Exceptions

Pablo Tesone - Guille Polito - Fundación Uqbar

What we expect?

The Good Path

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

My Language Stack

drawing convention:
stacks grow down

What we expect?

The Good Path

```
program xxx {
1 x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

My Language Stack

1.program xxx

drawing convention:
stacks grow down

What we expect?

The Good Path

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

1

My Language Stack

1.program xxx

drawing convention:
stacks grow down

What we expect?

The Good Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
2   self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

My Language Stack

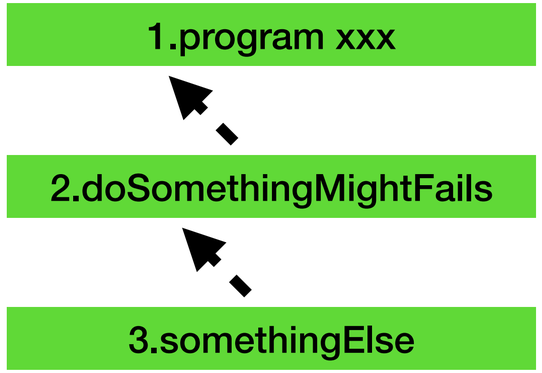
drawing convention:
stacks grow down

What we expect?

The Good Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
2   self.somethingElse()
  }

  method somethingElse(){
3   if(...)
      throw new MyException()
  }
}
```

My Language Stack



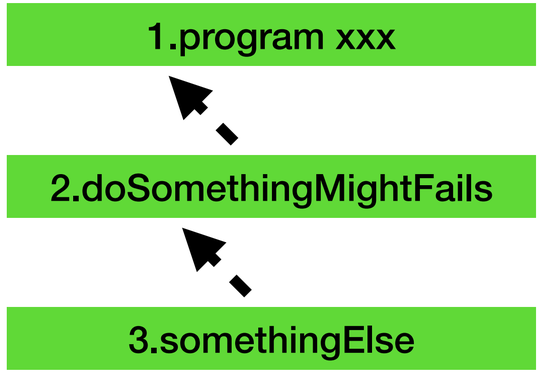
drawing convention:
stacks grow down

What we expect?

The Good Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
2   self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
3 }
}
```

My Language Stack



drawing convention:
stacks grow down

What we expect?

The Good Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
2 }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

My Language Stack

drawing convention:
stacks grow down

What we expect?

The Good Path

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

1

My Language Stack

1.program xxx

drawing convention:
stacks grow down

What we expect?

The Exception Path

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

My Language Stack

drawing convention:
stacks grow down

What we expect?

The Exception Path

```
program xxx {
1   x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

My Language Stack

1.program xxx

drawing convention:
stacks grow down

What we expect?

The Exception Path

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

1

My Language Stack

1.program xxx

drawing convention:
stacks grow down

What we expect?

The Exception Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
2   self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

My Language Stack

drawing convention:
stacks grow down

What we expect?

The Exception Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
2   self.somethingElse()
  }

  method somethingElse(){
3   if(...)
      throw new MyException()
  }
}
```

drawing convention:
stacks grow down

What we expect?

The Exception Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
2   self.somethingElse()
  }

  method somethingElse(){
    if(...)
3     throw new MyException()
  }
}
```

drawing convention:
stacks grow down

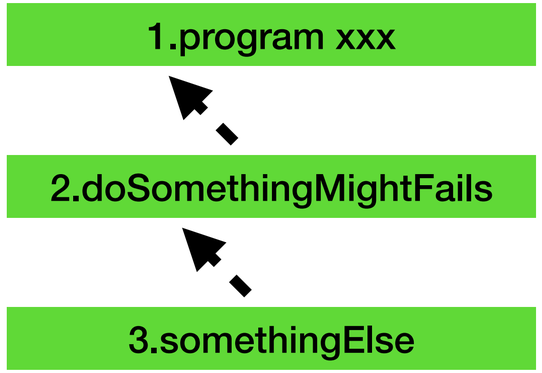
What we expect?

The Exception Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
2   self.somethingElse()
  }

  method somethingElse(){
    if(...)
3     throw new MyException()
  }
}
```

We need to come back here

My Language Stack



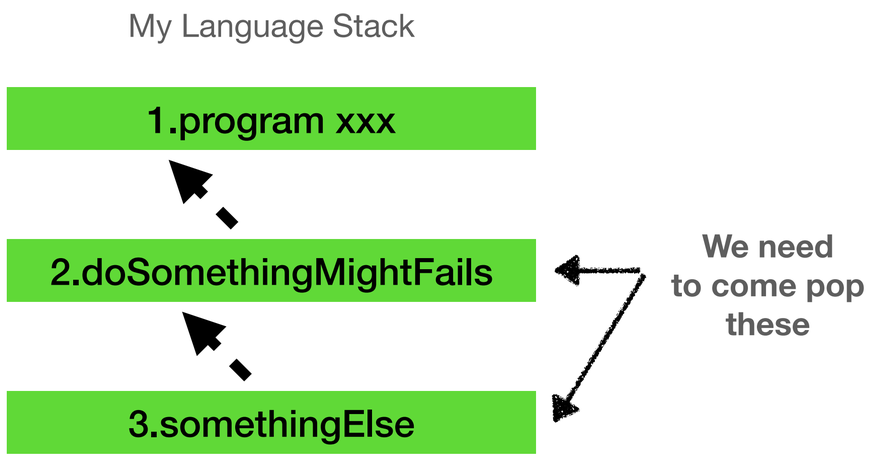
drawing convention: stacks grow down

What we expect?

The Exception Path

