A TYPESCRIPT FAN'S KOTLINJS ADVENTURES

should you make the switch?

POWERED BY INSTIL.

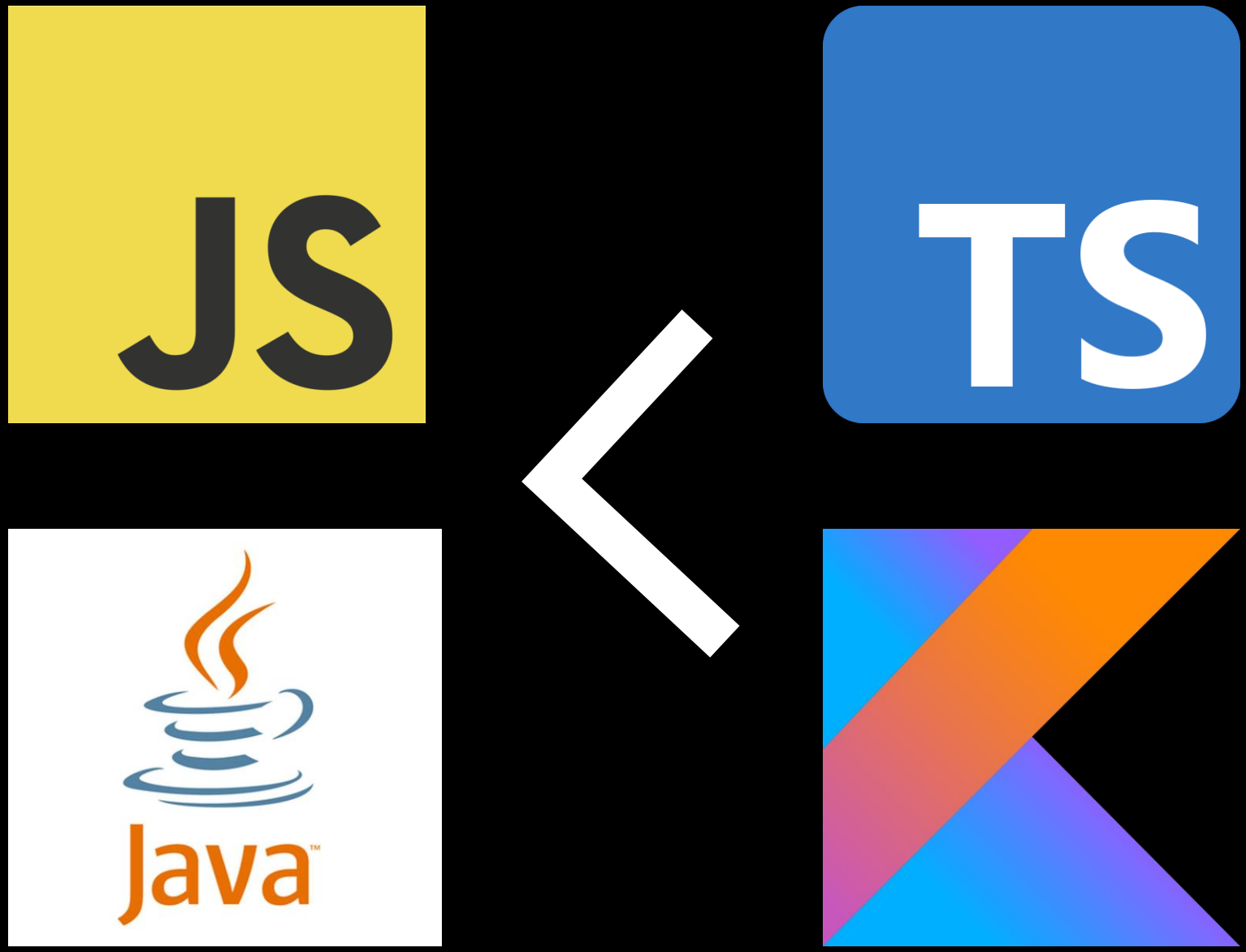
@BoyleEamonn

@GarthGilmour

INSTIL

we love typescript and kotlin
they solve real problems for us

- **Neither reinvents the wheel**
- **TS brings types and compilation to JS**
  - Improves upon what is already ubiquitous
  - Leverage JS knowledge and community
- **Kotlin improves upon Java in many ways**
  - But interop with Java is a design goal
  - Learning curve from Java is easy
  - Takes features from many languages



Kotlin (JVM)	Kotlin (Android)	Kotlin (JS)	Kotlin (Native)
Java Bytecode	Dalvik Bytecode	JavaScript Interpreter	Machine Code & Runtime
JVM	ART	Browser	OS

