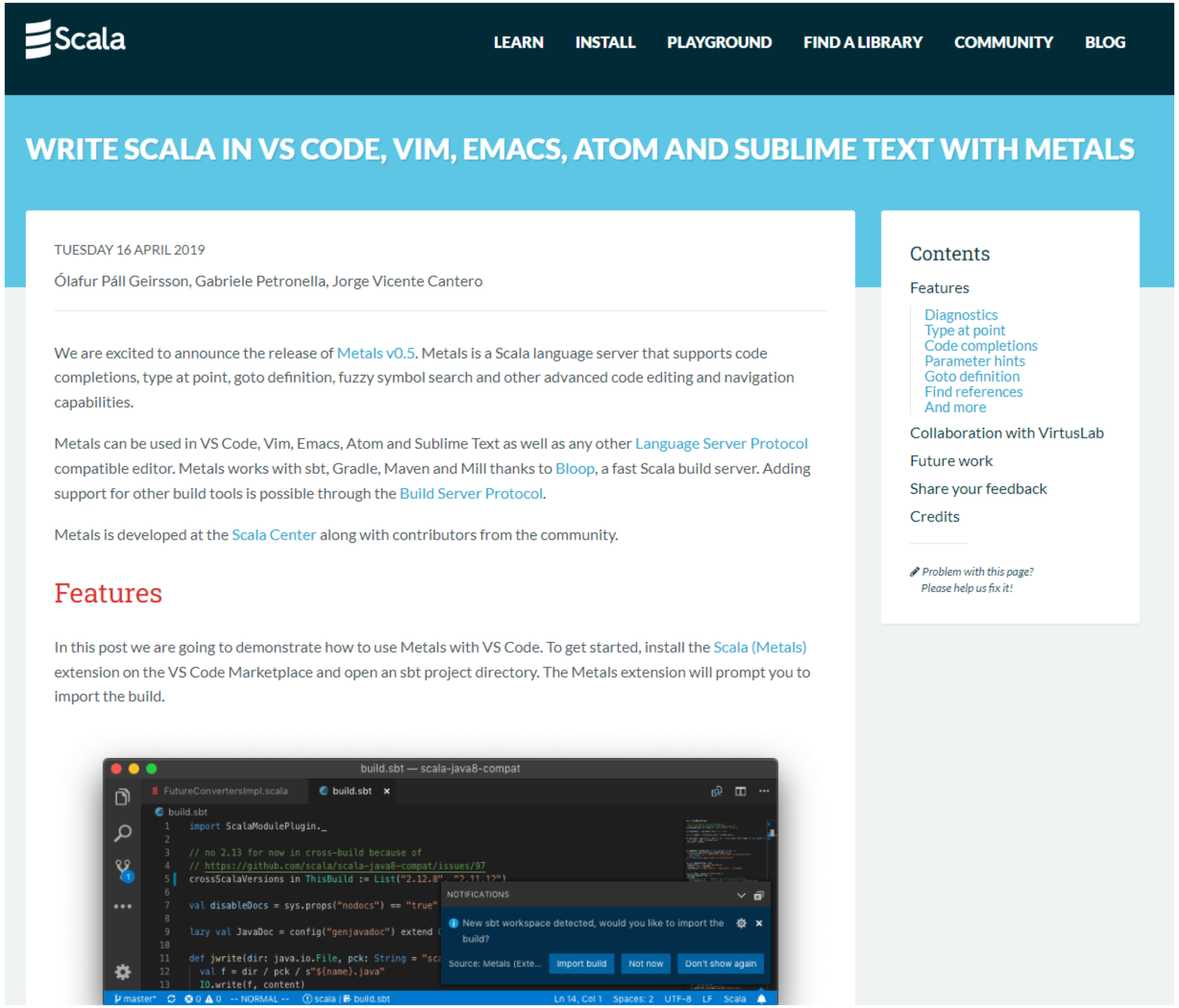


We use Scala LTS version: 3.3.4

Build system sbt will download Scala

Visual Studio Code with Scala Plugin

- https://www.scala-lang.org/2019/04/16/metals.html


Scala

LEARN INSTALL PLAYGROUND FIND A LIBRARY COMMUNITY BLOG

WRITE SCALA IN VS CODE, VIM, EMACS, ATOM AND SUBLIME TEXT WITH METALS

TUESDAY 16 APRIL 2019

Ólafur Páll Geirsson, Gabriele Petronella, Jorge Vicente Cantero

We are excited to announce the release of Metals v0.5. Metals is a Scala language server that supports code completions, type at point, goto definition, fuzzy symbol search and other advanced code editing and navigation capabilities.

Metals can be used in VS Code, Vim, Emacs, Atom and Sublime Text as well as any other Language Server Protocol compatible editor. Metals works with sbt, Gradle, Maven and Mill thanks to Bloop, a fast Scala build server. Adding support for other build tools is possible through the Build Server Protocol.

Metals is developed at the Scala Center along with contributors from the community.

Features

In this post we are going to demonstrate how to use Metals with VS Code. To get started, install the Scala (Metals) extension on the VS Code Marketplace and open an sbt project directory. The Metals extension will prompt you to import the build.

Contents

Features
- Diagnostics
- Type at point
- Code completions
- Parameter hints
- Goto definition
- Find references
- And more

Collaboration with VirtusLab

Future work

Share your feedback

Credits

Problem with this page? Please help us fix it!

