Advanced generics

A general-purpose function for printing an array of objects?

```
fun printObjects(objects: Array<Any>) {
    objects.forEach {
        println(it)
    }
}
```

This should allow us to print an arbitrary array of non-null object references, right?
Every class and interface is a subtype of `Any`

Not true! (False)

```
fun printObjects(objects: Array<Any>) {
    objects.forEach {
        println(it)
    }
}

fun main() {
    val myStrings: Array<String> = arrayOf("Hello", "World")
    printObjects(myStrings)
}
```

Compiler error: Type mismatch.
Required: Array<Any>
Found: Array<String>

Generics and subtyping

`String` is a subtype of `Any`
However, `Array<String>` is not a subtype of `Array<Any>`

More generally: if U is a (strict) subtype of T, `Array<U>` is not a subtype of `Array<T>`

Even more generally: if C is a generic class or interface and U is a (strict) subtype of T, then `C<U>` is not necessarily a subtype of `C<T>`

Why?

Why can't `Array<String>` be a subtype of `Array<Any>`?

Consider this function:

```
fun replaceFirst (objects: Array<Any>, replacement: Any) {
    objects[0] = replacement
}
```

`Int` is a subtype of `Any`, so we can pass an `Int` as `replacement`

If `Array<String>` were a subtype of `Array<Any>`, we could pass an `Array<String>` as `objects`

We would then be trying to plug an Int into an array of Strings – **not OK**

Fortunately, this is not allowed in Kotlin

```
fun replaceFirst(objects: Array<Any>, replacement: Any) {
    objects[0] = replacement
}

fun main() {
    val myStrings: Array<String> = arrayOf<String>(
        "Hello",
        "World",
    )
    replaceFirst(myStrings, 42)
}
```

Compiler error: Type mismatch.
Required: Array<Any>
Found: Array<String>

Unfortunately, this is allowed in Java

Arrays in Java are declared via `[]`

`Object` in Java is like `Any?` in Kotlin

```
public class ArrayExample {

    static void replaceFirst(Object[] objects, Object replacement) {
        objects[0] = replacement;
    }

    public static void main(String[] args) {
        final String[] myStrings = { "Hello", "World" };
        replaceFirst(myStrings, 42);
    }
}
```

A static method is not invoked on any particular object – a bit like a Kotlin top-level function (but in Java every method must be declared inside a class or interface)

```
public class ArrayExample {

    static void replaceFirst(Object[] objects, Object replacement) {
        objects[0] = replacement;
    }

    public static void main(String[] args) {
        final String[] myStrings = { "Hello", "World" };
        replaceFirst(myStrings, 42);
    }
}
```

In Java, `String[]` is a subtype of `Object[]`, so this is allowed

What happens at runtime? The attempt to store an `Integer` in a `String` array leads to an `ArrayStoreException`

Back to Kotlin

Easy to support printing arbitrary arrays of objects:

```
fun <T> printObjects(objects: Array<T>) {
    objects.forEach {
        println(it)
    }
}

fun main() {
    val myStrings: Array<String> = arrayOf("Hello", "World")
    printObjects(myStrings)
}
```

`T` is some unspecified type – the only guarantee is that `T` is a subtype of `Any?`

`printObjects` can work on an `Array<T>` for any `T` – here we give an `Array<String>`, which is fine

Wildcards

Because we do not care what T is here, we can use a wildcard:

```
fun printObjects(objects: Array<*>) {
    objects.forEach {
        println(it)
    }
}
```

`*` is a wildcard: it is useful when we do not want to use the type anywhere else in our code, so we do not need a type variable

Good replacement in an array

Easy to support replacement in an array with an element of same type:

```
fun <T> replaceFirst(objects: Array<T>, replacement: T) {
    objects[0] = replacement
}

fun main() {
    val myStrings: Array<String> = arrayOf("Hello", "World")
    printObjects(myStrings)
    replaceFirst(myStrings, "Goodbye")
}
```

Bad replacement in an array

Replacing using an element of an inappropriate type does not work:

```
fun <T> replaceFirst(objects: Array<T>, replacement: T) {
    objects[0] = replacement
}

fun main() {
    val myStrings: Array<String> = arrayOf("Hello", "World")
    printObjects(myStrings)
    replaceFirst(myStrings, 42)
}
```