JVM

Android

Browser

Native

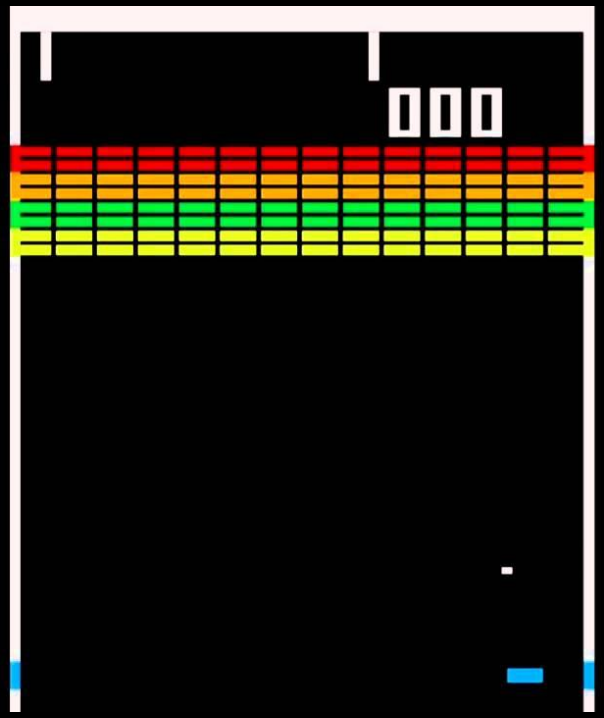
breaking down the problem

experiment: is kotlinjs worth it?
what problem does it solve?

- **Build an app in both TypeScript and KotlinJS**
  - Go beyond “hello world”
- **Incorporate common JS libraries**
- **Compare the experience**
  - Tooling
  - Language features
  - Community

