CSC 347 - Concepts of Programming Languages

Subtyping

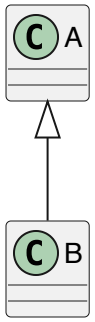
Instructor: James Riely

Learning Objectives

❓ What should the type relationship between parameterized types be?

- Identify read and write restrictions on parametric types

Subtyping



- Static and dynamic type

```
1 A x = new A ();
2 B y = new B ();
3 x = y; // B ok when A expected
```

- Method parameters

```
1 void aConsumer (A x) { ... }
2 aConsumer (new B());    // B ok when A expected
```

- Method results

```
1 B bProducer () { ... }
2 A x = bProducer (); // B ok when A expected
```

- Safe to use an instance of `B` when an `A` is expected
- `B` is a *subtype* of `A`: written `B<:A`
  - If `y:B` and `B<:A` then `y:A` (upcast)
- Subtyping is not just subclassing: parametric polymorphism

Subtyping Order: Top

- Subtyping relation `<:` is a *partial order* on types
  - *reflexive*: `X<:X`
  - *transitive*: if `Duck<:Bird` and `Bird<:Animal` then `Duck<:Animal`
- Some PLs have a `Top` type: `X<:Top` for all `X` (greater than all other types)
- In Java: `java.lang.Object` above reference types
- In Scala: `scala.Any` above all types, `scala.AnyRef` above reference types

```
1 val xs:List[AnyRef] = List ("hello", List(1))
2 val ys:List[Any]    = List ("hello", 1)
```

Subtyping Order: Bottom

- Most PLs do not have a `Bottom<:X` for all `X` (less than all other types)
- In Scala Type Hierarchy: `Nothing` is `Bottom` type
- ❓ What is `Bottom` useful for?
- Important for typing uses of `Nil`

```
1 val nil:List[String] = Nil
2 // Best type possible
3 val xs1:List[String] = "hello" :: nil
4 val xs2:List[Any]    = 1 :: nil
```

```
1 val nil:List[Nothing] = Nil
2 // Best type possible
3 val xs1:List[String] = "hello" :: nil
4 val xs2:List[Int]    = 1 :: nil
```

- `Nil : List[Nothing]` and `List[Nothing] <: List[X]`, so `(X :: List[Nothing]) : List[X]`

Type-Invariant Containers

```scala
enum Animal:
  case Bird(name: String)
  case Cat(name: String)
  def name: String
end Animal

case class Box[X](var content: X)
```

- Create concrete boxes

```scala
val birdBox: Box[Bird] = Box(Bird("Feathers"))
val catBox: Box[Cat] = Box(Cat("Fluffy"))
```

- Create an abstract box

```scala
var animalBox = Box[Animal](Bird("Feathers"))
animalBox.content = Cat("Fluffy")
```

- Alias boxes

```scala
animalBox = birdBox // does not compile
```

❓ Why not `animalBox = birdBox`?

- Does not compile because `Box[Bird]` is not a subtype of `Box[Animal]`
- Box is **invariant** in its type parameter `X`

Type-Invariant Containers

❓ What would go wrong if we allow assignment?

```
1 val birdBox: Box[Bird] = Box(Bird("Feathers"))
2 var animalBox: Box[Animal] = birdBox // does not compile, but let's assume it passes
```

```
1 val animal: Animal = animalBox.content  // ok, we can read bird as an animal
```

```
1 animalBox.content = Cat("Fluffy") // we can put a cat into an animal box
```

```
1 val bird: Bird = birdBox.content     // but now the bird box contains a cat!
```

❗ Writing to aliased container would create type violations!

