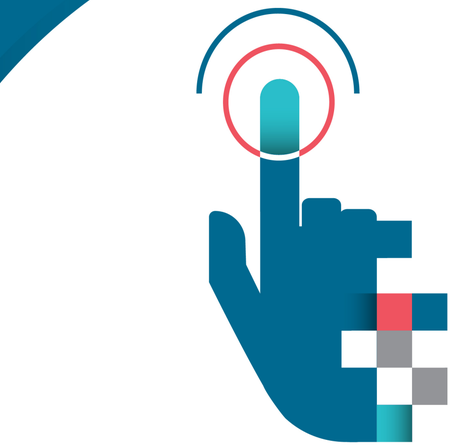
2019

1° ESCUELA DE ACTUALIZACIÓN EN TECNOLOGÍAS DE INFORMÁTICA

PROGRAMACIÓN EN KOTLIN PARA PLATAFORMAS JAVA Y ANDROID

LIC. EMMANUEL LAGARRIGUE LAZARTE

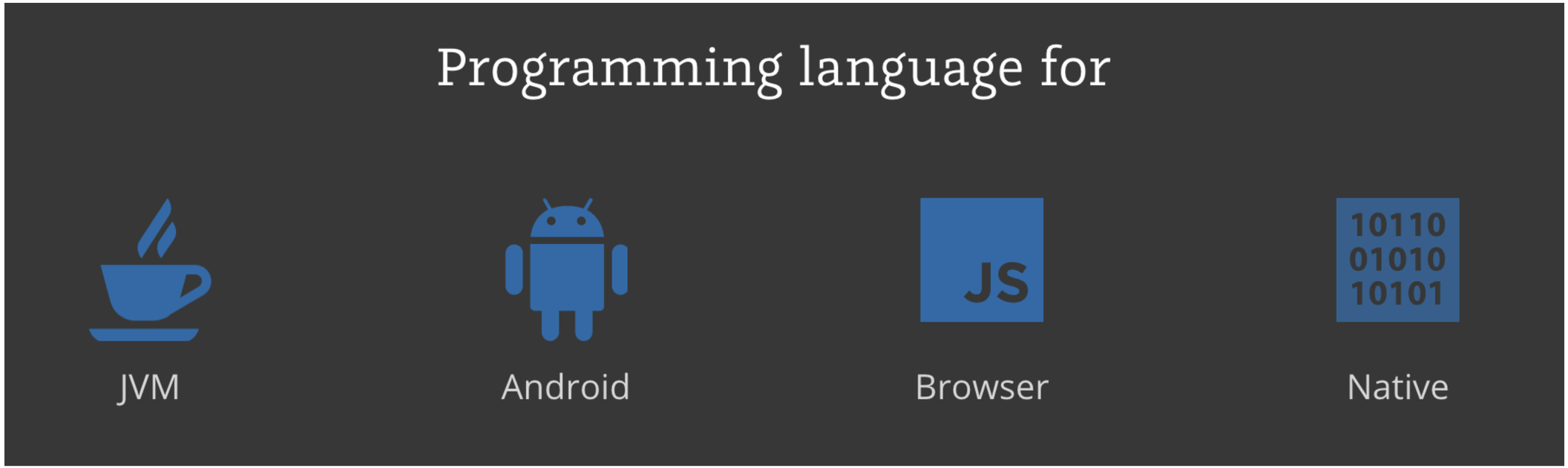


eati

DCIC DEPARTAMENTO DE CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

UNS UNIVERSIDAD NACIONAL DEL SUR

¿Qué es Kotlin?



https://kotlinlang.org/

¿Por qué Kotlin?

Concise

Drastically reduce the amount of boilerplate code.

See example

Safe

Avoid entire classes of errors such as null pointer exceptions.

See example

Interoperable

Leverage existing libraries for JVM, Android and the browser.

See example

Tool-friendly

Choose any Java IDE or build from the command line.

See example


Concise

Create a POJO with getters, setters, `equals()`, `hashCode()`, `toString()` and `copy()` in a single line:

```
data class Customer(val name: String, val email: String, val company: String)
```

Or filter a list using a lambda expression:

```
val positiveNumbers = list.filter { it > 0 }
```

Want a singleton? Create an object:

```
object ThisIsASingleton {
    val companyName: String = "JetBrains"
}
```


Safe

Get rid of those pesky NullPointerExceptions, you know, The Billion Dollar Mistake

```
var output: String
output = null   // Compilation error
```

Kotlin protects you from mistakenly operating on nullable types

```
val name: String? = null     // Nullable type
println(name.length())       // Compilation error
```

And if you check a type is right, the compiler will auto-cast it for you

```
fun calculateTotal(obj: Any) {
    if (obj is Invoice)
        obj.calculateTotal()
}
```

Interoperable

Use any existing library on the JVM, as there's 100% compatibility, including SAM support.

