August 25th 2015

The Zone, a general model for asynchronous contexts

Valérian Pittet

Outline

- The Zone
- Implementation
- Examples
  - Contextual error catching
  - Long stack traces
  - Asynchronous sequence diagram
  - Vert.x integration

The Zone

“The Zone is the asynchronous extension of a scope providing code hooks”

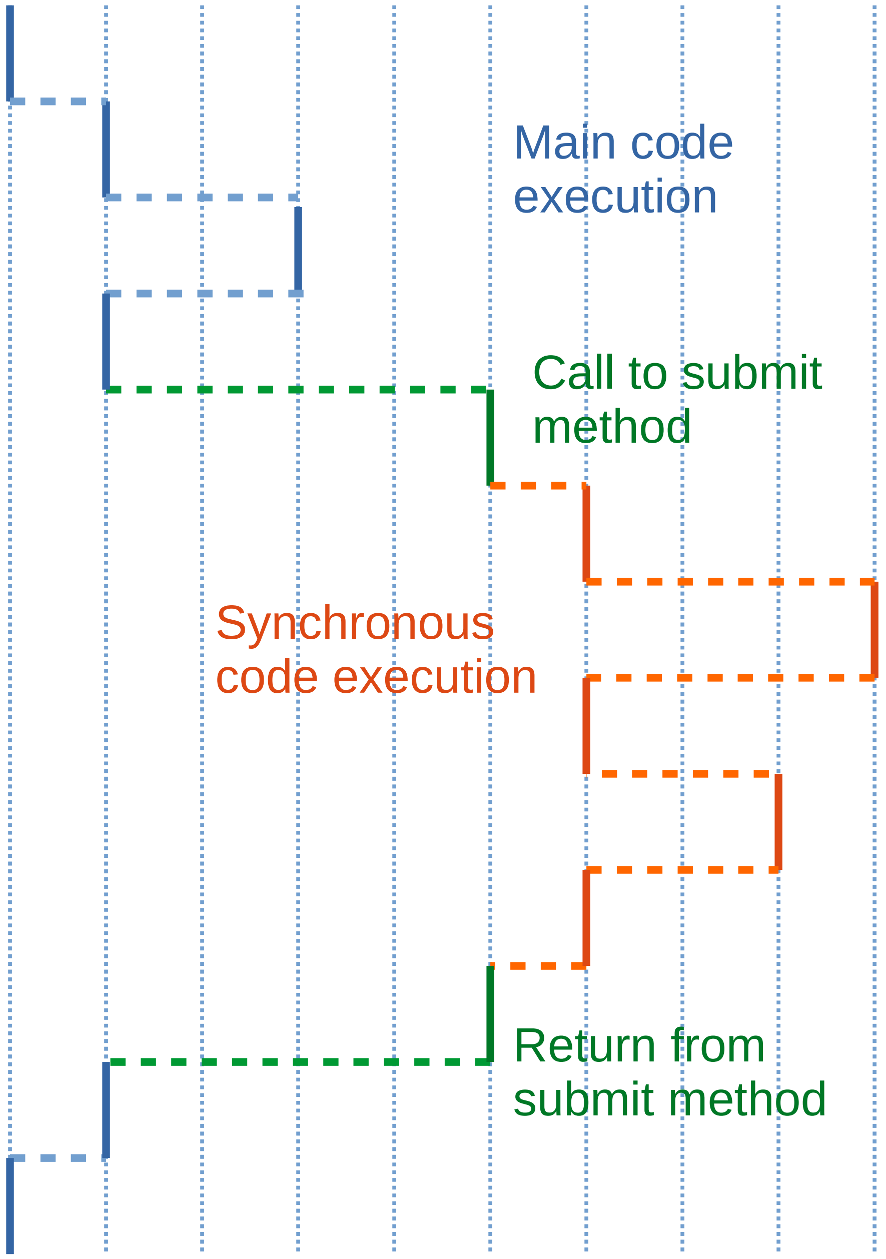
- Asynchronous definition
- Asynchronous scope
- Code hooks

Asynchronous

- Asynchronous: code that gets submitted and does not execute **in sequence** of its submission.
- Suppose main code runs the method:

```
submit(Task code);
```