2 Introduction to Scala

Installation and IDE

Language basics

Classes, objects, traits

Generics

Equal to Java

- Standard data types
  - but type names in upper case

```
Byte, Short, Int, Long, Float, Double, Char, Boolean
```

- Literals (some important)

```
1, 1S, 1289349348L, 3.14, 3.14F, 12.12E-12, 'a', '\n', '\u0044', ...
```

- java.lang.String for Strings

```
val s : String = "abc"
```

- Block structure and lexical scoping

```
{
  var a = ...
  if (a == y) then {
    val s = "is y"
    println(s)
  } else {
    val s = "not is y"
    println(s)
  }
}
```

Syntactical Differences to Java

Source files

- file ending **.scala**
- can contain arbitrary definitions: classes, traits, objects, methods, values
- package structure same as in Java

File `imp/imperative.scala`

```scala
package imp

abstract class Expr
abstract class Val(val x: AnyVal) extends Expr
case class IntVal(i: Int) extends Val(i)
case class BoolVal(b: Boolean) extends Val(b)
case class Var(name: String) extends Expr
case class BinExpr(op: String, left: Expr, right: Expr) extends Expr
...

def eval(expr: Expr, bds: Map[String, Val]) : Option[Val] = ...


val expr1 = BinExpr("+", Var("x"), IntVal(2))

object Main :
  def main(args : Array[String]) : Unit = ...
```

class definitions

method definition

value definition

object definition

Syntactical Differences to Java

- Semicolons are optional

```
val x = 1
val str = "ABC"
println(x)
```

- No brackets for methods without parameters

```
println
x.toString
string.toLowerCase
```

- Methods also in infix notations

```
if (list contains x) then ...
```

→

```
if (list.contains(x)) then ...
```

- Type declarations
  - with **:** in postfix notation

```
var y : Double = 1.0
y = 2.0
```

Syntactical Differences to Java

- Access modifiers
  - no modifier ➔ *public*
  - **`protected`**
  - **`private`**

public

private

```
class Person :
  ...
  private var name : String = ...
  override def toString : String = ...
```

public

Basic Language Elements

- Variables
  - immutable variables with **val**

```
val x = 1.0
```

final!

  - mutable variables with **var**

```
var y = 1.0
y = 2.0
```

  - with type declarations

```
var y : Double
y = 2.0
```

  - often optional and inferred

```
var y = 2.0
```

type Double inferred from value 2.0

- Methods with **def**
  - with type declarations for parameter (mandadory) and return type (optional)

paramter type

return type

=

```
def max(list : List[Int]) : Int  =  {
  var m = Integer.MIN_VALUE
  for (x <- list) {
    if (x > m) then m = x
  }
  m
}
```

block

```
def sign(x : Int) : Int  =
  if (x < 0) then -1
  else if (x == 0) then 0
  else +1
```

or single expression

Basic Language Elements

Classes, traits and objects

- class definitions with **class**

colon !

```
class Person :
  ...
```

indentation is significant

- with inheritance

```
class Student extends Person :
  ...
  override def toString : String = ...
```

override mandatory

- traits similar to interfaces with default methods

```
trait Writeable :
  def write(out: PrintStream) : Unit
  def writeln(out: PrintStream) : Unit = {
    write(out)
    out.println()
  }
```

- objects definitions are singletons

```
object HelloWorld extends App :
  println("Hallo World")
```

Scala 2 Compatibility

Allows braces instead of colon

braces !

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

```
class Student extends Person {
  ...
  override def toString : String = ...
}
```

```
trait Writeable {
  def write(out: PrintStream) : Unit
  def writeln(out: PrintStream) : Unit = {
    write(out)
    out.println()
  }
}
```

```
object HelloWorld extends App {
  println("Hallo World")
}
```

Scala Characteristics

from: Martin Odersky, Keynote ScalaDays, Berlin 2018

Types

- statically typed
- generics
- type inference

Objects

- purely object-oriented
- type hierarchy including all types
- dynamic binding of methods and fields

not purely functional!

Functions

- expressions
- first-class functions
- higher-order functions
- immutable data
- pattern matching

Types

Static, strong typing

Type inference

```
val list = List(2, 1, 4, 2)
```

```
➔ list : List[Int]
```

Generics

- generic types

type parameters in square brackets (!)

```
class Buffer[A] {... }
```

- generic methods

type parameters after method name

```
def compose[A, B, C](f : A => B, g : B => C) : A => C =
  x => g(f(x))
```

Object-oriented

Everything is an object

➔ No difference between built-in value types and reference types

- Methods for built-in types

```
1.toString
-1.abs
1.2.toInt
```

- Operators as methods

```
1.+(2)
```

➔ `1 + 2`

only conceptionally, compiled as in Java

- Methods as operators

```
val oneToN = 1 to n
```

➔ `1.to(n)`

- Comparison always by ==

```
x == 0
str == "abc"
this == that
```

Functional

Working with expressions

- `if` is expression with return value

```
val sign = if (x < 0) then -1 else if  (x > 0) then +1 else 0
```

- Blocks are expressions ➔ value of block is value of last expression

```
val max =
  {
    var m = Integer.MIN_VALUE
    for (x <- list) {
      if (x > m) m = x
    }
    m
  }
```

value returned

- All methods return a value (possibly `Unit` value `()`)

