Programming Languages and Techniques (CIS120)

Lecture 30

Exceptions

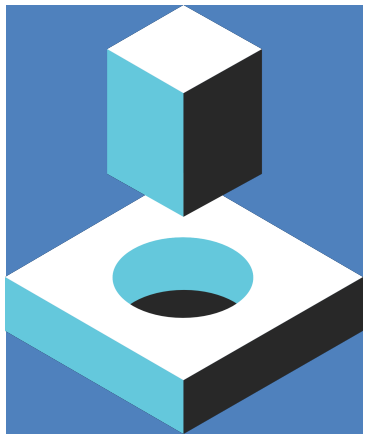
Chapter 27

Exceptions

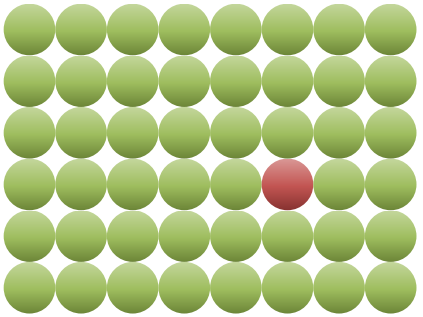
Dealing with the unexpected

Why do methods “fail”?

- Some methods expect their arguments to satisfy conditions
  - Input to `max` must be a nonempty list, Item must be non-null, more elements must be available when calling `next`, …
- Interfaces may be imprecise
  - Some Iterators don't support the "remove" operation
- External components of a system might fail
  - Try to open a file or resource that doesn't exist
- Resources might be exhausted
  - Program uses all of the computer's memory or disk space



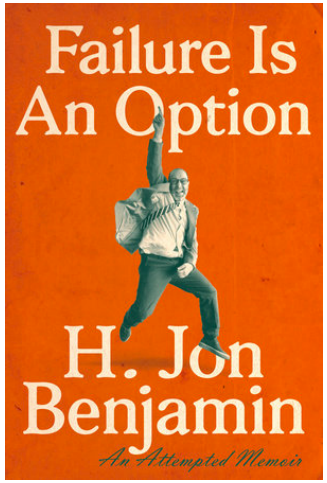
Error 404
Page Not Found!



- These are all *exceptional circumstances…*
  - How do we deal with them?

Ways to handle failure

- Return an error value (or default value)
  - e.g. Math.sqrt returns NaN ("not a number") if given input < 0
  - e.g. Many Java libraries return `null`
  - e.g. file reading method returns -1 if no more input available
  - *Caller is supposed to check return value, but it's easy to forget* ☹
  - *Use with caution – easy to introduce nasty bugs!* ☹
- Use an informative result
  - e.g. in OCaml we used **options** to signal potential failure
  - *Passes responsibility to caller, who must do the proper check to extract value*
- Use ***exceptions***
  - Available both in OCaml and Java
  - Any caller (not just the immediate one) can handle the exception
  - If an exception is not caught, the program terminates



Exceptions

- An exception is an *object* representing an abnormal condition
  - Its internal state describes what went wrong
  - e.g.: NullPointerException, IllegalArgumentException, IOException
  - Can define your own exception classes
- *Throwing* an exception is an *emergency exit* from the current context
  - The exception propagates up the invocation stack until it either reaches the top of the stack, in which case the program aborts with the error, or the exception is *caught*
- *Catching* an exception lets callers take appropriate actions to handle the abnormal circumstances
  - Java uses try / catch blocks to handle exceptions.

Example from Pennstagram HW

```
private void load(String filename) {
        ImageIcon icon;

        try {
            if ((new File(filename)).exists())
                icon = new ImageIcon(filename);
            else {
                java.net.URL u = new java.net.URL(filename);
                icon = new ImageIcon(u);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        …
}
```

Simplified Example

```
class C {
  public void foo() {
    this.bar();
    System.out.println("here in foo");
  }
  public void bar() {
    this.baz();
    System.out.println("here in bar");
  }
  public void baz() {
    throw new RuntimeException();
  }
}
```

What happens if we do `(new C()).foo()` ?

1. Program stops normally
2. Program stops with an uncaught exception
3. Program prints “here in bar”, then stops w/ exception
4. Program prints “here in bar”, then “here in foo”, then stops w/ exception

