Hope is postponed disappointment

November 19, 2023 – JSConf JP

Luca Mugnaini

Rakuten Group, Inc.

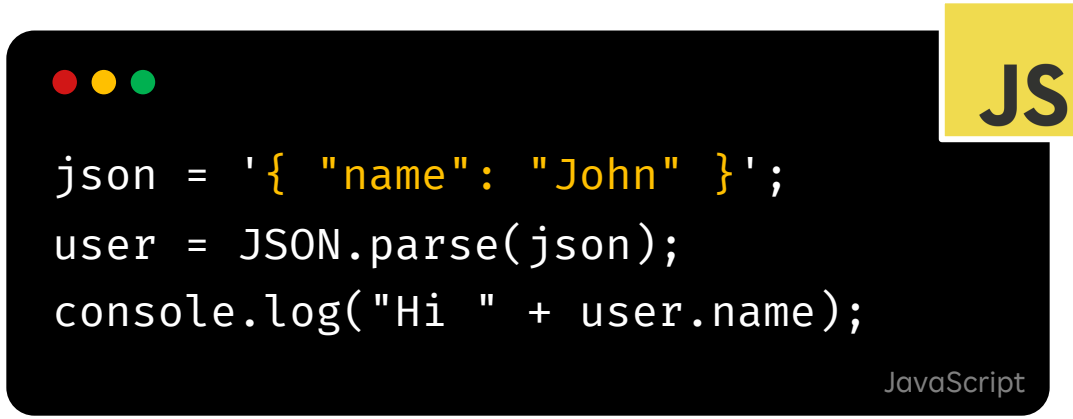
Rakuten

Hope is postponed disappointment

Luca Mugnaini

Landwasser Viaduct, Graubunden, Switzerland - Photo by Susanne Jutzeler
https://www.pexels.com/photo/17675783/
(November 8, 2023)

Source: https://wordart.com/ (November 8, 2023)
Hope is postponed disappointment


JS
json = '{ "name": "John" }';
user = JSON.parse(json);
console.log("Hi " + user.name);
JavaScript

```javascript
json = '{ "name": "John" }';
user = JSON.parse(json);
console.log("Hi " + user.name);
```

JavaScript

```
{ "name": "John" } -> Hi John
```

JavaScript

```
json = '{ "name": "John" }';
user = JSON.parse(json);
console.log("Hi " + user.name);
```

JavaScript

```
{ "name": "John" } -> Hi John
{ "n": "John" }    -> Hi undefined
```

JavaScript


```
{ "name": "John"
{ "n": "John" }
```

JavaScript

Source https://twitter.com/rickhanlonii/status/1718735181538795750 (November 8, 2023)

```
json = '{ "name": "Jo';
user = JSON.parse(json);
console.log("Hi " + user.name);
```

JavaScript

```
{ "name": "John" } -> Hi John
{ "n": "John" }    -> Hi undefined
{ "name": "Jo      -> Crashes with "SyntaxError: Unexpected end of JSON input"
```

JavaScript

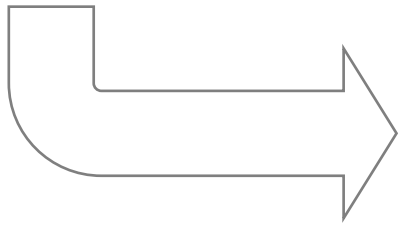
https://en.wikipedia.org/wiki/Montparnasse_derailment#/media/File:Train_wreck_at_Montparnasse_1895.jpg (November 8, 2023)

Handling errors requires discipline

Enhanced by disciplined developer

```javascript
json = '{ "name": "John" }';
user = JSON.parse(json);
console.log("Hi " + user.name);
```

JavaScript



```javascript
json = '{ "name": "John" }';
try {
    user = JSON.parse(json);
    if (!user.name) {
        throw new Error("Name is missing")
    }
    console.log("Hi " + user.name);
} catch(error) {
    console.log("Error: " + error.message);
}
```

JavaScript

Enhanced by disciplined developer

```javascript
json = '{ "name": "John" }';
try {
    user = JSON.parse(json);
    if (!user.name) {
        throw new Error("Name is missing")
    }
    console.log("Hi " + user.name);
} catch(error) {
    console.log("Error: " + error.message);
}
```

JavaScript

```
{ "name": "John" } -> Hi John
```

JavaScript

Enhanced by disciplined developer

```javascript
json = '{ "n": "John" }';
try {
    user = JSON.parse(json);
    if (!user.name) {
        throw new Error("Name is missing")
    }
    console.log("Hi " + user.name);
} catch(error) {
    console.log("Error: " + error.message);
}
```

JavaScript

```
{ "name": "John" } -> Hi John
{ "n": "John" }    -> Error: Name is missing
```

JavaScript

Enhanced by disciplined developer

```javascript
json = '{ "name": "Jo';
try {
    user = JSON.parse(json);
    if (!user.name) {
        throw new Error("Name is missing")
    }
    console.log("Hi " + user.name);
} catch(error) {
    console.log("Error: " + error.message);
}
```

JavaScript

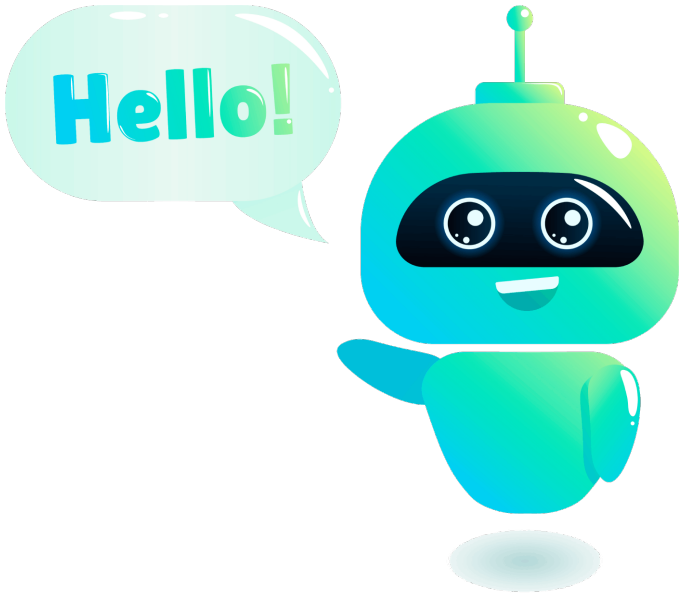
```
{ "name": "John" } -> Hi John
{ "n": "John" }    -> Error: Name is missing
{ "name": "Jo      -> Error: Unexpected end of JSON input
```

JavaScript

Can we replicate the **disciplined developers** with an assistant in the form of a **static type system compiler**?

```javascript
json = '{ "name": "Johm" }';
try {
    user = JSON.parse(json);
    if (!user.name) {
        throw new Error("Name is missing")
    }
    console.log("Hi " + user.name);
} catch(error) {
    console.log("Error: " + error.message);
}
```