Unit type with one value (≅ void)

```
def sign(x : Int) : Int =
  if  (x < 0) then -1
  else if (x > 0) then +1
       else 0
```

```
def write(list : List[Int]) : Unit = {
  for (x <- list) {
    System.out.print(x + "")
  }
}
```

Functional

Immutable data is the default

- immutable variables and parameters

```
val xx = 1
```

```
def maxx(list : List[Int]) : Int  = { ... }
```

parameters always immutable

- immutable data structures
  - immutable pairs and tuples

```
val a1 = ('a', 1)
```

  - functional lists

```
val list21 : List[Int] = 2 :: 1 :: Nil
val list321 : List[Int] = 3 :: list21
```

creates new set with element 3 prepended

  - functional sets

```
val set12 : Set[Int] = Set(1, 2)
val set123 = set12 + 3
```

+ creates new set with element 3 added

  - immutable maps
  - ...

Functional

Functions as first-class objects

- Lambdas

```
(x : Int) => x*2 + 1
```

- Generic function types

```
() => A
```
```
A => B
```
```
(A, B) => C
```
```
(A, B, C) => D
```
…

- Functions as parameters

```
def map[A, B](xs : List[A], fn : A => B) : List[B] =
  for (x <- xs) yield fn(x)
```

- Functions as return values

```
def compose[A, B, C](f : A => B, g : B => C) : A => C =
  x => g(f(x))
```

Imperative

- Mutable variables

```
var i = 1
i = 2
```

- **while** loop

```
while (i < 10) {
  ...
  i = i + 1
}
```

returns Unit value ()

- **for** loop without return

```
for (j <- 1 to 10) {
    ...
}
```

returns Unit value ()

- Exceptions

```
try {
  val x = s.toInt
} catch {
  case ne : NumberFormatException => println("Not a number")
  case e : Exception => println("Exception")
}
```

Scala Programming Style

Functional externally

- referential transparent functions

```
def fac(x: Int) : Int = {
  ...
}
```

- immutable data structures

```
class List[+T] extends Iterable[T] {
  def map[R](f : T => R) : List[R] = {
     ...
  }
}
```

Imperative internally

- with imperative internal implementations

```
def fac(x: Int) : Int = {
  var r = 1
  for (i <- 2 to x)  r = r * i
  r
}
```

more efficient compared to recursive solution

- with mutable internal implementation

```
class List[+T] extends Iterable[T] {
  def map[R](f : T => R) : List[R] = {
    val builder : Builder[T] =  new Builder[T]
    for (t <- this) builder.add((f(t))
    builder.build
  }
}
```

mutable builder

Rules

- **Use imperative** programming **internally** for efficiency reasons
- **Avoid non-local side effects** and **public access to mutable data** structures

JKU

Local Methods

- Methods defined within outer method
  - ➔ only locally defined
  - ➔ have access to outer function's parameters and variables

<u>Example</u>: `fact` with inner method `factAccumulate`

```
def fact(n : Int) : Int = {

  def factAccumulate(i : Int, accumulator : Int) : Int = {
    if (i > n) {
      accumulator
    } else {
      factAccumulate(i + 1, i * accumulator)
    }
  }

  factAccumulate(1, 1)
}
```

Positional and Named Arguments

```
def speed(distance: Float, time: Float) : Float = {
  distance / time
}
```

Method calls

- Position of arguments

```
speed(1200, 10)
```

- With name of parameters

```
speed(distance = 1200, time = 10)
```

```
speed(time = 10, distance = 1200)
```

- Mixed: First positional, then by name

```
speed(1200, time = 10)
```

Default Values for Parameters

- Definition of default values in method declarations

```
def printTime(out: java.io.PrintStream = Console.out, divisor : Int = 1) = {
  out.println("time = "+ System.currentTimeMillis() / divisor)
}
```

  - with default values

```
printTime()
```

  - with new values

```
printTime(System.err, 1000)
```

  - mixed

```
printTime(System.err)
printTime(divisor = 1000, out = System.err)
printTime(divisor = 1000)
printTime(System.err, divisor = 1000)
```

JKU

Vararg Parameter

- Varargs: last parameter can be repeated
  - type declaration with **<*Type*>***
  - within method represented as array **Array[<*Type*>]**

a number of string values!

```
def printLines(lines : String*) = {
  for (line <- lines) {
    println(line)
  }
}
```

a number of string values!

```
printLines( "This is the first line",
  "and this the second",
  "…")
```

String Interpolator

Insertion of computed values in string

- string with s prefix
- ***$expression*** for insertions
- with **${*expression*}** for complex expressions
- inserts **toString** of value of expression

```
s"$name = ${eval(expr, bdgs)}"
```

