{ this is Kotlin }

Advanced Topics

Tiberiu Tofan

SAM Interfaces

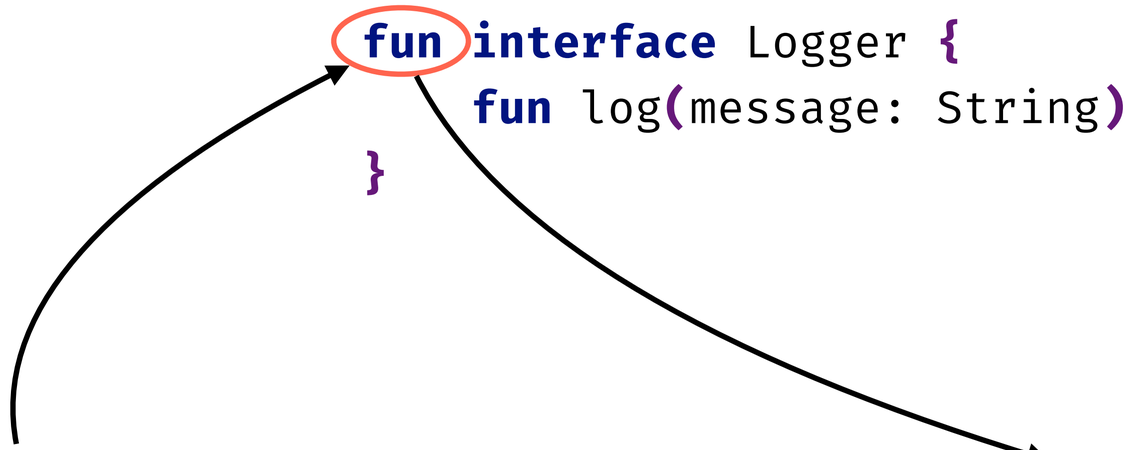
single abstract method

```
interface Logger {
    fun log(message: String)
}
```

```
val infoLogger = object : Logger {
    override fun log(message: String) {
        println("[INFO] $message")
    }
}
```

SAM Interfaces

single abstract method




```
val infoLogger = object : Logger {
    override fun log(message: String) {
        println("[INFO] $message")
    }
}
```

```
val infoLogger = Logger {
    println("[INFO] $it")
}
```

SAM Interfaces

functional interface

```java
//Java
@FunctionalInterface
public interface Runnable
```

```kotlin
val messanger = Runnable { println("I was executed") }
```

Recursion

a function calls itself

```
/**
 * Eliminates duplicate chars from a String
 * E.g.: deduplicate("abcbaabc") is "abc"
 */
fun deduplicate(s: String): String = run {
     fun dedup(s: String, acc: String): String = when {
        s.isEmpty() -> acc
        s.first() in acc -> dedup(s.drop(1), acc)
        else -> dedup(s.drop(1), acc + s.first())
    }
    dedup(s, "")
}
```

Recursion

watch out the stack frames

```
/**
 * Eliminates duplicate chars from a String
 * E.g.: deduplicate("abcbaabc") is "abc"
 */
fun deduplicate(s: String): String = run {
     fun dedup(s: String, acc: String): String = when {
         s.isEmpty() -> acc
         s.first() in acc -> dedup(s.drop(1), acc)
         else -> dedup(s.drop(1), acc + s.first())
     }
     dedup(s, "")
}

deduplicate(s)
```

Exception in thread "main"
java.lang.StackOverflowError

let's assume `s.length` is 5000

Recursion

tail recursion

- the call to itself can only be the last operation
- the compiler converts the recursion to a loop
- the compilation fails if the function cannot be optimized

```
/**
 * Eliminates dupl
 * E.g.: deduplica
 */
fun deduplicate(s: String): String = run {
    tailrec fun dedup(s: String, acc: String): String = when {
        s.isEmpty() -> acc
        s.first() in acc -> dedup(s.drop(1), acc)
        else -> dedup(s.drop(1), acc + s.first())
    }
    dedup(s, "")
}

deduplicate(s)
```

now `s.length` could be as big as the heap allows

Inline functions

motivation

- runtime overhead of higher order functions:
  - each function is an object
  - a function may capture a closure
  - memory and invocation overhead