```
program xxx {
  x = new MyClass()

  try{
1   x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
2   self.somethingElse()
  }

  method somethingElse(){
    if(...)
3     throw new MyException()
  }
}
```

We need to come back here



drawing convention: stacks grow down

What we expect?

The Exception Path

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

1

My Language Stack

1.program xxx

drawing convention:
stacks grow down

Things to Resolve…

Binding Variables

My Language Stack

1.program xxx

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

1

We need to bind this variable to the thrown exception

They Are in Different Stack Frames

drawing convention: stacks grow down

More things to Resolve…

Many catch in a try…

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  } catch b: OtherException {
    ...
  } catch e: Exception {
    ...
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

Decide where to come back

Many possible places… what is the rule?

We should continue to the callers, if we have non matching one…

Nesting Try (in same method / different methods)

drawing convention:
stacks grow down

More things to Resolve…

Many catch in a try…

```
program xxx {
  x = new MyClass()

  try{
    x.doSomethingMightFails()
  } catch e : MyException {
    x.doSomethingOnError(e)
  } catch b: OtherException {
    ...
  } catch e: Exception {
    ...
  }
}

class MyClass(){
  method doSomethingMightFails(){
    self.somethingElse()
  }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

Correctly Scoping the Variables

drawing convention:
stacks grow down

More things to Resolve…

Finally / Ensure

```
program xxx {
  x = new MyClass()

  try {
    a.doSomethingMightFails()
  }
  catch e : MyException {
    ...
  } then always {
    “something to do”
  }
}

class MyClass(){
  method doSomethingMightFails(){

  try {
    self.somethingElse()
  } then always {
    “something to do”
  }
 }

  method somethingElse(){
    if(...)
      throw new MyException()
  }
}
```