JKU

2 Introduction to Scala

Installation and IDE

Language basics

Classes, objects, traits

Generics

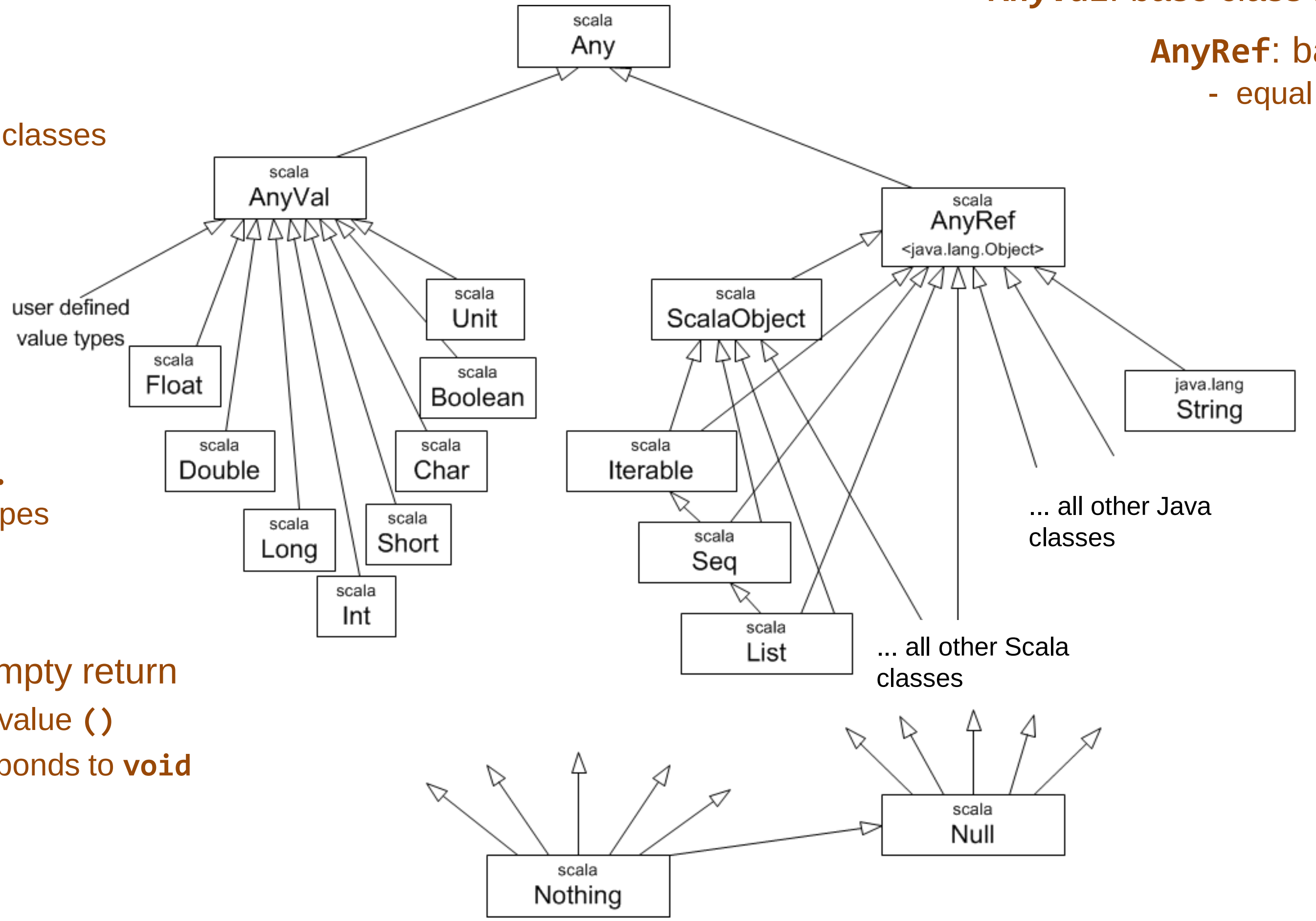
Scala Type Hierarchy

`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: wrapper 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!**

Class Any

- Abstract base class for all types

```
package scala

abstract class Any {
  final def == (that: Any): Boolean  =
    if (null eq this) then null eq that else this equals that

  final def != (that: Any): Boolean  =  !(this == that)

  def equals(that: Any): Boolean

  def hashCode: Int = …

  def toString: String = …

  def isInstanceOf[A]: Boolean

  def asInstanceOf[A]: A = this match {
    case x: A => x
    case _ => if (this eq null) then this else throw new ClassCastException()
  }
}
```

use == for all equality tests !

generic type parameter

equality

hashcode

toString

typtests and typcasts

Example: Typetests and Typcasts

```
if (x.isInstanceOf[Int]) then x.asInstanceOf[Int] + 1
```

Classes AnyVal and AnyRef

- AnyVal: Base class for value types

```
class AnyVal extends Any
```

- AnyRef: Base class for reference types (= `java.lang.Object`)

```
class AnyRef extends Any {

  def equals(that: Any): Boolean      = this eq that
  def hashCode: Int = …

  def toString: String  = …


  final def eq(that: AnyRef): Boolean = …

  final def ne(that: AnyRef): Boolean = !(this eq that)

  def synchronized[T](body: => T): T
     // execute `body` while locking `this`.

}
```

equals and hashCode

toString

should be overridden

reference equality

synchronized as method (!?)

Class Definitions

- Class parameters
  - **parameters** of primary constructor
  - plus **private fields**
- Class body
  - **field and methods** declarations
  - plus **code of constructor**

Classes have no static members!

class parameters

colon (!)

```
class Car(model: String, year: Int, initial : Int) :

  private var miles: Int = initial

  def getModel = model
  def getYear = year

  //...
  println("Car " + model + " year " + year + " created ")
```

field and method declarations

constructor code

- Instantiation with arguments for class parameters

```
val bmw = new Car("BMW", 2019, 0)
```

- alternatively without new

```
val bmw = Car("BMW", 2019, 0)
```

new can be omitted !

