

Designing Alternative Workflows Instead of Defending from Errors



Zoran Horvat

PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com



Status Code as Return Value

Status code

Discrete value returned from an operation to indicate completion status.

Caller must check status

Invites bugs

Increases caller's complexity

Caller has more things to do



Exception Execution Flow

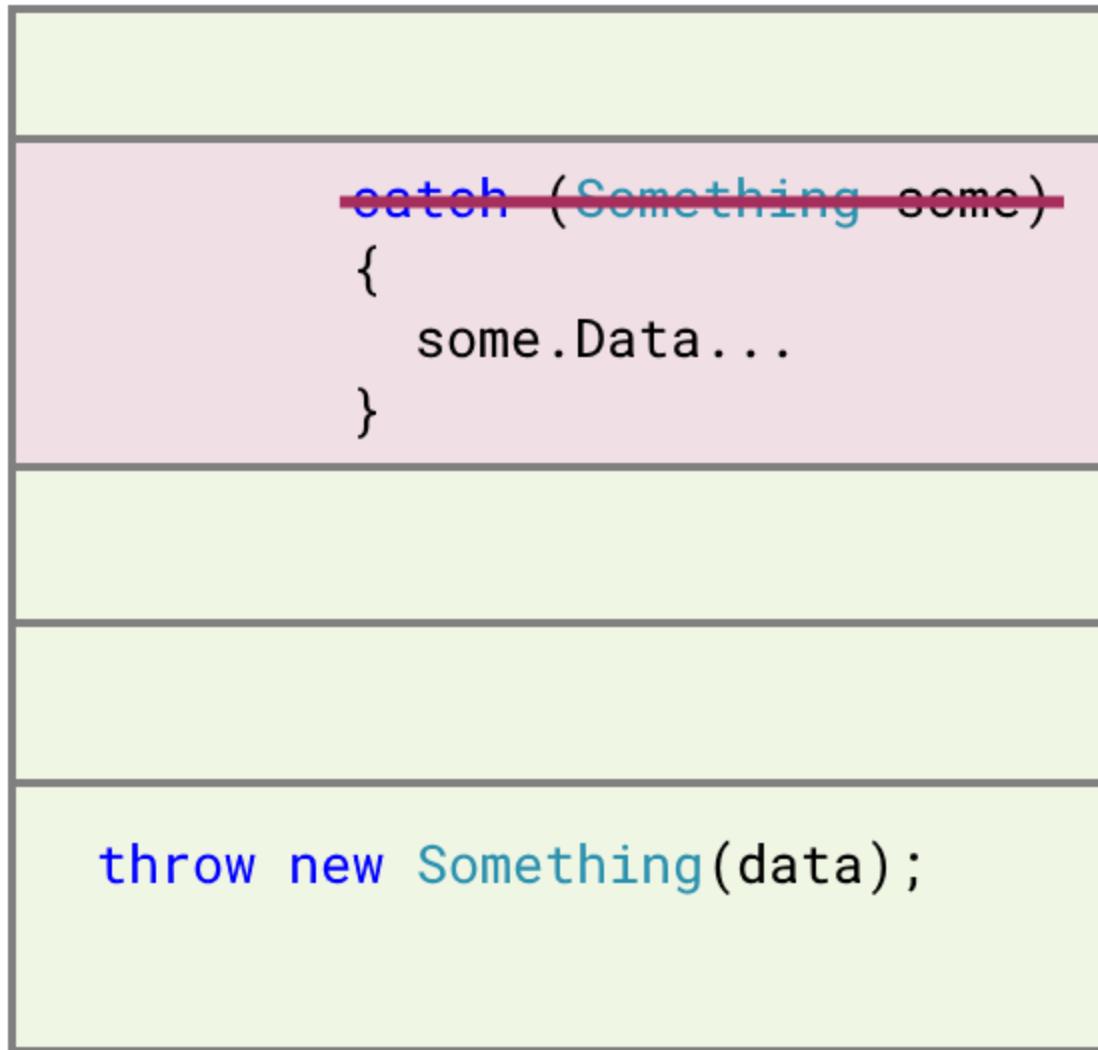
```
catch (Something some)
{
    some.Data...
}
```