JavaScript

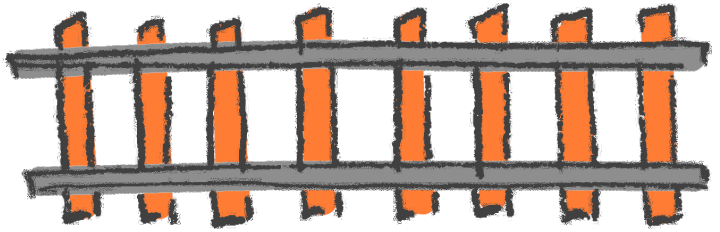


Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

Railway Metaphor

IN

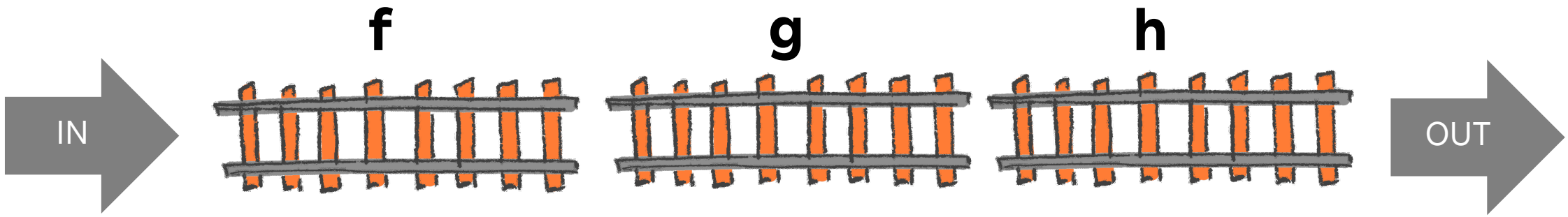
function

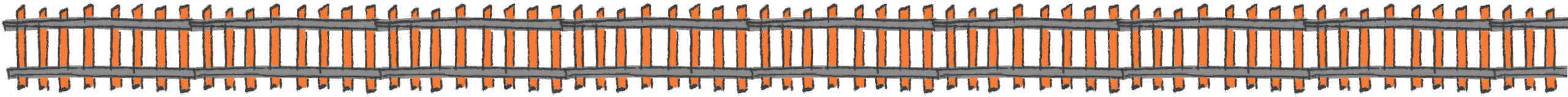


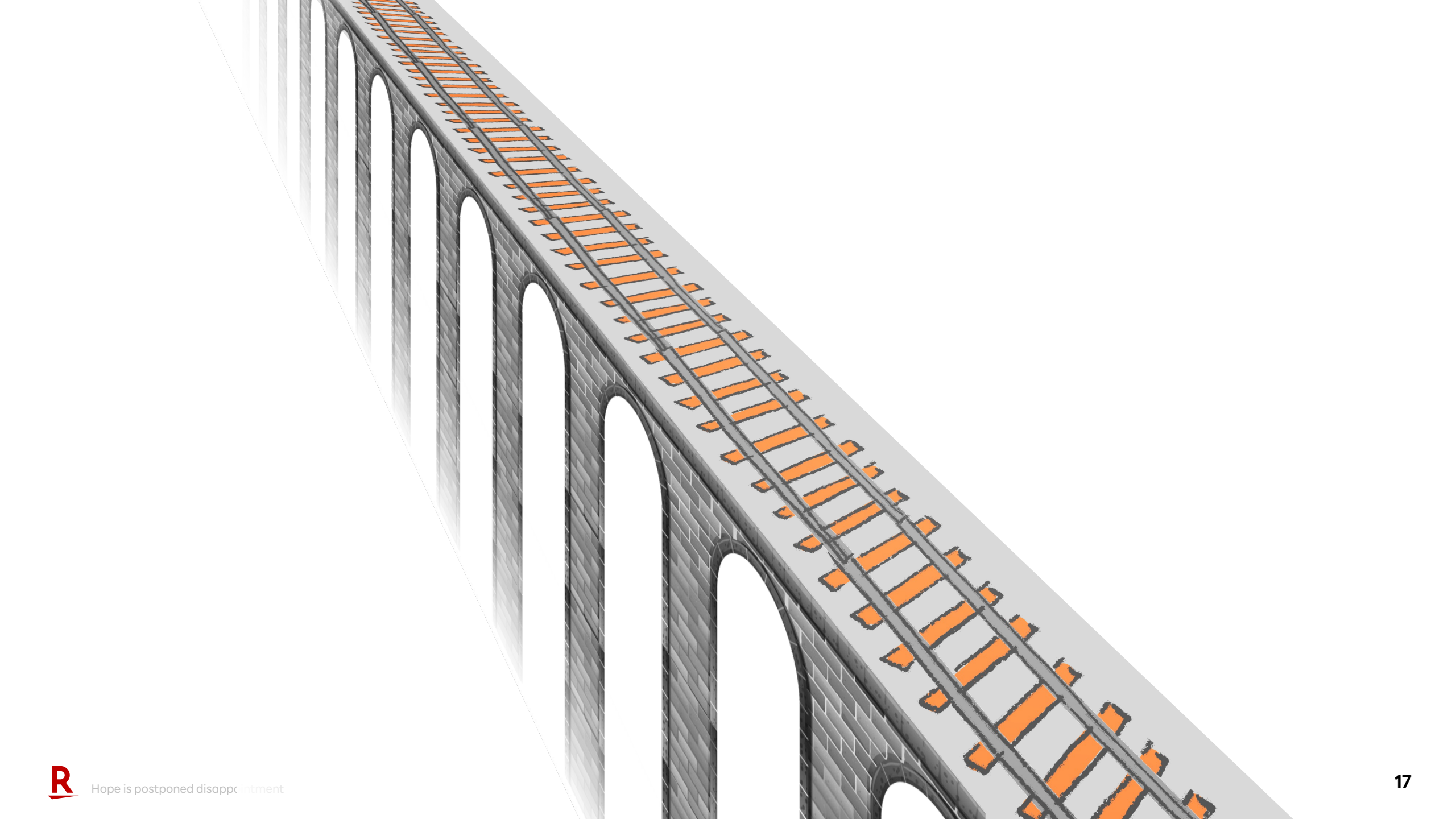
OUT

Railway Metaphor – Function composition

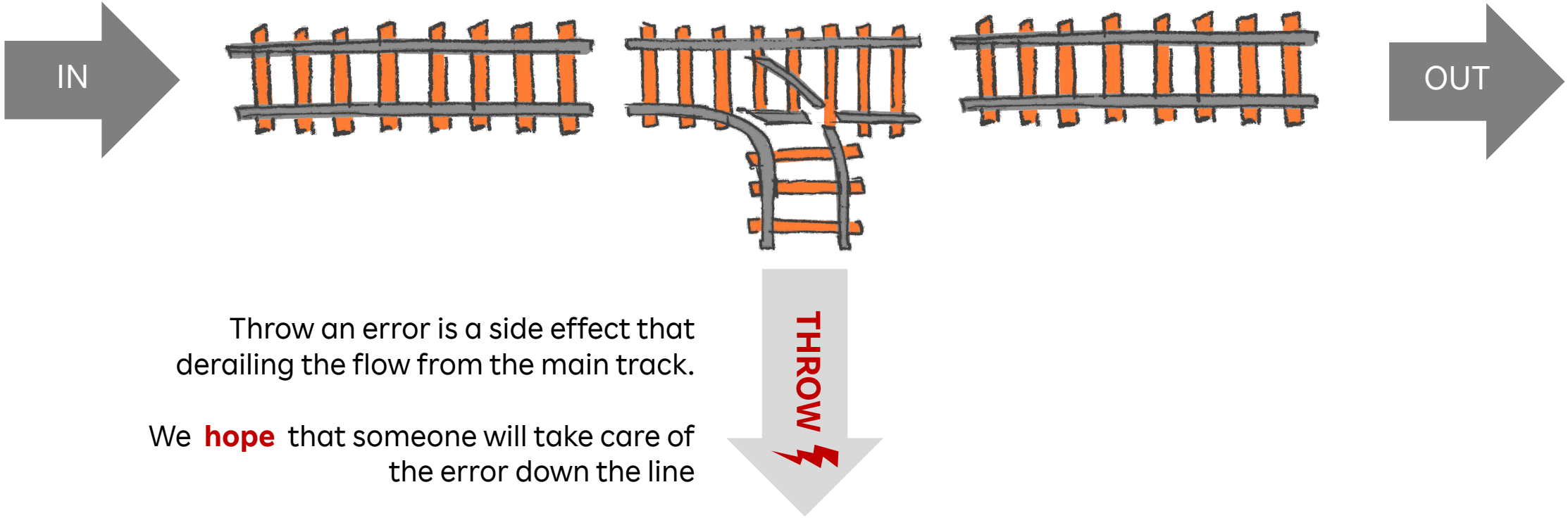
```
OUT = h ( g ( f ( IN ) ) )
```





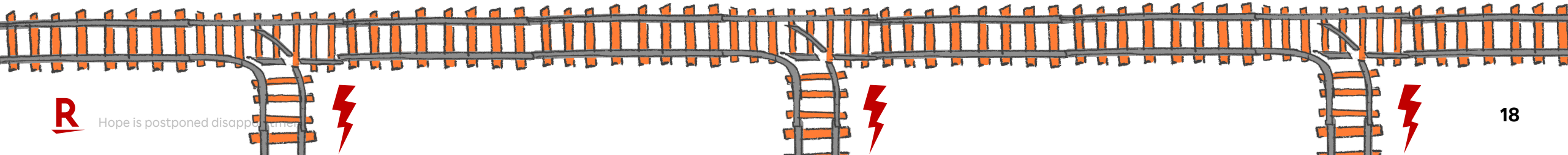


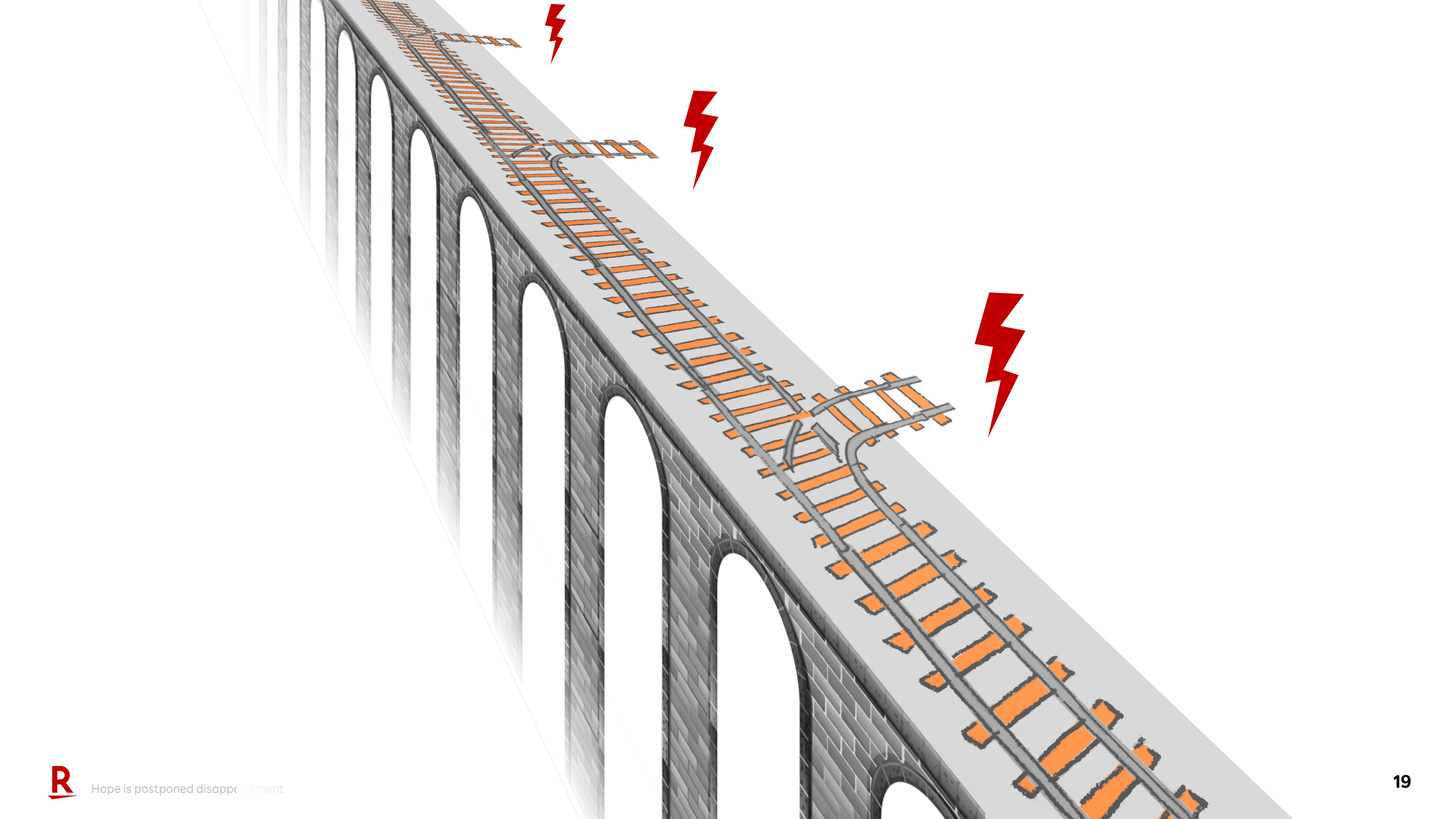
Throwing exceptions

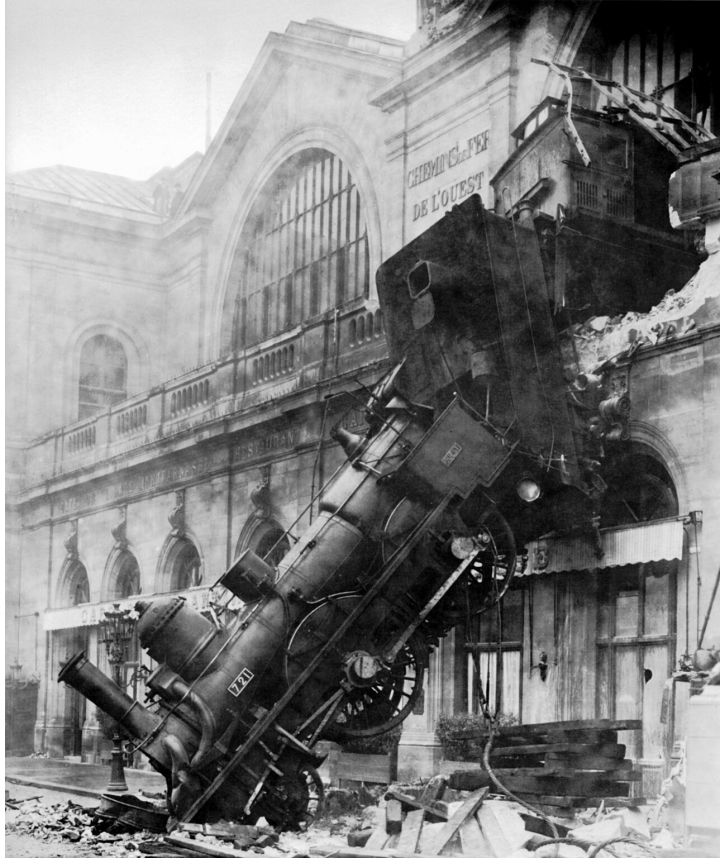


Throw an error is a side effect that derailing the flow from the main track.