Compiler error: The integer literal does not conform to the expected type String

`Array<T>` vs. `Array<Any?>`

`Array<T>` represents an array whose elements have some type, but we have no idea what the type is
We can only store a reference of type `T` in an `Array<T>`

`Array<Any?>` represents a completely flexible array that can contain anything: nulls, references to `String`s, references to `Int`s
We can store anything we like in an `Array<Any?>`

Bounded generics

Next, we will study how to write code that is generic with respect to types that are subtypes of certain other types

To help with examples, let's use a good old `Shape` inheritance hierarchy:

```
abstract class Shape {
    abstract val area: Double
}

class Rectangle : Shape() { … }

class Circle : Shape() { … }
```

Finding the largest area among an array of shapes

```
fun largestArea(shapes: Array<Shape>): Double =
    shapes.map(Shape::area).max()
```

This works if we have precisely an `Array<Shape>`:

```
fun main() {
    val myShapes: Array<Shape> = arrayOf(
        Circle(…), Circle(…), Rectangle(…),
    )
    println(largestArea(myShapes))
}
```

But it does not work if we have e.g. an `Array<Circle>`:

```
fun main() {
    val myCircles: Array<Circle> = arrayOf(
        Circle(…), Circle(…),
    )
    println(largestArea(myCircles))
}
```

Compile error: Type mismatch.
Required: Array<Shape>
Found: Array<Circle>

What is the problem?

Although `Circle` is a subtype of `Shape`, `Array<Circle>` is not a subtype of `Array<Shape>`

We could not use an `Array<String>` where an `Array<Any>` was required

Similarly, we cannot use an `Array<Circle>` where an `Array<Shape>` is required

A plain generic type `T` does not work

```
fun <T> largestArea(shapes: Array<T>): Double =
    shapes.map(Shape::area).max()
```

Compile error: Type mismatch.
Required: Shape
Found: T

```
fun main() {
    val myCircles: Array<Circle> = arrayOf(
        Circle(…), Circle(…),
    )
    println(largestArea(myCircles))
}
```

Compiler is happy:
`Array<Circle>` matches `Array<T>`

Solution: bounded generic

Solution: bounded generic

This says that T is some subtype of Shape – i.e. Shape is an **upper bound** for T

```
fun <T : Shape> largestArea(shapes: Array<T>): Double =
    shapes.map(Shape::area).max()
```

Compiler is happy: T is some subtype of Shape, so it has an area method

```
fun main() {
    val myCircles: Array<Circle> = arrayOf(
        Circle(…), Circle(…),
    )
    println(largestArea(myCircles))
}
```

Compiler is happy: Array<Circle> matches Array<T>

Finding the largest shape

Finding the largest shape

This does not just return a Shape? – it returns a T? where T is some particular subtype of Shape

```
fun <T : Shape> largestShape(shapes: Array<T>): T? {
    var result: T? = null
    for (shape in shapes) {
        if (result == null || shape.area > result.area) {
            result = shape
        }
    }
    return result
}

fun main() {
    val myCircles: Array<Circle> = arrayOf(Circle(…), Circle(…))
    val largestCircle: Circle? = largestShape(myCircles)
}
```

We put in an Array<Circle>

We get back a Circle?

Declaring a non-null type parameter

To require only that a type parameter is non-null, use Any as an upper bound:

```
class NonNullPair<S : Any, T : Any>(
    val first: S,
    val second: T,
)
```

S and T can be arbitrary types, except that they must be subtypes of Any

Since Any is not nullable, S and T are required to be non-nullable

Need multiple upper bounds?

Use where:

```
fun <T> sortPageElements(pageElements: List<T>)
        where T : Comparable<T>,
              T : PageElement {
    ...
}
```

This specifies two bounds for T: T must be:

- Comparable with other Ts
- A kind of PageElement

`List<String>` is a subtype of `List<Any>`

Recall that this does not compile:

```
fun printObjects(objects: Array<Any>) {
    objects.forEach {
        println(it)
    }
}

fun main() {
    val myStrings: Array<String> = arrayOf("Hello", "World")
    printObjects(myStrings)
}
```

However, this is allowed:

```kotlin
fun printObjects(objects: List<Any>) {
    objects.forEach {
        println(it)
    }
}

fun main() {
    val myStrings: List<String> = listOf ("Hello", "World")
    printObjects(myStrings)
}
```

What is special about `List<T>` ?