Class Definitions

Overloaded Constructors

- Definition with keyword **this**
- must call primary constructor

```
class Car(model: String, year: Int, initial : Int)  :

  private var miles: Int = initial

  def this(model: String, year: Int) = {
     this(model, year, 0)
  }

  def this(model: String) = {
     this(model, 2021, 0)
  }

  ...
```

Class Definitions

Inheritance

- **extends** with **call to constructor** of superclass
- **override** mandatory for **overriding concrete** members
- **abstract** classes and members supported

```
abstract class Vehicle(model: String, initial: Int) :
  protected var miles = initial
  def getMiles = miles
  override def toString : String = model + " with miles " + miles
  def drive(distance : Int) = miles += distance
```

call superclass constructor

```
class Car(model: String, year: Int, initial : Int) extends Vehicle(model, initial) :
  private val FULL = 20.0
  private val MILAGE = 50.0
  private var fuelLevel: Double = FULL;

  override def toString : String = super.toString + " fuel " + fuelLevel
  override def drive(distance: Int) = {
    super.drive(distance)
    fuelLevel = fuelLevel - distance / MILAGE
  }
  def refill() = { fuelLevel = FULL }
```

override!

Members

- Class members can be
  - **val** – immutable variable
  - **var** – mutable variable
  - **def** – method

val and var with getter and setter methods

- All members are **dynamically bound**
  - also **val** and **var** variables

in distinction to Java

- All members **can be abstract**

```
abstract class AbstractClass :
  def abstractMethod : ReturnType
  var abstractVar : VarType
  val abstractVal : ValType
```

abstract because no definition

- All members **can be overridden**

```
abstract class Shape :
  val pos : Point
  def draw : Unit
  ...
```

```
class Group(elems : Shape*) extends Shape :
  override val pos = new Point(minX(elems), minY(elems))
  override def draw = { /*...*/ }
```

override

Singleton Objects

Definition of singleton objects

- with keyword **`object`**
- with **`extends`** from superclass

```
object MyCar extends Car("Qasqai", 2011, 113409)
```

- possibly with class body

```
object MyCar extends Car("Qasqai", 2011, 113409) :
  val owner : String = "Me"
  override def toString : String = "This is my car with " + getMiles + " miles"
```

- Accessing singleton by object name

```
MyCar.drive(125)
println(MyCar.toString)
```

Specific constraints and properties of objects

- **cannot** have **class parameters**
- **cannot be extended**
- **same name as class** allowed (= ***companion object*** for the class)

Companion-Objects

- Singleton object with same name as class is companion to class
  - must be in same file as class
  - closely belongs to class
  - can access private members
  - has methods and fields similar to static members
  - often with apply method for constructing objects of class (can be called without method name)

```
class Car(model: String, initial: Int, mileage: Double)
  extends Vehicle(model, initial) { ...}
```

```
object Car {
  var carPool: List[Car] = List()

  def apply(model: String, year: Int, mileage: Double) = {
    val car = new Car(model, year, mileage)
    carPool = car :: carPool
    car
  }
}
```

apply method:
can be called for an object **without** method name

```
Car("Opel", 45000, 53.1)
Car("BMW", 0, 62.4)
val car1 = Car.carPool(0)
val car2 = Car.carPool(1)
```

corresponds to

```
Car.apply("Opel", 45000, 53.1)
```

Main-class is Object

- Object with **main** method

```
object MyApp :
  def main(args : Array[String]) : Unit = {
    println("Hallo World")
  }
```

- Object extending **App**

```
object MyApp2 extends App :
  println("Hallo World")
```

Scala Traits

- Traits are abstract types similar to interfaces with default implementations in Java

```
trait Writeable {
  def write(out: PrintStream) : Unit

  def writeln(out: PrintStream) : Unit = {
    write(out)
    out.println()
  }
}
```

no body = abstract

concrete with body

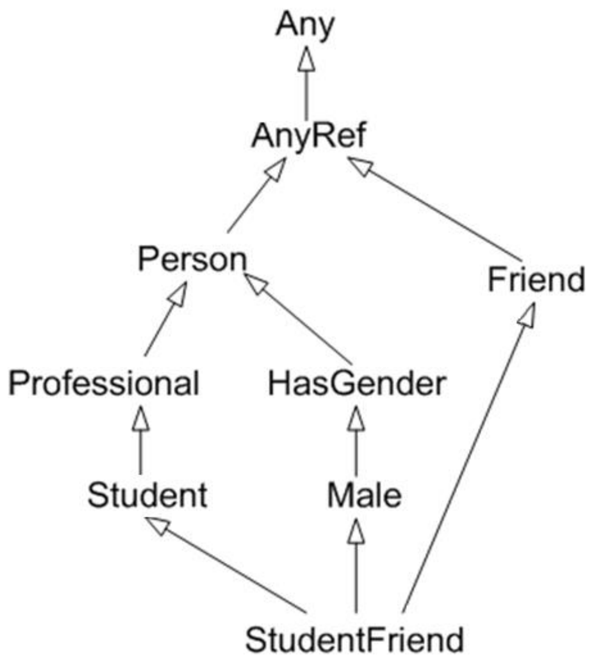
- inheriting from traits

use **with** for multiple supertypes

```
class Group(elems : Shape*) extends Shape with Writeable :
  ...
  override def write : Unit = {
    System.out.println("Group: " + elems)
  }
```