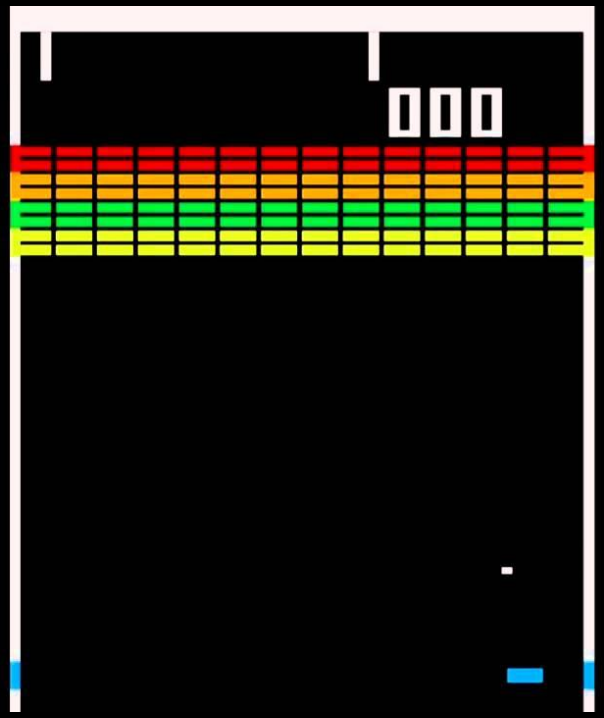
breakout clone

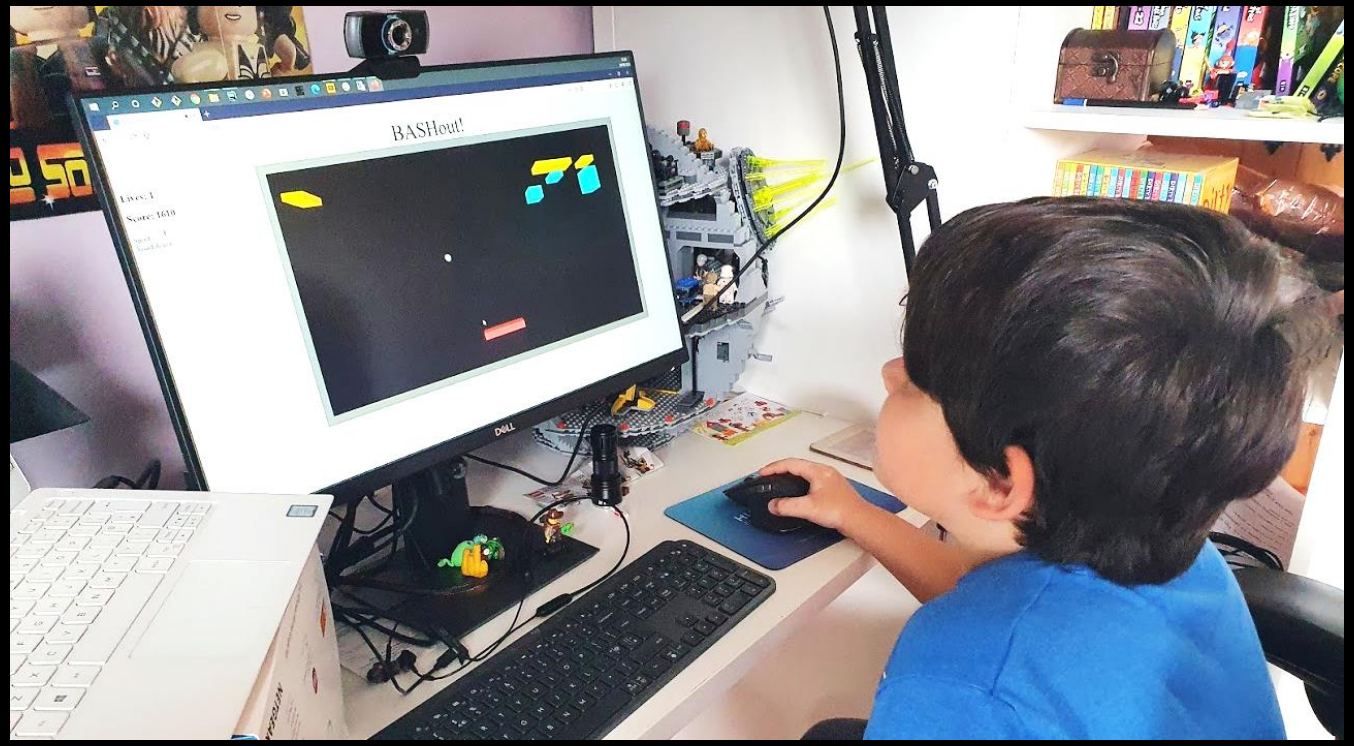
browser based clone



breakout clone

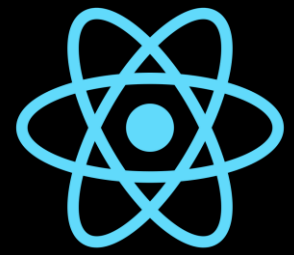
browser based clone

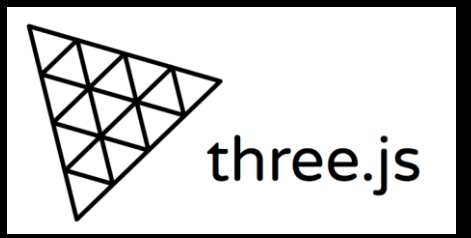




moving parts

- **React**
  - https://reactjs.org
- **Redux**
  - https://redux.js.org
- **React-Three-Fiber**
  - https://github.com/react-spring/react-three-fiber