Answer: 2

Abstract Stack Machine

Workspace

```
(new C()).foo();
```

Stack

Heap

Abstract Stack Machine

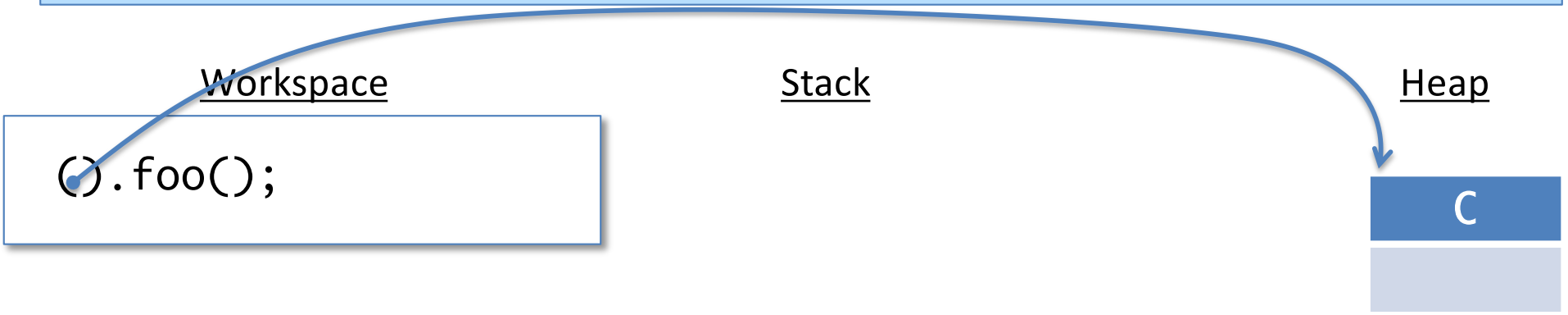
Workspace

```
(new C()).foo();
```

Stack

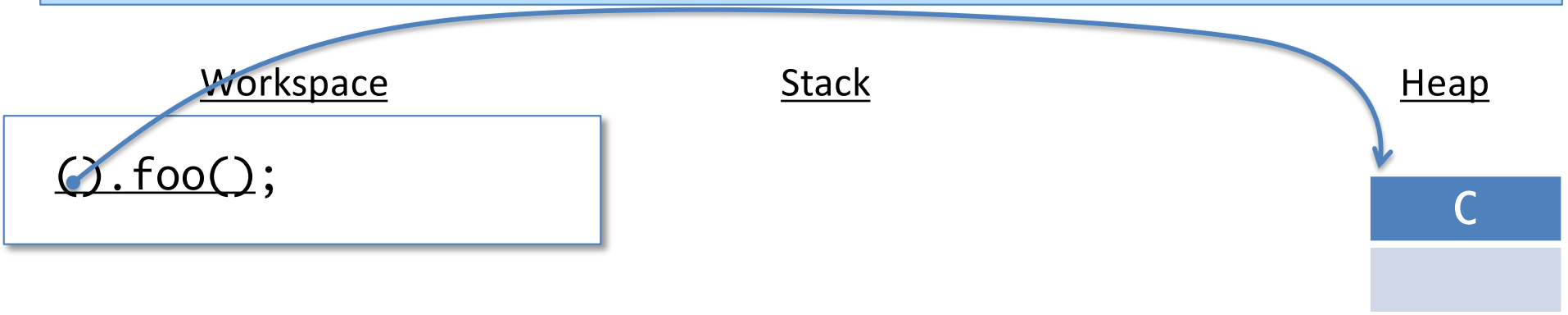
Heap

Abstract Stack Machine

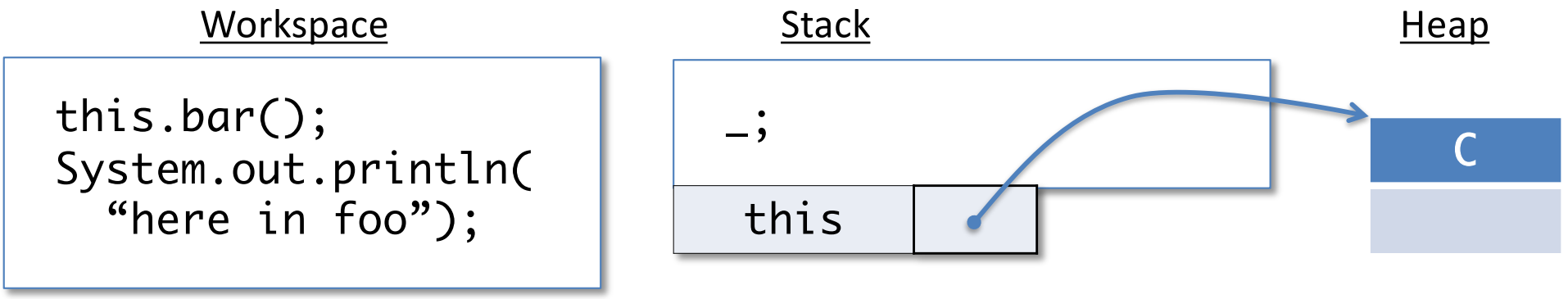


Allocate a new instance of C in the heap. (Skipping details of trivial constructor for C.)

Abstract Stack Machine



Abstract Stack Machine



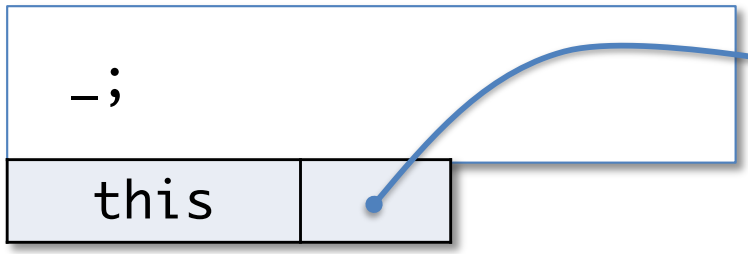
Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack. Use the dynamic class to lookup the method body from the class table.