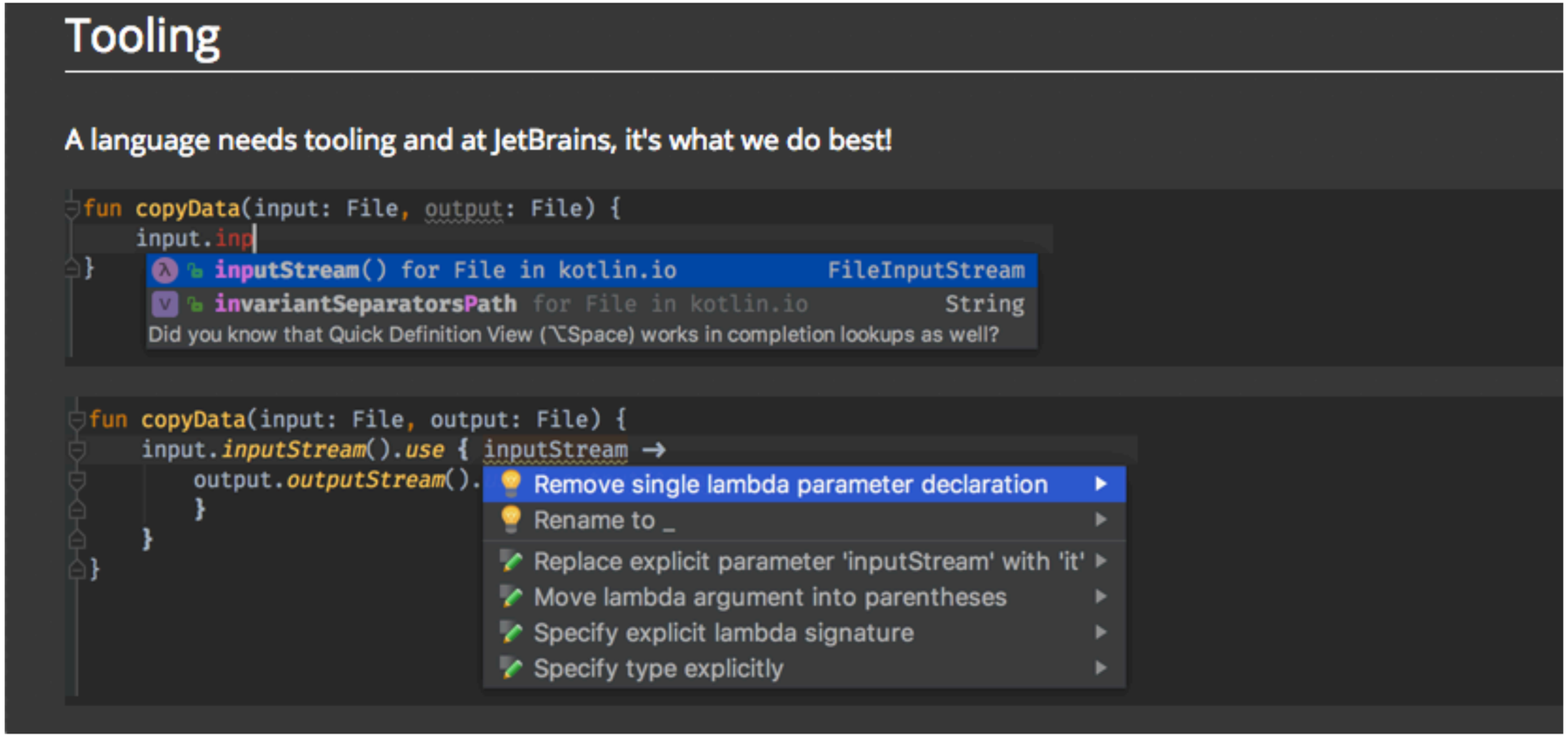
```
import io.reactivex.Flowable
import io.reactivex.schedulers.Schedulers

Flowable
    .fromCallable {
        Thread.sleep(1000) //  imitate expensive computation
        "Done"
    }
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.single())
    .subscribe(::println, Throwable::printStackTrace)
```

Target either the JVM or JavaScript. Write code in Kotlin and decide where you want to deploy to

```
import kotlin.browser.window

fun onLoad() {
    window.document.body!!.innerHTML += "<br/>Hello, Kotlin!"
}
```

Tooling

A language needs tooling and at JetBrains, it's what we do best!




¿Cómo es Kotlin?