declarative ui

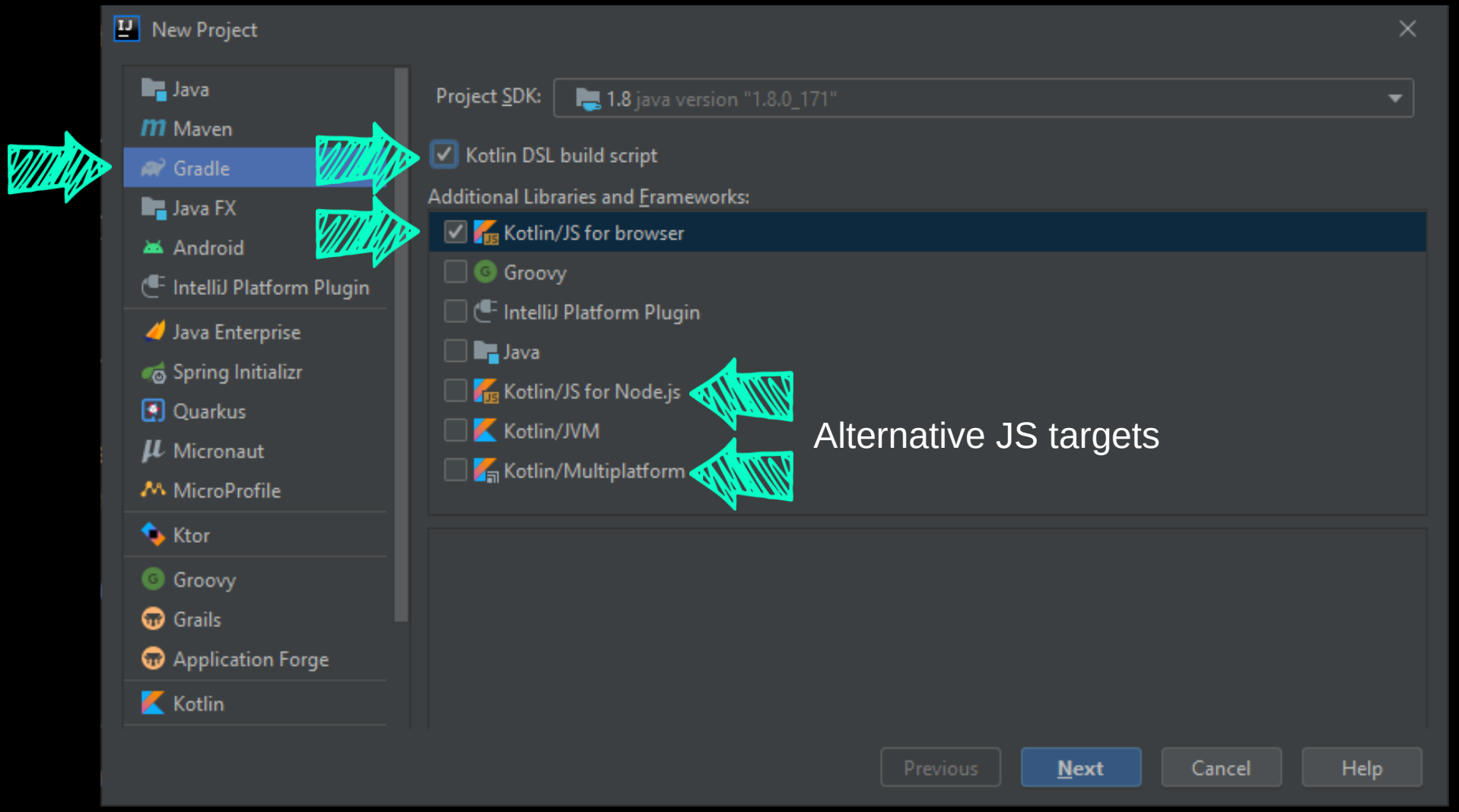
state separation

```
const geometry = new THREE.BoxGeometry(BAT_WIDTH, 1, 1);

export const Bat: FC = () => {
  const position = useSelector((state: State) => state.batPosition);

  return (
    <mesh position={[position, 0, BAT_Z]} geometry={geometry}>
      {woodMaterial}
    </mesh>
  );
};
```