A question – not meant to indicate nullable!

```kotlin
public interface List<out T> : Collection<T> {
    ...
}
```

This says:

- `List` is generic with respect to a type parameter `T`
- `T` can only be used in **out** positions in the members of `List`

Out positions are positions that allow `List` to **produce** a `T`, and include:

- return types
- `val` properties

After construction, List cannot **consume** `T`s: `T` cannot be used as a method parameter type or `var` property type

Constructor parameters of type `T` are allowed

`List<out T>` can only produce `T` s

Given access to a `List<out T>` we can obtain Ts by:

- Calling methods that return T directly (e.g. get)
- Calling methods or accessing properties that return T indirectly (e.g. `subList`, `iterator`)

There is no way to put a T into a `List<out T>`

- This would require passing a T as a parameter or setting a var property

- Not allowed: T can only be used in out positions

More generally …

A class or interface `C<out T>` can only **produce** `T`s

- Methods of `C` can return `T`, `T?`, `D<out T>`, etc.

`C<out T>` cannot **consume** elements of type `T`

- This would require receiving `T`s via parameters, via producers of `T`s, or via `var` property of type `T`
- **Not allowed:** `T` can only be used in **out** positions

`D` is some other class or interface that produces `T` s

Subtyping is covariant with respect to `out` type parameters

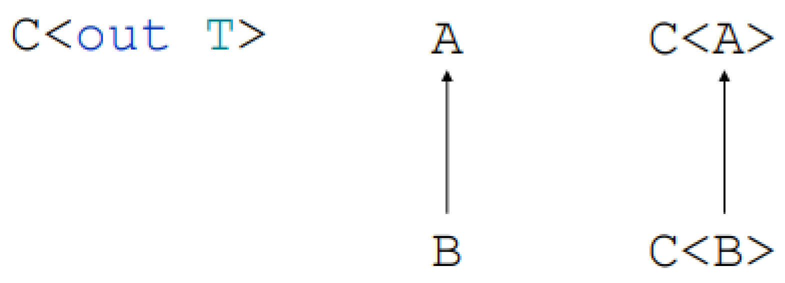
Intuition:

- A client of `List<Any>` expects to obtain `Any` s from a list
- If we give the client a `List<String>` it will obtain `String` s
- A `String` is an `Any` , so this is fine!

Rule:

- Suppose we have a generic class or interface `C<out T>`
- Let B be a subtype of A
- Then `C<B>` is a subtype of `C<A>`

Subtyping is covariant with respect to out type parameters



Subtyping is covariant here (same direction)

This is called declaration-site covariance: `out` occurs where the generic parameter is declared

`List<String>` is a subtype of `List<Any>`

Because `List` is declared as `List<out T>`, this is allowed:

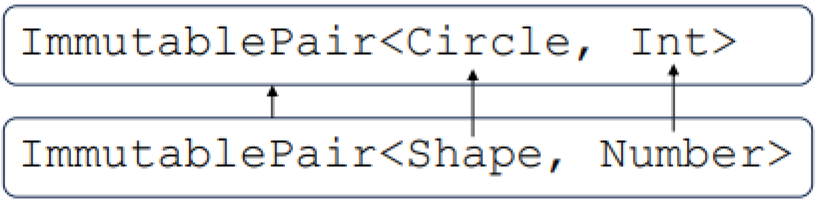
```
fun printObjects(objects: List<Any>) {
    objects.forEach {
        println(it)
    }
}

fun main() {
    val myStrings: List<String> = listOf ("Hello", "World")
    printObjects(myStrings)
}
```

Example: immutable pair

```
class ImmutablePair<out S, out T>(
    val first: S,
    val second: T,
)
```

Fine: S and T are only used in **out** positions



Arrows are wrong way round!