Parallel
execution
flow

```
throw new Something(data);
```

Execution
flow

Exception Execution Flow

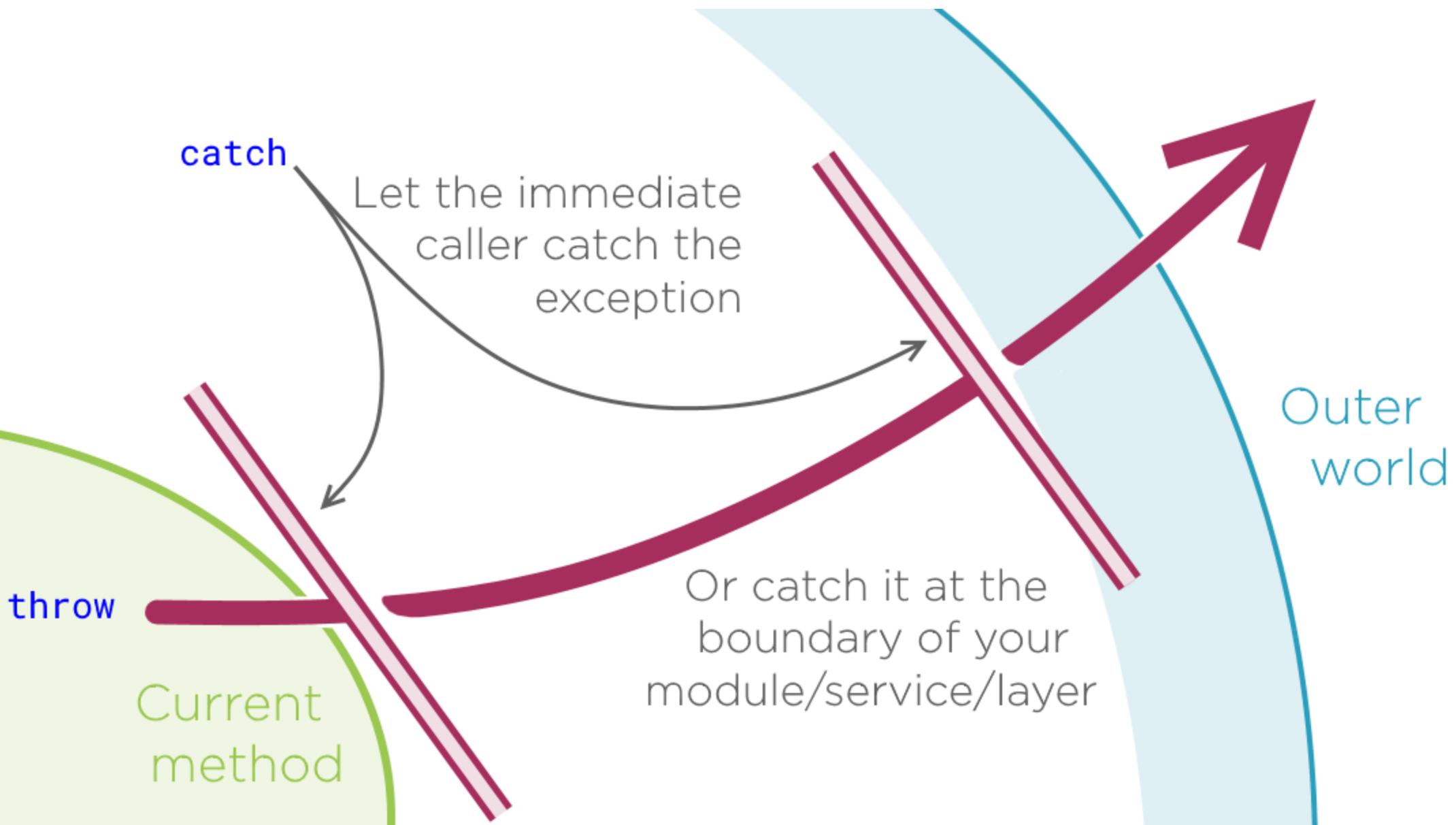