We **hope** that someone will take care of the error down the line







https://en.wikipedia.org/wiki/Montparnasse_derailment#/media/File:Train_wreck_at_Montparnasse_1895.jpg (November 8, 2023)

The Result Type

A type holding an **ok value**
or an **error value**

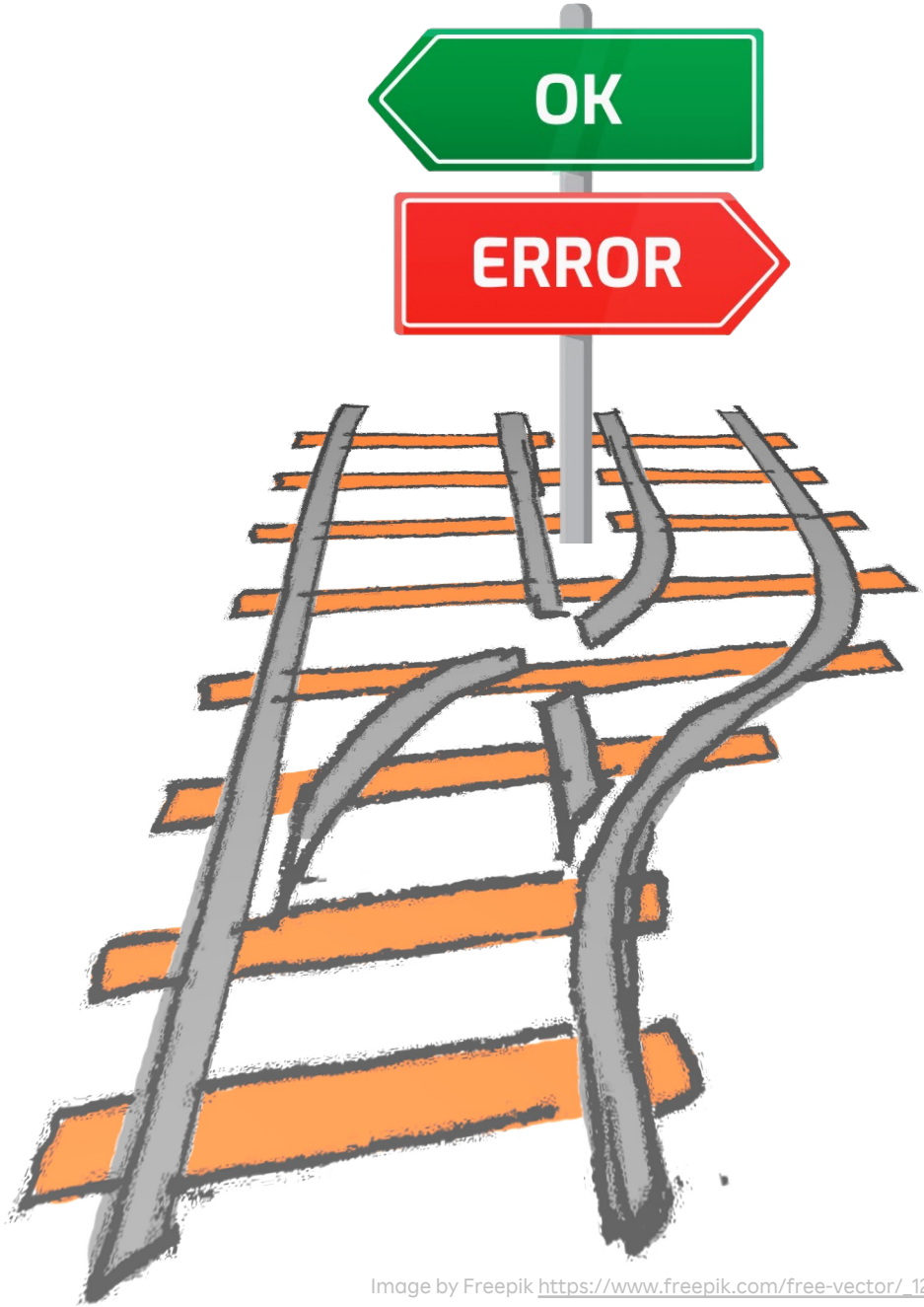


Image by Freepik https://www.freepik.com/free-vector/_1253756.htm (November 8, 2023)

The Result Type

A type holding an **ok value**
or an **error value**

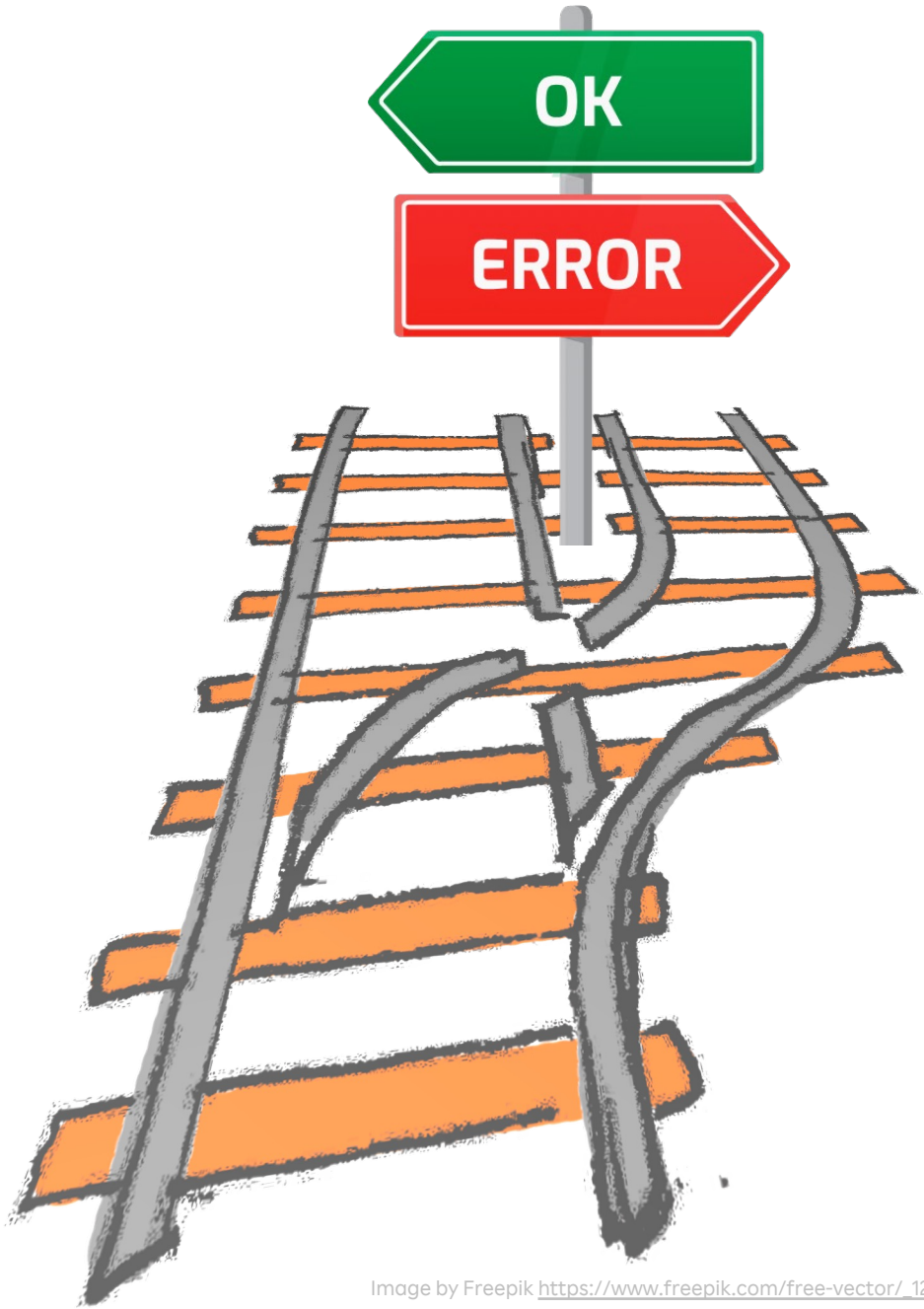
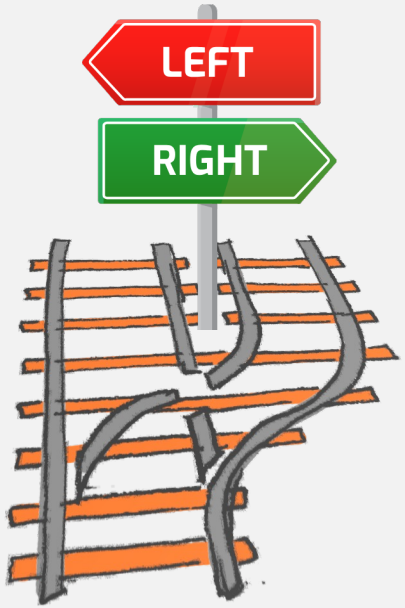
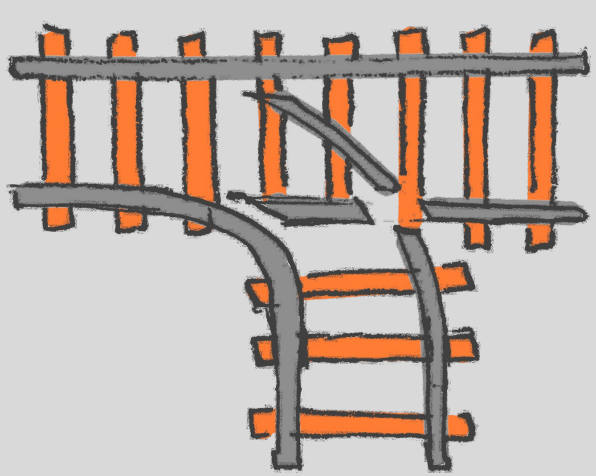


Image by Freepik https://www.freepik.com/free-vector/_1253756.htm (November 8, 2023)