and compare executions when “`code`” is run synchronously and asynchronously

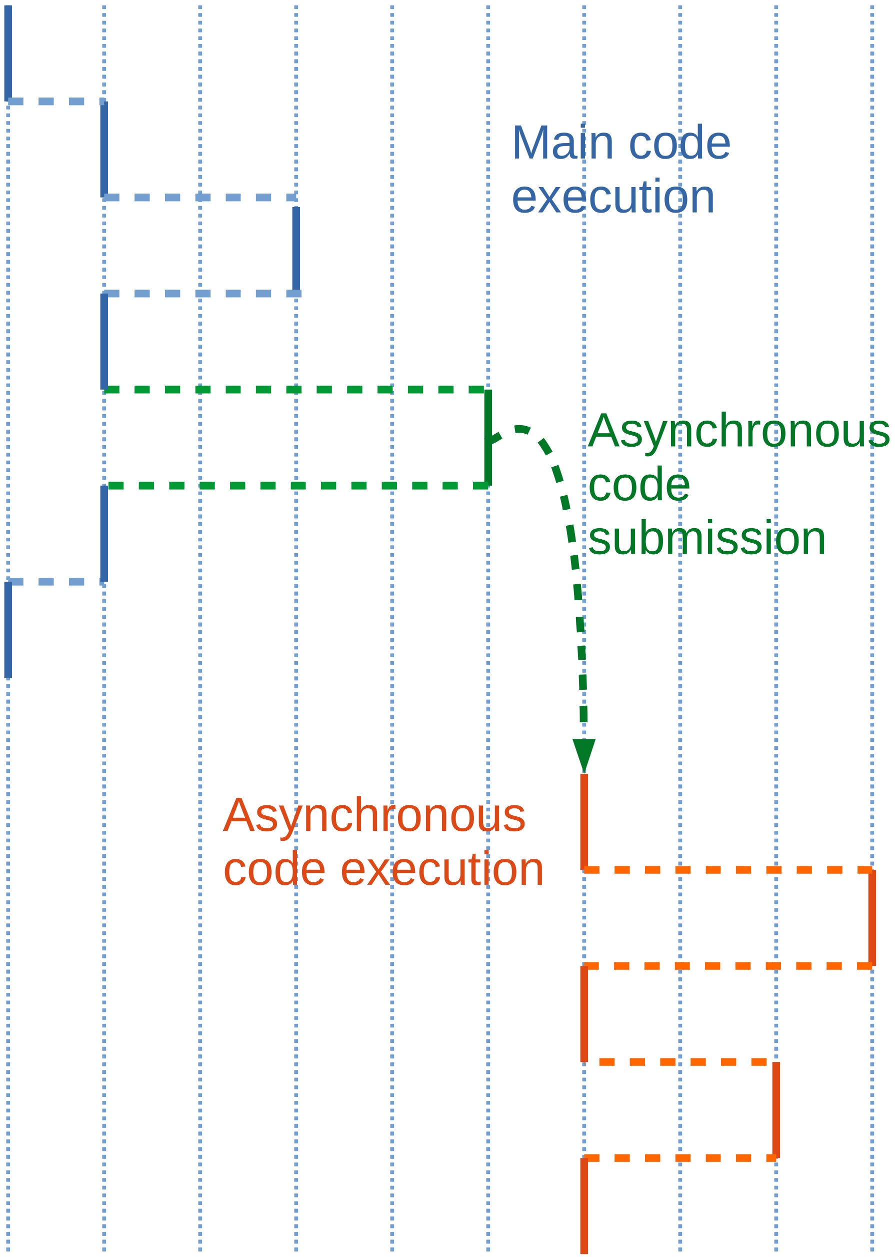

Main code execution

Call to submit method

Synchronous code execution

Return from submit method

- Call to submit method **does** appear in the stack of executing code
- Submitted code executes **synchronously**
- Submit method returns **after** submitted code's completion



- Call to submit method **does not** appear in the stack of executing code
- Submitted code executes **asynchronously**
- Submit method returns **independently** of submitted code's completion

Asynchronous scope

- Conventional scope is bound to execution sequence
- Conventional scope is not preserved on in asynchronous submission

- Code **submitted** into a Zone executes **inside** this Zone.

Zone values

As a scope, the Zone can define key-value bindings.

- Access bindings from enclosing Zones.
- Shadow bindings from enclosing Zones with new bindings.
- Immutable bindings

Code hooks

“Provide few, powerful and general properties instead of many specific ones.”

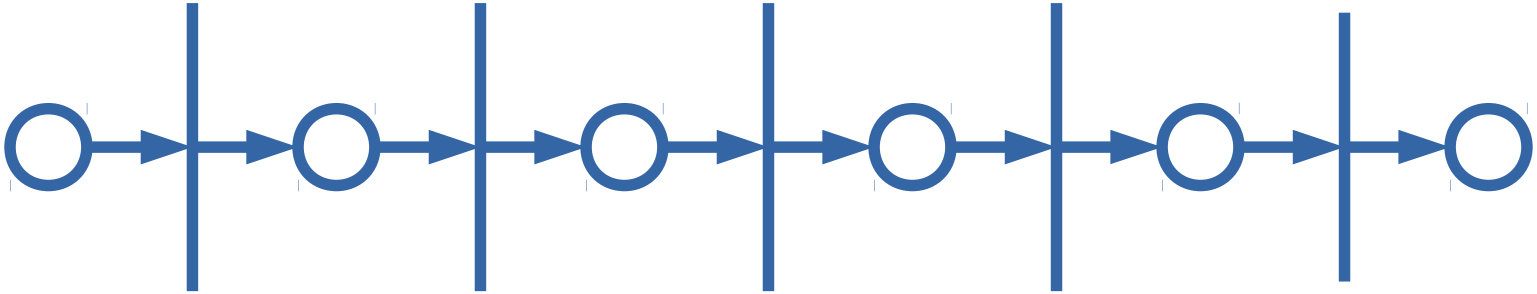
1. Entering and exiting the Zone: crossing hooks.
2. Submitted code manipulation: around hooks.

Crossing hooks

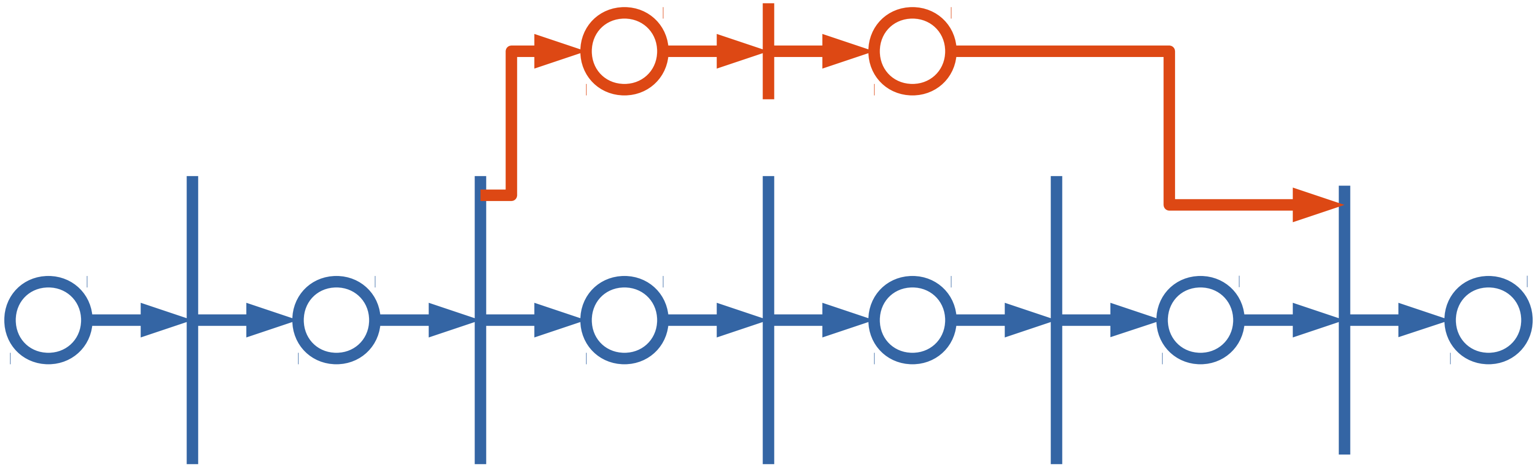
- Enter-exit, a better solution than begin-end
- Well defined in any situation

Let's see an illustration, using the Petri net representation.

