CSC 347 - Concepts of Programming Languages

Subtyping

Instructor: Stefan Mitsch

Learning Objectives

❓ What should the type relationship between parameterized types be?

- Understand read and write restrictions on parametric types

Checked Casts

- Recall lecture on safety

```
 1 class A { int x; }
 2 class B extends A { float y; }
 3 class C extends A { char c; }
 4
 5 void f (B b) {
 6   A a = b;        // upcast always safe
 7 }
 8 void g (A a) {
 9   B b = (B) a;    // downcast must be checked
10 }
11 f (new B()); // OK
12 g (new C()); // ClassCastException
```

❓ What makes upcasting safe? What makes downcasting unsafe?

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

- Subtyping relation `<:` is a *partial order* on types
  - *reflexive*: `X<:X`
  - *transitive*: if `Duck<:Bird` and `Bird<:Animal` then `Duck<:Animal`

Subtyping Order: Top

- 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
- In Scala: `scala.AnyRef` above reference types

```
1 import java.io.FileInputStream
2
3 val xs:List[AnyRef] = List ("hello", FileInputStream ("a.txt"))
4 val ys:List[Any]    = List ("hello", 1)
```

Subtyping Order: Bottom

- Most PLs do not have a `Bottom` type: `Bottom<:X` for all `X` (less than all other types)
- Recall Scala type hierarchy
- In Scala: `Bottom` is `scala.Nothing`
- Important for typing uses of `Nil`
  - `Nil:List[Nothing]`
  - `List[Nothing]<:List[X]`

```
1 val mynil1:List[Int]     = Nil
2 val    xs1:List[Any]     = "hello"::mynil1 // Best type possible
3
4 val mynil2:List[Nothing] = Nil
5 val    xs2:List[String]  = "hello"::mynil2 // Best type possible
```

Scala Lists

```
1 class A {  def f () = 1 }
2 class B extends A:
3   override def f () = 2
4            def g () = 3
5
6 var as:List[A] = List (A(), B()) // OK, because B <: A
7 var bs:List[B] = List (B(), B()) // OK
```

```
1 as = bs     // ok?
2 as(1).f()   // result: 2
```

❓ Why is `as = bs` ok?

- Ok, because `List[B] <: List[A]`
- `List` is *covariant*
- If `B<:A` then `List[B]<:List[A]`
- Generally, type constructor `T[-]` is *covariant* if `B<:A` implies `T[B]<:T[A]`

Scala Arrays

```
1 class A {  def f () = 1 }
2 class B extends A:
3   override def f () = 2
4            def g () = 3
5
6 var as:Array[A] = Array (A(), B()) // OK
7 var bs:Array[B] = Array (B(), B()) // OK
```

```
1 as = bs
2      ^
3 error: type mismatch;
4        found   : Array[B]
5        required: Array[A]
6   Note: B <: A, but class Array is invariant in type T.
7   You may wish to investigate a wildcard type such as '? <: A'
```

❓ Why?

Invariance

```
1 class A {  def f () = 1 }
2 class B extends A:
3   override def f () = 2
4            def g () = 3
5
6 var as:Array[A] = Array (A(), B()) // OK
7 var bs:Array[B] = Array (B(), B()) // OK
```

```
1 as = bs          // ERROR, because Array[B] NOT <: Array[A]
2 as(0) = A()      // OK, because as:Array[A]
3 bs(0).g()        // Unsafe access
```

- Covariance only safe for *read only* structure

Wildcard?

```
1 class A {  def f () = 1 }
2 class B extends A:
3   override def f () = 2
4            def g () = 3
5
6 var as:Array[? <: A] = Array (A(), B()) // OK
7 var bs:Array[B]       = Array (B(), B()) // OK
```

```
1 as = bs          // OK, because Array[B] <: Array[? <: A]
2 as(0) = A()
3         ^
4 error: type mismatch;
5        found   : A
6        required: ?1.T where ?1 is an unknown value of type Array[? <: A]
```

- Covariance only safe for *read only* structure, no way to write an array of type `Array[? <: A]`

Wildcard?

```
1 class A {  def f () = 1 }
2 class B extends A:
3   override def f () = 2
4            def g () = 3
5
6 var as:Array[A]       = Array (A(), B()) // OK
7 var bs:Array[? >: B] = Array (B(), B()) // OK
```

```
1 bs = as          // OK, because Array[A] <: Array[? >: B]
2 bs(0) = B()      // OK
3 bs(0).g()        // Unsafe access
4        ^
5 error: value g is not a member of Array[? >: B]#T
```

- Contravariance safe for *write only* structure, degrade the type of read

Wildcard?

```
1 class A {  def f () = 1 }
2 class B extends A:
3   override def f () = 2
4            def g () = 3
5
6 var as:Array[A]       = Array (A(), B()) // OK
7 var bs:Array[? >: B] = Array (B(), B()) // OK
```

```
1 bs = as          // OK, because Array[A] <: Array[? >: B]
2 bs(0) = B()      // OK
3 bs(0)            // OK
```

```
1 val res1: Array[? >: B]#T = B@d271a54
```

- Contravariance safe for *write only* structure, degrade the type of read

Java Arrays Covariant

```
1 public class Driver {
2   public static void main (String[] args) {
3     B[] bs = new B[] { new B (), new B () };
4     A[] as = bs;       // OK, because covariant
5     as[0] = new A (); // ArrayStoreException
6     bs[0].g();
7   }
8 }
```

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

- Java arrays are covariant
- 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
```

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

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

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

Three Types

```
1 class A {}            // Animal
2 class B extends A {} // Bird
3 class D extends B {} // Duck
```

Variance Annotations

```
1 trait Source[+X] { def get    () : X    } // Covariant
2 trait Sink  [-X] { def put (x:X) : Unit } // Contravariant
3 class Ref   [ X] (var contents:X)         // Invariant
4   extends Source[X] with Sink[X] {
5     def get ()    = contents
6     def put (x:X) = contents = x
7 }
```

```
1 val ref : Ref   [B] = Ref[B] (B())
2 val src : Source[A] = ref
3 val snk : Sink  [D] = ref
```

```
1 val d = D()
2 snk.put(d)
3 val r = ref.get()
4 val s = src.get()
```

```
1 val d: D = D@595713f3
2 val r: B = D@595713f3
3 val s: A = D@595713f3
```

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
```

Variance when `B<:A`

	Covariant	Contravariant	Invariant
Relationship	T[B]<:T[A]	T[A]<:T[B]	T[A]=T[B]
Collection	List[X]		Array[X]
Annotations	T[+X]	T[-X]	T[X]
Functions	Unit=>X	X=>Unit	X=>X

Summary

- *Subtype polymorphism*: `A<:B` (`A` is a subtype of `B`)
- *Parametric polymorphism*: `T[X]` parameterize class `T` with type parameter `X`
- Outside the scope of this course
  - Bounded polymorphism (Java, C#, Scala, Flow)
  - Java's use-site variance and wildcards
  - Adhoc polymorphism, typeclasses, implicit params
  - Check out Scala and Typescript