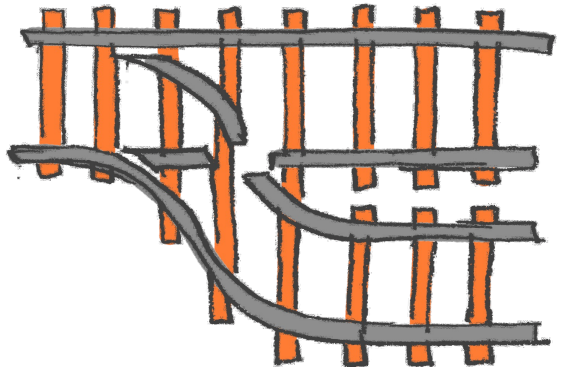
The Either Type





value

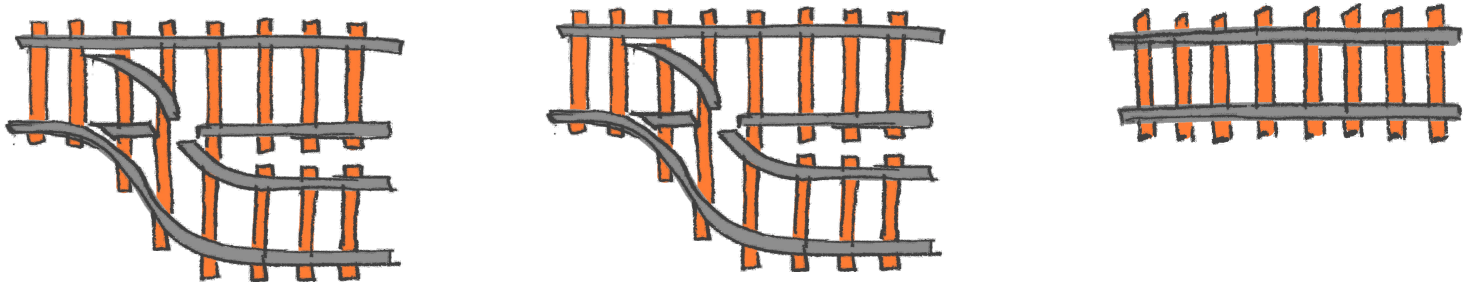
throw



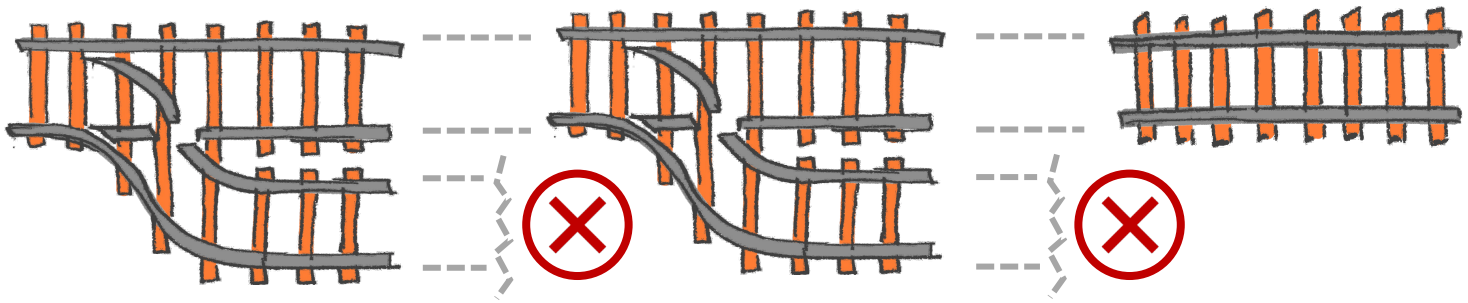
Ok value

Err error

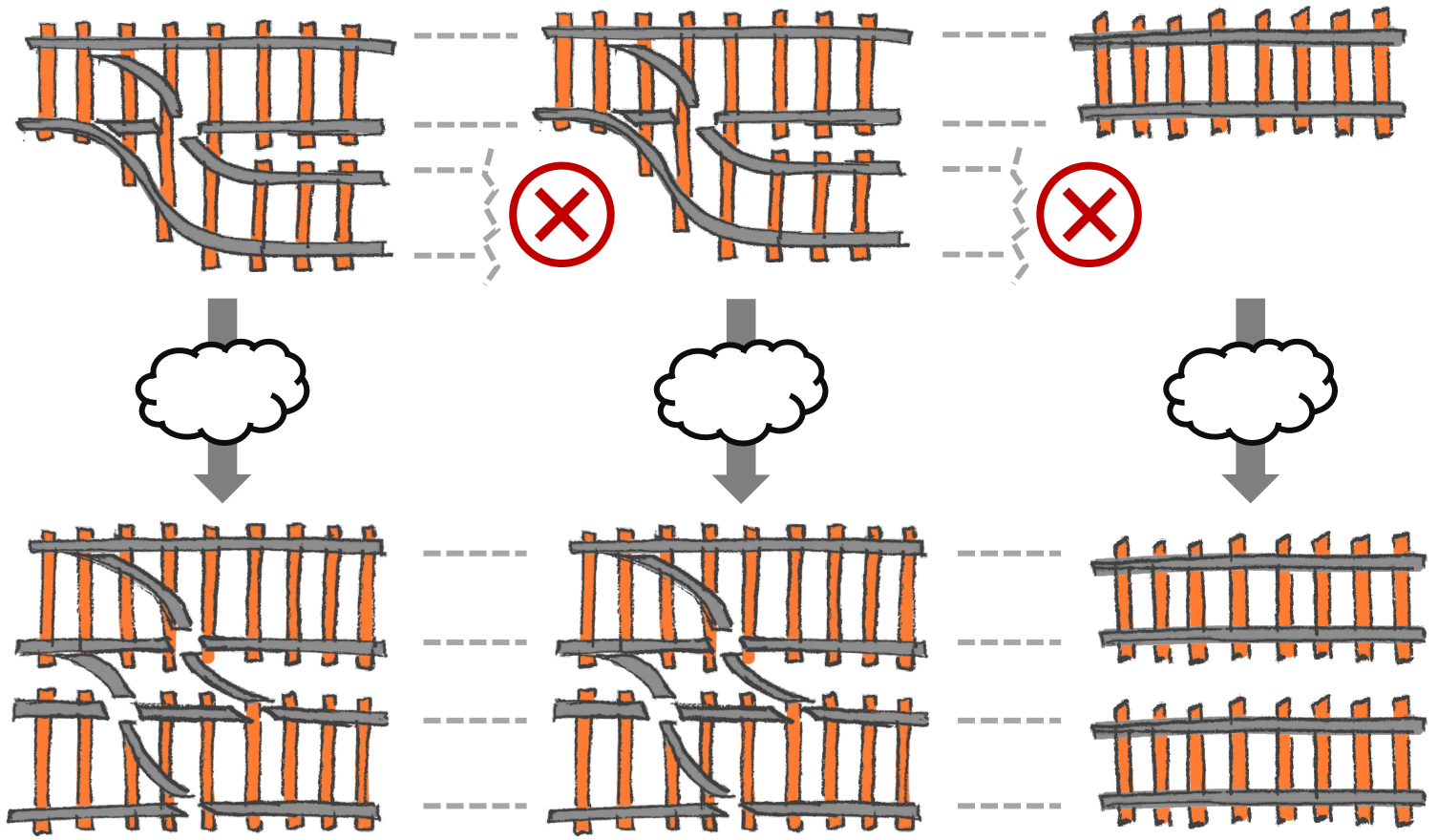
The Result Type - Composition



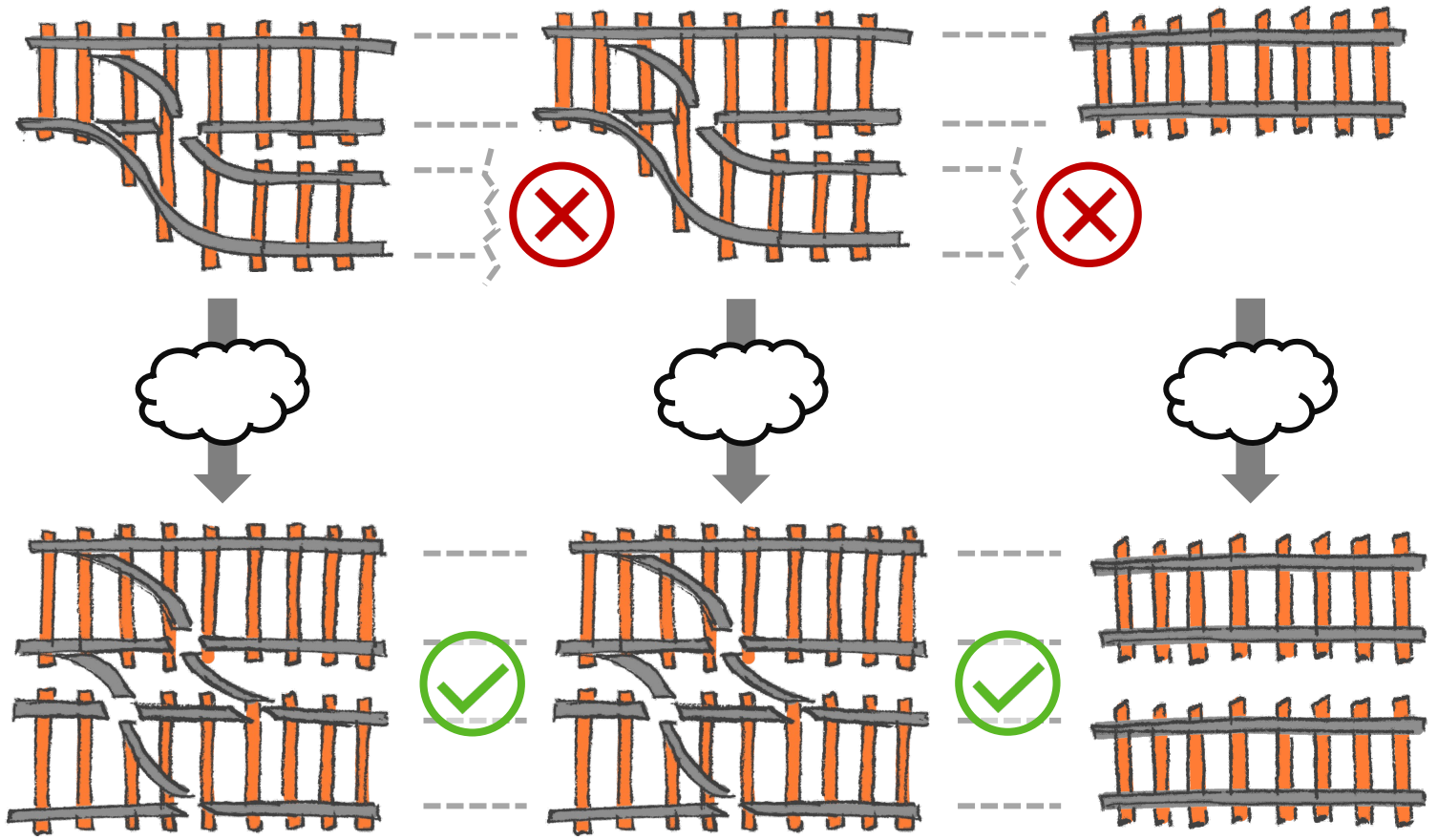
The Result Type - Composition

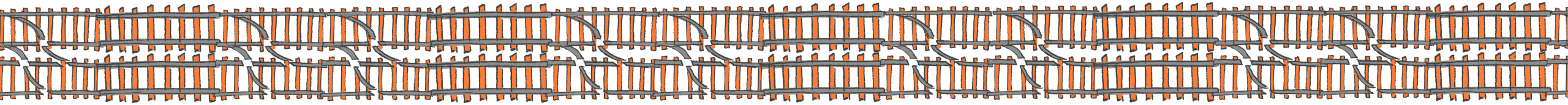


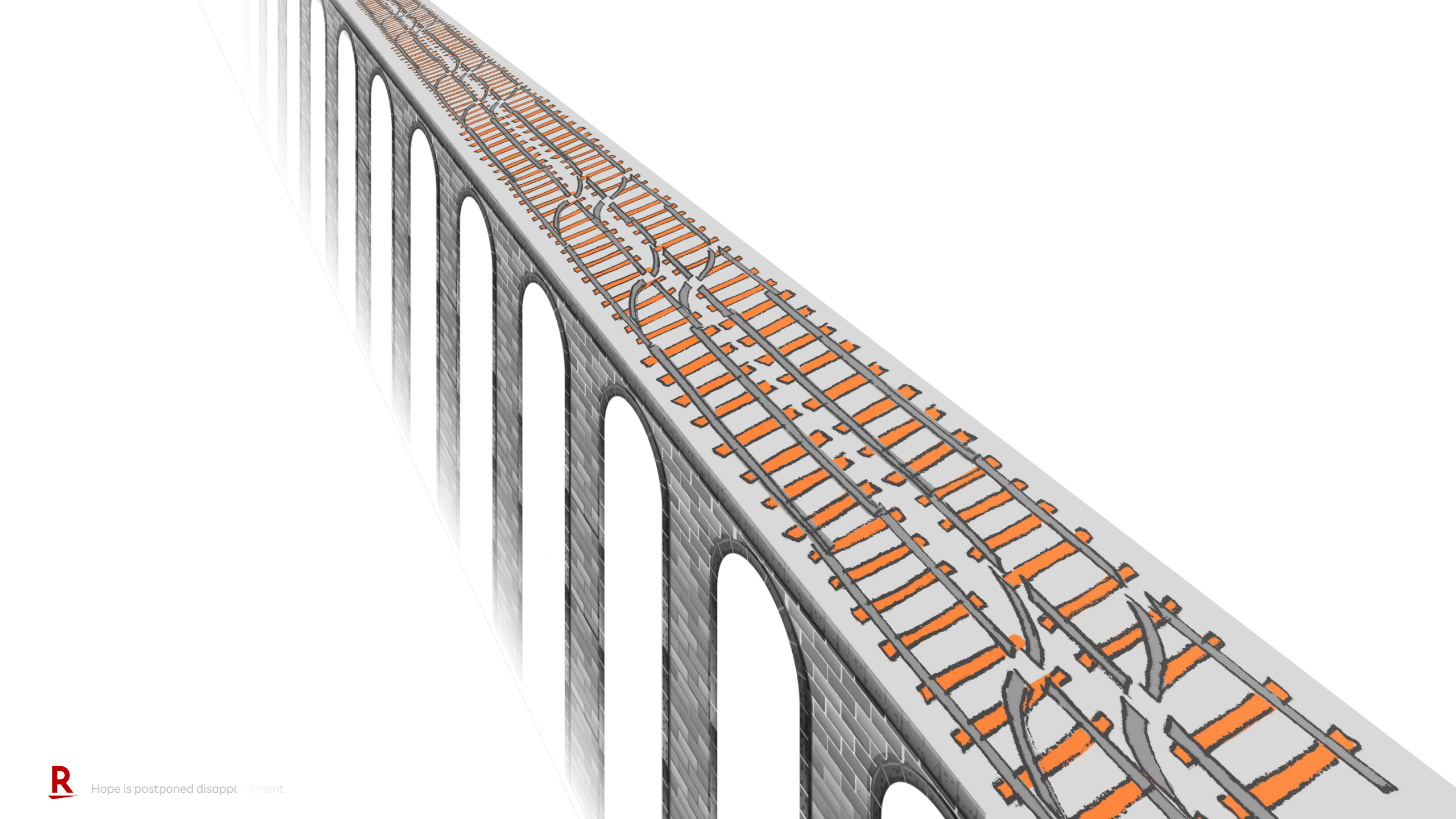
The Result Type - Composition



The Result Type - Composition







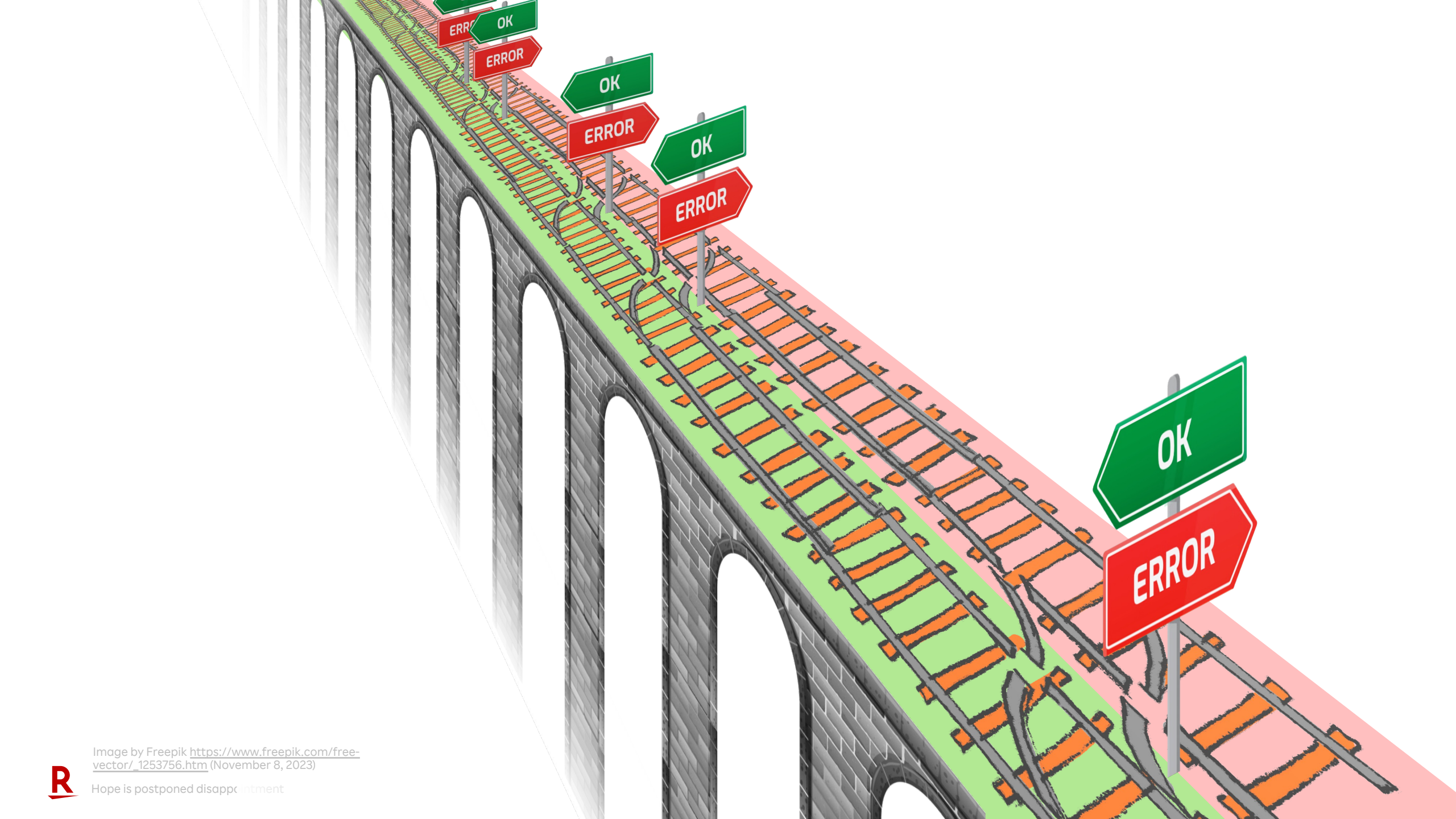


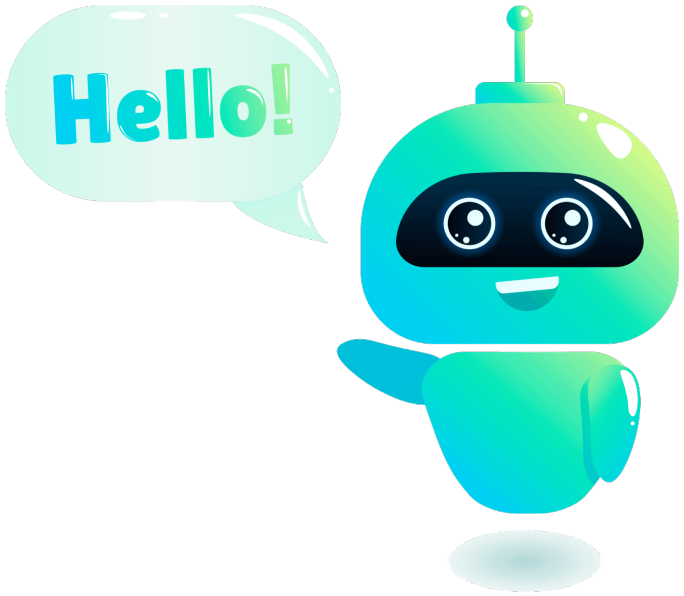
Image by Freepik https://www.freepik.com/free-vector/_1253756.htm (November 8, 2023)

Hope is postponed disappointment

Can we replicate the **disciplined developers** with an assistant in the form of a **static type system compiler**?

```javascript
json = '{ "name": "Johm" }';
try {
    user = JSON.parse(json);
    if (!user.name) {
        throw new Error("Name is missing")
    }
    console.log("Hi " + user.name);
} catch(error) {
    console.log("Error: " + error.message);
}
```

JavaScript



Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

Using the Result type in Elm

The analog of **`JSON.parse()`** in Elm returns a **Result Type** because it is a function that can fail

Using the returned value directly will cause a **type mismatch** error

```
Elm Compiler error:

This `user` value is a: Result Error User
But I need a record with a `name` field!
```

Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

Using the Result type in Elm

We need to unbox the returned value before using it (via pattern matching):

```elm
case result of
   Ok user -> text ("Hi " ++ user.name)
```

Elm

We forgot to consider the error case but **`case..of`** is an exhaustive switch statement so the code will not compile

```
Elm Compiler error:

This `case` does not have branches for all
possibilities. Missing possibilities include Err
```

Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

Using the Result type in Elm

```elm
case result of
    Ok user -> text ("Hi " ++ user.name)
    Err error -> text ("Error: " ++ errorToString error)
```

Elm

The **ResultType** and the **compiler guided us** in the right direction, regardeless our discipline level

From JavaScript to Elm, with the assistance of the compiler

```javascript
json = '{ "name": "Johm" }';
try {
    user = JSON.parse(json);
    if (!user.name) {
        throw new Error("Name is missing")
    }
    console.log("Hi " + user.name);
} catch(error) {
    console.log("Error: " + error.message);
}
```

JavaScript

```elm
type alias User = { name : String }

parseJSON : String -> Result Error User
parseJSON =
    field "name" string |> map User |> decodeString

json = "{ \"name\": \"John\" }"
result = parseJSON json

main = case result of
    Ok user ->
        text ("Hi " ++ user.name)

    Err error ->
        text ("Error: " ++ errorTo
```

Elm

It looks good!

Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

Let's try to achieve a similar result using the type system of TypeScript

Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

The Result Type TypeScript (and other languages)

```TypeScript
// TypeScript
type Result <E, T> = { tag: 'Ok'; value: T } | { tag: 'Err'; error: E }
```

The Result Type TypeScript (and other languages)

```typescript
// TypeScript
type Result <E, T> = { tag: 'Ok'; value: T } | { tag: 'Err'; error: E }
```

TypeScript

```elm
-- Elm
type Result error value = Ok value | Err error
```

Elm

```kotlin
// Kotlin
sealed class Result <out E, out T> {
    data class  Ok <T>(val value: T) : Result<Nothing, T>()
    data class Err <E>(val error: E) : Result<E, Nothing>()
}
```

Kotlin

From Elm to TypeScript

```elm
-- type Result error value = Ok value | Err error

type alias User = { name : String }

parseJSON : String -> Result Error User
parseJSON =
    field "name" string |> map User |> decodeString




json = "{ \"name\": \"John\" }"
result = parseJSON json

main = case result of
    Ok user ->
        text ("Hi " ++ user.name)

    Err error ->
        text ("Error: " ++ errorToString error)
```

Elm


```typescript
type Result <Err, Ok> = { kind: 'Ok'; value: Ok } | { kind: 'Err'; error: Err };

type User = { name: string };

function parseJSON (json: string | undefined): Result <string, User> {
  try {
    if (json === undefined) {
      return { kind: 'Err', error: 'JSON is undefined' };
    }
    const user = JSON.parse(json) as User;
    if (!user || !user.name) {
      return { kind: 'Err', error: '"name" key is missing' };
    }
    return { kind: 'Ok', value: user };
  } catch (error) {
    return { kind: 'Err', error: `Invalid JSON ${error}` };
  }
}

const json = '{ "name": "John" }';
const result = parseJSON(json);

switch (result.kind) {
  case 'Ok':
    console.log("Hi " + result.value.name);