Multiple Inheritance with Traits: Linearization

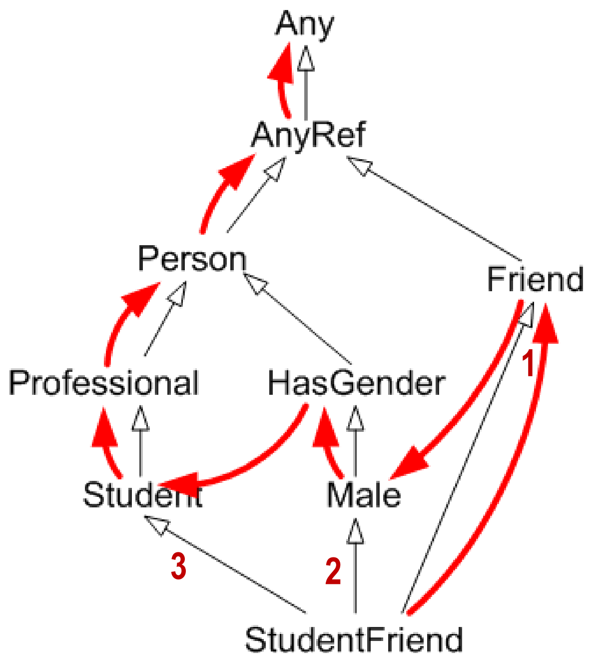
- Classes can extend one class and many traits
- Linearization of inheritance hierarchy
  - subtypes always before super types
  - right more specific than left
- Benefits
  - inheritance conflicts are avoided
  - super-calls are resolved




```
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
```

Multiple Inheritance with Traits: Linearization

- Classes can extend one class and many traits
- Linearization of inheritance hierarchy
  - subtypes always before super types
  - right more specific than left
- Benefits
  - inheritance conflicts are avoided
  - super-calls are resolved




```
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
```

```
StudentFriend,
Friend,
Male,
HasGender,
Student,
Professional,
Person,
AnyRef,
Any
```

most special

2 Introduction to Scala

Installation and IDE

Language basics

Classes, objects, traits

Generics

Type Parameters in Scala

- Type parameters for generic classes, traits and methods

```
class Class[T]
```

```
trait Trait[T]
```

```
def method[T](x : T)
```

Type parameters in square brackets [ ]

Example: Generic functional stack

```
class Stack[T] {
  ...
  def isEmpty : Boolean = ...
  def top : T = ...
  def pop : Stack[T] = ...
  def push(elem : T) : Stack[T] = ...
}
```

pop and **push** return new Stack object

```
var stringStack: Stack[String] = new Stack[String]()

stringStack = stringStack.push("Mike")
val frank = stringStack.top
stringStack = stringStack.pop
```

```
var intStack : Stack[Int] = new Stack[Int]()

intStack = intStack.push(1);
val one = intStack.top
intStack = intStack.pop
```

Type Bounds in Scala

- **Upper type bound** with `<:` — extends
- **Lower type bound** with `>:` — super

extends!

```
def max[T <: Ordered[T]](x: T, y: T) : T = {
    if (x >= y) x
    else y
}
```

```
class SortedList[T <: Ordered[T]] {
  ...
  def add(elem: T)  = {
    var i = elems.length - 1;
    while (i >= 0 && elem > elems(i)) {
      i -= 1
    }
    elems.insert(i, elem)
  }
}
```

```
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
  ...
}
```

Variance of Generic Types [1/2]

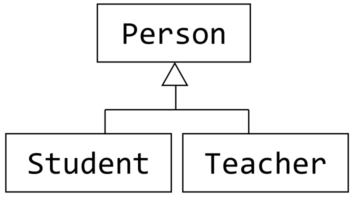
```
class List<T> {
   public void add(T elem) {...}
   public T get(int i) {...}
}
```

Recall

- **Co-variant output parameters and returns** are safe
- **Contra-variant input parameters** are safe

➔ **Co- and contravariant assignments** of generic types in Java **not allowed**

Co-variant assignment in Java: Problem analysis



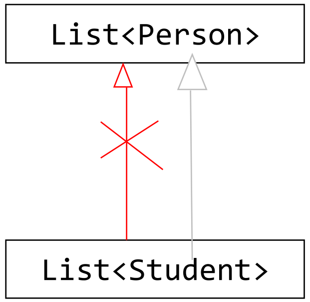

```
List<Student> stdts = new ArrayList<Student>();
```

```
List<Person> persons = stdts;

persons.add(new Teacher(...));

Student student = stdts.get(0);
```

Error: co-variant assignment not allowed

input not safe

ClassCastException



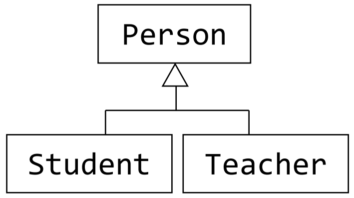
Variance of Generic Types [2/2]

```
class List<T> {
   public void add(T elem) {...}
   public T get(int i) {...}
}
```

Recall

- **Co-variant output parameters and returns** are safe
- **Contra-variant input parameters** are safe

➔ **Co- and contravariant assignments** of generic types in Java **not allowed**



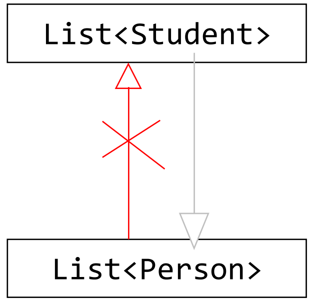
Contra-variant assignment in Java: Problem analysis

```
List<Person> persons = new ArrayList<Person>();
persons.add(new Teacher(…));
```

```
List<Student> stdts = persons;

Student s = stdts.get(0);
```