Abstract Stack Machine

Workspace

```
this.bar();
System.out.println(
  “here in foo”);
```

Stack

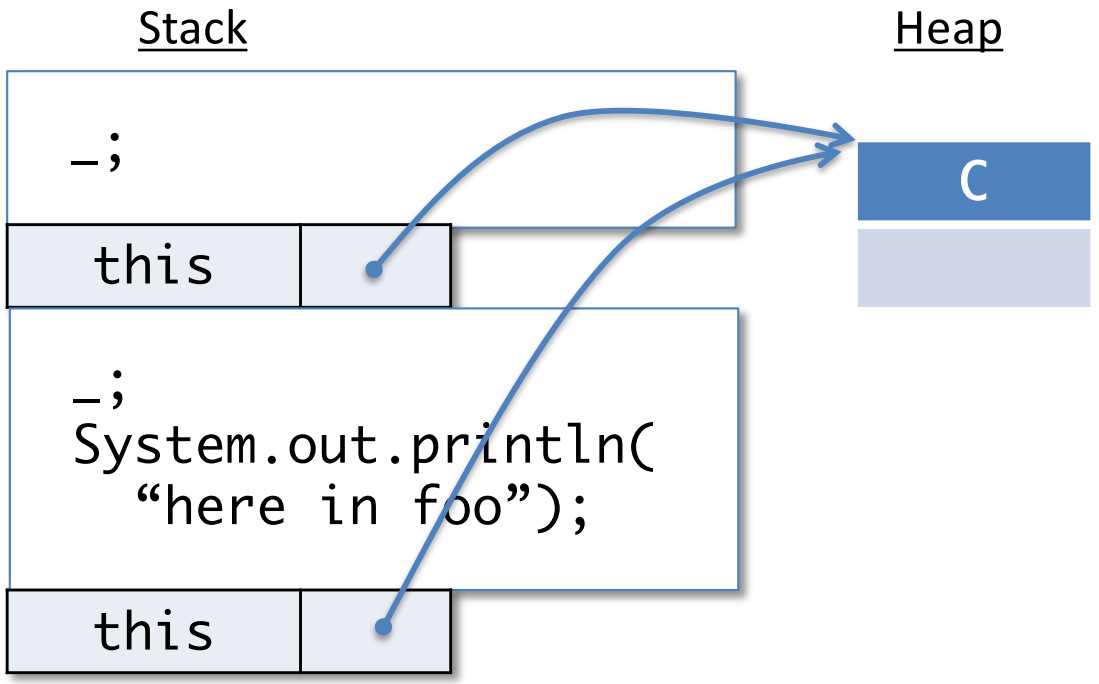


Heap

Abstract Stack Machine

Workspace

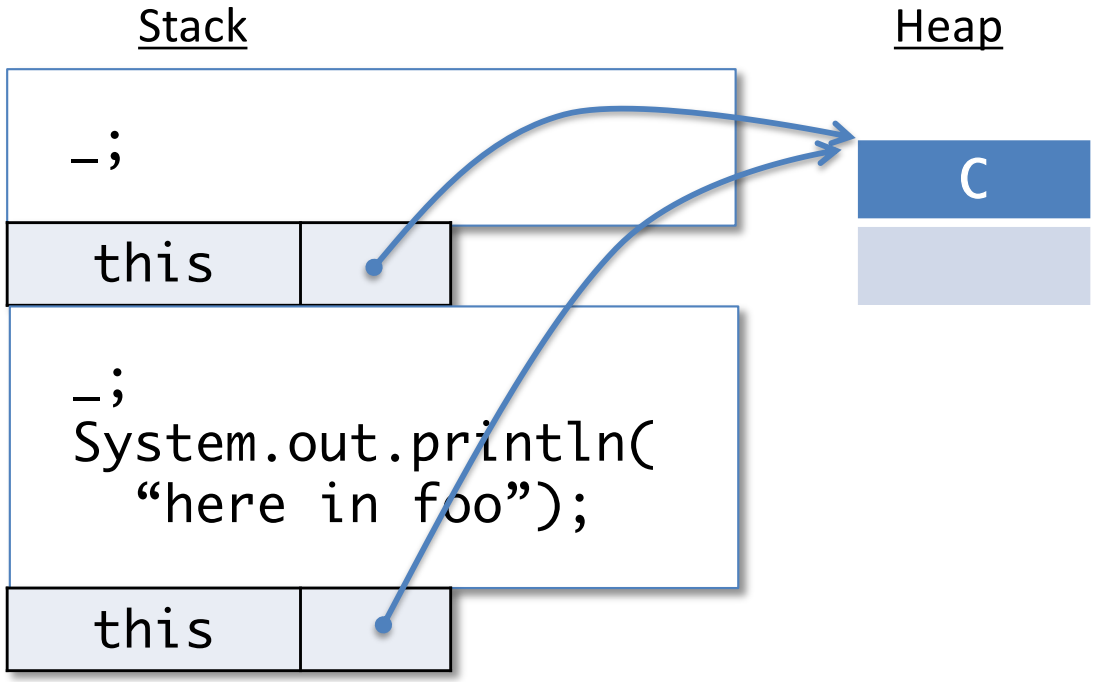
```
this.baz();
System.out.println(
  “here in bar”);
```