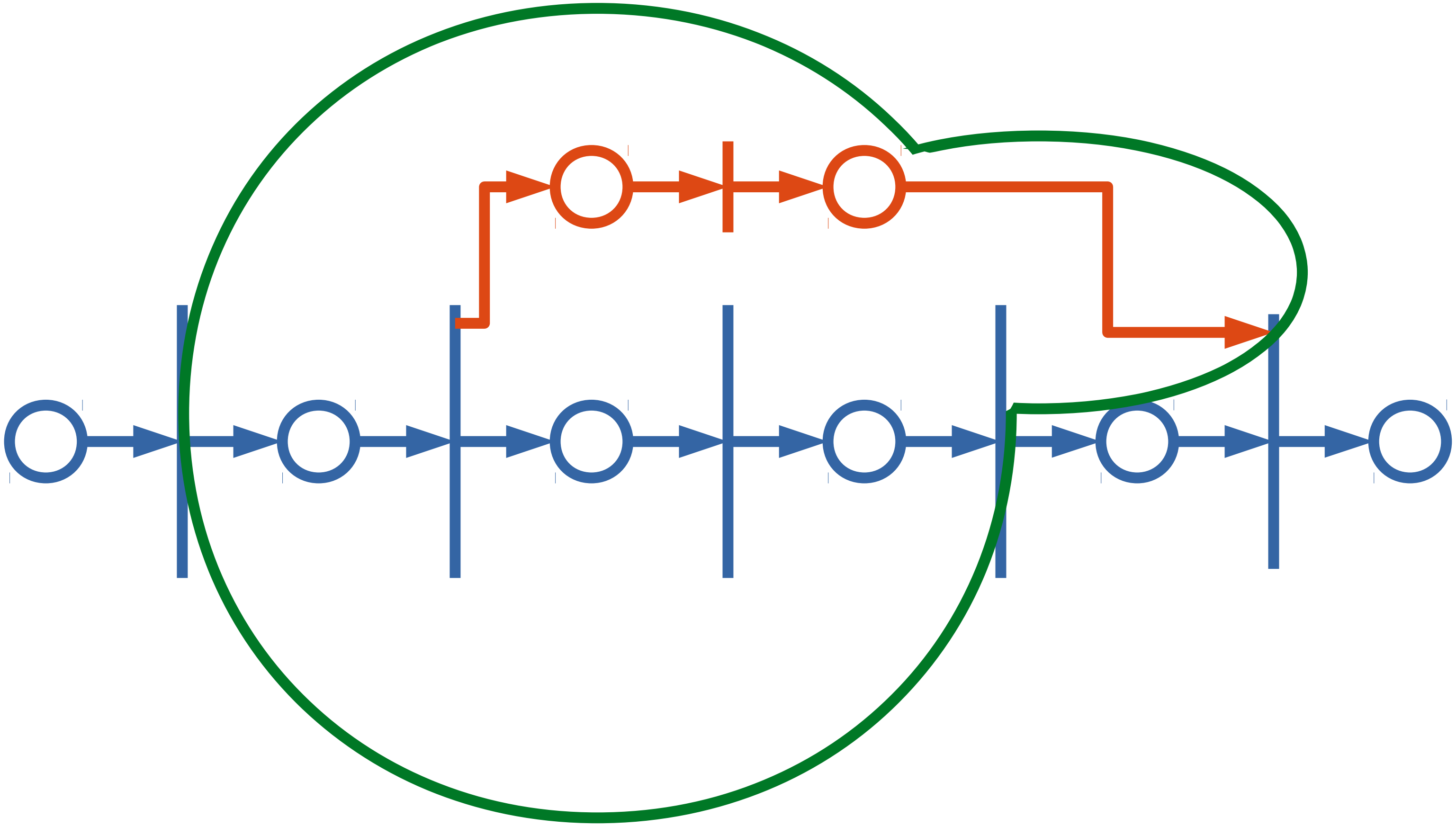
- Main task



- Main task
- Asynchronous task



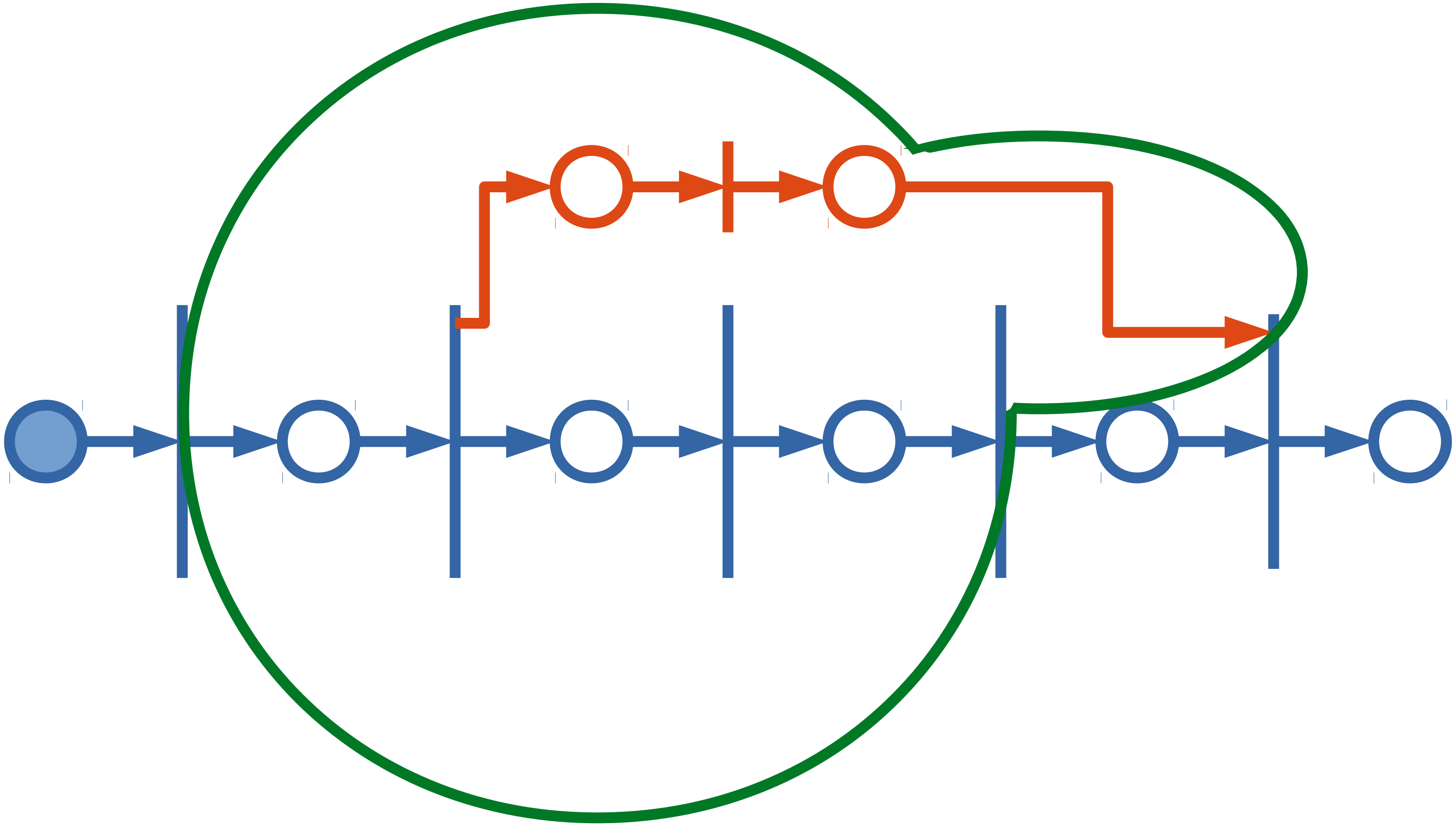
- Main task
- Asynchronous task
- The Zone