Call stack

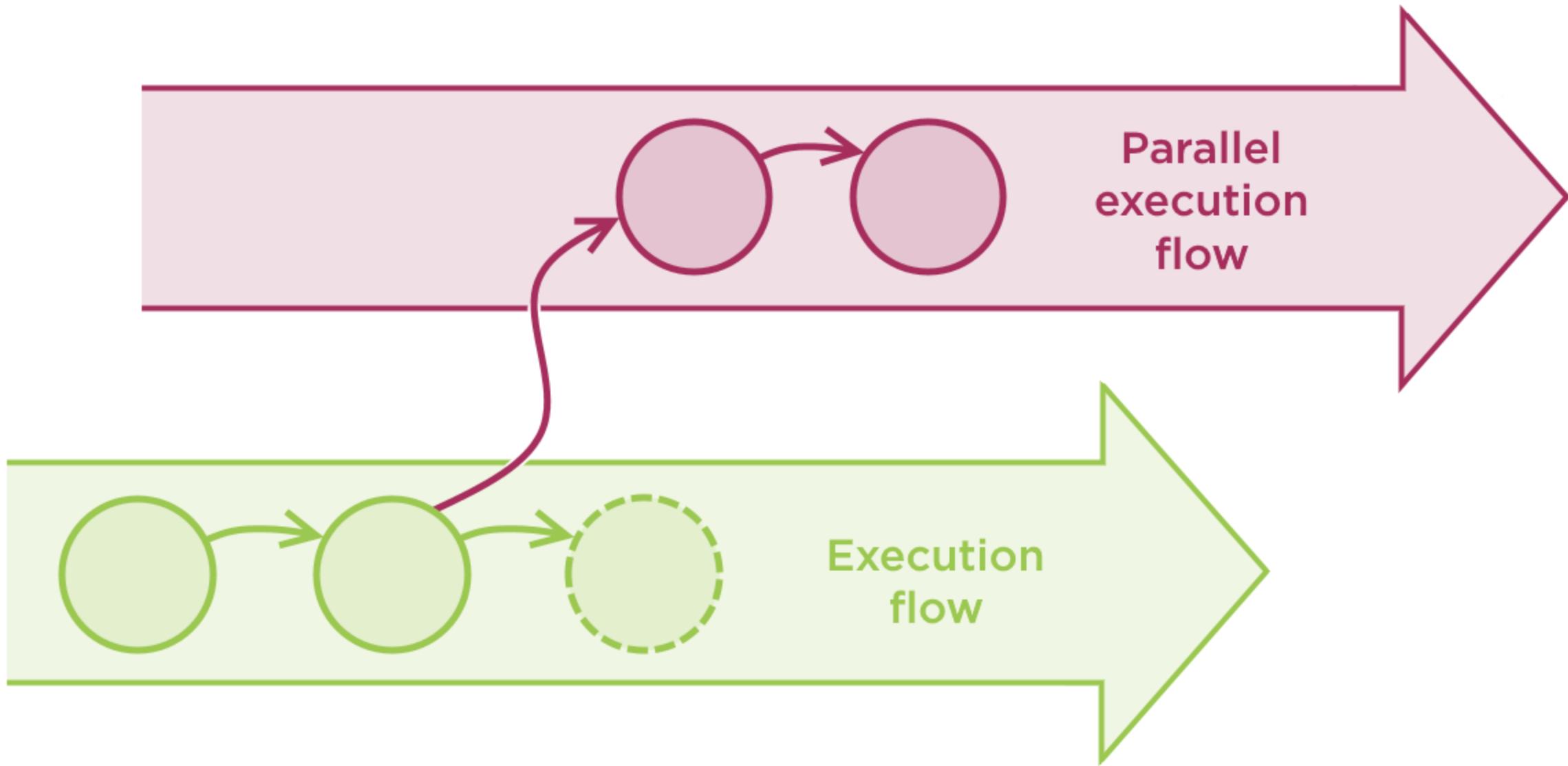
Intended recipient might forget
to catch the exception