creating a kotlinjs react project



add packages from multiple sources

kotlinjs, multiplatform and npm

```
dependencies {
    implementation(kotlin("stdlib-js"))

    implementation("org.jetbrains:kotlin-react:16.13.0-pre.94-kotlin-1.3.70")
    implementation("org.jetbrains:kotlin-react-dom:16.13.0-pre.94-kotlin-1.3.70")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-common:1.3.5")
    . . .
    implementation(npm("react-three-fiber", "4.2.20"))
    implementation(npm("react-use-gesture", "7.0.15"))
    implementation(npm("redux", "4.0.5"))
    implementation(npm("three", "0.119.1"))
}
```

get down to coding

main

```
import kotlin.browser.document
import react.redux.provider
import redux.store
import components.App

fun main() {
    react.dom.render(document.getElementById("root")) {
        provider(store) {
            App()
        }
    }
}
```

get down to coding

```
import kotlin.browser.document
import react.redux.provider
import redux.store
import components.App

fun main() {
    react.dom.render(document.getElementById("root")) {
        provider(store) {
            App()
        }
    }
}
```

get down to coding

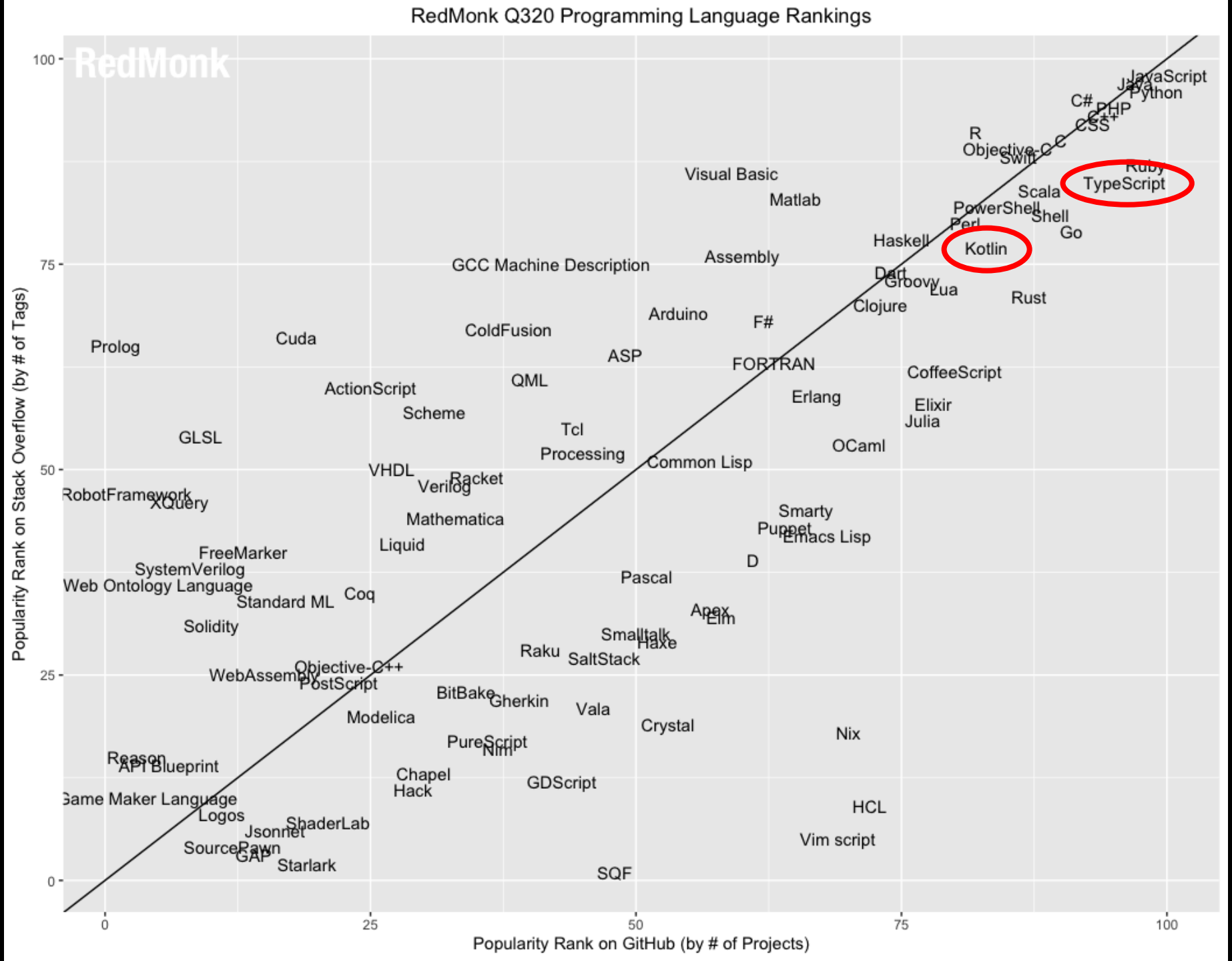
```
import kotlin.browser.document
import react.redux.provider
import redux.store
import components.App

fun main() {
    react.dom.render(document.getElementById("root")) {
        provider(store) {
            App()
        }
    }
}
```