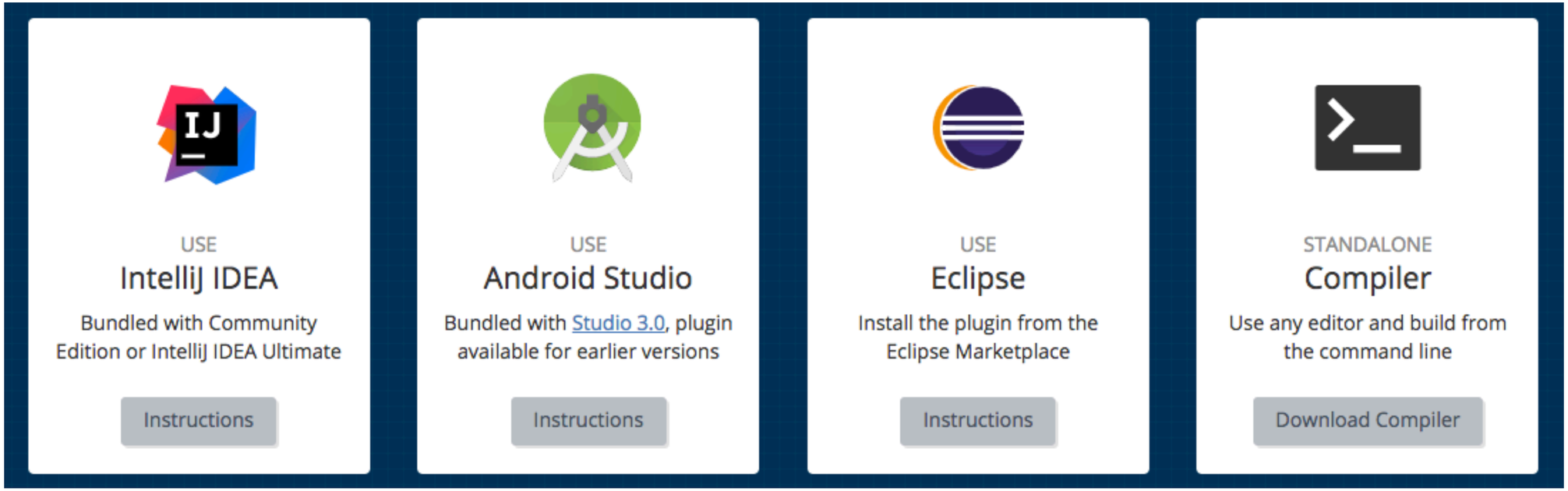
- Kotlin, al igual que Java, es estáticamente tipado.
- Kotlin, a diferencia de Java, tiene inferencia de tipos.

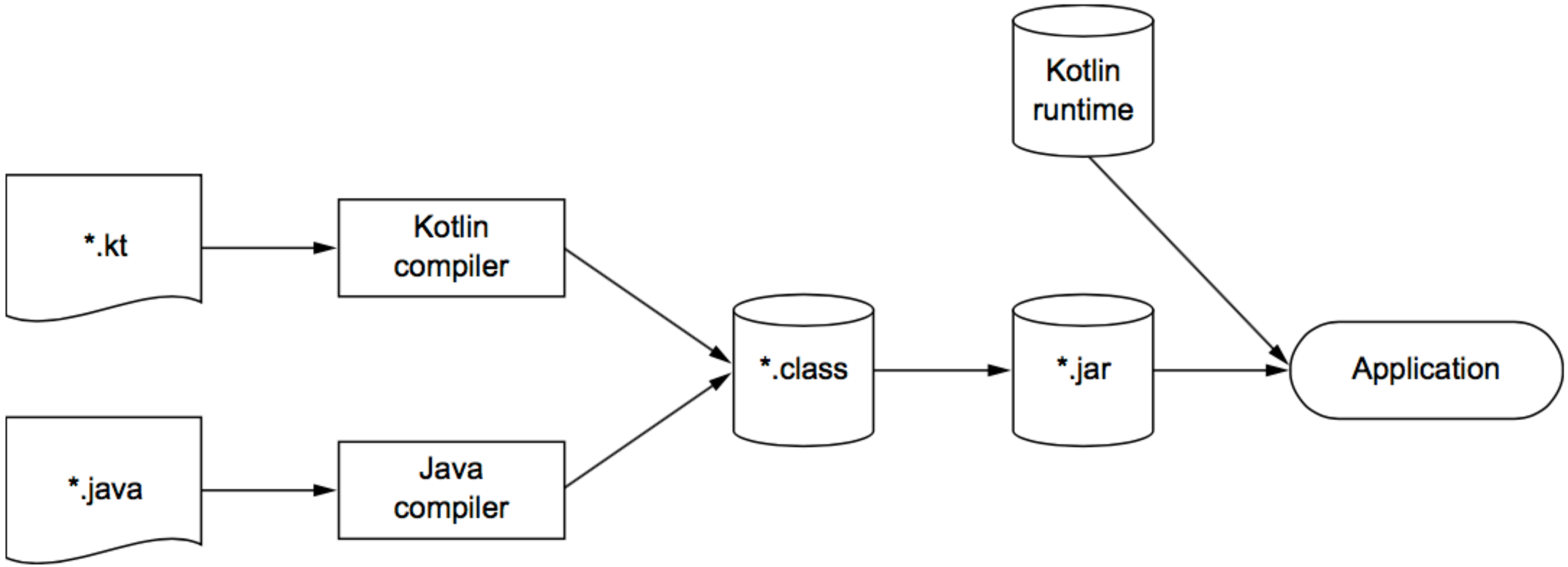
¿Cómo es Kotlin?