- Main task
- Asynchronous task
- The Zone

Step 1

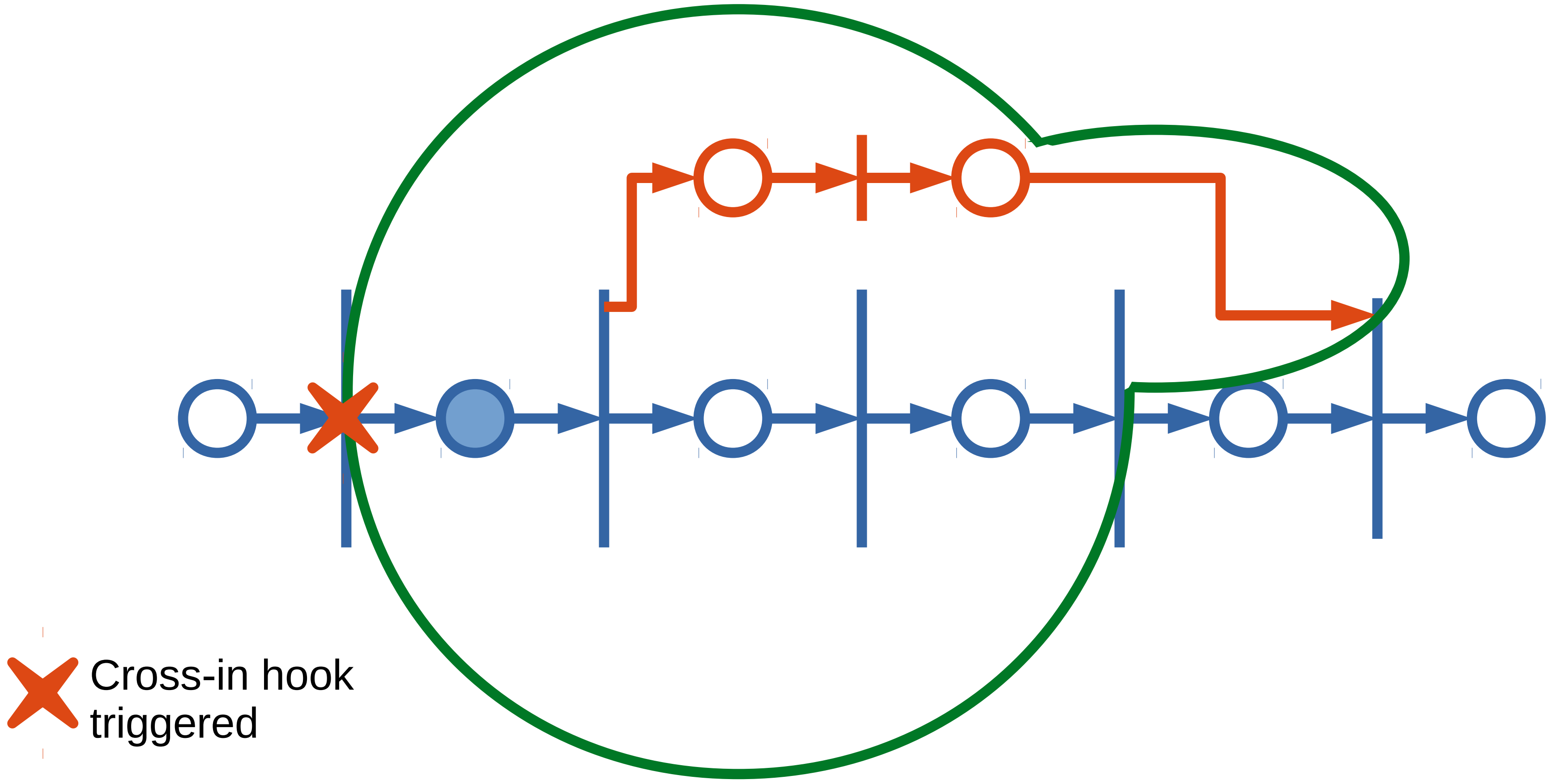
Start main task



- Main task
- Asynchronous task
- The Zone

Step 2

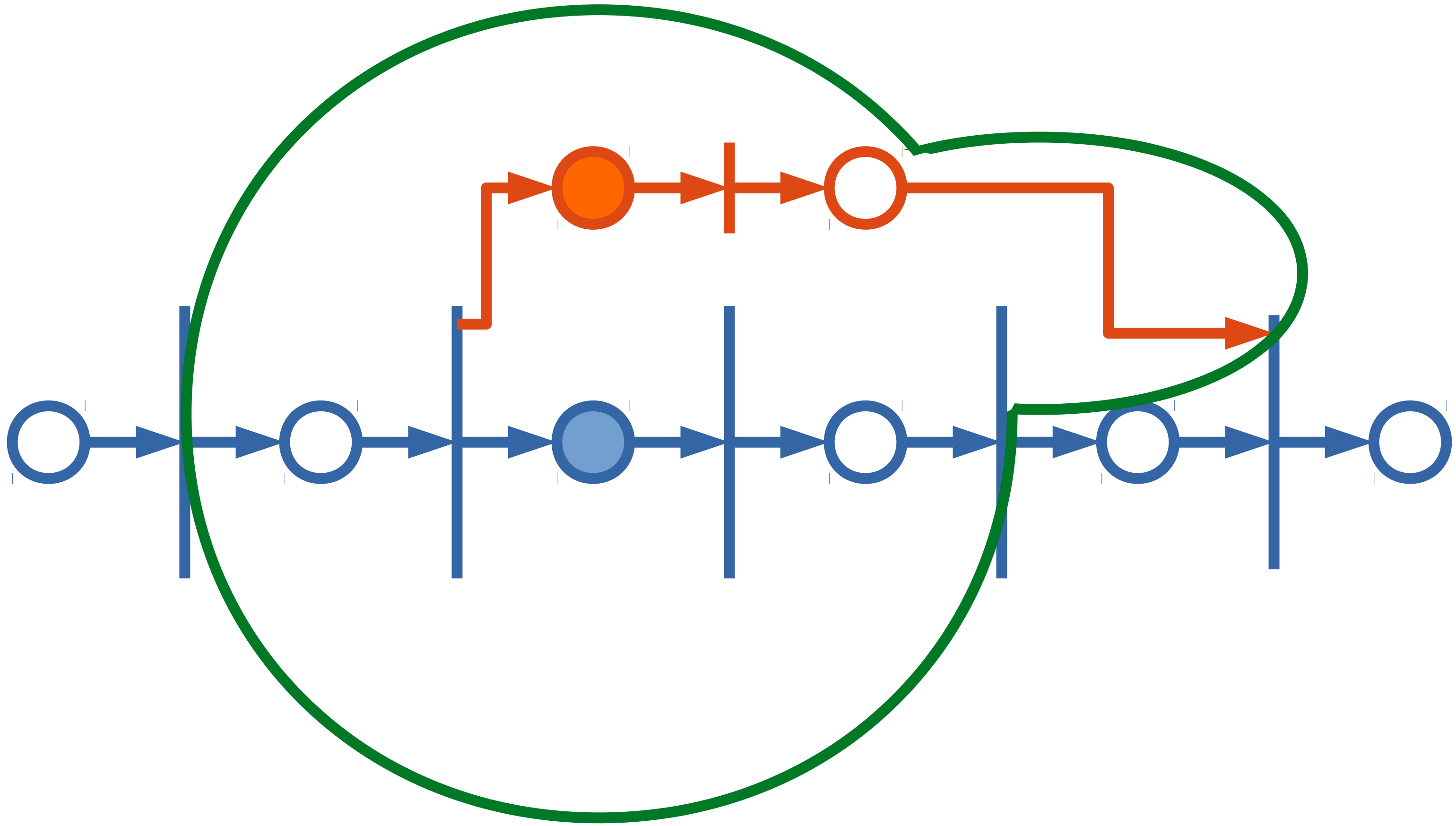
Token crosses inside the Zone