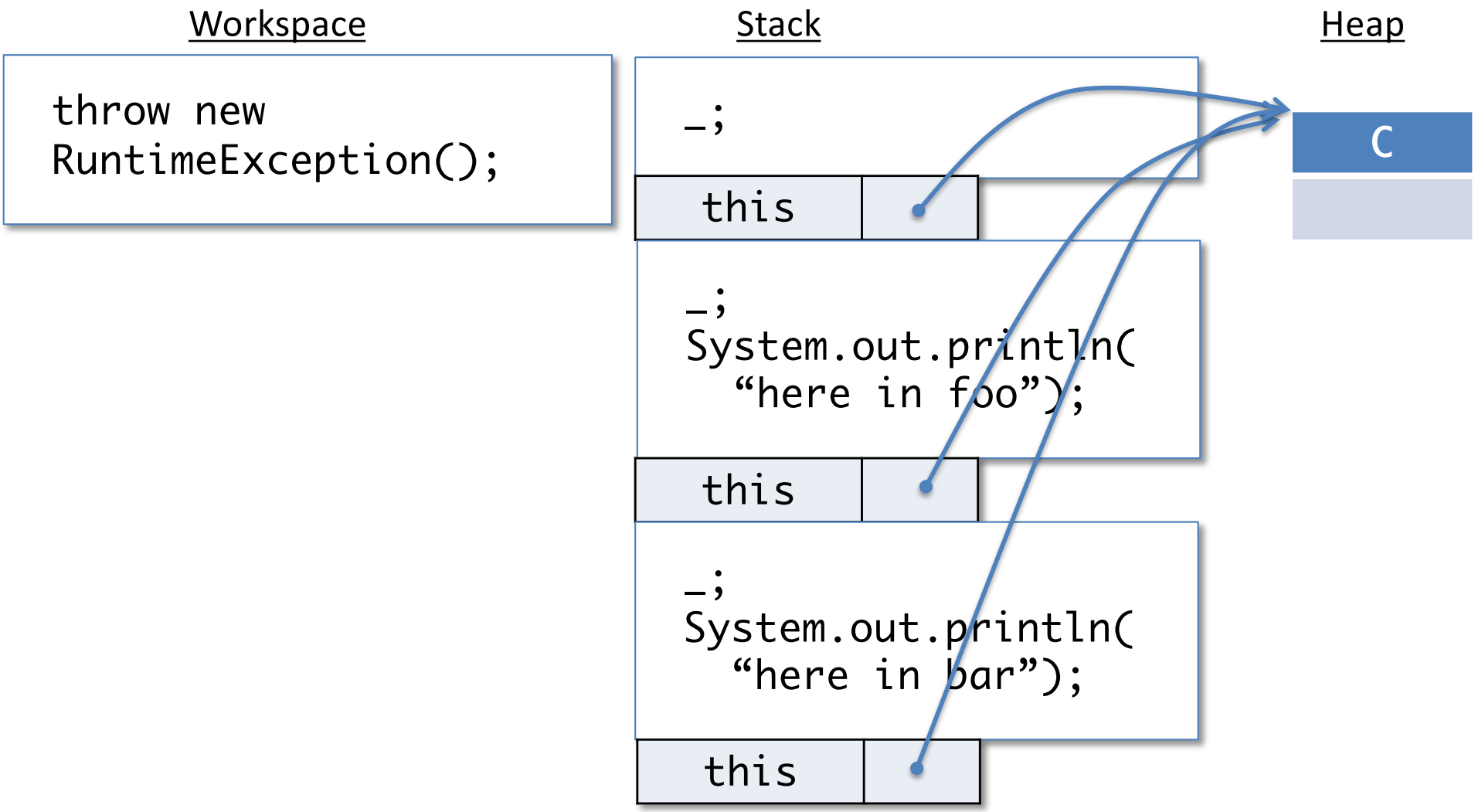
Abstract Stack Machine

Workspace

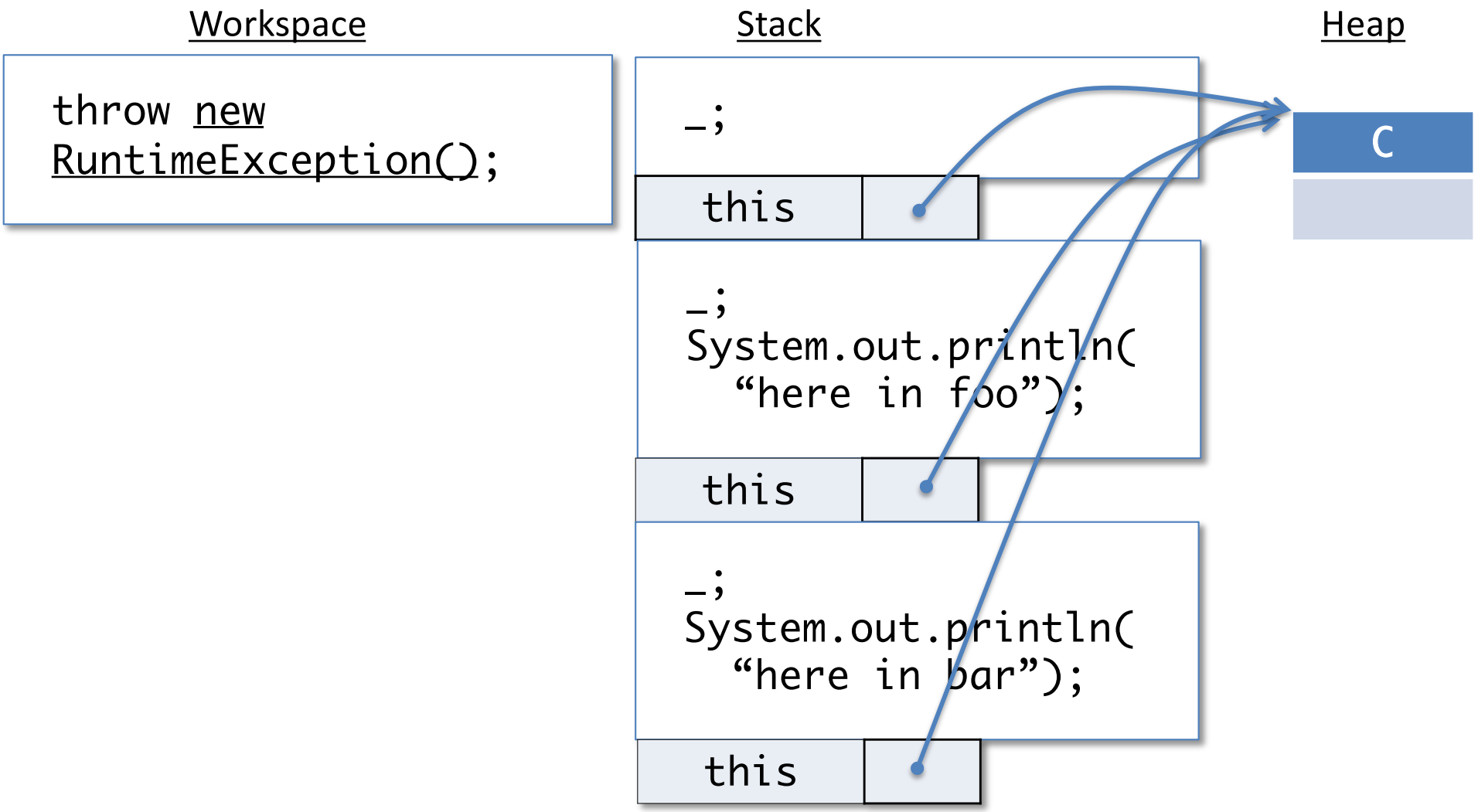
```
this.baz();
System.out.println(
  “here in bar”);
```