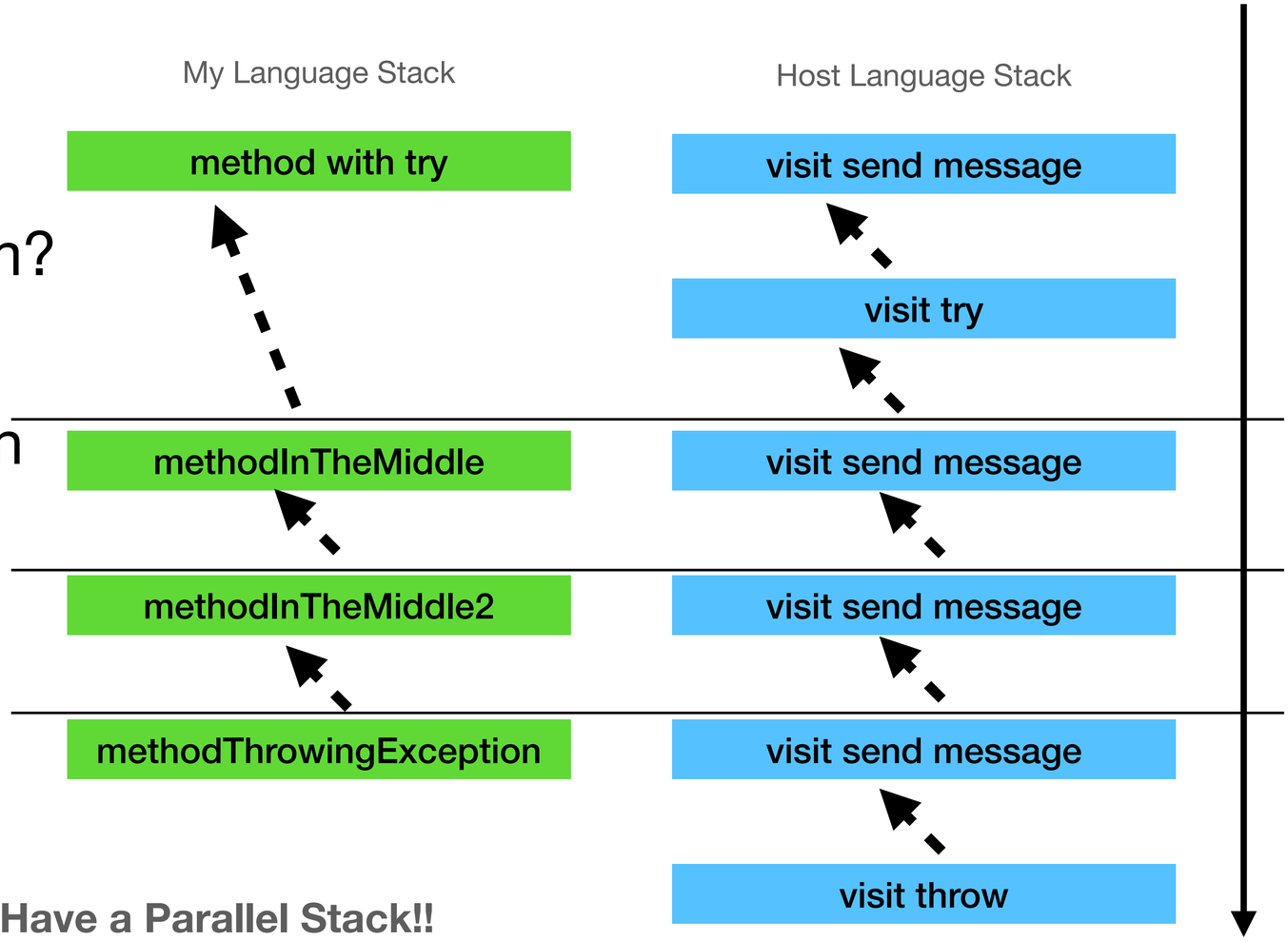
Next This One

First This One

- We need to handle the finalisation blocks.
- They should be solved in correct order
- The state of the execution should be consistent: e.g., we need to be in the correct context
- Executed always

One Approach: Using Host Language Exceptions

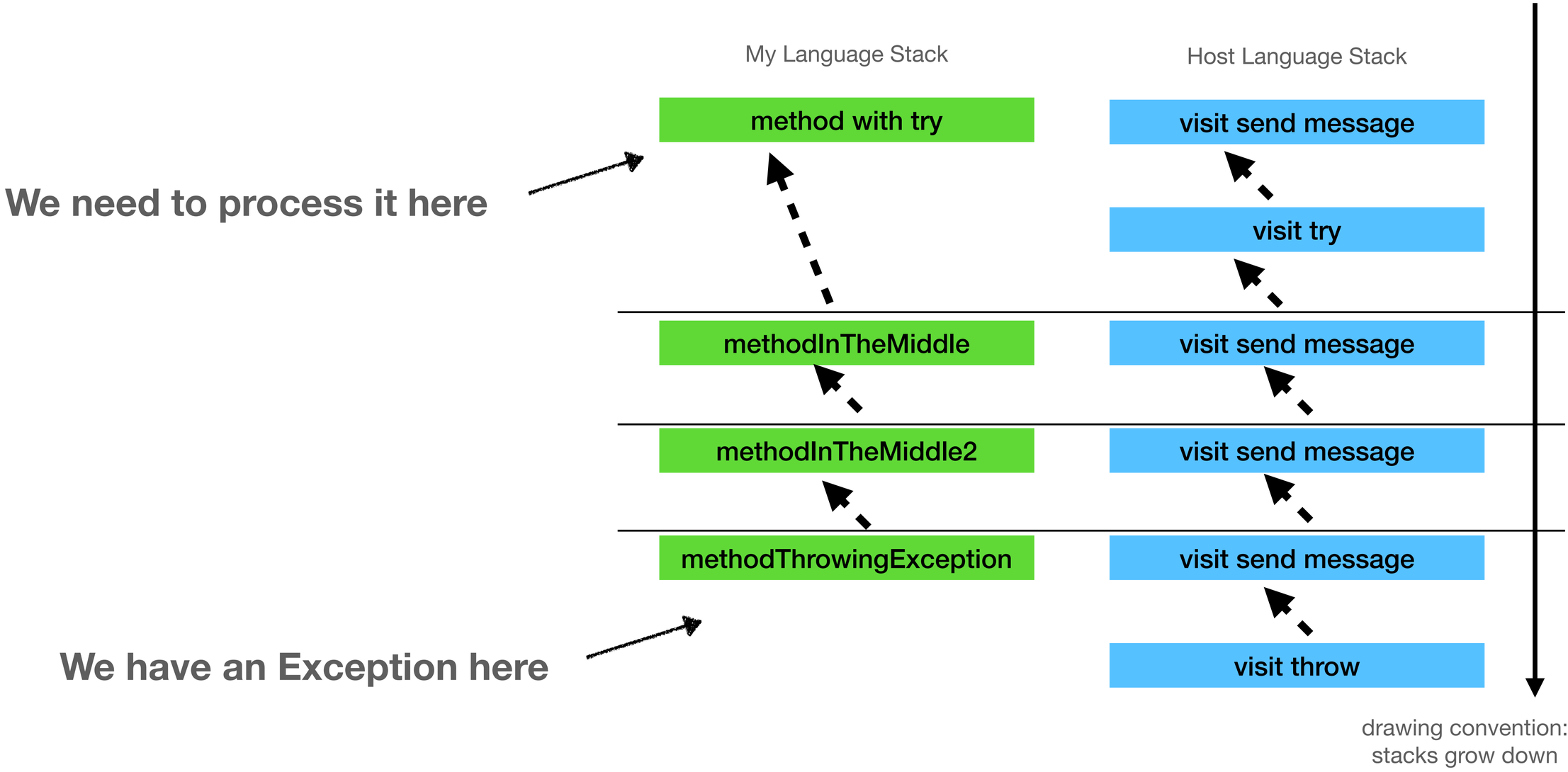
- We have a language that has exceptions already, why not use them?
- The plan is to wrap our language exception and throw a host exception with it
- If we see, we have a parallel stack