- Soporte de tipos Nullable
- Orientado a Objetos
- Soporte de Programación Funcional:
  - Function Types
  - Lambda expressions
  - Conjunto de APIs para trabajar con objetos y colecciones

¿Cómo es Kotlin?

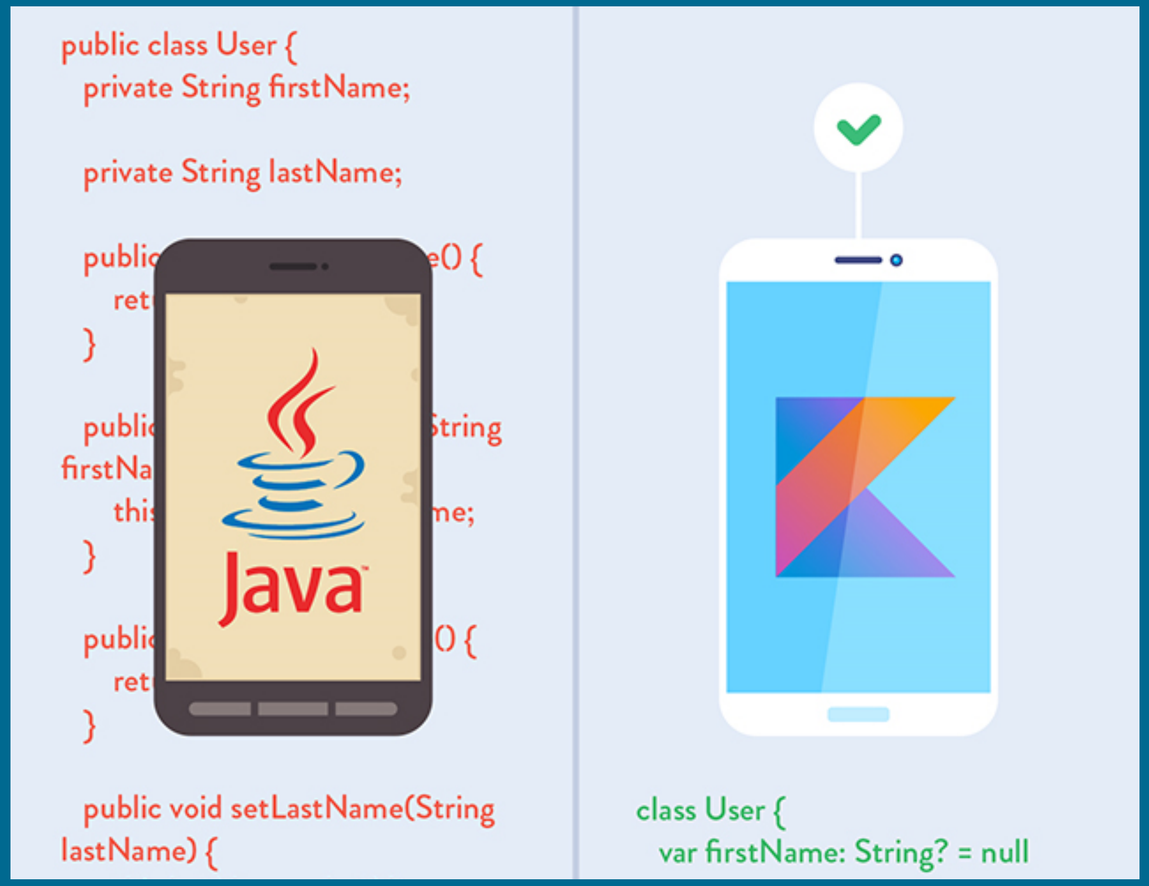
- Gratis y Open Source http://github.com/ jetbrains/ kotlin
- https://try.kotl.in