Error: contra-variant assignment not allowed

return not safe: ClassCastException



Solution in Java: Use-site Variance

```
public class List<T> {
  public T get(int idx) {...}
  public void add(T x) {...}
}
```

Co-variance with upper-bounded wildcard

```
List<Student> students = new List<Student>();
students.add(new Student(..));
```

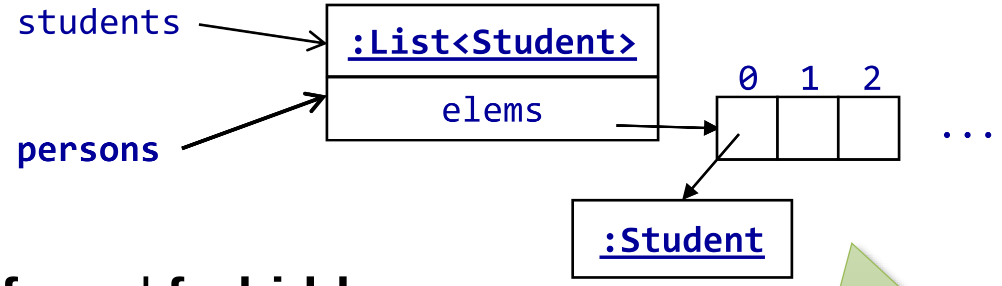
```
List<? extends Person> persons = students;
```

Variable with **upper-bounded wildcard** allows **co-variant** assignments **eingeschränkt**

- method with **generic return** are **typesafe** and **allowed**

```
Person p = persons.get(0);
```

persons.get guarantees **Person**,
Student compatible with **Person**



- method with **generic input parameter not typesafe** and **forbidden**

~~persons.add(new Person());~~

persons.add would allow **Person**,
but is not compatible with **Student**

No **add** for variable **persons** allowed!

Solution in Java: Use-site Variance

```java
public class List<T> {
  public T get(int idx) {...}
  public void add(T x) {...}
}
```

Contra-variance with lower-bounded wildcard

```java
List<Person> persons = new List<Person>();
persons.add(new Person(..));
```

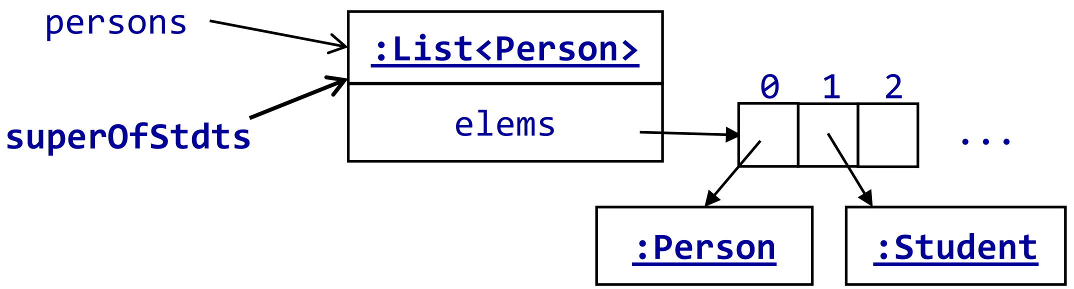
```java
List<? super Student> superOfStdts = persons;
```

Variable with **lower-bounded wildcards** allows **contra-variant** assignments

- method with **generic input parameter typesafe** and **allows**

```java
superOfStdts.add(new Student(...));
```

students.add allows **Student**,
is compatible with **Person**



- method with **generic return** are **not typesafe** and **forbidden**

~~`Student s = superOfStdts.get(0);`~~

students.get forbidden,
because **Student** cannot be guaranteed

No ***get*** for variable **students** allowed!

JKU

Use-Site Variance in Scala

Use-site variance declarations in Scala

- co-variant `_ <: T` — ? **extends** T
- contra-variant `_ >: T` — ? **super** T

<u>Example</u>: Co-variant assignment of **ListBuffer** — **ListBuffer** is a mutable linked list

```
val students : ListBuffer[Student] = ListBuffer();

val persons : ListBuffer[_ <: Person] = students
```

co-variant assignment

```
val p : Person = persons.head
```

head safe!

~~`persons.insert(0, new Teacher(...));`~~

use of insert not allowed !

Example Use-Site Variance in Scala

Example: copy elements from ListBuffer into ListBuffer

ListBuffer analogous to Java's **LinkedList**

- Variant 1: **upper bounded wildcard**

```
def copyTo[T](from: ListBuffer[_ <: T], to: ListBuffer[T]) = {
  for (t <- from) {
    to.addOne(t)
  }
}
```

- Variant 2: **lower bounded wildcard**

```
def copyTo[T](from: ListBuffer[T], to: ListBuffer[_ >: T]) = {
  for (t <- from) {
    to.addOne(t)
  }
}
```

```
val students : ListBuffer[Student] = ...
val persons  : ListBuffer[Person] = ...
```

```
copyTo(students, persons)
```