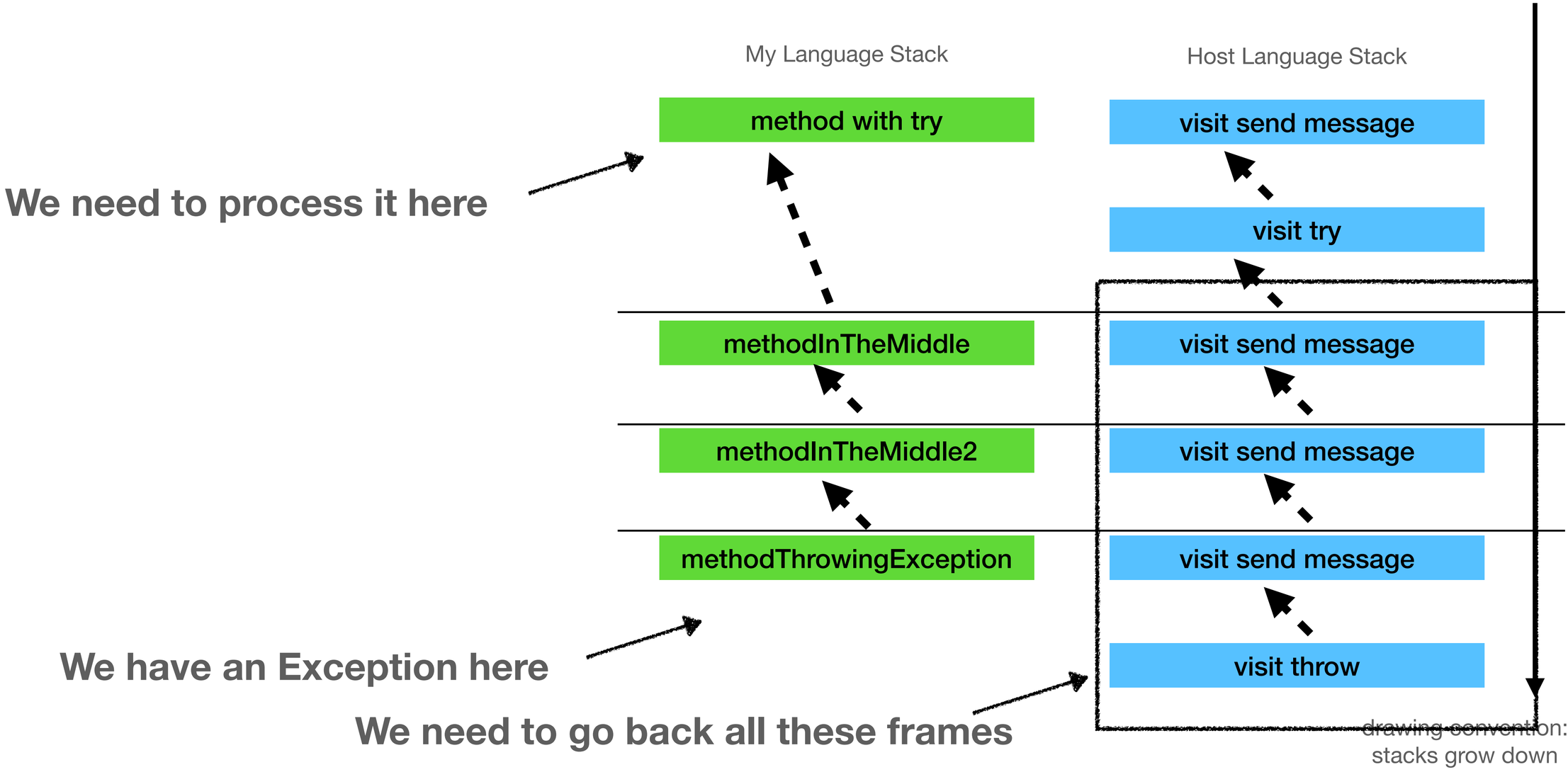
We Have a Parallel Stack!!

drawing convention:
stacks grow down

One Approach: Using Host Language Exceptions



One Approach: Using Host Language Exceptions



Using Host Language Exceptions

We have a Parallel Stack… and we use it

```
visitTryNode: aTryNode

   | returningValue |

   [[ returningValue := self visit: aTryNode tryExpression ]
      on: WrappingError
      do: [ :e |
         (self doCatch: e languageException in: aTryNode)
            ifTrue: [ “… we visit the correct catch …”]
            ifFalse: [ e pass ]
      ]
   ] ensure: [ “… visit the finally node ”  ]

   ^ returningValue
```