Kotlin build process

Kotlin Basics

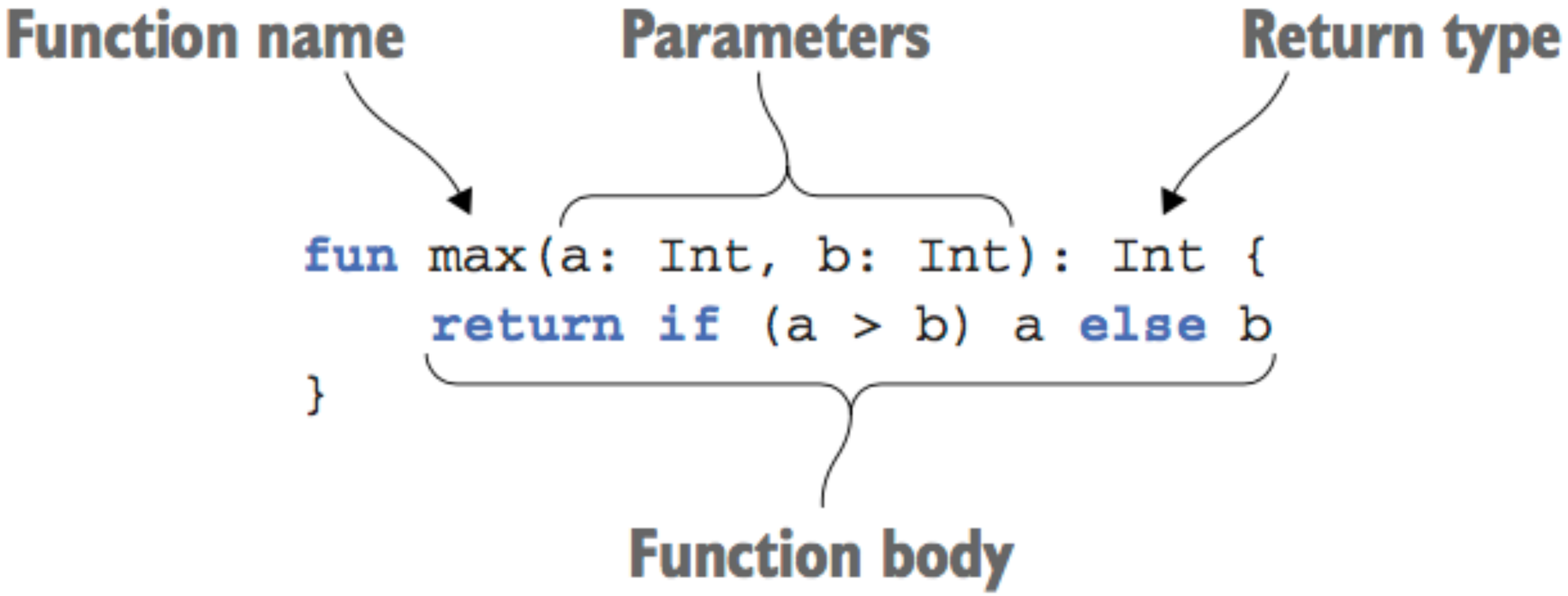



Funciones y Variables

```
fun main(args: Array<String>) {
    println("Hello, world!")
}
```


Funciones

```
fun max(a: Int, b: Int): Int {
    return if (a > b) a else b
}

>>> println(max(1, 2))
2
```


Function name
Parameters
Return type
fun max(a: Int, b: Int): Int {
return if (a > b) a else b
}
Function body

Sentencias y Expresiones

- En Kotlin, **if** es una expresión, no una sentencia
- Las expresiones tienen valor y pueden utilizarse en otras expresiones
- En Kotlin, la mayoría de las estructuras son expresiones, excepto por los loops

Cuerpos expresiones

- Si el cuerpo de una función consiste de una sola expresión, se puede utilizar dicha expresión como cuerpo sin los corchetes

```
fun max(a: Int, b: Int): Int = if (a > b) a else b
```

Cuerpos expresiones

- Se puede omitir el tipo, ya que es inferido por el tipo de la expresión

```
fun max(a: Int, b: Int) = if (a > b) a else b
```


Variables

```
val question =
    "The Ultimate Question of Life, the Universe, and Everything"

val answer = 42


val answer: Int = 42


val answer: Int
answer = 42
```

Variables Mutables e Inmutables

- **val (value)**: Referencia inmutable. Una variable declarada con **val** no puede ser re asignada luego de ser inicializada. Idem **final** en java.
- **var (variable)**: Referencia mutable. El valor de la variable puede cambiar. Idem declaración normal en java. El tipo no puede cambiar.

String templates

```
fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Kotlin"
    println("Hello, $name!")
}
```

```
println("Hello, ${args[0]}!")
```

```
println("Hello, ${if (args.size > 0) args[0] else "someone"}!")
```

Clases y propiedades

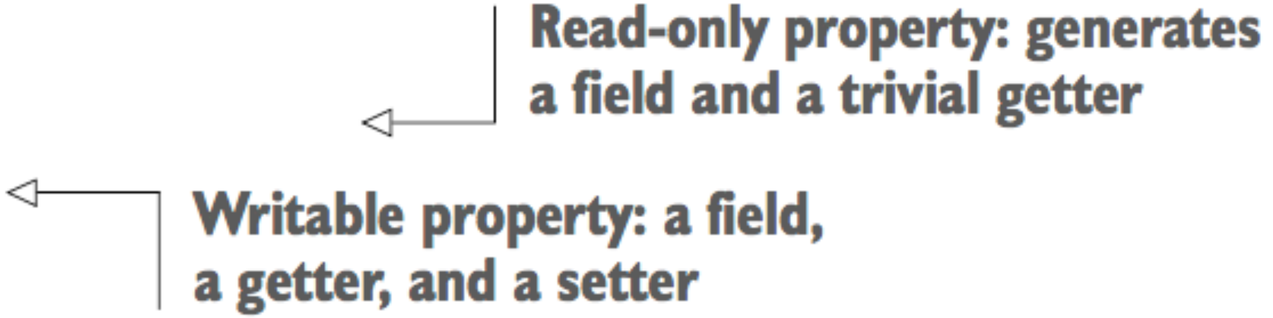
```java
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Kotlin