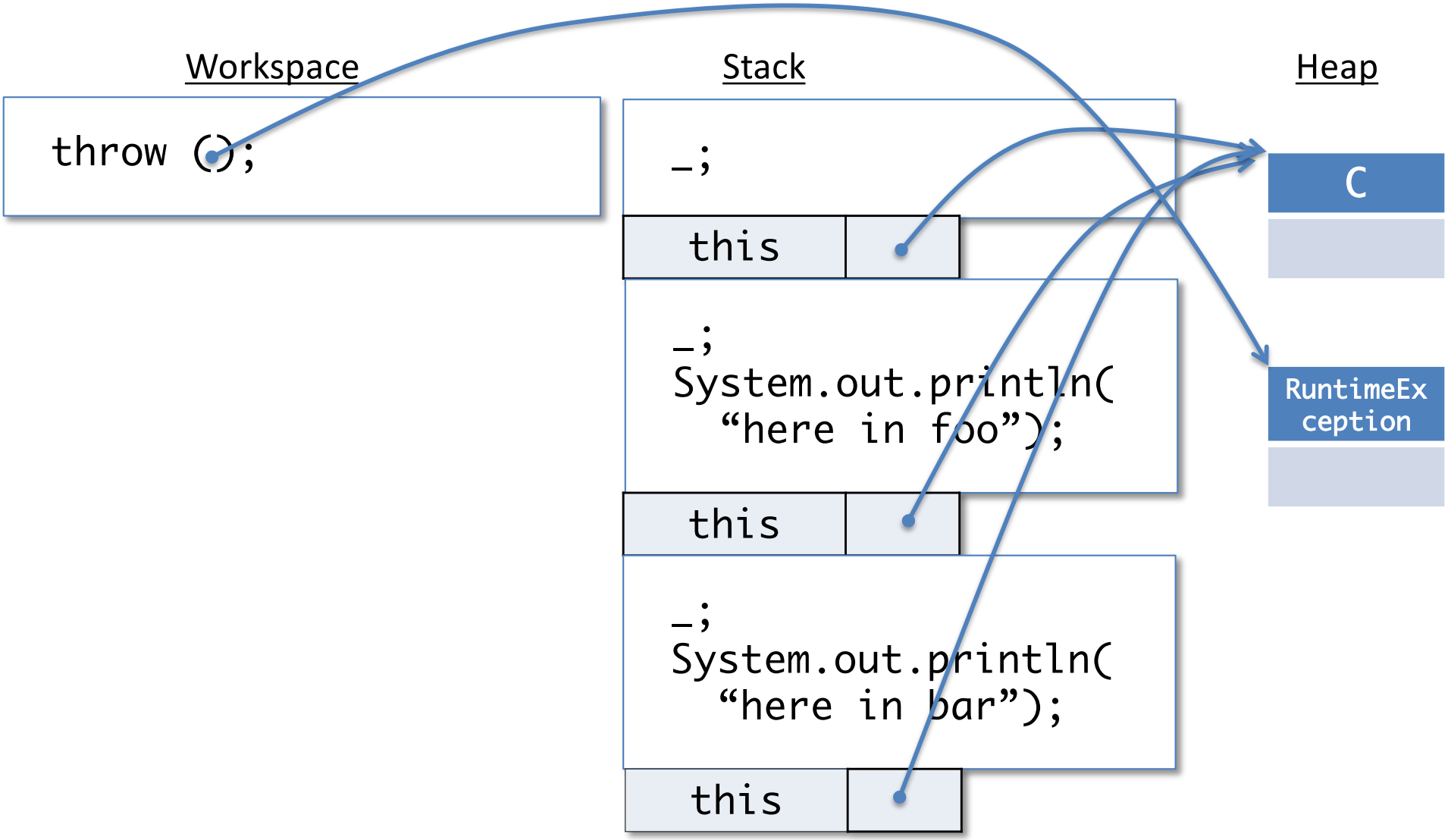
Abstract Stack Machine



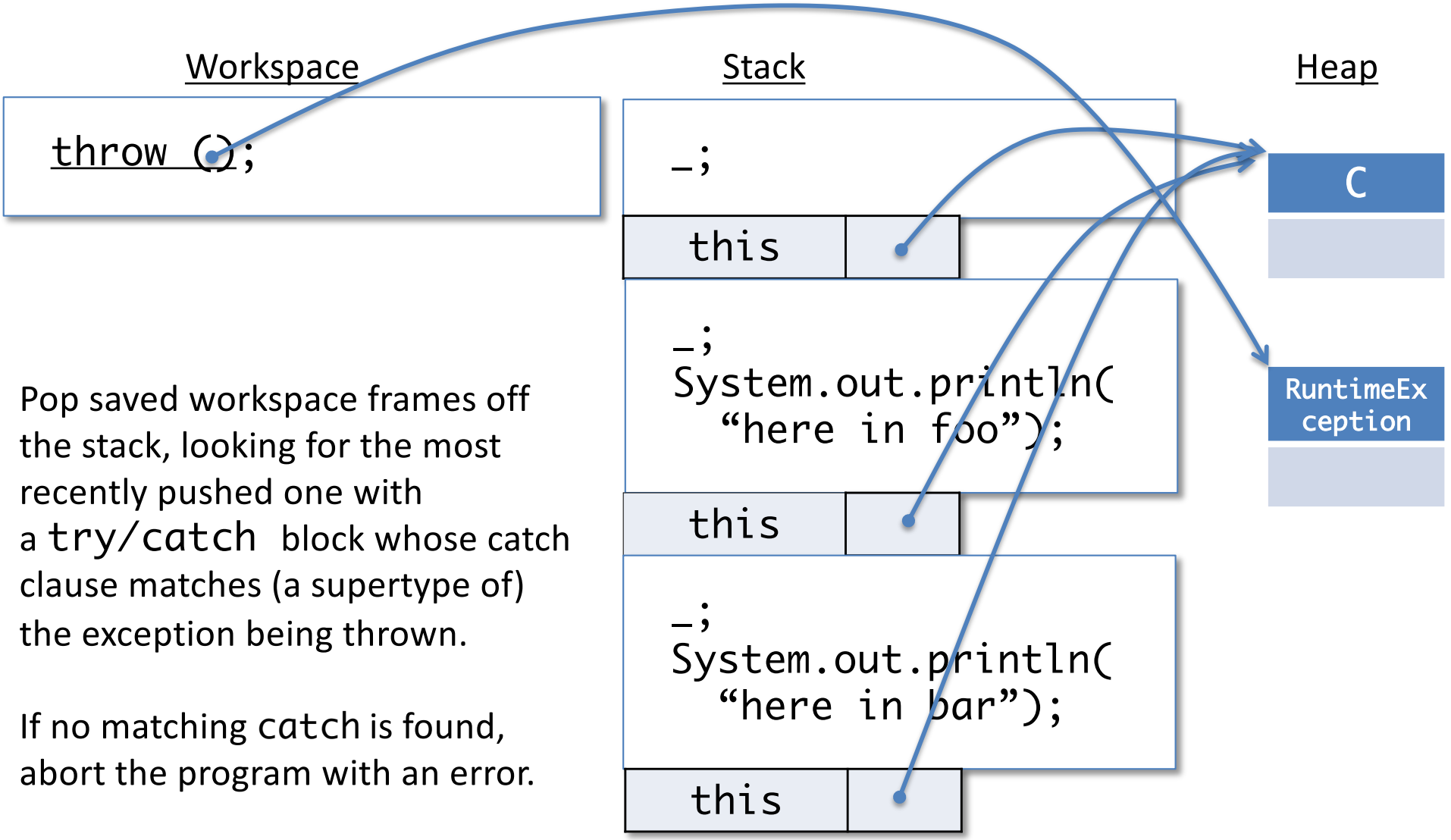
Abstract Stack Machine



Abstract Stack Machine



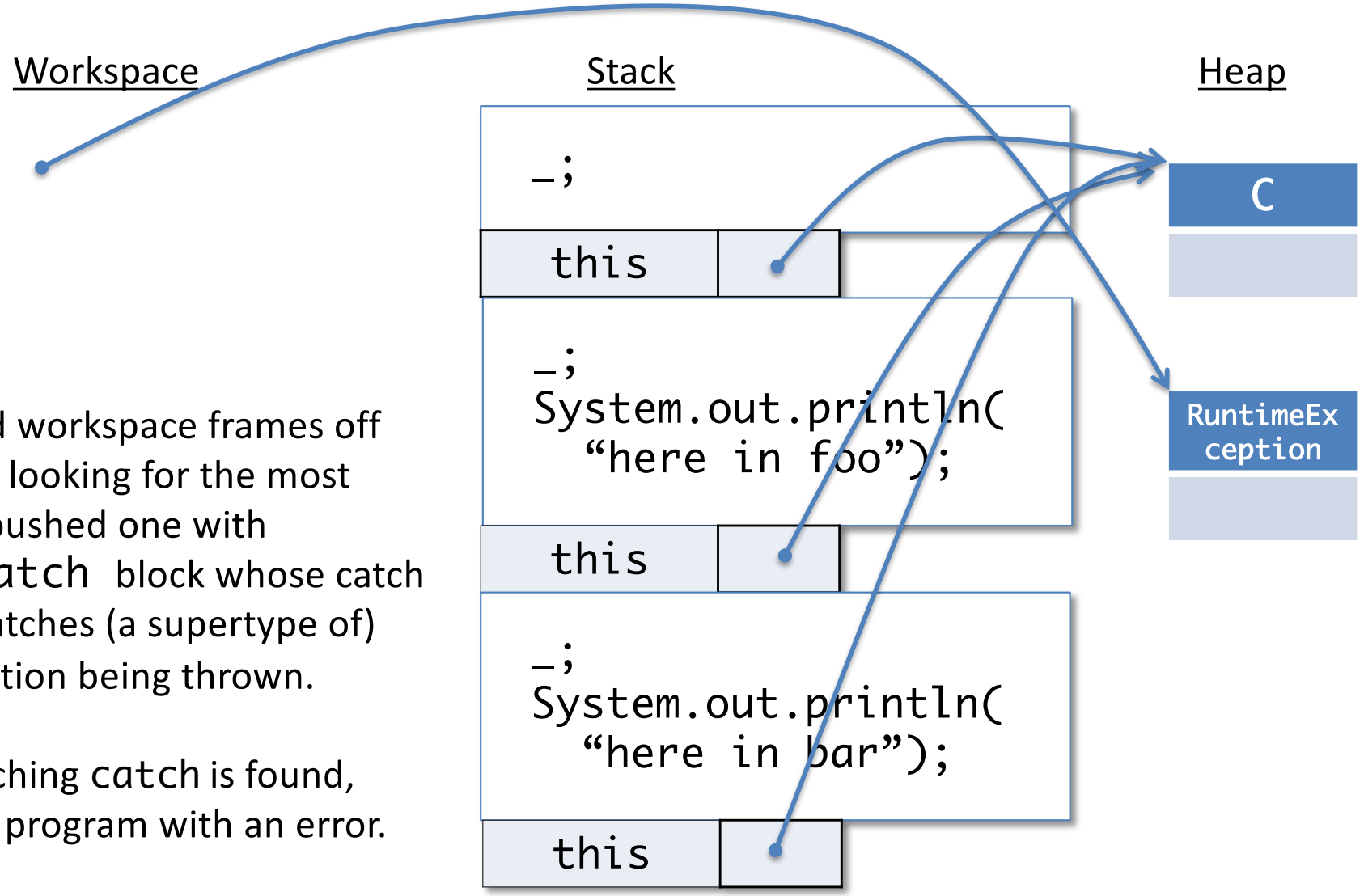
Abstract Stack Machine



Pop saved workspace frames off the stack, looking for the most recently pushed one with a `try/catch` block whose catch clause matches (a supertype of) the exception being thrown.

If no matching `catch` is found, abort the program with an error.

Abstract Stack Machine



Pop saved workspace frames off the stack, looking for the most recently pushed one with a `try/catch` block whose catch clause matches (a supertype of) the exception being thrown.