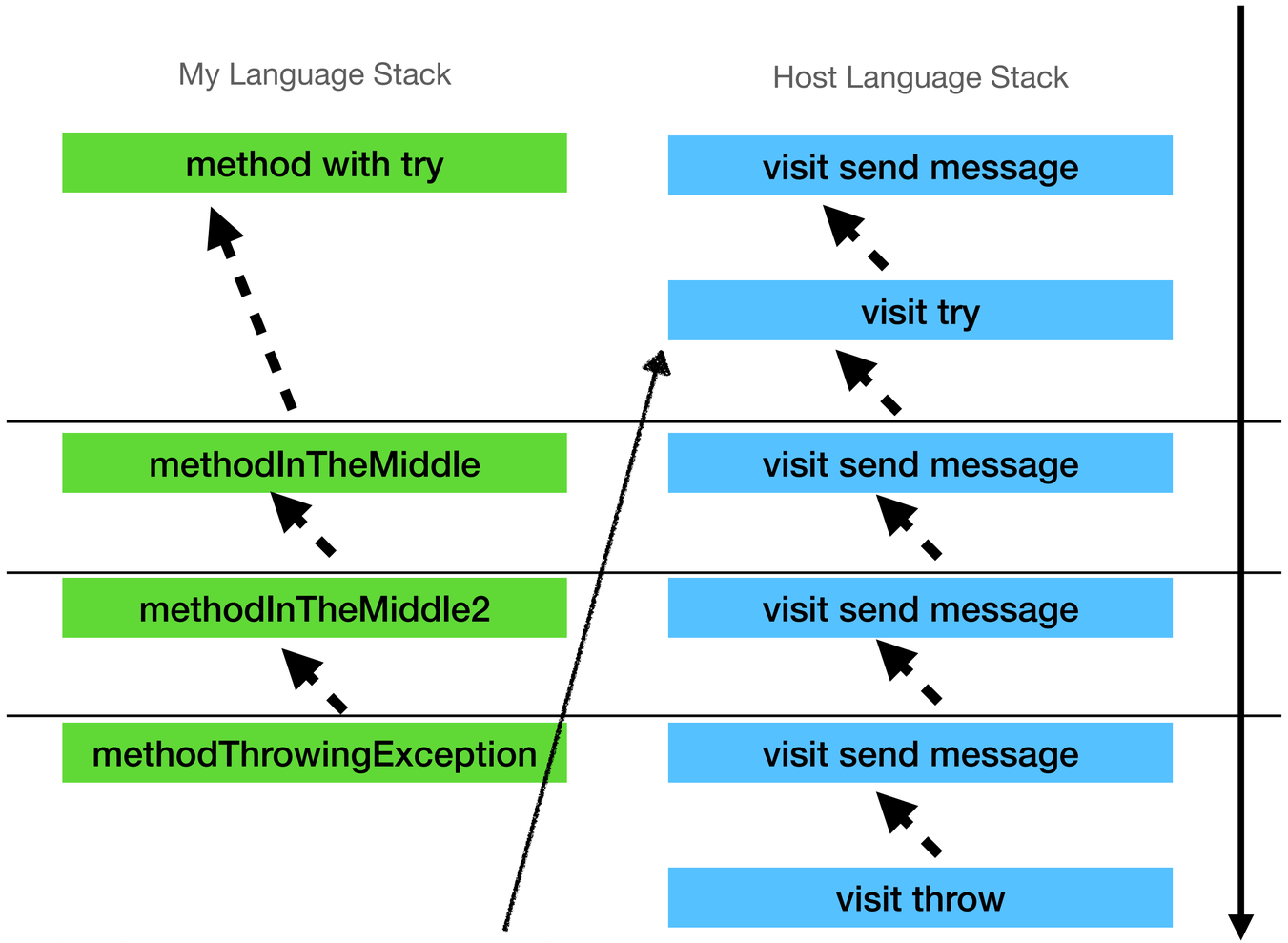
We catch the Exception here

```
visitThrow: aThrow

   | exception |

   exception := self visit: aThrow expression.

   (WrappingError wrap: exception) signal.
```

We wrap the language Exception with a host one

Using Host Language Exceptions

```
visitTryNode: aTryNode

   | returningValue |

   [[ returningValue := self visit: aTryNode tryExpression ]
      on: WrappingError
      do: [ :e |
         (self doCatch: e languageException in: aTryNode)
            ifTrue: [ "… we visit the correct catch …"]
            ifFalse: [ e pass ]
      ]
   ] ensure: [ "… visit the finally node "  ]

   ^ returningValue


visitThrow: aThrow

   | exception |

   exception := self visit: aThrow expression.

   (WrappingError wrap: exception) signal.
```