```kotlin
class Person(val name: String)
```

Propiedades

```
class Person(
    val name: String,
    var isMarried: Boolean
)
```



```
/* Java */
>>> Person person = new Person("Bob", true);
>>> System.out.println(person.getName());
Bob
>>> System.out.println(person.isMarried());
true
```

Kotlin

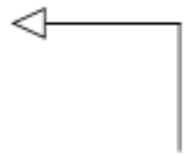
```
>>> val person = Person("Bob", true)
>>> println(person.name)
Bob
>>> println(person.isMarried)
true
```

Call the constructor without the "new" keyword.

You access the property directly, but the getter is invoked.

Custom Accessors

```
class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
        get() {
            return height == width
        }
}
```



Property getter declaration


Enums

```
enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

Enums

```
enum class Color(
        val r: Int, val g: Int, val b: Int
) {
    RED(255, 0, 0), ORANGE(255, 165, 0),
    YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),
    INDIGO(75, 0, 130), VIOLET(238, 130, 238);

    fun rgb() = (r * 256 + g) * 256 + b
}
>>> println(Color.BLUE.rgb())
255
```

Declares properties of enum constants

Specifies property values when each constant is created

The semicolon here is required.

Defines a method on the enum class

When enums

```
fun getMnemonic(color: Color) =
    when (color) {
        Color.RED -> "Richard"
        Color.ORANGE -> "Of"
        Color.YELLOW -> "York"
        Color.GREEN -> "Gave"
        Color.BLUE -> "Battle"
        Color.INDIGO -> "In"
        Color.VIOLET -> "Vain"
    }

>>> println(getMnemonic(Color.BLUE))
Battle
```

Returns a "when" expression directly

Returns the corresponding string if the color equals the enum constant


When enums

```
fun getWarmth(color: Color) = when(color) {
    Color.RED, Color.ORANGE, Color.YELLOW -> "warm"
    Color.GREEN -> "neutral"
    Color.BLUE, Color.INDIGO, Color.VIOLET -> "cold"
}

>>> println(getWarmth(Color.ORANGE))
warm
```

When cualquier objeto

```
fun mix(c1: Color, c2: Color) =
        when (setOf(c1, c2)) {
            setOf(RED, YELLOW)  -> ORANGE
            setOf(YELLOW, BLUE)  -> GREEN
            setOf(BLUE, VIOLET)  -> INDIGO
            else -> throw Exception("Dirty color")
        }

>>> println(mix(BLUE, YELLOW))
GREEN
```

Enumerates pairs of colors that can be mixed

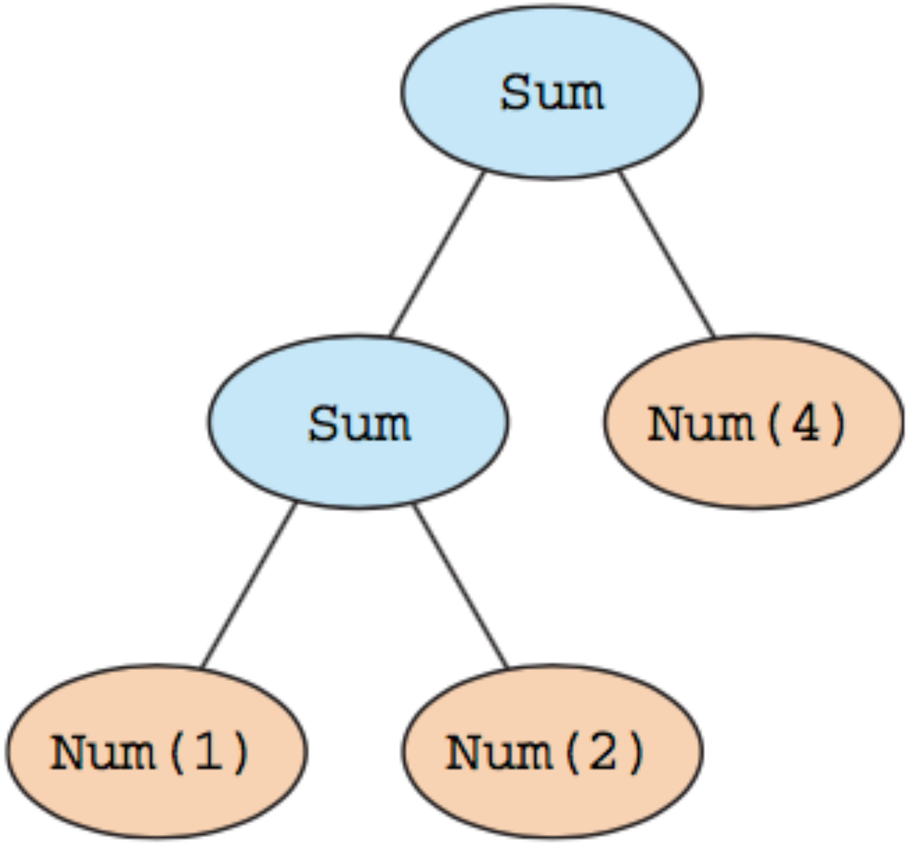
An argument of the "when" expression can be any object. It's checked for equality with the branch conditions.