    break;
  case 'Err':
    console.log("Error: " + result.error);
    break;
  default:
    const exhaustiveCheck: never = result;
    break;
}
```

TypeScript

```elm
-- type Result error value = Ok value | Err error

type alias User = { name : String }

parseJSON : String -> Result Error User
parseJSON =
    field "name" string |> map User |> decodeString




json = "{ \"name\": \"John\" }"
result = parseJSON json

main = case result of
    Ok user ->
        text ("Hi " ++ user.name)

    Err error ->
        text ("Error: " ++ errorToString error)
```

Elm

```typescript
type Result <Err, Ok> = { kind: 'Ok'; value: Ok } | { kind: 'Err'; error: Err };

type User = { name: string };

function parseJSON (json: string | undefined): Result <string, User> {
  try {
    if (json === undefined) {
      return { kind: 'Err', error: 'JSON is undefined' };
    }
    const user = JSON.parse(json) as User;
    if (!user || !user.name) {
      return { kind: 'Err', error: '"name" key is missing' };
    }
    return { kind: 'Ok', value: user };
  } catch (error) {
    return { kind: 'Err', error: `Invalid JSON ${error}` };
  }
}

const json = '{ "name": "John" }';
const result = parseJSON(json);

switch (result.kind) {
  case 'Ok':
    console.log("Hi " + result.value.name);
    break;
  case 'Err':
    console.log("Error: " + result.error);
    break;
  default:
    const exhaustiveCheck: never = result;
    break;
}
```

TypeScript

```elm
-- type Result error value = Ok value | Err error

type alias User = { name : String }

parseJSON : String -> Result Error User
parseJSON =
    field "name" string |> map User |> decodeString
```


```elm
json = "{ \"name\": \"John\" }"
result = parseJSON json

main = case result of
    Ok user ->
        text ("Hi " ++ user.name)

    Err error ->
        text ("Error: " ++ errorToString error)
```

Elm

```typescript
type Result <Err, Ok> = { kind: 'Ok'; value: Ok } | { kind: 'Err'; error: Err };

type User = { name: string };

function parseJSON (json: string | undefined): Result <string, User> {
  try {
    if (json === undefined) {
      return { kind: 'Err', error: 'JSON is undefined' };
    }
    const user = JSON.parse(json) as User;
    if (!user || !user.name) {
      return { kind: 'Err', error: '"name" key is missing' };
    }
    return { kind: 'Ok', value: user };
  } catch (error) {
    return { kind: 'Err', error: `Invalid JSON ${error}` };
  }
}

const json = '{ "name": "John" }';
const result = parseJSON(json);

switch (result.kind) {
  case 'Ok':
    console.log("Hi " + result.value.name);
    break;
  case 'Err':
    console.log("Error: " + result.error);
    break;
  default:
    const exhaustiveCheck: never = result;
    break;
}
```

TypeScript

```elm
-- type Result error value = Ok value | Err error

type alias User = { name : String }

parseJSON : String -> Result Error User
parseJSON =
    field "name" string |> map User |> decodeString



json = "{ \"name\": \"John\" }"
result = parseJSON json

main = case result of
    Ok user ->
        text ("Hi " ++ user.name)

    Err error ->
        text ("Error: " ++ errorToString error)
```

Elm

```typescript
type Result <Err, Ok> = { kind: 'Ok'; value: Ok } | { kind: 'Err'; error: Err };

type User = { name: string };

function parseJSON (json: string | undefined): Result <string, User> {
  try {
    if (json === undefined) {
      return { kind: 'Err', error: 'JSON is undefined' };
    }
    const user = JSON.parse(json) as User;
    if (!user || !user.name) {
      return { kind: 'Err', error: '"name" key is missing' };
    }
    return { kind: 'Ok', value: user };
  } catch (error) {
    return { kind: 'Err', error: `Invalid JSON ${error}` };
  }
}

const json = '{ "name": "John" }';
const result = parseJSON(json);

switch (result.kind) {
  case 'Ok':
    console.log("Hi " + result.value.name);
    break;
  case 'Err':
    console.log("Error: " + result.error);
    break;
  default:
    const exhaustiveCheck: never = result;
    break;
}
```

TypeScript

```elm
-- type Result error value = Ok value | Err error

type alias User = { name : String }

parseJSON : String -> Result Error User
parseJSON =
    field "name" string |> map User |> decodeString

json = "{ \"name\": \"John\" }"
result = parseJSON json

main = case resul
    Ok user ->
        text ("Hi

    Err error ->
        text ("Er
```

```typescript
type Result <Err, Ok> = { kind: 'Ok'; value: Ok } | { kind: 'Err'; error: Err };

type User = { name: string };

function parseJSON (json: string | undefined): Result <string, User> {
  try {
  } catch (error) {
    return { kind: 'Err', error: `Invalid JSON ${error}` };
  }
}

const json = '{ "name": "John" }';
const result = parseJSON(json);
```

Elm Compiler error:

This `user` value is a: **Result Error User**

But I need a record with a `name` field!