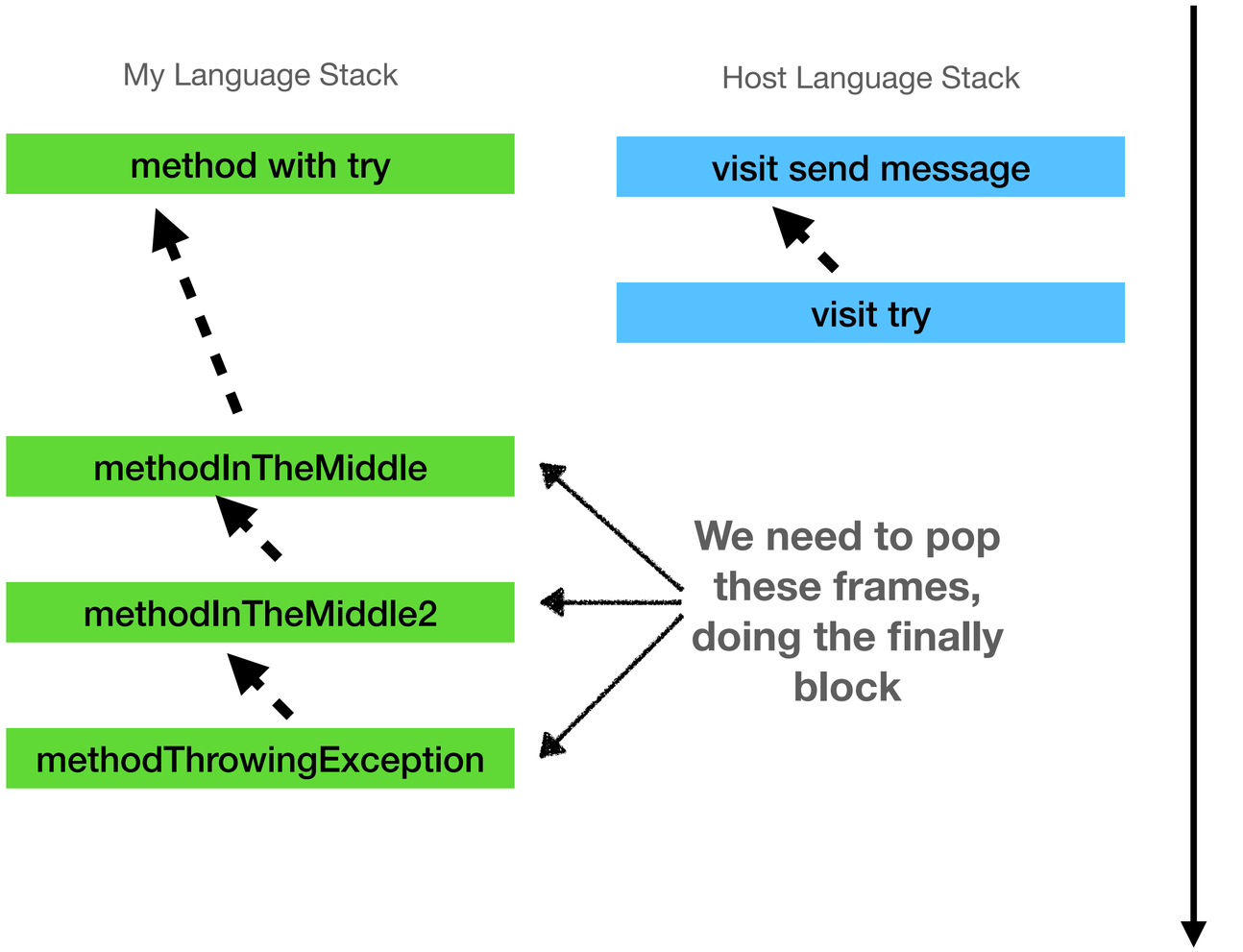


When the exception is thrown, we return there

drawing convention:
stacks grow down

Using Host Language Exceptions

```
visitTryNode: aTryNode

   | returningValue |

   [[ returningValue := self visit: aTryNode tryExpression ]
      on: WrappingError
      do: [ :e |
         (self doCatch: e languageException in: aTryNode)
            ifTrue: [ "… we visit the correct catch …"]
            ifFalse: [ e pass ]
      ]
   ] ensure: [ "… visit the finally node "  ]

   ^ returningValue


visitThrow: aThrow

   | exception |

   exception := self visit: aThrow expression.

   (WrappingError wrap: exception) signal.
```



drawing convention: stacks grow down

Using Host Language Exceptions

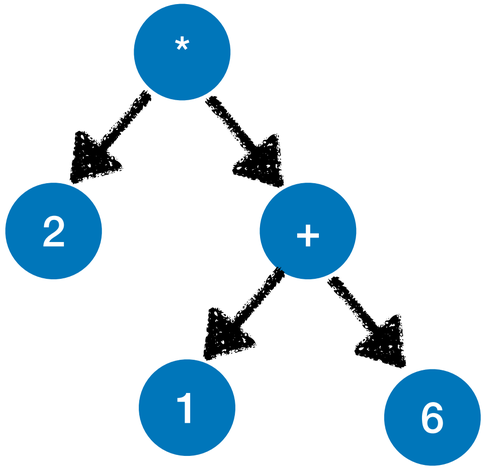
```
visitTryNode: aTryNode

   | returningValue |

   [[ returningValue := self visit: aTryNode tryExpression ]
      on: WrappingError
      do: [ :e |
         (self doCatch: e languageException in: aTryNode)
            ifTrue: [ “… we visit the correct catch …”]
            ifFalse: [ e pass ]
      ]
   ] ensure: [ “… visit the finally node ”  ]

   ^ returningValue


visitThrow: aThrow

   | exception |

   exception := self visit: aThrow expression.

   (WrappingError wrap: exception) signal.
```