An idea:
If there is anything to tell,
use the return value to tell it

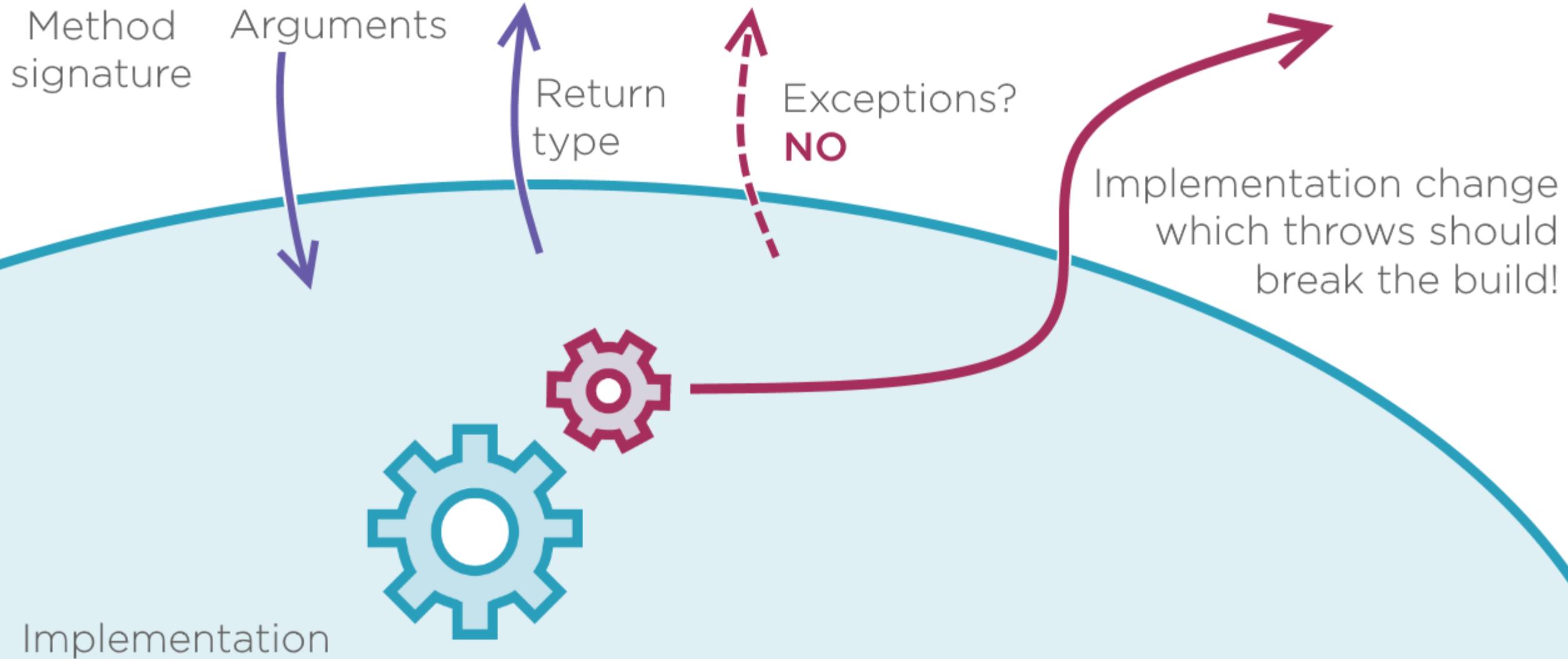
Exception Execution Flow



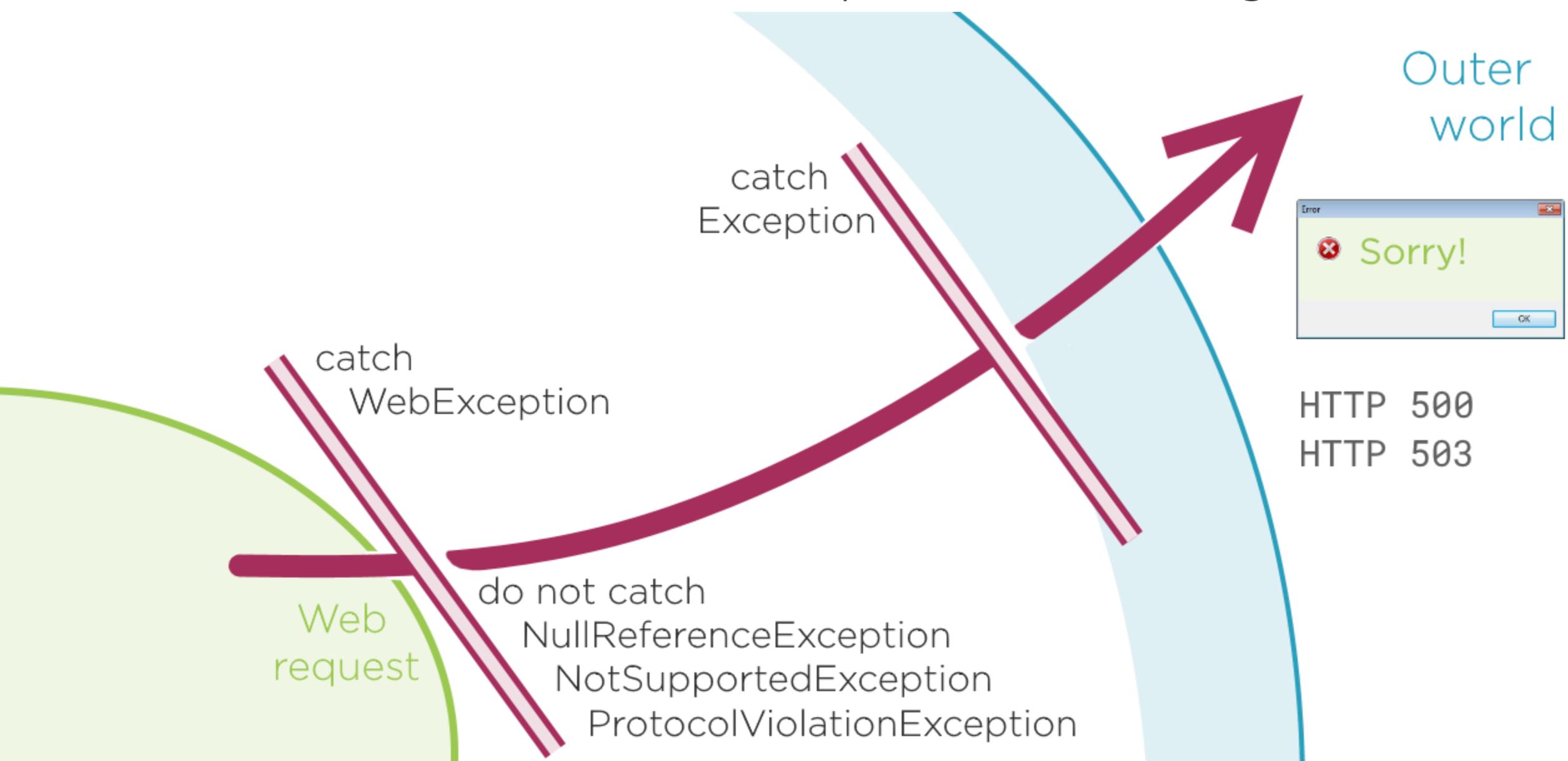
Exception Execution Flow



Exception Execution Flow



The Rules of Exception Handling

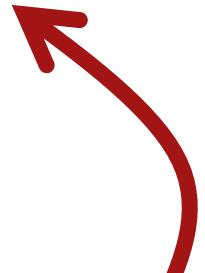


The Rules of Exception Handling

```
void Process(string uri)
{
    try
    {
        // fetch uri
    }
    catch (WebException)
    {
    }
}
```

```
try
{
    Process("http://bad.uri");
}
catch (WebException ex)
{
    ...
}
```

My guess is that it's
Process the one who
threw the exception



The Rules of Exception Handling

```
void Process(string uri)
{
    try
    {
        // fetch uri
    }
    catch (WebException)
    {
    }
    catch (ProtocolViolationException)
    {
    }
}
```

```
try
{
    Process("http://bad.uri");
}
catch (WebException ex)
{
    ...
}
```

Expected exception

Unexpected exception



Summary



Indicating errors back to the caller

Exceptions are flawed

- Parallel execution flow may introduce uncertainty and instability

Indicate result through return type only

- Unify both flows through the result



Summary



Polymorphic positive/negative flows

- One type hierarchy contains both kinds of subclasses
- Polymorphism is a powerful tool in object-oriented languages
- Fitting positive and negative concepts into a single type may look awkward



Summary



Operation result type

- Separates positive and negative component inside one object
- Either contains positive component, or a negative component, but not both
- Consumer has to branch around ***what*** the operation result holds



Summary



Either type

- Comes from functional languages
- Contains two components, like operation result does
- Encapsulates knowledge which component is the one present
- Consumer must cover both operations
- Consumer never knows which strategy was actually executed



Summary



Either<TLeft, TRight>
clean example of defensive design

What are the principles of
defensive design then?

- Have objects
- Do not defend
- Do not fork on errors

Final words

Defensive design yields shorter code

Add practices one at a time

Start feeling better...

... and stop defending