TypeScript Compiler error:

Property '**value**' does not exist on type '**Result<string, User>**'

TypeScript

Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

Hope is postponed disappointment

```elm
-- type Result error value = Ok value | Err error

type alias User = { name : String }

parseJSON : String -> Result Error User
parseJSON =
    field "name" string |> map User |> decodeString



json = "{ \"name\": \"John\" }"
result = parseJSON json

main = case resul
    Ok user ->
        text ("Hi

    Err error ->
        text ("Er
```

```typescript
type Result <Err, Ok> = { kind: 'Ok'; value: Ok } | { kind: 'Err'; error: Err };

type User = { name: string };

function parseJSON (json: string | undefined): Result <string, User> {
  try {
  } catch (error) {
    return { kind: 'Err', error: `Invalid JSON ${error}` };
  }
}

const json = '{ "name": "John" }';
const result = parseJSON(json);
```

Elm Compiler error:

This `case` does not have branches for all possibilities. Missing possibilities include **Err**

TypeScript Compiler error:

Type '**{ kind: "Err"; error: string; }**' is not assignable to type '**never**'.

TypeScript

Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

Hope is postponed disappointment

```elm
type alias User = { name : String }

parseJSON : String -> Result Error User
parseJSON =
    field "name" string |> map User |> decodeString

json = "{ \"name\": \"John\" }"
result = parseJSON json

main = case result of
    Ok user ->
        text ("Hi " ++ user.name)

    Err error ->
        text ("Error: " ++ errorTo
```

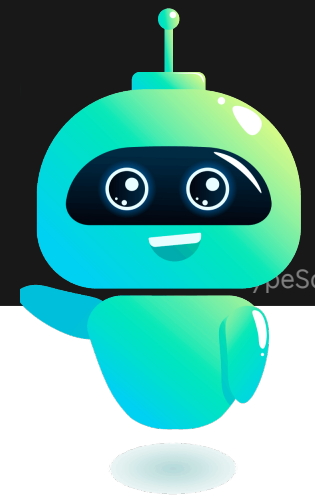
Elm

It looks good!

```typescript
type Result <Err, Ok> = { kind: 'Ok'; value: Ok } | { kind: 'Err'; error: Err };

type User = { name: string };

function parseJSON (json: string | undefined): Result <string, User> {
  try {
    if (json === undefined) {
      return { kind: 'Err', error: 'JSON is undefined' };
    }
    const user = JSON.parse(json) as User;
    if (!user || !user.name) {
      return { kind: 'Err', error: '"name" key is missing' };
    }
    return { kind: 'Ok', value: user };
  } catch (error) {
    return { kind: 'Err', error: `Invalid JSON ${error}` };
  }
}

const json = '{ "name": "John" }';
const result = parseJSON(json);

switch (result.kind) {
  case 'Ok':
    console.log("Hi " + result.value.name);
    break;
  case 'Err':
    console.log("Error: " + result.error);
    break;
  default:
    const exhaustiveCheck: never = result;
    break;
}
```

TypeScript

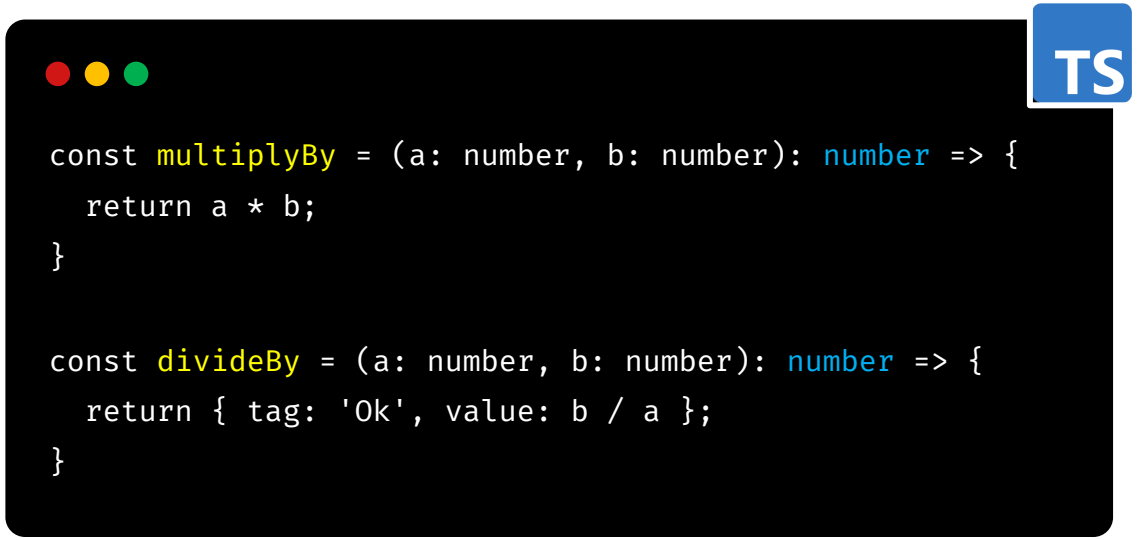
It looks good too!



Robot graphic by svstudioart on Freepik https://www.freepik.com/free-vector/_13830698.htm (November 8, 2023)

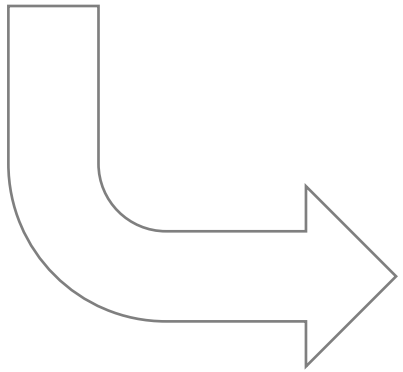
Do we need to rewrite all our functions?

No, only the one that fail

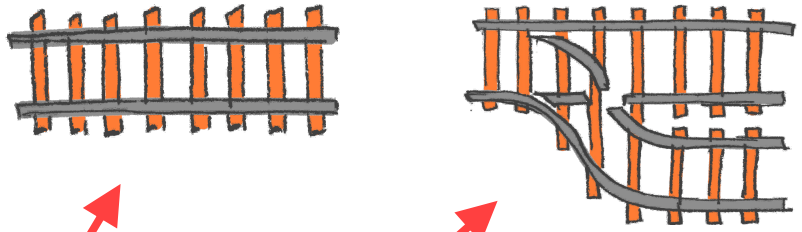


```ts
const multiplyBy = (a: number, b: number): number => {
  return a * b;
}

const divideBy = (a: number, b: number): number => {
  return { tag: 'Ok', value: b / a };
}
```

```ts
const multiplyBy = (a: number, b: number): number => {
  return a * b;
}

const divideBy = (a: number, b: number): number => {
  return { tag: 'Ok', value: b / a };
}
```

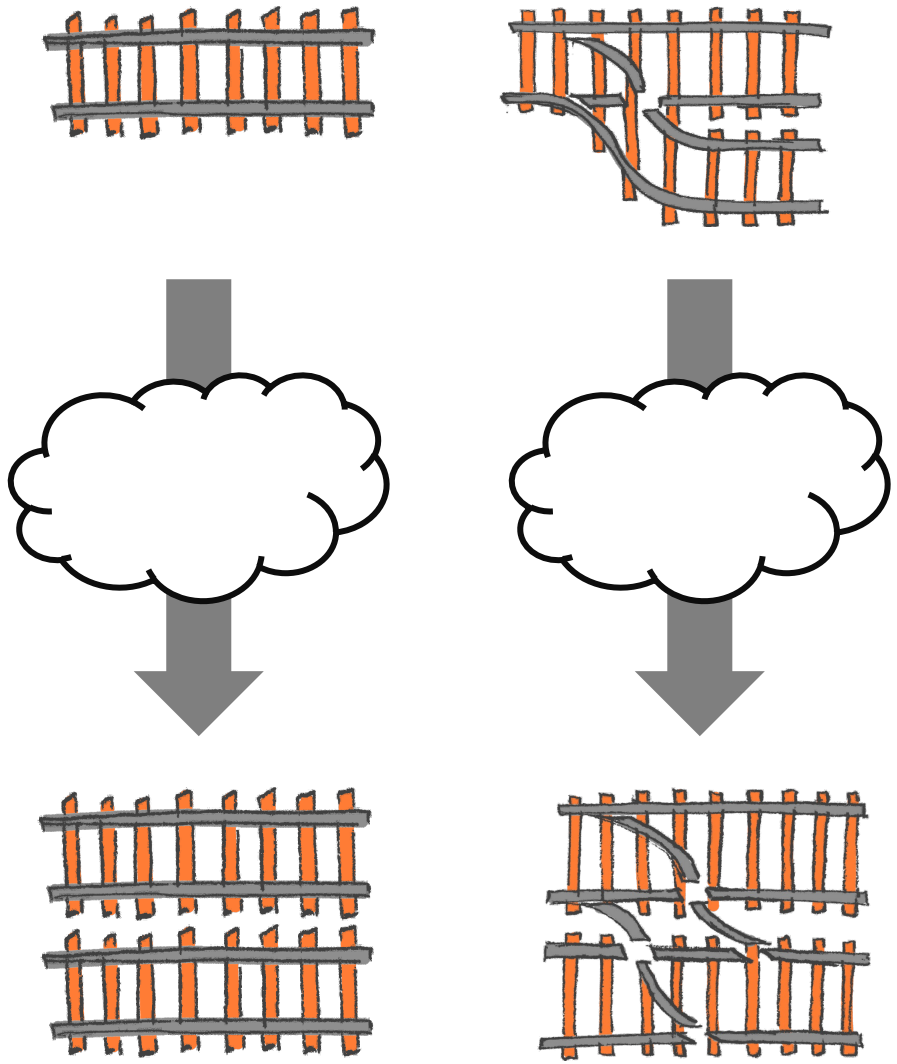


```ts
const multiplyBy = (a: number, b: number): number => {
  return a * b;
}

const divideBy = (a: number, b: number): Result<string, number> => {
  if (a === 0) {
    return { tag: 'Err', error: 'Divided by 0' };
  } else {
    return { tag: 'Ok', value: b / a };
  }
}
```