- Main task
- Asynchronous task
- The Zone

Step 3

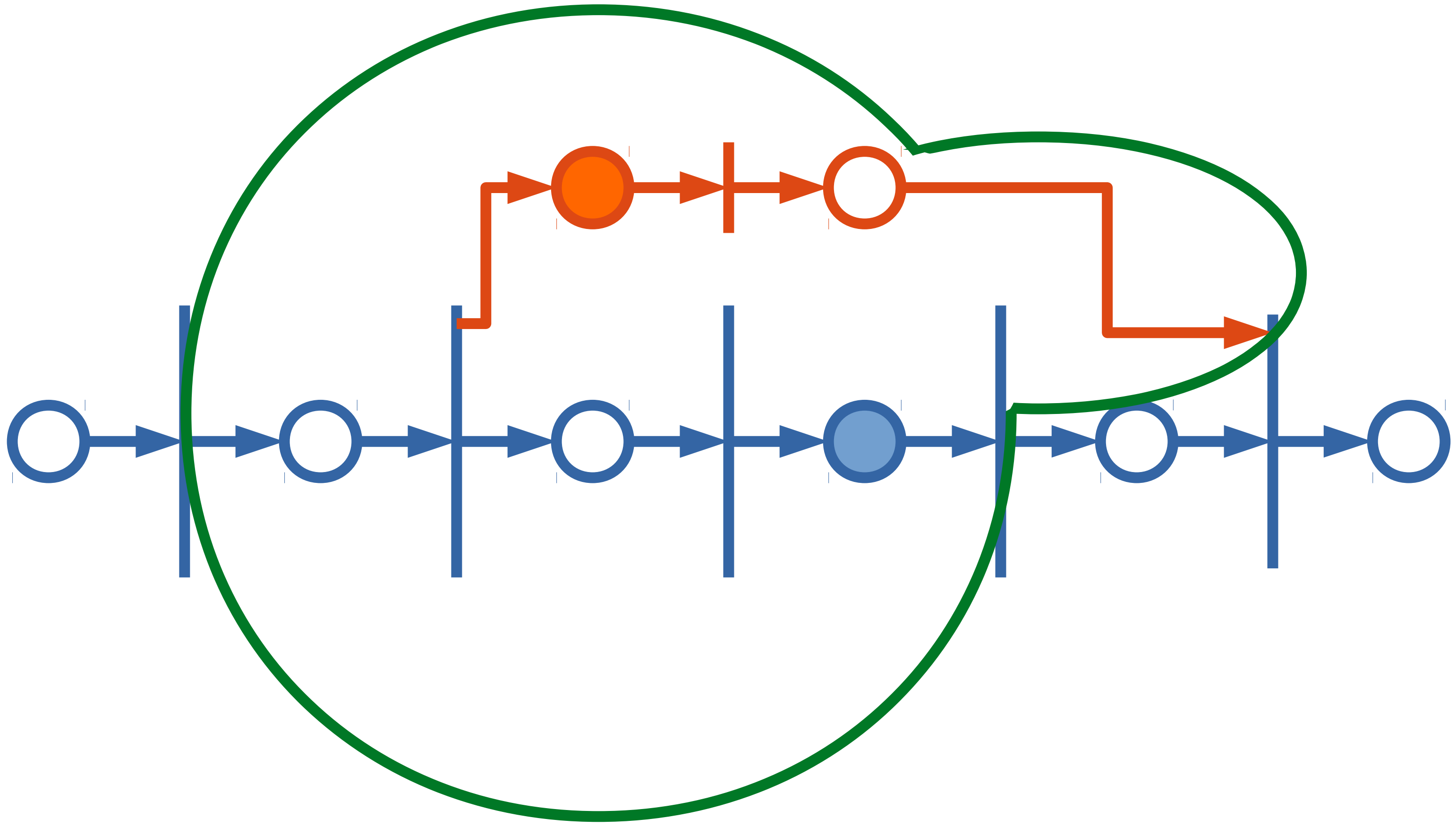
Asynchronous task submission



- Main task
- Asynchronous task
- The Zone

Step 4

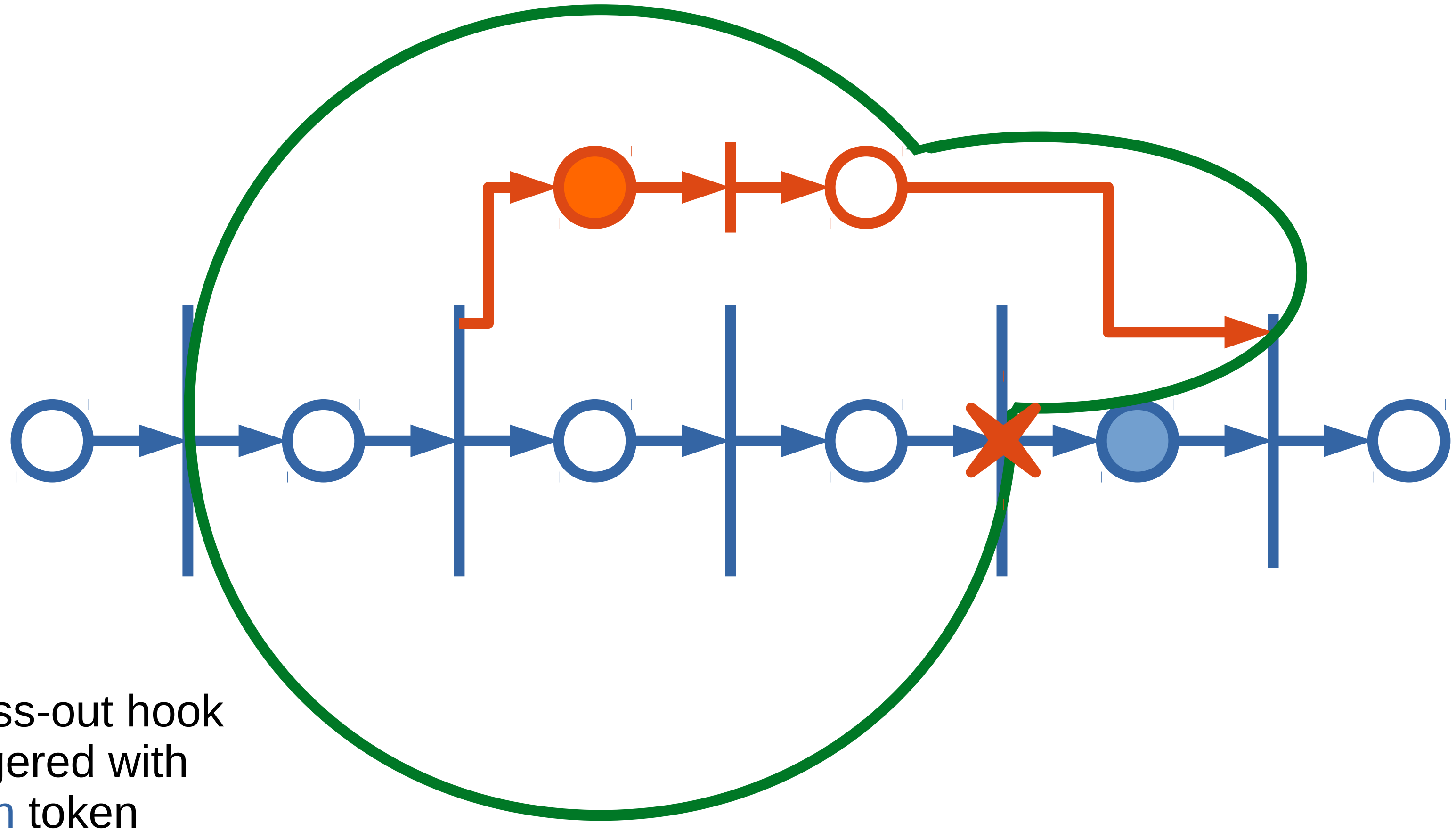
Main task executes normally