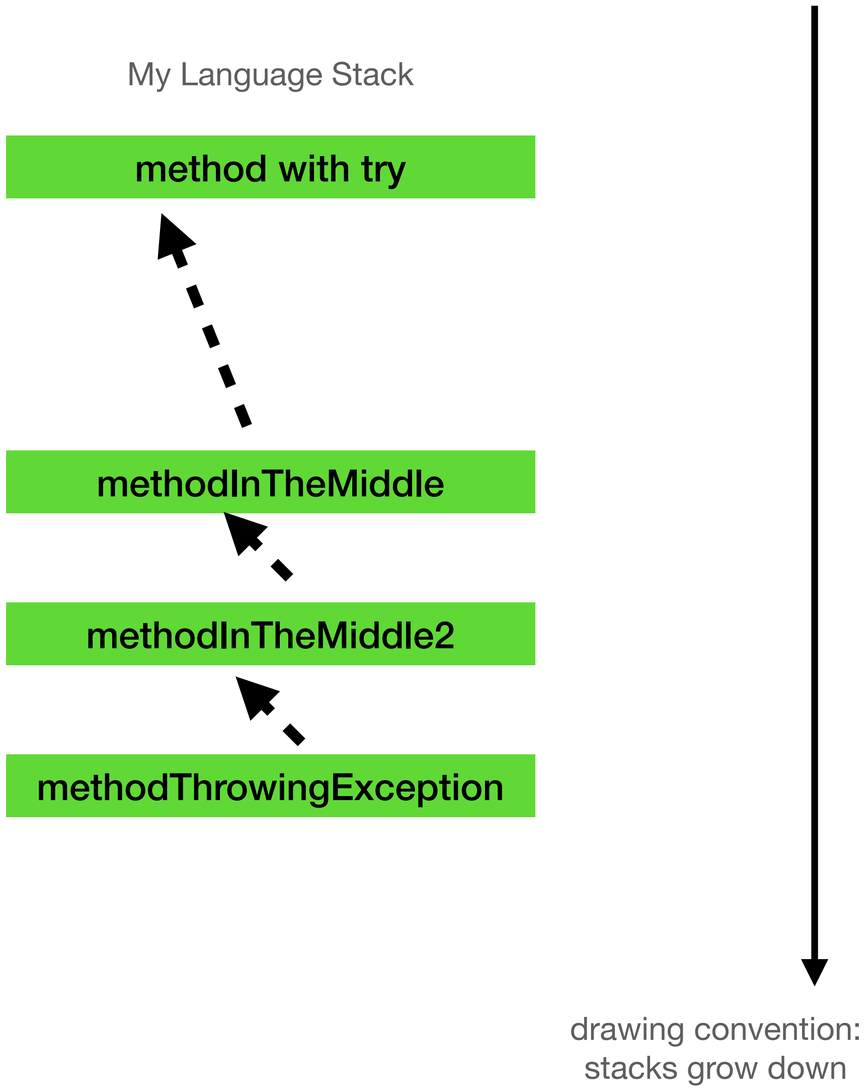
- We need a language with exceptions
- We need to pop the call-stack correctly
- We need to have a parallel stack
- Used with a recursive interpreter

drawing convention:
stacks grow down

First: Loop Based Interpreters

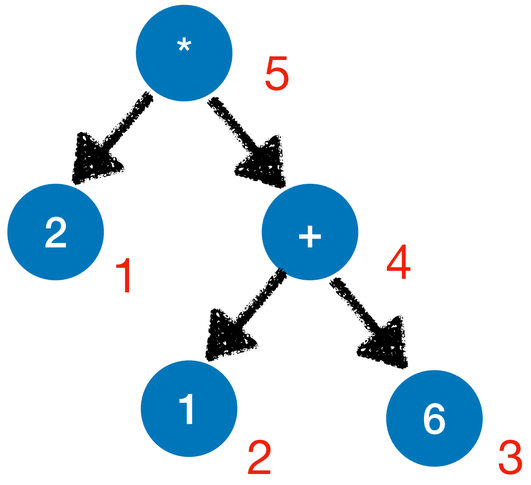
- A big loop, we execute one node at the time
- We keep the execution state in the call- stack
- We need to decide which is the next node to execute
- We have not limit in the depth of the stack
- Easier for Bytecode interpreters
- We see, that we can store intermediate results in the stack





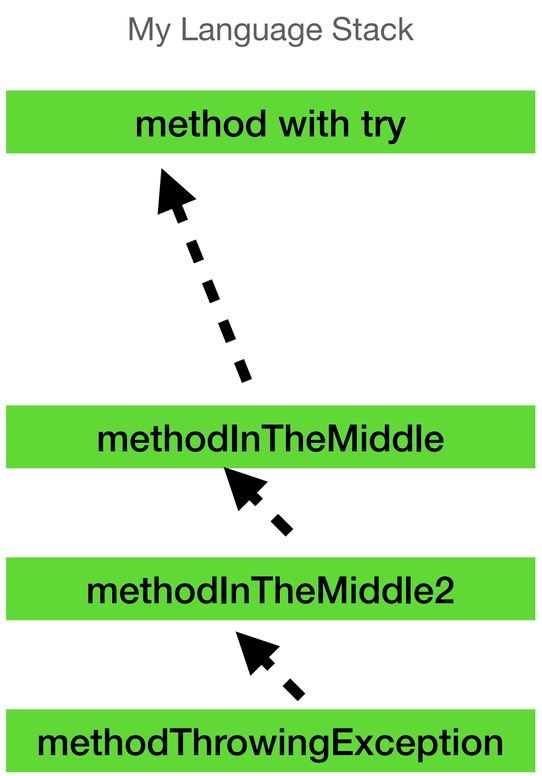
First: Loop Based Interpreters

- A big loop, we execute one node at the time
- We keep the execution state in the call- stack
- We need to decide which is the next node to execute
- We have not limit in the depth of the stack
- Easier for Bytecode interpreters
- We see, that we can store intermediate results in the stack