Example: mutable pair

```
class MutablePair<out S, out T>(
    var first: S,
    var second: T,
)
```

Compile error: Type parameter S is declared as 'out' but occurs in 'invariant' position
Compile error: Type parameter T is declared as 'out' but occurs in 'invariant' position

Declaration-site contravariance

This does not compile:

```kotlin
interface Order<T> {
    fun lessThan(first: T, second: T): Boolean
}

fun sortStrings(strings: List<String>, order: Order<String>) {
    …
}

fun main() {
    val strings: List<String> = listOf("Hello", "There", "John")
    val anyOrder = object : Order<Any> {
        override fun lessThan(first: Any, second: Any): Boolean =
            first.hashCode() < second.hashCode()
    }
    sortStrings(strings, anyOrder)
}
```

Compile error: Order<Any> is not a subtype of Order<String>

However, this does compile:

```kotlin
interface Order<in T> {
    fun lessThan(first: T, second: T): Boolean
}

fun sortStrings(strings: List<String>, order: Order<String>) {
    …
}

fun main() {
    val strings: List<String> = listOf("Hello", "There", "John")
    val anyOrder = object : Order<Any> {
        override fun lessThan(first: Any, second: Any): Boolean =
            first.hashCode() < second.hashCode()
    }
    sortStrings(strings, anyOrder)
}
```

Compiler is happy: thanks to i
is not a subtype of Order<Strin

Compiler is happy: thanks to `in`, `Order<Any>` is a subtype of `Order<String>`

```
interface Order<in T> {
    ...
}
```

This says:

- `Order` is generic with respect to a type parameter `T`
- `T` can only be used in **in** positions in the members of `Order`

In positions are positions that allow `Order` to **consume** `T`s, and include:

- parameter types

`T` cannot be used in places that would **produce** `T`s, such as method return types or property types

`Order<in T>` can only consume `T`s

Given access to an `Order<in T>` we can pass in `T`s by:

- Calling methods that accept `T` directly or indirectly

There is no way to get a T from an `Order<in T>`

- This would require a `T` being returned from a method or provided by a property
- Not allowed: T can only be used in **in** positions

More generally …

A class or interface `C<in T>` can only consume `T`s

- Methods of `C` can take parameters of type `T`, `T?`, `D<in T>`

`C<in T>` cannot produce elements of type `T`

- This would require returning values of type `T`, providing properties of type `T`, or consuming producers of type `T`
- **Not allowed:** `T` can only be used in **in** positions

Subtyping is contravariant with respect to in type parameters

`C<in T>`

$$\begin{tikzcd} A & C\langle A\rangle \arrow[d] \\ B \arrow[u] & C\langle B\rangle \end{tikzcd}$$

Subtyping is contravariant here (opposite direction)

This is called **declaration-site contravariance**: `in` occurs where the generic parameter is declared

Without in or out, subtyping is invariant

`C<T>`

$$\begin{tikzcd} A & C\langle A\rangle \arrow[d, shift left, "\times"] \\ B \arrow[u] & C\langle B\rangle \arrow[u, shift left, "\times"] \end{tikzcd}$$

No subtyping (unless `A` and `B` are the same)

```
// We don't care what these are, but they would potentially produce
or consume Ts.
class ExistingProducer<out T>
class ExistingConsumer<in T>
// Both a producer and a consumer
class ExistingInvariant<T>
class Producer<out T>(
    val something: T, // Fine: reading "something" will produce a T
    //var somethingElse: T, // Not OK: writing to "somethingElse"
would consume a T
) {
    fun foo(): T = something // Fine: returning a T is a case of
"producing"
    //fun bar(input: T) { } // Not OK: this would consume a T
    //fun baz(higherOrder: () -> T) { } // Not OK: because we could
consume a T by calling "higherOrder"
    // This is fine. If we *had* a T, we could then pass it to
"higherOrder".
    // But we don't *obtain* a T via this function call.
    fun zap(higherOrder: (T) -> String) { }
```

```
    // This is fine: it allows us to produce Ts by yielding another
producer of Ts.
    // No possibility of consuming Ts.
    fun zop(): ExistingProducer<T>? {
        return null // In reality this would return something more
interesting
    }
    // This is not OK: I might have a reference to this consumer.
Then you can put
    // things into the consumer, and I can get them via my
reference. Gives me an indirect way
    // to consume Ts.
    // See the ExampleProducer class below for more details.
//    fun xox(): ExistingConsumer<T>? {
//        return null // In reality this would return something
more interesting
//    }
    // This is fine. It gives me a consumer of Ts. I can put things
into that consumer.
    // But the consumer does not produce Ts, so I cannot consume Ts
from it!
    fun yoy(consumer: ExistingConsumer<T>) {
    }
    // Not OK: this returns something that can both consume and
produce Ts,
    // therefore is has the same problems as xox above
//    fun wow(): ExistingInvariant<T>? {
//        return null // This would be more interesting in practice
//    }
}
fun main() {
    val myList: List<String> = emptyList()
    // Why does this work? Doesn't this consume a string? Take a
look at
    // Collections.kt to see the hack that allows this.
    myList.contains("Hello")
    // Why is this allowed? Because it returns a *copy* - take a
look at the implementation of toMutableList to see this
    myList.toMutableList()
}
class ProducerExample<out T> {
    // This is allowed: even though T is used in a property
declaration, the property is private.
    private val captured: MutableList<T> = mutableListOf()
```

```
    // This is allowed. Although it reveals the mutable list, it
reveals it with only its "producing" functions
    // available, because the return type is MutableList<out T>
    fun revealCapturedAsProducer(): MutableList<out T> = captured
    // This is not allowed - this would return a reference to
"captured" that could *consume* Ts.
    //fun revealCapturedAsConsumer(): MutableList<in T> = captured
}
```