Type-Covariant Containers

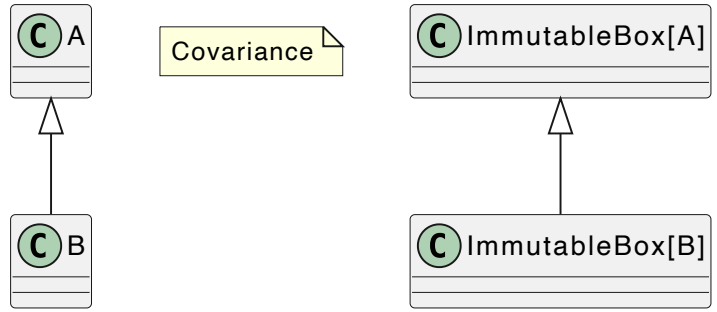
❓ What if we disallow modifying boxes?

```
1 case class ImmutableBox[+X](val content: X)
```

- Create and alias boxes

```
1 val birdBox: ImmutableBox[Bird] = ImmutableBox(Bird("Feathers"))
2 val animalBox: ImmutableBox[Animal] = birdBox // compiles
3 val animal = animalBox.content                // reading ok
4 animalBox.content = Cat("Fluffy") // immutable, cannot write
```

- `ImmutableBox` is **covariant** in `X`
- Covariance type annotation `+X` can only be applied to read-only structures



Covariant Boxes

❓ What is the practical relevance?

```
1 def printAnimalName(b: ImmutableBox[Animal]) =
2   println(b.content.name)
```

- Print some concrete boxes

```
1 val birdBox: ImmutableBox[Bird] = ImmutableBox(Bird("Feathers"))
2 val catBox: ImmutableBox[Cat] = ImmutableBox(Cat("Fluffy"))
3
4 printAnimalName(birdBox)
5 printAnimalName(catBox)
```

Scala Lists are Covariant

❓ What is the practical relevance?

```scala
def printAnimalNames(animals: List[Animal]) =
  for a <- animals do println(a.name)
```

- Print some concrete lists

```scala
val birds: List[Bird] = List(Bird("Feathers"), Bird("Chirpy"))
val cats: List[Cat]   = List(Cat("Fluffy"), Cat("Whiskers"))

printAnimalNames(birds)
printAnimalNames(cats)
```

- 💡 `List` must be covariant to allow `printAnimalNames` with concrete arguments of type `List[Bird]` and `List[Cat]`!

Scala Arrays are Invariant

```
1 def printAnimalNames(animals: Array[Animal]) =
2   for a <- animals do println(a.name)
3
4 val birds: Array[Bird] = Array[Bird](Bird("Feathers"), Bird("Chirpy"))
5 printAnimalNames(birds)
```

- Compile error

```
1 printAnimalNames(birds)
2                  ^^^^^
3                  Found: (birds : Array[Bird])
4                  Required: Array[Animal]
```

- Scala and Java arrays are readable and writable
  - Scala arrays are invariant: compile error when writing to alias
  - Java arrays are covariant: runtime exception when writing to alias

Covariant Wildcard

❓ How can we still print arrays generically?

- Use a covariant wildcard to make array read-only

```
1 def printAnimalNames(animals: Array[? <: Animal]) =
2   for a <- animals do println(a.name)
3
4 val birds: Array[Bird] = Array[Bird](Bird("Feathers"), Bird("Chirpy"))
5 printAnimalNames(birds) // compiles
```

- Cannot write to `Array[? <: Animal]`

```
1 animals(0) = Cat("Fluffy")
2                ^^^^^^^^^^^^
3                Found:    Cat
4                Required: animals.T
```

What about Writing?

```
1 def putCat(box: Box[Cat]) =
2   box.content = Cat("Fluffy")
```

- Unable to pass in a box of animal

```
1 val animalBox: Box[Animal] = Box(Cat("Whiskers"))
2 putCat(animalBox)
3         ^^^^^^^^^
4         Found:    (animalBox : Box[Animal])
5         Required: Box[Cat]
```

- Because the animal box could be a bird box

```
1 val animalBox: Box[Animal] = Box(Bird("Feathers"))
2 putCat(animalBox)
```