We need to traverse the graph correctly

We keep all information in the stack



drawing convention: stacks grow down

First: Loop Based Interpreters

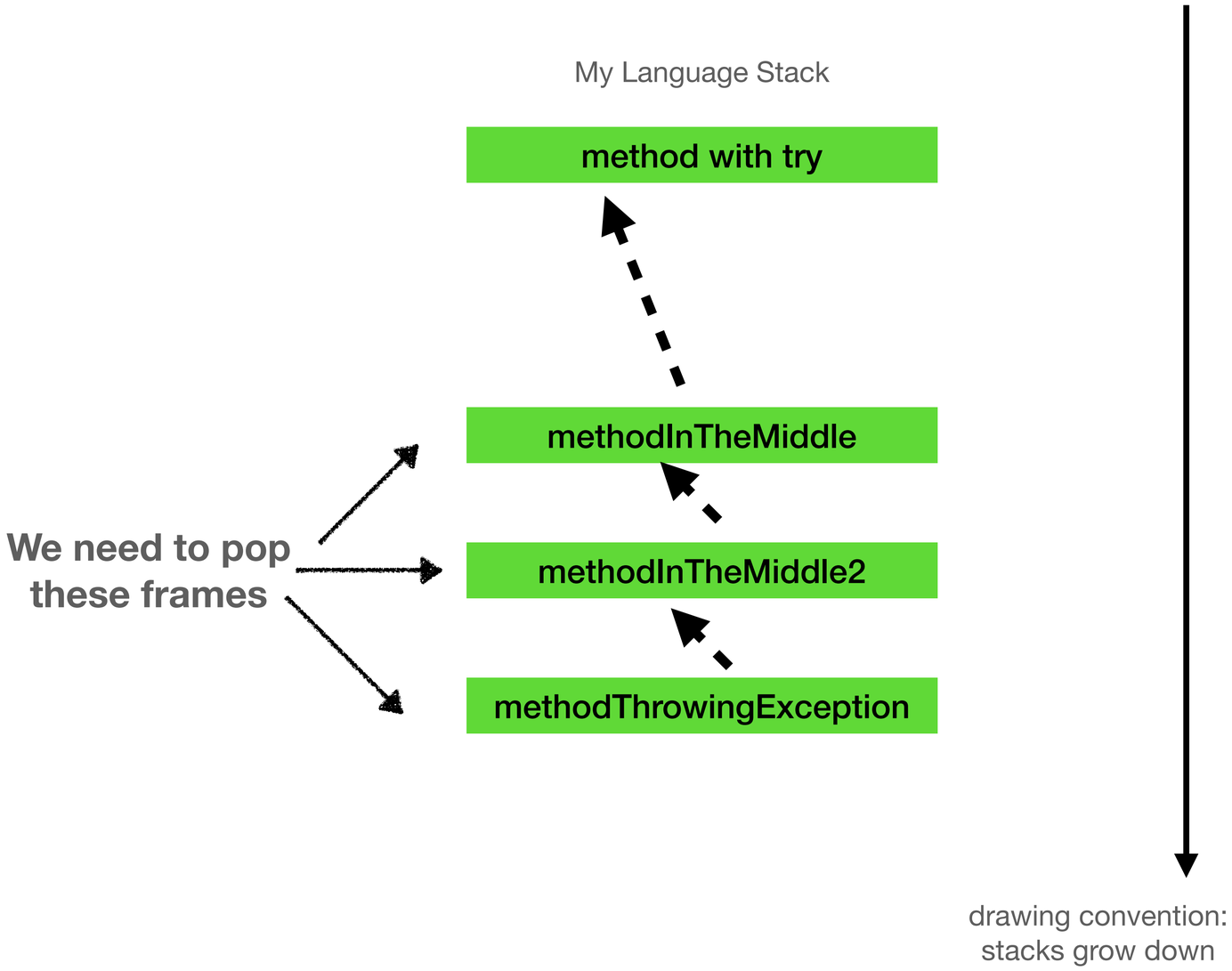
```
while(true){
   switch(currentNode){
      self.visit(currentNode)
   }
}

Interpreter >> visitSendMessage: aNode
      // push a stack-frame
      // activate a method
      // update currentNode

Interpreter >> visitLiteral: aNode
      // push literal value in stack
      // update currentNode

Interpreter >> visitThrow: aNode
      // find handling frame
      // pop other frames
      // update currentNode
 ...
```

- We need to linearise the nodes. We always need to have the next node.
- All intermediate state in the stack
- We push all arguments and return value

drawing convention:
stacks grow down

Now… back to exceptions: Walking Back the stack

- We use the state in the Call-Stack
- When we have a throw, we need to find the stack that will continue the execution
- Used in languages without exceptions.
- When we have a language with deeper than possible in host language



Now… back to exceptions: Walking Back the stack

- This is more complex in recursive interpreters
- Easier in loop based interpreters
- Easier to implement Finally / Ensure and Resume
- We need to mark the frames with finalisation or try, to evaluate them
- We need to keep all the state for handling exceptions in the stack
- When executed a ensure/finally block, in an exception, we need to continue walking back the stack