Array subtyping is invariant

T occurs in an **out** position

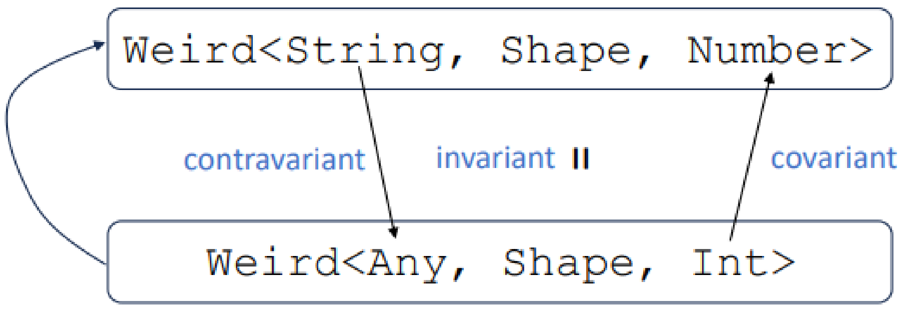
```
class Array<T> {
    operator fun get(index: Int): T { … }
    operator fun set(index: Int, value: T) { … }
    …
}
```

T occurs in an **in** position

The declaration site for T must be **invariant**, because T is used in both **in** and **out** positions – an Array<T> can both consume and produce Ts

A generic class can use a mixture of in, out and invariant type parameters

```
class Weird<in S, T, out U>
```



More for theoretical interest than of practical importance!

Declaration-site variance is limited

Consider this function to copy certain elements from one array to another

```
fun <T> copyElements(
    source: Array<T>,
    destination: Array<T>,
    elementsToCopy: Int,
) {
    for (i in 0..<elementsToCopy) {
        destination[i] = source[i]
    }
}
```

We cannot use this to copy from an Array<String> to an Array<Any>

```
fun main() {
    val strings = arrayOf("A", "B", "C")
    val anys = arrayOf("E", 1, Unit, 3.0, emptyList<Int>())
    copyElements<Any>(strings, anys, 3)
}
```

Compiler error: Type mismatch.
Required: Any
Found: String

We cannot use this to copy from an Array<String> to an Array<Any>

```
fun main() {
    val strings = arrayOf("A", "B", "C")
    val anys = arrayOf("E", 1, Unit, 3.0, emptyList<Int>())
    copyElements<String>(strings, anys, 3)
}
```

Compiler error: Type mismatch.
Required: String
Found: Any

We cannot use this to copy from an Array<String> to an Array<Any>

Intuitively, we **should** be able to perform such a copy, which involves:

- Reading Strings from an Array<String> array
- Writing Strings (which are Anys) to an Array<Any>

Use-site variance solves the problem

We can use in and out at the use site of a generic class or interface, to restrict access to the methods and properties that are available

```
fun <T> copyElements(
    source: Array<out T>,
    destination: Array<in T>,
    elementsToCopy: Int,
) {
    for (i in 0..<elementsToCopy) {
        destination[i] = source[i]
    }
}
```