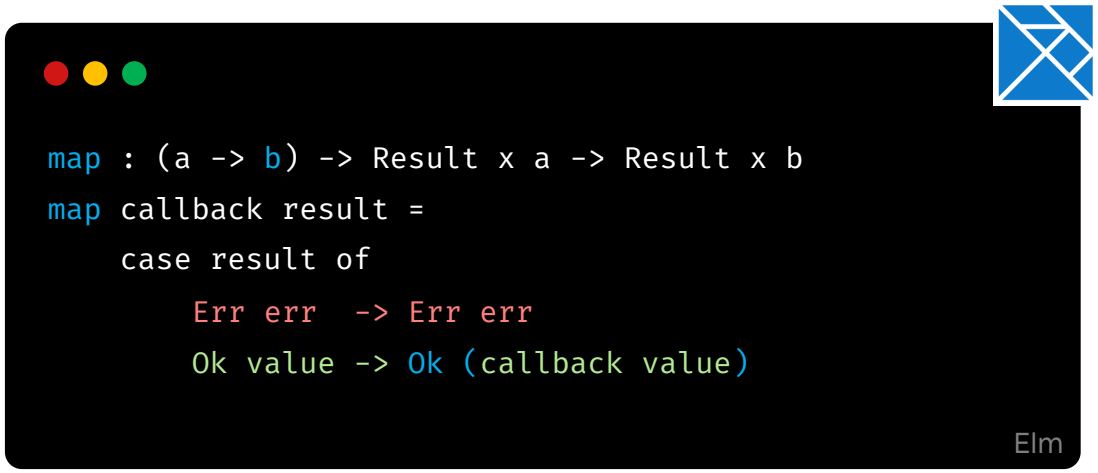


```ts
const multiplyBy = (a: number, b: number): number => {
  return a * b;
}

const divideBy = (a: number, b: number): Result<string, number> => {
  if (a === 0) {
    return { tag: 'Err', error: 'Divided by 0' };
  } else {
    return { tag: 'Ok', value: b / a };
  }
}
```

The Result Type - Helpers

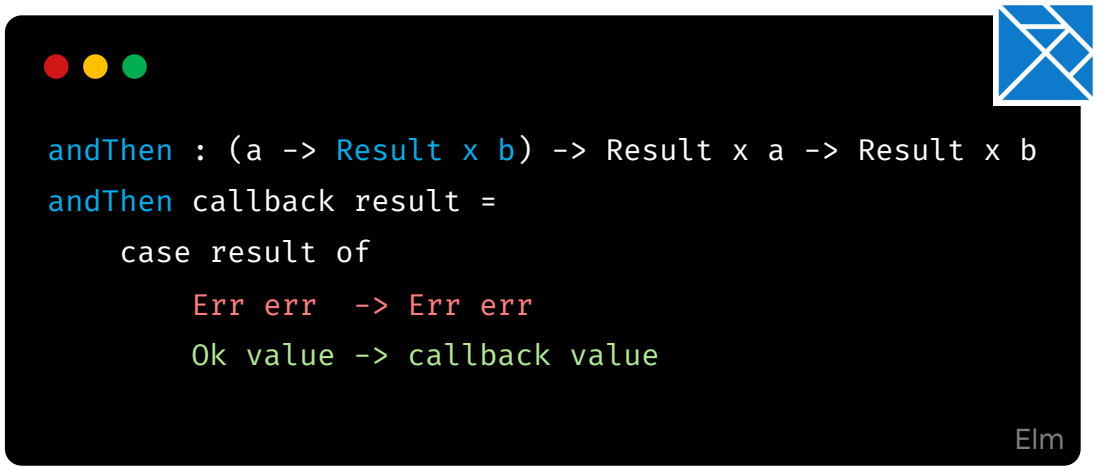


The Result Type - Helpers



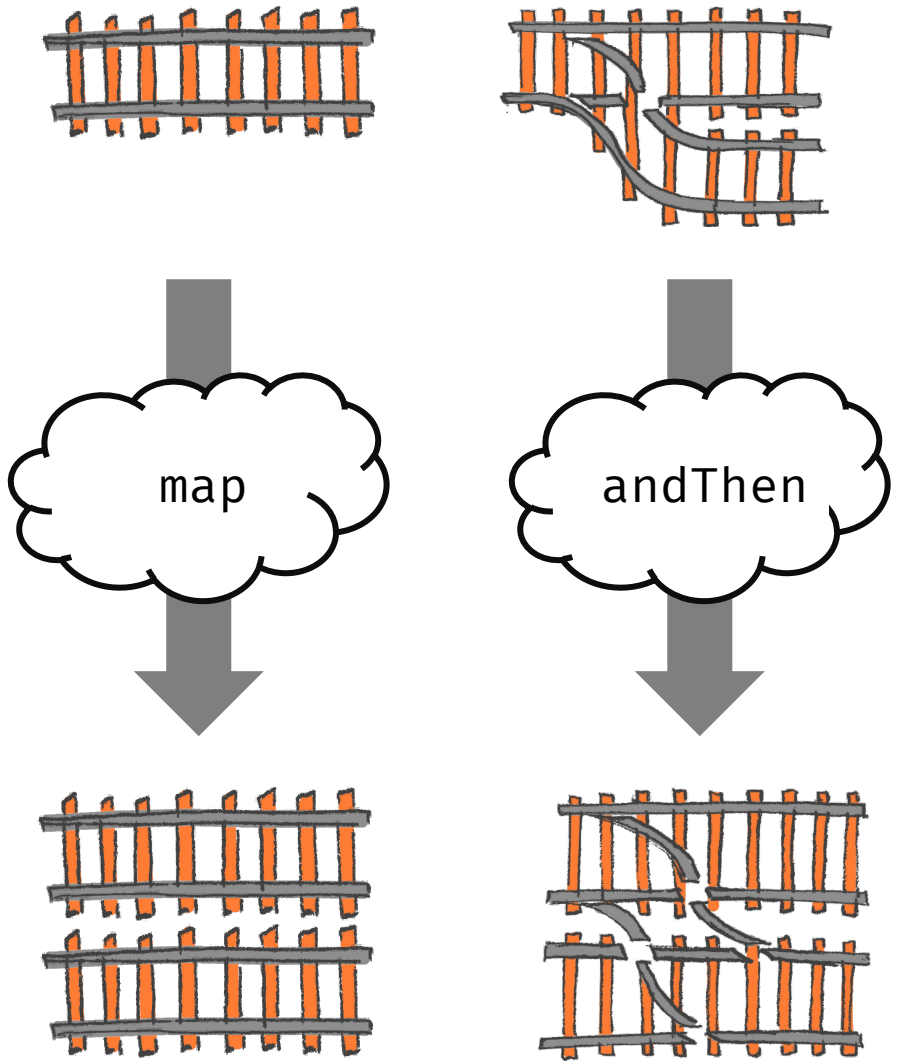
```elm
map : (a -> b) -> Result x a -> Result x b
map callback result =
    case result of
        Err err  -> Err err
        Ok value -> Ok (callback value)
```

Elm

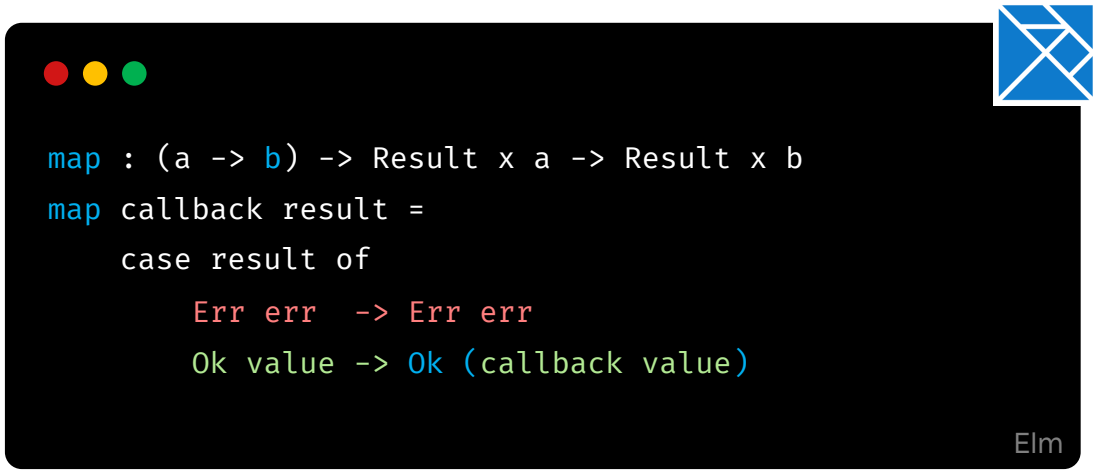


```elm
andThen : (a -> Result x b) -> Result x a -> Result x b
andThen callback result =
    case result of
        Err err  -> Err err
        Ok value -> callback value
```

Elm

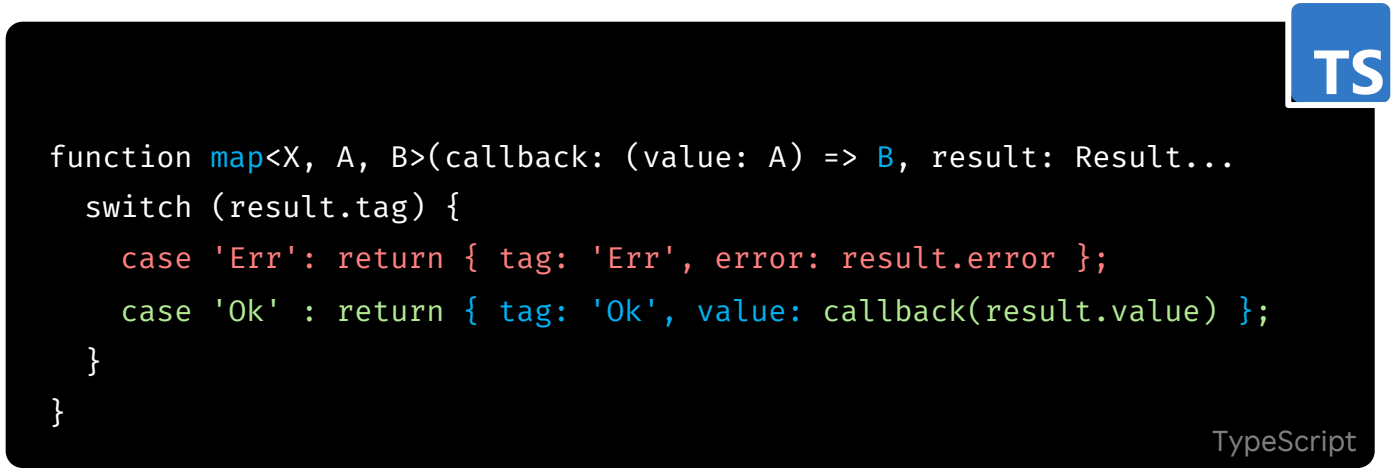


The Result Type - Helpers



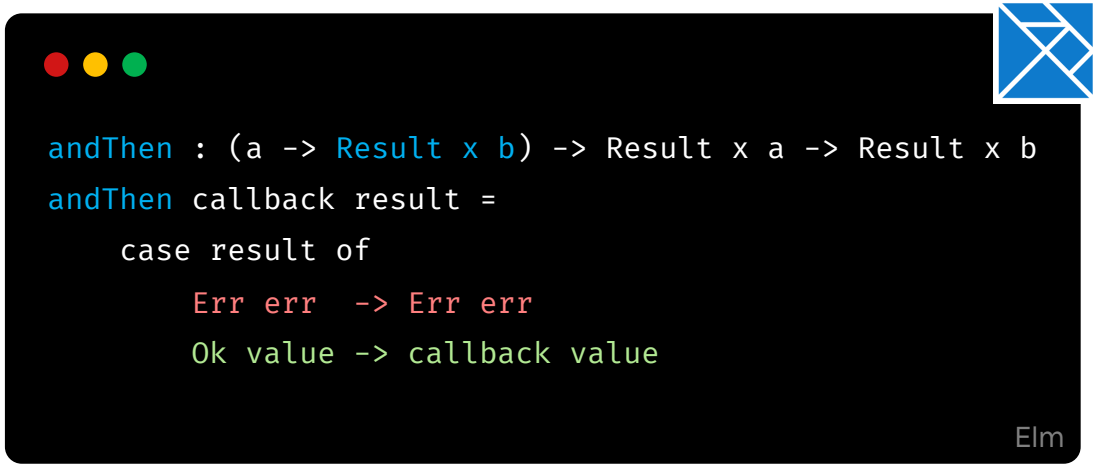
```elm
map : (a -> b) -> Result x a -> Result x b
map callback result =
    case result of
        Err err  -> Err err
        Ok value -> Ok (callback value)
```