- Main task
- Asynchronous task
- The Zone

Step 5

Token crosses
outside the Zone

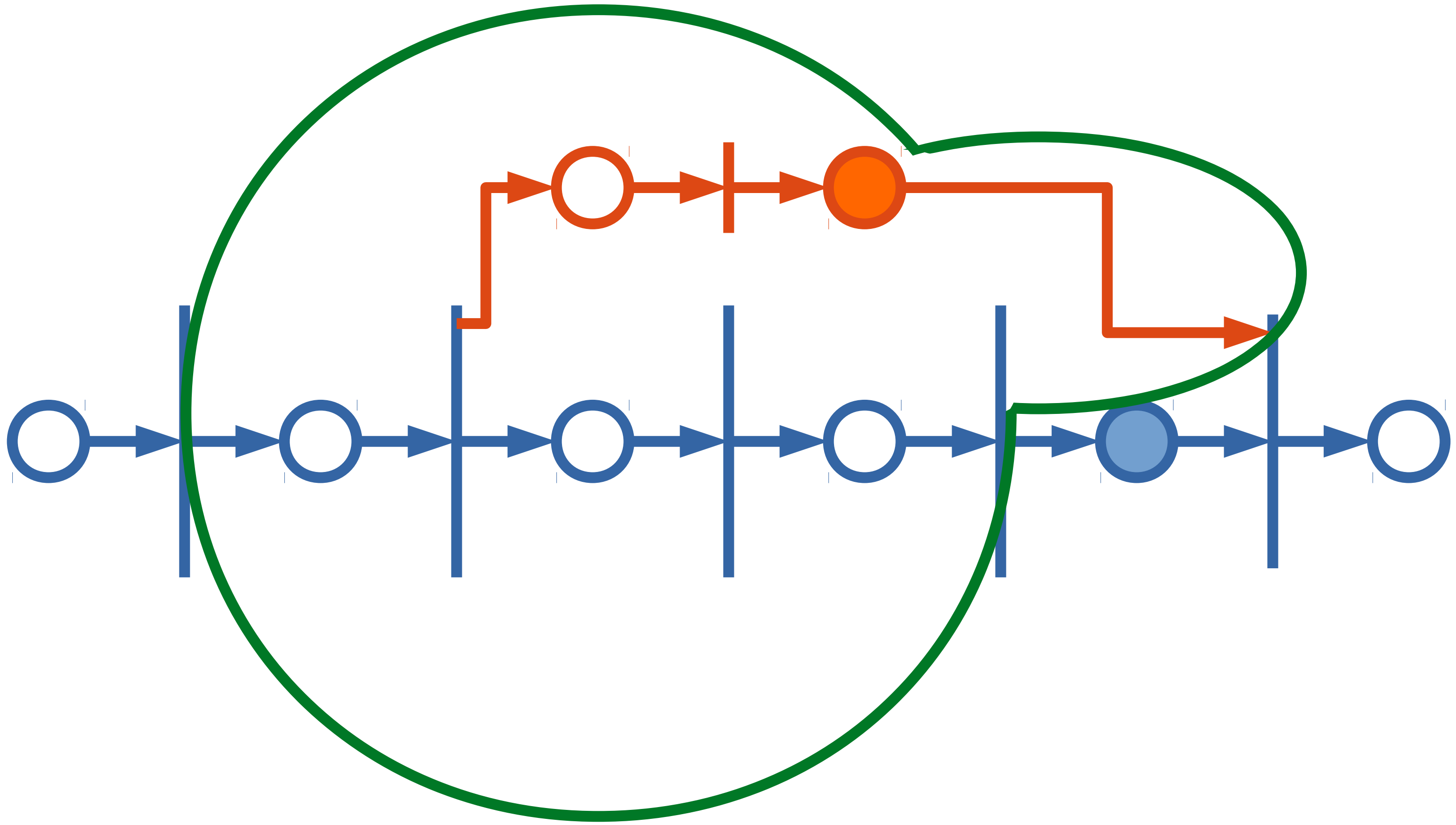


✗ Cross-out hook
triggered with
main token

- Main task
- Asynchronous task
- The Zone

Step 4'