Contravariance

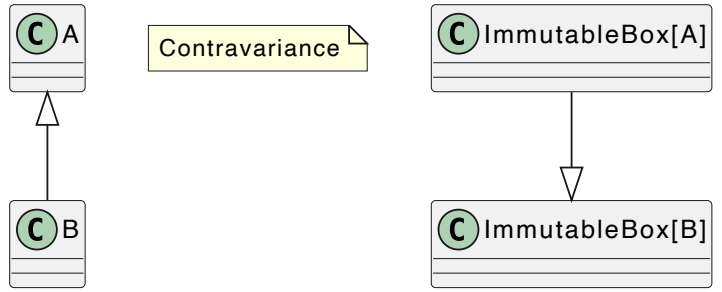
❓ What if we disallow reading?

```
1 class MutableBox[-X](content: X):
2   private var x: X = content
3   def put(elem: X) : Unit = x = elem
```

- Create and alias boxes

```
1 def putCat(box: MutableBox[Cat]) = box.put(Cat("Fluffy"))
2 val animalBox: MutableBox[Animal] = new MutableBox(Cat("Whiskers"))
3 putCat(animalBox)
```

- `MutableBox` is **contravariant** in `X`
- Contravariance type annotation `-X` can only be applied to write-only structures



Contravariance allows Degraded Read

```
1 class MutableBox[-X](content: X):
2   private var x: X = content
3   def put(elem: X) : Unit = x = elem
4   def get : Any = x
5
6 def putCat(box: MutableBox[Cat]) = box.put(Cat("Fluffy"))
```

- Create box and modify it

```
1 val animalBox: MutableBox[Animal] = new MutableBox(Bird("Feathers"))
2 var animal: Any = animalBox.get
3
4 putCat(animalBox)
5 animal = animalBox.get
```

Contravariant Wildcard

```
1 def addFluffy(cats: List[Cat]): List[Cat] =
2   Cat("Fluffy") :: cats
```

- Add to a list of cats

```
1 val cats: List[Cat] = List(Cat("Whiskers"))
2 val moreCats: List[Cat] = addFluffy(cats) // ok
```

- Add to a list of animals

```
1 val animals: List[Animal] = List(Bird("Feathers"), Cat("Whiskers"))
2 addFluffy(animals)
3           ^^^^^^^
4           Found:    (animals : List[Animal])
5           Required: List[Cat]
```

Contravariant Wildcard

```
1 def addFluffy(cats: List[? >: Cat]) =
2   Cat("Fluffy") :: cats
```

- Contravariant wildcard `? >: X`
- Add to a list of animals

```
1 val animals: List[Animal] = List(Bird("Feathers"), Cat("Whiskers"))
2 val more: List[Any] = addFluffy(animals)
```

- ❓ Can we do better than `List[Any]`?

Contravariant Type Parameter

- Use a contravariant type parameter rather than a wildcard

```
1 def addFluffy[X >: Cat](cats: List[X]): List[X] =
2   Cat("Fluffy") :: cats
```

- Add to a list of animals

```
1 val animals: List[Animal] = List(Bird("Feathers"), Cat("Whiskers"))
2 val moreAnimals: List[Animal] = addFluffy(animals)
```

Unusual Use Case: Programming with Types

Peano arithmetic in types

```
1 trait Nat
2 class _0 extends Nat
3 class Succ[A <: Nat] extends Nat
4
5 type _1 = Succ[_0]
6 type _2 = Succ[_1]
7 // ...
```

Comparison types

```
1 trait <[A <: Nat, B <: Nat]
```

```
1 object < {
2   // base case: 0 is less than any successor
3   given [B <: Nat] => <[_0, Succ[B]]
4   // inductive case: if A < B then Succ[A] < Succ[B]
5   given [A <: Nat, B <: Nat] => <[A, B] => <[Succ[A], Succ[B]]
6   // helper method to summon an instance (a proof)
7   def apply[A <: Nat, B <: Nat](using lt: <[A, B]) = lt
8 }
```

- Compiler infers type of `p` given types in `object <`

```
1 val p = <[_0, _1]   // 0 < 1
2 // <[_1, _0] // 1 < 0 does not compile
```

Summary

- *Subtype polymorphism*: `B<:A` (`B` is a subtype of `A`)
- *Parametric polymorphism*: `T[X]` parameterize class `T` with type parameter `X`
- Covariant, contravariant, and invariant parametric types for `B<:A`