Round 1

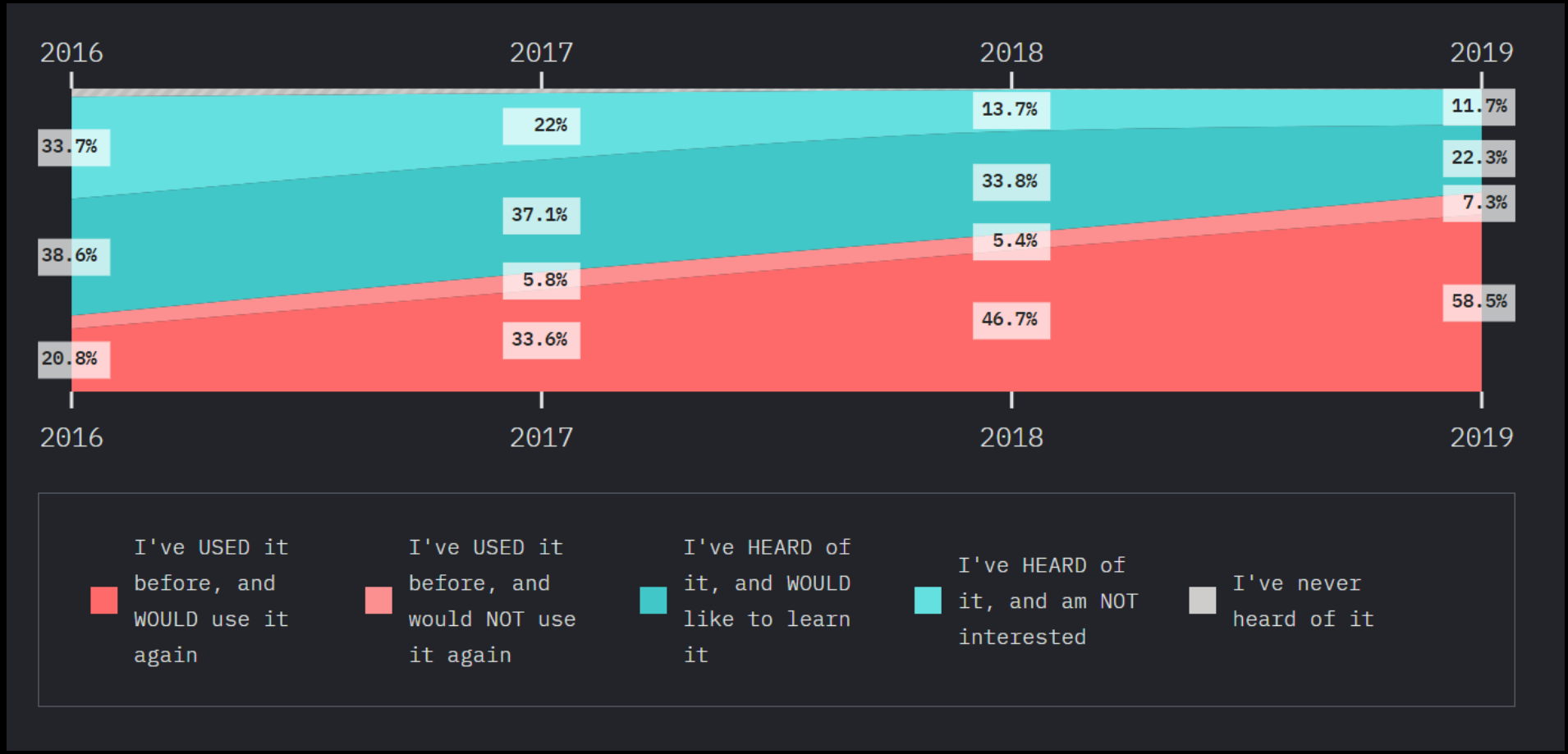
community

1 JavaScript
2 Python
3 Java
4 PHP
5 C++
5 C#
7 Ruby
7 CSS
9 TypeScript
10 C
11 Swift
11 Objective-C
13 R
14 Scala
15 Go
15 Shell
17 PowerShell
18 Perl
19 Kotlin
20 Rust



state of javascript 2019

typescript well established



community, maturity and support

typescript > kotlin

- **TypeScript is more popular than KotlinJS**
- **As a superset of JS, reusing knowledge and assets is easier**
  - And the transition for JS developers to TS is easier
- **TypeScript is well established in the JavaScript world**
  - Many libraries include TypeScript definitions
  - DefinitelyTyped contains many more

Round 2

interop with javascript

importing npm js packages
gradle dsl

- Only a few first class wrappers provided
- It is easy to add NPM packages yourself

```
dependencies {
    . . .
    implementation(npm("react-three-fiber", "4.2.20"))
    implementation(npm("react-use-gesture", "7.0.15"))
    implementation(npm("three", "0.119.1"))
}
```

- But how easy is it to consume that code in Kotlin?

really easy
external declarations

```
@file:JsModule("react-three-fiber")
@file:JsNonModule

...

external val Canvas: RClass<RProps>

external fun extend(objects: Any)

external fun useFrame(callback: (dynamic, Double) -> Unit)

external fun useThree(): dynamic

external interface PointerEvent {
    val uv: Vector2
}
```

Specify the NPM package

Define any items you wish to use

dukat

dukat

no, not this one

```ts
export interface BasicInterface {
    readonly field1: number;
    method1(): boolean;
}

export function buildInterface(): BasicInterface;

export type ReadOnlyBasicInterface = Readonly<BasicInterface>;
```

```ts
external interface BasicInterface {
    var field1: Number
    fun method1(): Boolean
}

external fun buildInterface(): BasicInterface

typealias ReadOnlyBasicInterface = Readonly<BasicInterface>
```

```ts
export type MyReadOnly<T> = {
    readonly [K in keyof T]: T[K];
}

export type ReadonlyDummyInterface = MyReadOnly<DummyInterface>;
```

```ts
typealias MyReadOnly<T> = Any

typealias ReadonlyDummyInterface = MyReadOnly<DummyInterface>
```

dynamic and jsobject
get out of jail

- **KotlinJS supports a `dynamic` type**
  - This can be used to quickly patch over APIs

```
external fun useFrame(callback: (dynamic, Double) -> Unit)
```

- **It also has a helper function to create objects on the fly**

```
Block(jsObject {
    position = brick.location.toVector3()
    color = brick.color
})
```

interop with javascript

typescript > kotlinjs

- **As a superset, TypeScript has to win this one**
- **The type system is geared to support JavaScript**
  - Lots of libraries already provide TypeScript definition files
- **However, writing external declaration in KotlinJS is easy**
- **Dukat does a good job but can’t cover every case**
  - You may have to write custom translation code on top

Round 3

jsx vs dsl

```tsx
export const Hud: FC = () => {
  // ...

  return (
    <div>
      <h2>Lives: {lives}</h2>
      <h2>Score: {score}</h2>
      <div>
        <label>Speed:</label>
        <input type="range" min={1} max={50} value={speedScalar * 10}
               onChange={e => /* ... */}/>
      </div>
      <div>
        <label>Sound Active:</label>
        <input type="checkbox" checked={soundActive}
               onChange={e => /* ... */}/>
      </div>
    </div>
  );
};
```

```kotlin
val Hud = FC {
    // ...

    div {
        h2 { +"Lives: $lives" }
        h2 { +"Score: $score" }
        div {
            label { +"Speed:" }
            input(type = InputType.range) {
                attrs {
                    min = "1"
                    max = "50"
                    value = (speedScalar * 10).toString()
                    onChangeFunction = {/* ... */}
                }}}
        div {
            label { +"Sound Active:" }
            input(type = InputType.checkBox) {
                attrs {
                    checked = soundActive
                    onChangeFunction = {/* ... */}
                }}}}
}
fun RBuilder.Hud() = child(Hud)
```