Executed if none of the other branches were matched

When nada

```
fun mixOptimized(c1: Color, c2: Color) =
    when {
        (c1 == RED && c2 == YELLOW) ||
        (c1 == YELLOW && c2 == RED) ->
            ORANGE

        (c1 == YELLOW && c2 == BLUE) ||
        (c1 == BLUE && c2 == YELLOW) ->
            GREEN

        (c1 == BLUE && c2 == VIOLET) ||
        (c1 == VIOLET && c2 == BLUE) ->
            INDIGO

        else -> throw Exception("Dirty color")
    }
>>> println(mixOptimized(BLUE, YELLOW))
GREEN
```

No argument for “when”



$$(1 + 2) + 4$$

class Expr { }

```
class Num extends Expr {
  private int value;

  Num(int value) {
    this.value = value;
  }

  int getValue() { return value; }
}
```

```
class Sum extends Expr {
  private Expr left;
  private Expr right;

  Sum(Expr left, Expr right) {
    this.left = left;
    this.right = right;
  }

  Expr getLeft() { return left; }

  Expr getRight() { return right; }
}
```

```
static int eval(Expr e) {

  if (e instanceof Num) {
    Num n = (Num) e;
    return n.getValue();
  }

  if (e instanceof Sum) {
    Sum s = (Sum) e;
    return eval(s.getLeft()) + eval(s.getRight());
  }

  throw new IllegalArgumentException();
}
```

Smart casts: combinando checkeo de tipos y cast

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr
```

Simple value object class with one property, value, implementing the Expr interface

The argument of a Sum operation can be any Expr: either Num or another Sum

```
fun eval(e: Expr): Int {
    if (e is Num) {
        val n = e as Num
        return n.value
    }
    if (e is Sum) {
        return eval(e.right) + eval(e.left)
    }
    throw IllegalArgumentException("Unknown expression")
}
```

This explicit cast to Num is redundant.

The variable e is smart-cast.

Refactoring: **expresión de retorno**

```
fun eval(e: Expr): Int =
    if (e is Num) {
        e.value
    } else if (e is Sum) {
        eval(e.right) + eval(e.left)
    } else {
        throw IllegalArgumentException("Unknown expression")
    }

>>> println(eval(Sum(Num(1), Num(2))))
3
```

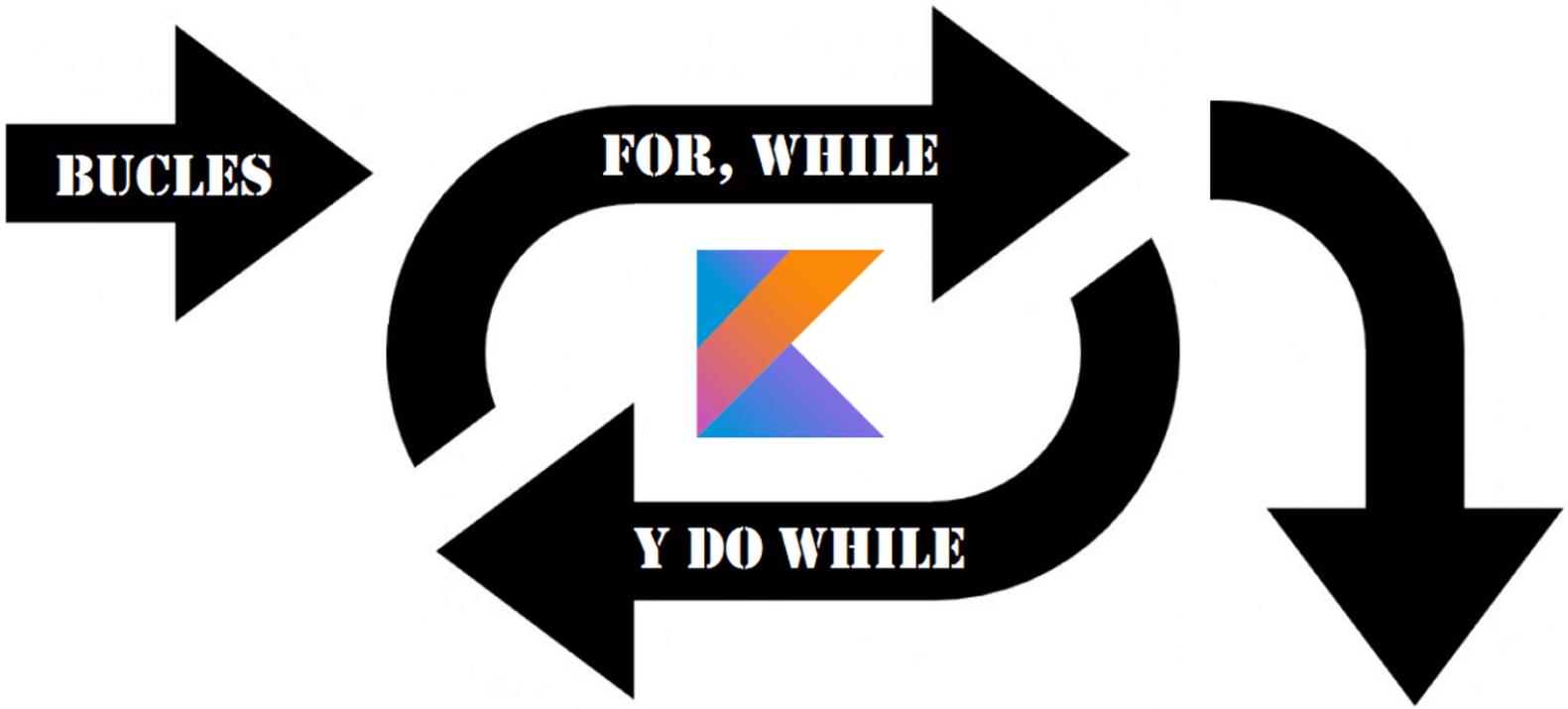
Refactoring: **if** por **when**