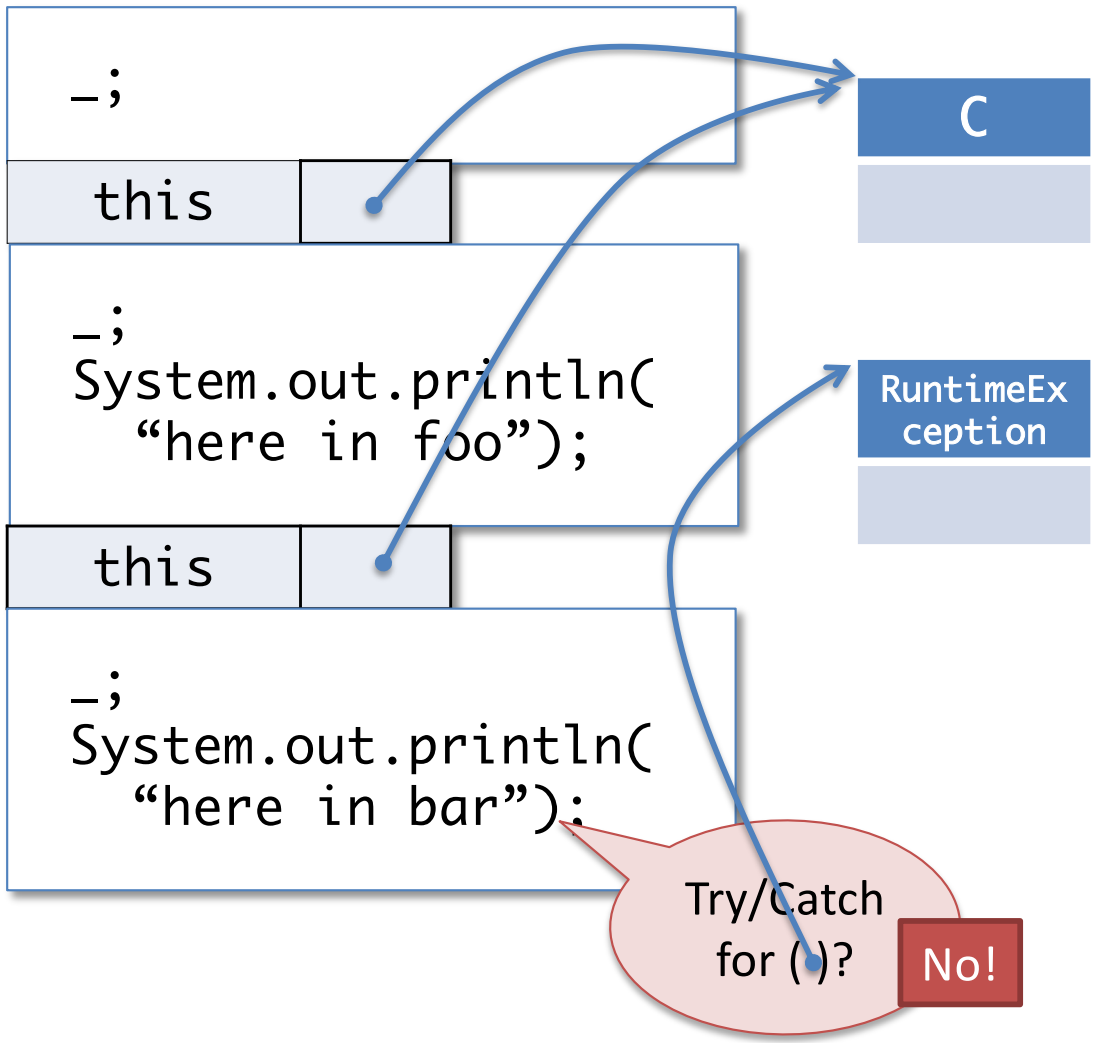
If no matching `catch` is found, abort the program with an error.

Abstract Stack Machine

Workspace

Stack

Heap



Pop saved workspace frames off the stack, looking for the most recently pushed one with a `try/catch` block whose catch clause matches (a supertype of) the exception being thrown.

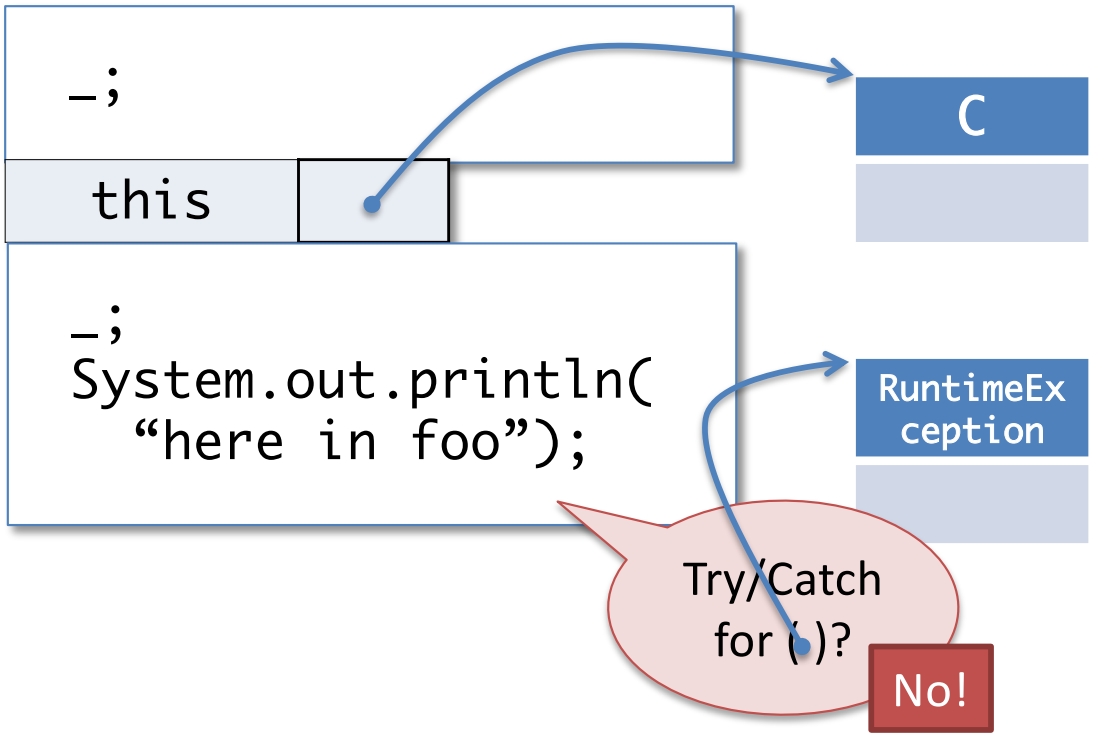
If no matching `catch` is found, abort the program with an error.