It’s pretty cool that we can create this

Extension function

```
inline fun RBuilder.div(
  classes: String? = null,
  block: RDOMBuilder<DIV>.() -> Unit
): ReactElement
```

Optional param via default

Last param is a
lambda with receiver

```
// Here, “this” is the containing object
div {
 // Here, “this” is a RDOMBuilder<DIV>
}
```

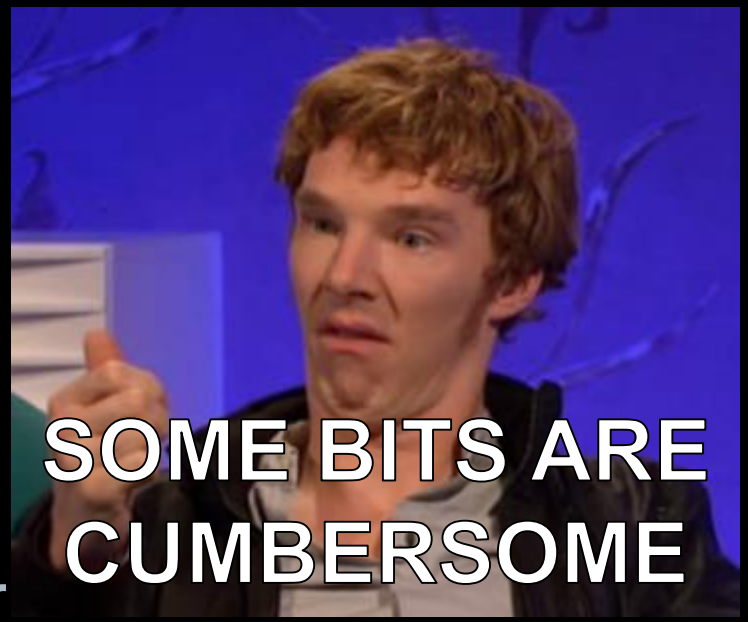
```
inline fun RBuilder.div(
  classes: String? = null,
  block: RDOMBuilder<DIV>.() -> Unit
): ReactElement
```

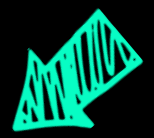
```
// Here, “this” is the containing object
div {
 // Here, “this” is a RDOMBuilder<DIV>
}
```

As I said, this flexibility is pretty cool

```
val Hud = FC {
    // ...

    div {
        h2 { +"Lives: $lives" }
        h2 { +"Score: $score" }
        div {
            label { +"Speed:" }
            input(type = InputType.range) {
                attrs {
                    min = "1"
                    max = "50"
                    value = (speedScalar * 10).toString()
                    onChangeFunction = {/* ... */}
                }}}
        // ...
}
fun RBuilder.Hud() = child(Hud)
```





Overloaded operators to attach data

```
val Hud = FC {
    // ...

    div {
        h2 { +"Lives: $lives" }
        h2 { +"Score: $score" }
        div {
            label { +"Speed:" }
            input(type = InputType.range) {
                attrs {
                    min = "1"
                    max = "50"
                    value = (speedScalar * 10).toString()
                    onChangeFunction = {/* ... */}
                }}}
        // ...
}
fun RBuilder.Hud() = child(Hud)
```

```
val Hud = FC {
    // ...

    div {
        h2 { +"Lives: $lives" }
        h2 { +"Score: $score" }
        div {
            label { +"Speed:" }
            input(type = InputType.range) {
                attrs {
                    min = "1"
                    max = "50"
                    value = (speedScalar * 10).toString()
                    onChangeFunction = {/* ... */}
                }}}
        // ...
    }
}
fun RBuilder.Hud() = child(Hud)
```

Attributes inside a `attrs` block

```
val Hud = FC {
    // ...

    div {
        h2 { +"Lives: $lives" }
        h2 { +"Score: $score" }
        div {
            label { +"Speed:" }
            input(type = InputType.range) {
                attrs {
                    min = "1"
                    max = "50"
                    value = (speedScalar * 10).toString()
                    onChangeFunction = {/* ... */}
                }}}
        // ...
}
fun RBuilder.Hud() = child(Hud)
```

Additional extension function required

No union (or intersection) types in Kotlin

String attribute types

```
val Hud = FC {
    // ...

    div {
        h2 { +"Lives: $lives" }
        h2 { +"Score: $score" }
        div {
            label { +"Speed:" }
            input(type = InputType.range) {
                attrs {
                    min = "1"
                    max = "50"
                    value = (speedScalar * 10).toString()
                    onChangeFunction = {/* ... */}
                }}}
        // ...
    }
}
fun RBuilder.Hud() = child(Hud)
```

```ts
interface InputHTMLAttributes<T> extends HTMLAttributes<T> {
    max?: number | string;
    min?: number | string;
    value?: string | ReadonlyArray<string> | number;
    ...
}
```