  ❗ this doesn’t mean you should avoid using higher order functions
- type erasure

Inline functions

declaration

inline

```
inline fun applyTwice(i: Int, f: (Int) -> Int): Int = f(f(i))
```

Inline functions

compilation

```
fun applyTwice(i: Int, f: (Int) -> Int): Int = f(f(i))
```

```
fun main() {
    val res = applyTwice(2) {
        it * 2
    }
}
```


```
public static final Integer applyTwice(
        Integer i,
        Function1<Integer, Integer> f) {
    return f.invoke(f.invoke(i));
}
public static final void main() {
    Integer res = applyTwice(2,
        new Function1<Integer, Integer>() {
            @Override
            public Integer invoke(Integer i) {
                return i * 2;
            }
        });
}
```

* pseudo-code that describes the worst case scenario, the compiler might do some extra optimizations where possible

Inline functions

compilation

```
inline fun applyTwice(i: Int, f: (Int) -> Int): Int = f(f(i))
```

```
fun main() {
    val res = applyTwice(2) {
        it * 2
    }
}
```


```
public static void main()
{
    int i = 2;
    int it = i * 2;
    int res = it * 2;
}
```

* pseudo-code that describes the worst case scenario, the compiler might do some extra optimizations where possible

Could a recursive function be inline?

Inline Classes

- creating a new class has a performance penalty
  - additional heap allocation
  - for primitives, we lose runtime optimizations
- inline classes wrap around a single property - that becomes the runtime type
- inline classes are erased at compile time
- used for type safety

Inline Classes

```
@JvmInline
value class Iban(val value: String) {
    init {
        require(value.matches("""[A-Z]{2}\d{2}[A-Z]{4}\d{16}""".toRegex()))
    }
}
```

```
val iban = Iban("RO10TRUE1234567812345678")
val ibanValue = iban.value
```

Inline Classes

```
@JvmInline
value class Iban(private val value: String) {
    init {
        require(value.matches("""[A-Z]{2}\d{2}[A-Z]{4}\d{16}""".toRegex()))
    }

    operator fun invoke(): String = value
}


val iban = Iban("RO10TRUE1234567812345678")
val ibanValue = iban()
```

Primitives

`Byte`, `Int`, `Long`, `Float`, `Double`, `Boolean`, etc are classes, but:

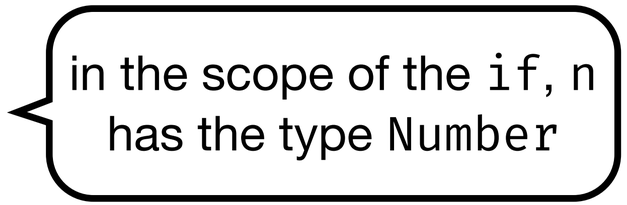
- the compiler may optimize and use primitives instead
- for performance critical code:
  - avoid boxing/unboxing - e.g. working with lists
  - avoid nullable types (primitives cannot be null)
  - use `IntArray`, `DoubleArray`, etc.

❗ **reusability, readability, maintainability** are *usually* more important than performance

Smart Cast

nullable to non nullable

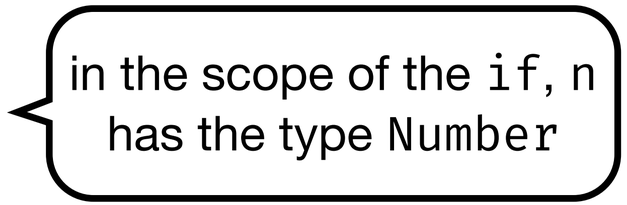
```
fun smartCast(n: Number?) {
    if (n != null) {
        println(n.toDouble())
    }
}
```



Smart Cast

nullable to non nullable

```
fun smartCast(n: Number?) {
    if (n is Number) {
        println(n.toDouble())
    }
}
```