Example Use-Site Variance in Scala

Example: copy elements from ListBuffer into ListBuffer

```
val stdts : ListBuffer[Student] = ...
val persons : ListBuffer[Person] = ... copyTo(stdts, persons)
```

- Variant 1: **upper bounded wildcard**

```
def copyTo1[T](from: ListBuffer[_ <: T], to: ListBuffer[T]) = {
  for (t <- from) {
    to.addOne(t)
  }
}
```

- Variant 2: **lower bounded wildcard**

```
def copyTo2[T](from: ListBuffer[T], to: ListBuffer[_ >: T]) = {
  for (t <- from) {
    to.addOne(t)
  }
}
```

- Variant 3: **two type variables** with **upper bound**

```
def copyTo3[T, F <: T](from: ListBuffer[F], to: ListBuffer[T]) = {
  for (t <- from) {
    to.addOne(t)
  }
}
```

- Variant 4: **two type variables** with **lower bound**

```
def copyTo4[F, T >: F](from: ListBuffer[F], to: ListBuffer[T]) = {
  for (t <- from) {
    to.addOne(t)
  }
}
```

super allowed!

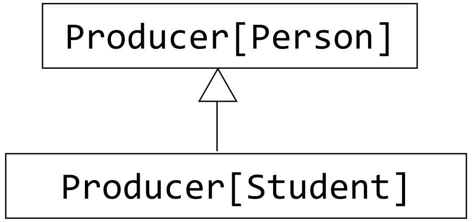
Declaration-site Variance in Scala

Variance declared together with type parameter in class/trait

- **Co-variant type parameter**
  - specified with **+**
  - only for type parameters for **return values**
  - class is **co-variant** for **co-variant type parameter**

co-variant positions

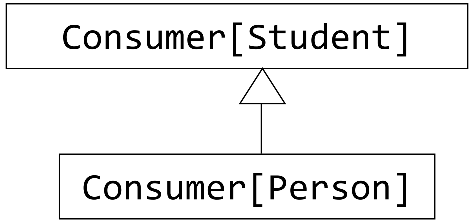
```
class Producer[+T] {
  def produce : T
}
```



- **Contra-variant type parameter**
  - specified with **-**
  - for type parameters for **input parameters**
  - class is **contra-variant** for **contra-variant type parameter**

contra-variant positions

```
class Consumer[-T] {
  def consume(t : T)
}
```



Declaration-site Variance in Scala

Example: Immutable stack

- co-variant type parameter T allowed as return type

Stack co-variant regarding type parameter T

```
class Stack[+T] {
  …
  def top : T = …
  def pop : Stack[T] = …
```

```
var stdts : Stack[Student] = Stack(new Student("Hans"))
var persons : Stack[Person] =  stdts
val p : Person = persons.top
persons = persons.pop
```

safe!

- but co-variant type parameter cannot be used as type for input parameters

```
class Stack[+T] {
  …

  def push(elem : T) = …
}
```

parameter of co-variant type not allowed !

Declaration-site Variance in Scala

Example: Immutable stack

push: returns new stack with new type for elements

- use **new type variable** with **type bound**
- return **new Stack** with **new type parameter**
- exploit **type inference** for **inferring type of returned Stack**

```
class Stack[+T] {
  ...
  def push [U >: T] (elem : U) : Stack[U] = new Stack[U] { … }
}
```

Generic type U with U supertype von T

Result Stack[U] !!

Example:

```
val students = new Stack[Student]
val persons = students.push(new Teacher(…))
```

1) Type parameter **U = Person**

2) Result of push is of type **Stack[Person]**

Type inference:

```
push[U >: T](elem : U) : Stack[U]
```

`T = Student`
`elem = new Teacher`
➔ U supertype of `Student` and `Teacher`
➔ U = `Person`
➔ return type is `Stack[Person]`

Declaration-site Variance in Scala

Example: Function1

- **contra-variant** type parameter **P** for **input**
- **co-variant** type parameter **R** for return

Contra-variant for T and co-variant for R

```
trait Function1[-T, +R] {
  def apply(x : T) : R
  ...
}
```

Person => String
subtype of
Student => Any

Type declaration
T => R
is trait type
Function1[-T,+R]

Example:

Function1[-A, +B]

```
def map[A, B](xs : List[A], fn : A => B) : List[B] =
  for (x <- xs) yield fn(x)
```

```
val students = List[Student]
val personNameFn : Person => String = (p: Person) => p.name
```

```
val infos : List<Any> = map(students, personNameFn)
```

Type inference:

A = Student, B = Any
fn : Function1[-Student, +Any]
personNameFn : Function1[Person, String]
➔ concrete Parameter **personNameFn** compatible with formal parameter **Function1[-Student, +Any]**

JKU

Declaration-Site vs. Use-Site Variance

Declaration-Site

- class specifies variance

```
class Stack[+T] { ... }
```

- use of type parameter restricted in class

```
class Stack[+T] {
  def top : T = ..
  def push(elem : T) = ...
}
```

- no use-site variance annotations needed

```
val ps : Stack[Person] = Stack[Student]
```

- class cannot have invalid operations

Use-Site

- no variance specification in class

```
class Stack[T] { ... }
```

- no restrictions in class

```
class Stack[T] {
  def top : T = ..
  def push(elem : T) = ...
}
```

- use-site variance annotations needed

```
Stack[_ <: Person] ps =
                 Stack[Student]();
```

- use of invalid operations are forbidden

```
ps.push(Teacher());
```