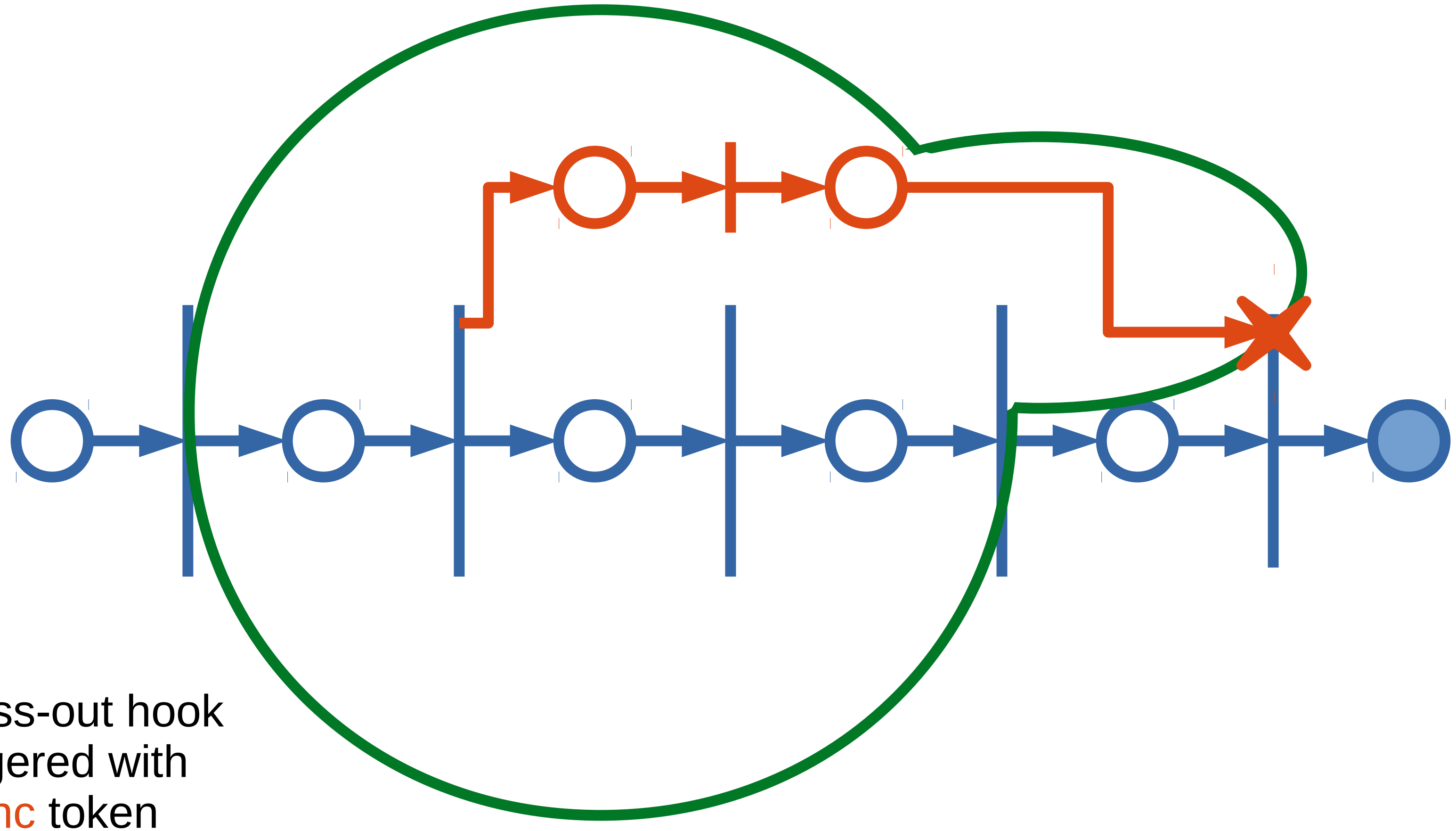
Asynchronous task execution



- Main task
- Asynchronous task
- The Zone

Step 6

Synchronization



❌ Cross-out hook triggered with async token

Around hooks

Manipulate code executed inside a Zone through a function *f* : `task → task`

- Synchronously

```
myCode = () → { /* code */ };
myZone.run(myCode); // run f(myCode)
```

Around hooks

Manipulate code executed inside a Zone through a function *g* : `task → task`

- Asynchronously

```
myCode = () → { /* code */ };
myZone.bind(myCode);// bind g(myCode)
```

Around hooks: Inheritance

- Around hooks from parent Zones are inherited and applied by any child Zone.

- Around hooks are not repeated.
  If the same hook appears twice in the parent chain, only first applies.

Implementation

“Some code to ease understanding of incoming demos”

1. Core classes

2. Zone interface

Core class: `Zone`

- Implements binding mechanisms and hooks application

- Knows its enclosing (parent) Zone

- Defines code hooks and key-value bindings

Core class: `Token`

- Representation of the Petri net token.
- May contain a result or an error.

```
interface Token<T>

class VoidToken extends Token<Void>

class ResultToken<T> extends Token<T>

class ErrorToken extends Token<Void>
```

Core class: `ZonedToken`

- Tuple containing a `Token` and the `Zone` it comes from.
- Used to determine which Zones are crossed by tokens

```
class ZonedToken<T> {
    Token<T> token;
    Zone zone;
}
```

Core class: `Task`

`Task` represents a block of code with optionally inputs and output.

```
interface Task<T, U, V> {
  V apply(T t, U u);
}
```

Core class: `ZonedTask`

- Represents a `Task` bound to a `Zone`
- Inputs and output are wrapped in `ZonedTokens`

```
Task<Int, Double, String>
```

becomes

```
ZonedTask<ZonedToken<Int>,
          ZonedToken<Double>,
          ZonedToken<String>>
```

Zone Interface

The Zone is an abstract class

- **Implements** binding mechanisms
- **Defines** hook and lookup function

One extends the abstract Zone to implement his custom behavior.

Zone interface: `getValue`

- Used to lookup for a key
- Returns optional result to distinguish `null` value from no match for a key.

```
<T> Option<T> getValue(Key<T> key);
```

Zone interface: `crossIn`

- Defines operation on cross-in hook.
- Called whenever a token crosses inside the Zone.
- Can replace crossing token.

```
<T> Token<T> crossIn(Token<T> tkn);
```

Zone interface: `crossOut`

- Defines operation on cross-out hook.
- Called whenever a token crosses outside the Zone.
- Can replace crossing token.

```
<T> Token<T> crossOut(Token<T> tkn);
```

Zone interface: `getHook`

- Does not implements operation on hook.