	Covariant	Contravariant	Invariant
	Readable	Writable	Read-write
Relationship	T[B] <: T[A]	T[A] <: T[B]	T[A]<>T[B]
Box	ImmutableBox[X]	MutableBox[X]	Box[X]
Collection	List[X]		Array[X]
Annotations	T[+X]	T[-X]	T[X]
Functions	Unit=>X	X=>Unit	X=>X

Java Arrays are Covariant

```
1 public static void main (String[] args) {
2   B[] bs = new B[] { new B (), new B () };
3   A[] as = bs;       // OK, because covariant
4   as[0] = new A (); // ArrayStoreException
5   bs[0].g();
6 }
```

```
1 $ javac Driver.java
2 $ java Driver
3 Exception in thread "main" java.lang.ArrayStoreException: A
4   at Driver.main(Driver.java:5)
```

- Every assignment to object array dynamically checked!

Why?

- For example, to sort

```
1 static void sort(Object[] xs) { ... }
2 String[] ss = ...;
3 sort(ss); // requires covariance
```

- With Java Generics

```
1 static <X extends Comparable<? super X>> void sort(X[] xs) { ... }
```

- In Scala

```
1 def sort[X <: Comparable[? >: X]] (xs: Array[X]) = ...
```

- Use

```
1 class A implements Comparable<A>{}
2 class B extends A {}
3 B[] bs = ...;
4 sort(bs); // B's are comparable as A's
```

Variance Annotations Example

```scala
trait Source[+X] { def get    () : X    } // Covariant: read-only
trait Sink  [-X] { def put (x:X) : Unit } // Contravariant: write-only
```

```scala
class Ref   [ X] (var contents:X)         // Invariant: read-write
  extends Source[X] with Sink[X]:
    def get ()    = contents
    def put (x:X) = contents = x
```

- Create type hierarchy

```scala
abstract class Animal { def name: String }
class Bird(val name: String) extends Animal
class Cat(val name: String) extends Animal
class Duck(name: String) extends Bird(name)
```

- Create aliases

```scala
val ref: Ref[Bird] = Ref(Bird("Feathers"))
val src: Source[Animal] = ref
val snk: Sink[Duck] = ref
```

- Write to `snk`, read from `ref` and `src`

```scala
val d = Duck("Quack")
snk.put(d)
val r = ref.get()
val s = src.get()
```

* What are the static and dynamic types of `d`, `r`, and `s`?

Variance Annotations Example

```
1 trait Function[-X,+Y] { def apply(x:X) : Y    } // Contravariant in X, covariant in Y
```

- Covariant use

```
1 // List[Int] <: Seq[Int], so Function[Int,List[Int]] <: Function[Int,Seq[Int]]
2 val increment: Function[Seq[Int],Seq[Int]] = new Function[Seq[Int],List[Int]]:
3   def apply(x: Seq[Int]) : List[Int] = x.map(_ + 1).toList
4 // Int <: Any, so Function[Seq[Int],Int] <: Function[Seq[Int],Any]
5 val length: Function[Seq[Int],Any] = new Function[Seq[Int],Int]:
6   def apply(x: Seq[Int]) : Int = x.length
```

- Contravariant use

```
1 def chain(f: Function[List[Int],Seq[Int]], g: Function[Seq[Int],Any]) =
2   (x:List[Int]) => g(f(x))
3 chain(increment, length)
```

Variance Annotations

```
1 trait Producer[+Y]     { def apply ()     : Y    } // Covariant
2 trait Consumer[-X]     { def apply (x:X) : Unit } // Contravariant
3 trait Function[-X,+Y] { def apply (x:X) : Y    } // Both
4 trait Operator[X]      { def apply (x:X) : X    } // Invariant
```

```
1 trait Producer[-Y]     { def apply ()     : Y    }
2                                ^
3 error: contravariant type Y occurs in covariant position
```

```
1 trait Consumer[+X]     { def apply (x:X) : Unit }
2                                     ^
3 error: covariant type X occurs in contravariant position
```