Type union

```ts
type PropsWithChildren<P> = P & { children?: ReactNode };
```

Type intersection

```
export type InterfaceUnion = First | Second;

export function interfaceUnionInput(input: InterfaceUnion): void;

export function interfaceUnionOutput(): InterfaceUnion;
```

```
// Type exports erased!

external fun interfaceUnionInput(input: First)

external fun interfaceUnionInput(input: Second)

external fun interfaceUnionOutput(): dynamic /* First | Second */
```

Kotlin supports proper overloads

Not supported on the return type

```ts
export type InterfaceIntersection = First & Second;

export function interfaceIntersectionInput(input: InterfaceIntersection): void;

export function interfaceIntersectionOutput(): InterfaceIntersection;
```

```ts
external fun interfaceIntersectionInput(input: First /* First & Second */)

external fun interfaceIntersectionOutput(): First /* First & Second */
```

Intersection dropped

useEffect

union return workaround

```
fun useEffect(
    dependencies: RDependenciesList? = null,
    effect: () -> Unit
) {
    // ...
}

fun useEffectWithCleanup(
    dependencies: RDependenciesList? = null,
    effect: () -> Rcleanup
) {
    // ...
}
```

mapped and conditional types

even more power in typescript

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};

type PromiseType<T extends Promise<any>> =
    T extends Promise<infer U> ? U : never;
```

Type conditional

jsx vs dsl

typescript > kotlin

- Kotlin's language features are more flexible for DSLs
- JSX is a more elegant solution for build React specifically
- TypeScript's advanced type system is very powerful

So much power!!

Round 4

async await vs coroutines

asynchronous programming
promises and async/await

- **Async await is a good async solution in JS & TS**
  - Engineered so it interops with Promises
  - Succinct

```
async function loadMap(url: string): Promise<void> {
  const response = await fetch(url);
  const map = await response.text();

  // ...
}
```

coroutines

kotlin > typescript

- **Kotlin's more general coroutines are better**
  - Works with other patterns than simply async
- **In KotlinJS, it works easily with Promises**

```
suspend fun loadMap(url: String) {
    val response = window.fetch(url).await()
    val map = response.text().await()

    // ...
}
```

coroutines

kotlin > typescript

- **Coroutines are more general and powerful**
  - They can be used with other patterns too
- **With suspend functions we don't need to "await"**

```
suspend fun loadMap(url: String) {
    val map = client.get<String>(url)

    // ...
}
```

Ktor Client

Round
5

elegant syntax

expressions

kotlin > typescript

- **Kotlin doesn't have the ternary, but has more**
  - **when** for basic pattern matching
  - **if** and **when** are expressions
  - **Unit** instead of void
  - Expression bodied functions
- **This creates more symmetry in code**

typescript
chained ternaries

```tsx
const App: FC = () => {
  // ...
  return (
    <div>
      // ...
      {gameState === GameState.Start ? <StartScreen/> :
       gameState === GameState.Playing ? <PlayingGame/> :
       gameState === GameState.Dead ? <DeathScreen/> :
       gameState === GameState.Win ? <WinScreen/> :
       null}
    </div>
  );
};
```

kotlin

when expression

```
val App = FC {
    // ...
    div {
        // ...
        when (gameState) {
            GameState.Start -> StartScreen()
            GameState.Playing -> PlayingScreen()
            GameState.Win -> WinScreen()
            GameState.Dead -> EndScreen()
        }
    }
}
```

destructuring

typescript > kotlin

- **Both languages support object destructuring**
  - Within blocks and in lambda parameters
- **However, Kotlin's is limited**
  - Supported via *`componentN`* methods
  - Fixed order to properties extracted
  - Data classes do this automatically

```
const {value, color} = brick;
```

Arbitrary properties extracted

```
const {value, location} = brick;
```

Destructuring on parameters 

```
export const Brick: FC<Props> = ({index}) => {
}
```

Array destructuring

```
const [first, ...remaining] = bricks;
```

conclusion

they’re both very good

but in different ways

comparison summary

both have advantages and disadvantages over each other

Name	Winner
Community	TypeScript
Coroutines	Kotlin
JSX vs React DSL	TypeScript
Multiplatform	Kotlin
Advanced Type System	TypeScript
Extensions	Kotlin
Interop with JS	TypeScript
Expressions	Kotlin
Destructuring	TypeScript
Standard Library	Kotlin
Tooling	TypeScript
Functions	Kotlin

conclusion

kotlinjs
new and upcoming

- New compiler backend
- Improved code emitting
- TypeScript definition files generated on compile
- More multiplatform libraries
- WebAssembly

Questions?

references

https://www.youtube.com/watch?v=fZUL8_kgHXg

more links

- https://kotlinlang.org/
- https://kotlinlang.org/docs/reference/js-overview.html
- https://kotlinlang.slack.com
- https://www.typescriptlang.org/
- https://www.typescriptlang.org/docs/handbook/intro.html