Smart Cast

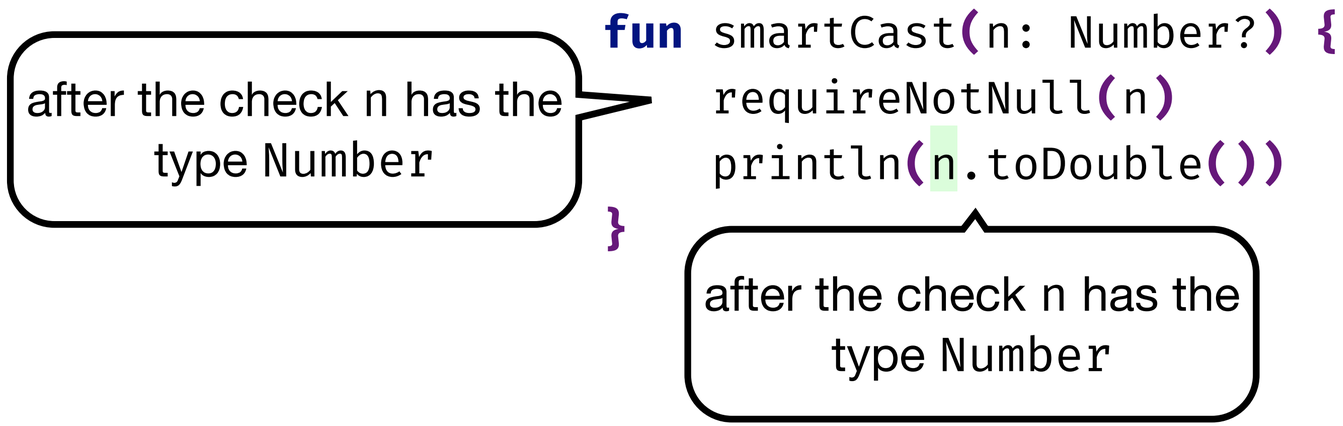
nullable to non nullable

```
fun smartCast(n: Number?) {
    n ?: throw IllegalArgumentException("Null not allowed")
    println(n.toDouble())
}
```

after the check n has the type Number

Smart Cast

nullable to non nullable



Smart Cast

properties

```
data class Star(val age: Number)

fun smartCast(star: Star) {
    when (star.age) {
        is Int -> println(1..star.age)
        is Double -> print(star.age.isFinite())
    }
}
```

star.age has now the type Int

star.age has now the type Double

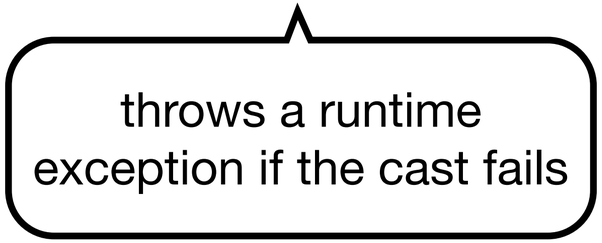
Smart Cast

- cannot use open properties
- cannot use interface properties
- cannot use properties with custom getters (because the implementation may change the type)
- cannot use mutable properties (because it can be changed in a different thread)
- solution: introduce a local variable

Explicit Cast

unsafe

```
fun unsafeCast(n: Number?) {
    val i: Int = n as Int
}
```



Explicit Cast

safe

```
fun safeCast(n: Number?) {
    val i: Int? = n as? Int
}
```

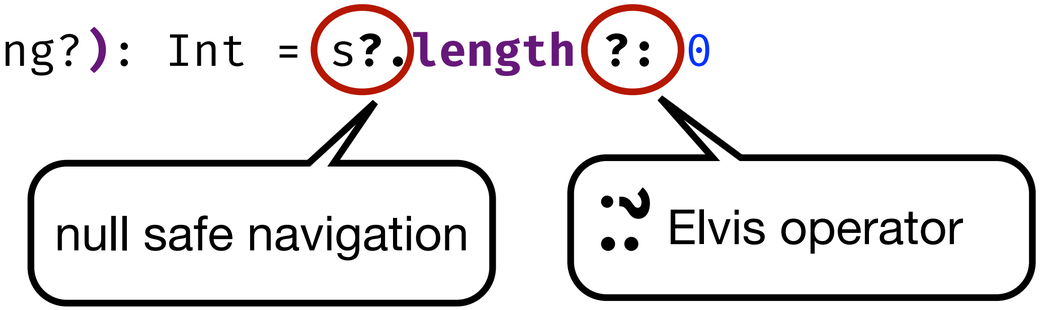
returns `null` if the cast fails

Dealing with `null`

why Kotlin doesn't need `Optional`

```
fun measureLength(s: String?): Int =
    Optional.ofNullable(s).map { it.length }.orElse(0)
```

```
fun measureLength(s: String?): Int = s?.length ?: 0
```



Dealing with `null`

why Kotlin doesn't need `Optional`

```
fun censorToEmpty(s: String?, censor: (String) -> String?): String =
    Optional.ofNullable(s).flatMap { Optional.ofNullable(censor(it)) }.orElse("")
```

```
fun censorToEmpty2(s: String?, censor: (String) -> String?): String =
    s?.let(censor) ?: ""
```

scope function