- Returns the function to apply on hooked `Task`. `AroundHook`: function from `Task` to `Task`
- Allows to implement union-inheritance

```
AroundHook getSyncHook();
AroundHook getAsyncHook();
```

Zone interface: `getHook`

```
AroundHook getSyncHook() {
  return (task) → {
    print(“hook being applied”);
    return (a, b) → {
      print(“hooked task starting...”);
      Result r = task.apply(a, b);
      print(“hooked task finished.”);
      return r;
    }
  }
}
```

Hook application log

Hooked task

Examples

- Contextual error handling
- Long stack traces
- Asynchronous sequence diagram
- Vert.x instrumentiation

Contextual error handling

- Use the Zone to implement asynchronous error handling.
- Ambiguous specification:
  - Instantly apply error handling of all parent Zone
  - Only catch error when it leaves the Zone

- Second one closer to the try-catch block

Scheme to illustrate ambiguousity

Contextual error handling

- Check in crossing out hook if the token contains an error.

- If yes, handle it.

Demo

Long stack traces

- Complement to contextual error handling.
- Stores call stack of asynchronous code submission.
- Adds this stack to the asynchronous errors.
- Easier debug if the error cause stands before asynchronous submission.

Long stack traces

- On each asynchronous submission, store current call stack in a Zone value.
- Wrap asynchronous code in try-catch block that
  - Catches all errors.
  - Update their stack traces.
  - Re-throw all errors.

Demo

Asynchronous Sequence Diagram

- Trace of asynchronous execution.
- Acyclic dependency graph.
- Each task is divided in subtasks where:
  - A subtask execution only has first **outgoing** dependencies.
  - Then only **incoming** dependencies.

Graph generation illustration

Asynchronous Sequence Diagram

- Run each submitted task in a new Zone. (Using an around hook)
- Give each task's Zone a unique Id.
- On crossing out hook:
  - Check source Zone's Id.
  - Check destination Zone's Id.
  - Store dependency between the two Zones in some data structure.

Demo

Vert.x integration

Vert.x is an asynchronous execution framework featuring:

- Callback-style asynchrony
- Communication between entities across a unique event bus.

Vert.x integration

- Identify all code submission method (asynchronous)
- Bind on-the-fly submitted code to the current Zone
- Wrap all messages on the event bus in a `ZonedToken`
- Use the AspectJ AOP programming framework to automatically modify the Vert.x library to bind asynchronous code to the Zone.

Demo vert.x with async sequence diagram

Thanks

- ELCA company
- Philipp Oser – ELCA supervisor
- Benoît Briot – ELCA supervisor
- Rachid Guerraoui – EPFL supervisor

And thank You for your attention !

Questions ?