```
fun eval(e: Expr): Int =
    when (e) {
        is Num ->
            e.value
        is Sum ->
            eval(e.right) + eval(e.left)
        else ->
            throw IllegalArgumentException("Unknown expression")
    }
```

Smart casts are applied here.

“when” branches that check the argument type

Ciclos **while** y **for**



do-while loops

```
while (condition) {
    /*...*/
}

do {
    /*...*/
} while (condition)
```

The body is executed while the condition is true.

The body is executed for the first time unconditionally. After that, it's executed while the condition is true.


Rangos y progresiones

- **rango: intervalo entre dos valores**

```
val oneToTen = 1..10
```


Rangos y progresiones

```
for (i in 1.. 100) {

}

for (i in 100 downTo 1 step 2) {

}
```

Iterando sobre mapas

```
val binaryReps = TreeMap<Char, String>()

for (c in 'A'..'F') {
    val binary = Integer.toBinaryString(c.toInt())
    binaryReps[c] = binary
}

for ((letter, binary) in binaryReps) {
    println("$letter = $binary")
}
```

Converts ASCII code to binary

Stores the value in a map by the c key

Uses TreeMap so the keys are sorted

Iterates over the characters from A to F using a range of characters

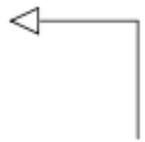
Iterates over a map, assigning the map key and value to two variables


Usando **in** …

```
fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun isNotDigit(c: Char) = c !in '0'..'9'

>>> println(isLetter('q'))
true
>>> println(isNotDigit('x'))
true
```

Usando **in** …

```
c in 'a'..'z'
```



Transforms to
a <= c && c <= z

Usando **in** en **when**

```
fun recognize(c: Char) = when (c) {
    in '0'..'9' -> "It's a digit!"
    in 'a'..'z', in 'A'..'Z' -> "It's a letter!"
    else -> "I don't know…"
}
>>> println(recognize('8'))
It's a digit!
```

You can combine multiple ranges.

Checks whether the value is in the range from 0 to 9

Usando **in** en rangos de Strings

```
>>> println("Kotlin" in "Java".."Scala")
true
```

The same as “Java” <= “Kotlin” && “Kotlin” <= “Scala”

```
>>> println("Kotlin" in setOf("Java", "Scala"))
false
```

This set doesn’t contain the string “Kotlin”.


Excepciones en Kotlin

```
if (percentage !in 0..100) {
    throw IllegalArgumentException(
        "A percentage value must be between 0 and 100: $percentage")
}
```

try, catch y finally

```
fun readNumber(reader: BufferedReader): Int? {
    try {
        val line = reader.readLine()
        return Integer.parseInt(line)
    }
    catch (e: NumberFormatException) {
        return null
    }
    finally {
        reader.close()
    }
}
```

You don't have to explicitly specify exceptions that can be thrown from this function.

The exception type is on the right.

"finally" works just as it does in Java.

```
>>> val reader = BufferedReader(StringReader("239"))
>>> println(readNumber(reader))
239
```

try como expresión

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        return
    }

    println(number)
}

>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
```

Becomes the value of the "try" expression

Nothing is printed.