`source` is restricted to **producing** `T`s

`destination` is restricted to **consuming** `T`s

`destination` **consumes a** `T`

`source` **produces a** `T`

Use-site variance

`Array<out T>` projects only those methods and properties in which T occurs in an out position
This means e.g. the `get` operator is available on `source`, but the `set` operator is not
`[]` calls `get`, which is available

`Array<in T>` projects only those methods and properties in which T occurs in an in position
This means e.g. the set operator is available on destination, but the get operator is not
Remember that `[]` calls set, which is available

Covariance and contravariance with use-site variance

Covariance: If a method expects an `Array<out T>` parameter, it is fine to provide an `Array<U>` parameter where U is a subtype of T
Contravariance: If a method takes an `Array<in T>` parameter, it is fine to provide an `Array<S>` parameter where S is a supertype of T
Array is just an example – this applies to generic classes and interfaces in general

Array copying now works!

```
fun <T> copyElements(
    source: Array<T>,
    destination: Array<T>,
    elementsToCopy: Int,
) {
    for (i in 0..<elementsToCopy) {
        destination[i] = source[i]
    }
}

fun main() {
    val strings = arrayOf("A", "B", "C")
    val anys = arrayOf("E", 1, Unit, 3.0, emptyList<Int>())
    copyElements(strings, anys, 3)
}
```

The type system protects us against mistakes

```
fun <T> copyElements(
    source: Array<T>,
    destination: Array<T>,
    elementsToCopy: Int,
) {
    for (i in 0..<elementsToCopy) {
        destination[i] = source[i]
    }
}

fun main() {
    val strings = arrayOf("A", "B", "C")
    val anys = arrayOf("E", 1, Unit, 3.0, emptyList<Int>())
    copyElements(anys, strings, 3)
}
```

Exercise: why doesn't this compile?

Implementing or extending a generic interface and remaining generic

`MutableList` is a generic interface

```
class ArrayList<T> : MutableList<T> {
    ...
}
```

`ArrayList` is a generic class that implements this interface

Implementing a generic interface using concrete types

`Comparable` is a generic interface

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}
```

`compareTo` in `CountryCode` is **specialised** to accept another `CountryCode`

```
class CountryCode : Comparable<CountryCode> {
    override fun compareTo(other: CountryCode): Int = ...
}
```

`CountryCode` is a non-generic class that implements `Comparable` specifically for `CountryCodes`

Type erasure

Java bytecode does not contain information about generic types

Why?

Because Java bytecode was designed before Java had generics!

Java bytecode is the foundation of the Java ecosystem – it is very difficult to change it

Remember that **backwards-compatibility** is critical in the context of widely-used software

Generics are great:

- They allow strong static checks by the compiler to ensure that you are using types properly
- Without generics, we would have to use `Any` everywhere and downcast all over the place
- Generics avoid `ClassCastExceptions` that would arise from mistakes with downcasting

But: when it generates bytecode, the compiler discards all of the generic information that you lovingly provided!

Consequences of type erasure

You cannot create an instance of a T, where T is a generic parameter:

```
val myT: T = T() // Does not compile
```

Due to type erasure, we don't know at runtime what T is

You cannot ask whether an expression is an instance of a generic type parameter T:

```
if (x is T) { // Does not compile

}
```

After type erasure, the byte code contains no information about T

You cannot ask whether the actual type of an object is some specific generic type:

```
if (x is List<Int>) { // Does not compile
    ...
}
```

Due to type erasure, whether x was a `List<Int>`, `List<String>`, etc., is unknown

```
if (x is List<*>) {
    ...
}
```

This is the best you can do:

In Java, it is not possible to create an array of generic element type T:

```
T[] myArray = new T[100]; // Does not compile
```

The best you can do is make an array of Object and cast it to an array of T:

```
T[] myArray = (T[]) new Object[100];
```

Impact of this on Kotlin

Kotlin's `Array` class is implemented via Java arrays

This is why the following does not work:

```
arrayOfNulls<T?>(100) // Does not compile
```

The best you can do is:

```
arrayOfNulls<Any?>(100) as Array<T?>
```

equivalent

Similar to Java: `(T[]) new Object[100]`

equivalent

Advanced topic: reified types

In parts of a Kotlin program that are guaranteed not to interact with Java code, it is possible to use a form of generics without type erasure