Elm



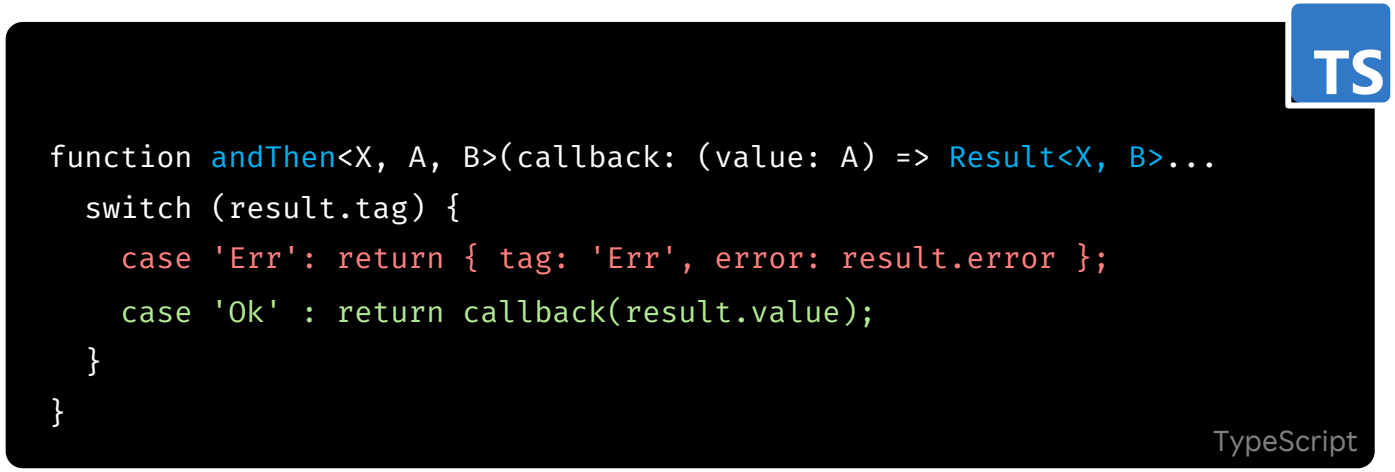
```typescript
function map<X, A, B>(callback: (value: A) => B, result: Result...
  switch (result.tag) {
    case 'Err': return { tag: 'Err', error: result.error };
    case 'Ok' : return { tag: 'Ok', value: callback(result.value) };
  }
}
```

TypeScript



```elm
andThen : (a -> Result x b) -> Result x a -> Result x b
andThen callback result =
    case result of
        Err err  -> Err err
        Ok value -> callback value
```

Elm

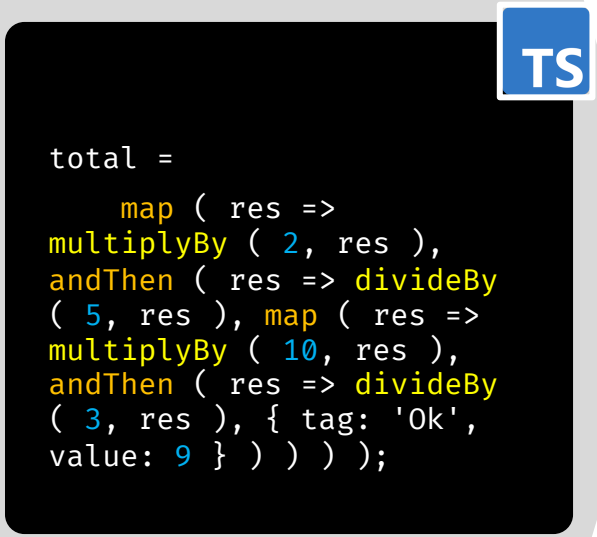


```typescript
function andThen<X, A, B>(callback: (value: A) => Result<X, B>...
  switch (result.tag) {
    case 'Err': return { tag: 'Err', error: result.error };
    case 'Ok' : return callback(result.value);
  }
}
```

TypeScript

```
9 / 3 * 10 / 5 * 2
```

```
9 / 3 * 10 / 5 * 2
```



```
total =
    map ( res =>
multiplyBy ( 2, res ),
andThen ( res => divideBy
( 5, res ), map ( res =>
multiplyBy ( 10, res ),
andThen ( res => divideBy
( 3, res ), { tag: 'Ok',
value: 9 } ) ) ) );
```

9 / 3 * 10 / 5 * 2

```ts
total =
    map ( res =>
multiplyBy ( 2, res ),
andThen ( res => divideBy
( 5, res ), map ( res =>
multiplyBy ( 10, res ),
andThen ( res => divideBy
( 3, res ), { tag: 'Ok',
value: 9 } ) ) ) );
```

```ts
total =
  map     ( res => multiplyBy (  2, res ),
  andThen ( res => divideBy   (  5, res ),
  map     ( res => multiplyBy ( 10, res ),
  andThen ( res => divideBy   (  3, res ),
              {tag: 'Ok', value: 9 })));
```

Partial application

```ts
multiplyBy = (a: number, b: number): number => a * b;
// multiplyBy( 3, 2 )  ->  6
```

```ts
const multiplyBy = (a: number) => (b: number): number => a * b;
// multiplyBy( 3 )      -> function: b => 3 * b
// multiplyBy( 3 )( 2 ) -> 6
```

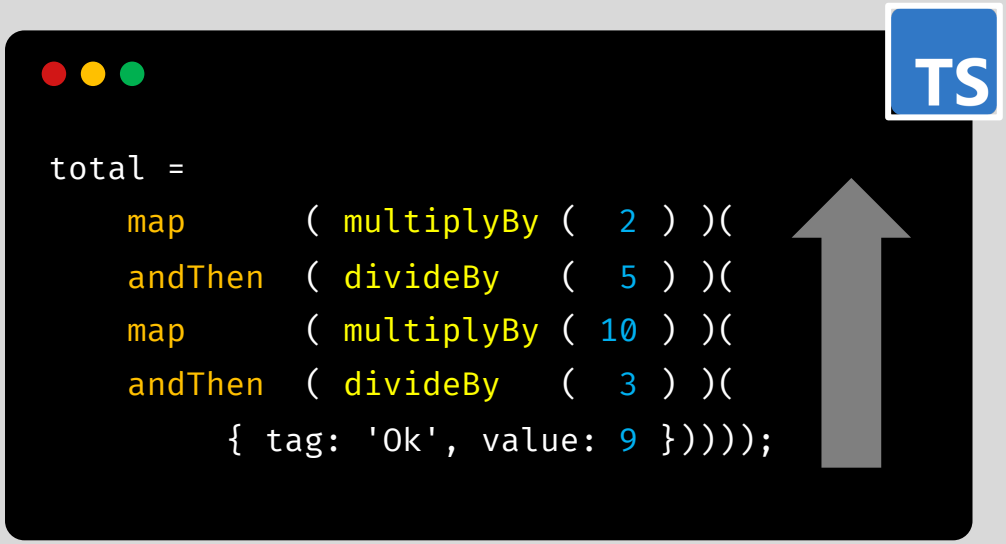
```
9 / 3 * 10 / 5 * 2
```

```ts
total =
    map ( res =>
multiplyBy ( 2, res ),
andThen ( res => divideBy
( 5, res ), map ( res =>
multiplyBy ( 10, res ),
andThen ( res => divideBy
( 3, res ), { tag: 'Ok',
value: 9 } ) ) ) );
```

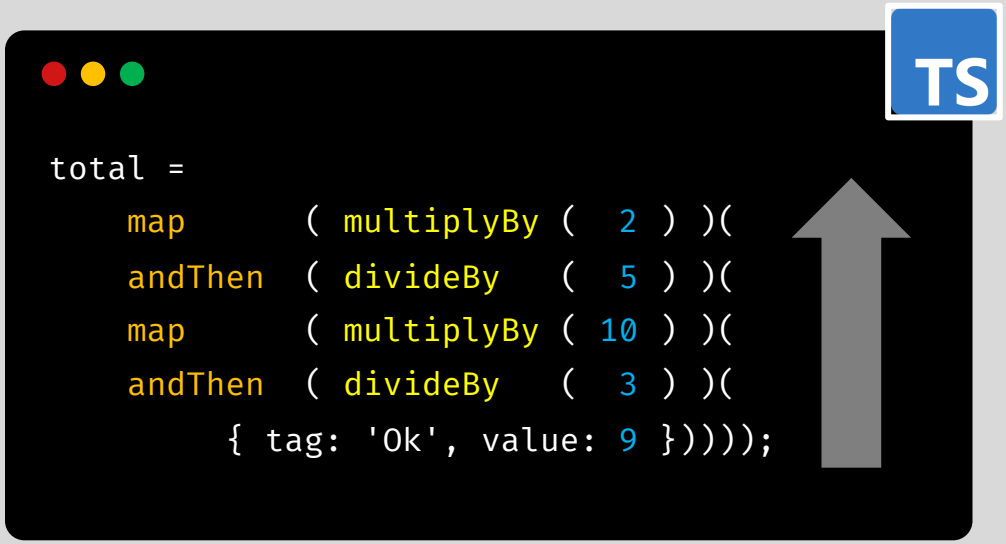
```ts
total =
  map     ( res => multiplyBy (  2, res ),
  andThen ( res => divideBy   (  5, res ),
  map     ( res => multiplyBy ( 10, res ),
  andThen ( res => divideBy   (  3, res ),
            {tag: 'Ok', value: 9 })));
```

```ts
total =
  map     ( multiplyBy (  2 ) )(
  andThen ( divideBy   (  5 ) )(
  map     ( multiplyBy ( 10 ) )(
  andThen ( divideBy   (  3 ) )(
      {tag: 'Ok', value: 9 })));
```

```
9 / 3 * 10 / 5 * 2
```


TS
total =
map (multiplyBy (2))(
andThen (divideBy (5))(
map (multiplyBy (10))(
andThen (divideBy (3))(
{ tag: 'Ok', value: 9 })))));

```
9 / 3 * 10 / 5 * 2
```



TC 39

proposal-pipeline-operator

Pipe Operator (|>) for JavaScript

Stage 2

https://github.com/tc39/proposal-pipeline-operator

(November 8, 2023)

TC 39

proposal-partial-application

Partial Application for JavaScript

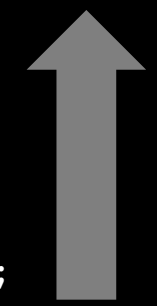
Stage 1

https://github.com/tc39/proposal-partial-application

(November 8, 2023)

The syntax is the paradigm

```
total =
    map      ( multiplyBy (  2 ) )(
    andThen  ( divideBy   (  5 ) )(
    map      ( multiplyBy ( 10 ) )(
    andThen  ( divideBy   (  3 ) )(
         { tag: 'Ok', value: 9 })))));
```



```
total =
    Ok 9
        |> andThen ( divideBy    3 )
        |> map     ( multiplyBy 10 )
        |> andThen ( divideBy    5 )
        |> map     ( multiplyBy  2 )
```

TC39

proposal-pipeline-operator

Pipe Operator (|>) for JavaScript

Stage 2

https://github.com/tc39/proposal-pipeline-operator

(November 8, 2023)

TC39

proposal-partial-application

Partial Application for JavaScript

Stage 1

https://github.com/tc39/proposal-partial-application

(November 8, 2023)

Third-party Libraries

For Elm

Built in the language

For TypeScript

- `NeverThrow`
- `ts-toolbelt`
- `ts-results`
- `purify-ts`
- `fp-ts`

For Kotlin

- `Arrow`
- `kotlin-result`
- `result4k`

The Result Type

A type holding an **ok value** or an **error value**

A way to **manage errors using the type system** without throwing exceptions, possibly **improving both the DX and the UX**.

code available at

github.com
/lucamug/hope

Photo by Johannes Hofmann on Unsplash
https://unsplash.com/photos/approaching-red-train-across-mountains-PM5a_R83-YQ
(November 8, 2023)


Rakuten