Abstract Stack Machine

Workspace

Stack

Heap

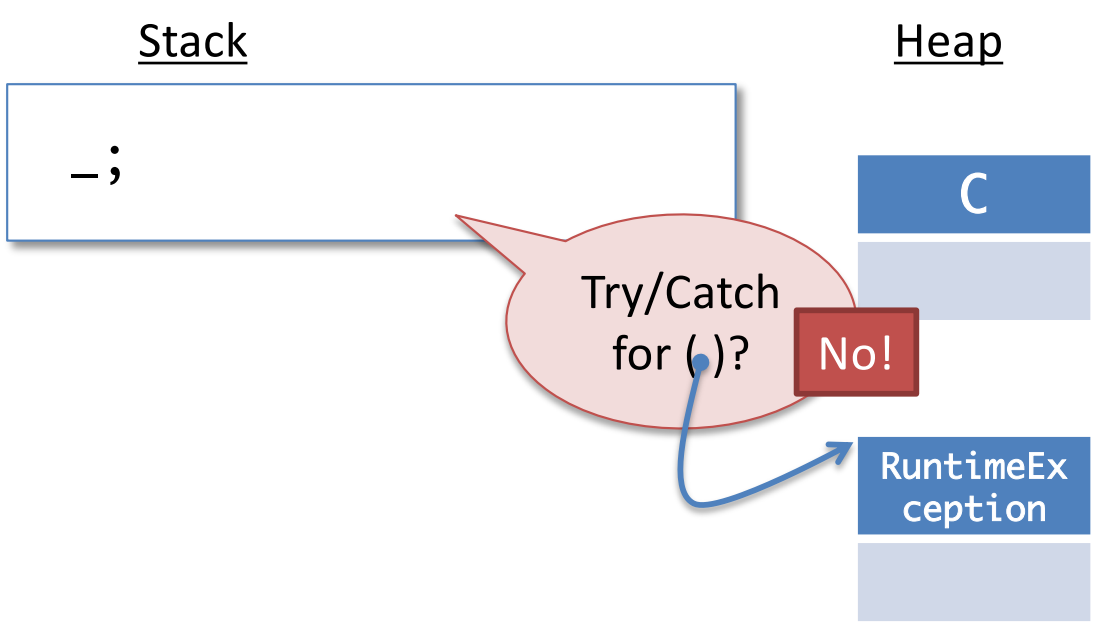


Discard the current workspace.

Then, pop saved workspace frames off the stack, looking for the most recently pushed one that contains a `try/catch` block whose catch clause declares a supertype of the exception being thrown.

If no matching catch is found, abort the program with an error.

Abstract Stack Machine



Discard the current workspace.

Then, pop saved workspace frames off the stack, looking for the most recently pushed one that contains a `try/catch` block whose catch clause declares a supertype of the exception being thrown.

If no matching catch is found, abort the program with an error.

Abstract Stack Machine

Workspace | Stack | Heap

Program terminated with uncaught exception ( )!

Discard the current workspace.

Then, pop saved workspace frames off the stack, looking for the most recently pushed one that contains a `try/catch` block whose catch clause declares a supertype of the exception being thrown.

If no matching catch is found, abort the program with an error.

Catching the Exception

```
class C {
  public void foo() {
    this.bar();
    System.out.println("here in foo");
  }
  public void bar() {
    try {
      this.baz();
    } catch (Exception e) { System.out.println("caught"); }
    System.out.println("here in bar");
  }
  public void baz() {
    throw new RuntimeException();
  }
}
```

- *Now* what happens if we do `(new C()).foo();`?

Abstract Stack Machine

Workspace

```
(new C()).foo();
```

Stack

Heap

Abstract Stack Machine

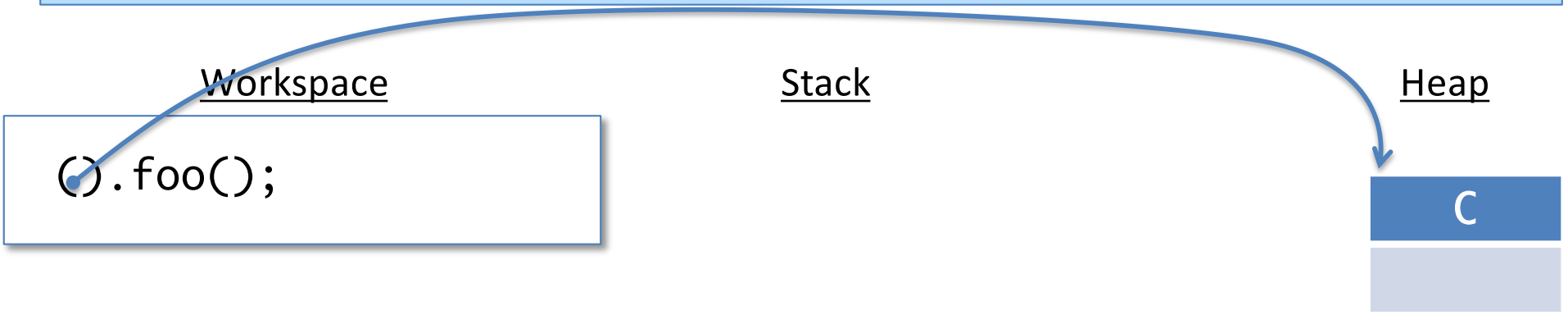
Workspace

```
(new C()).foo();
```

Stack

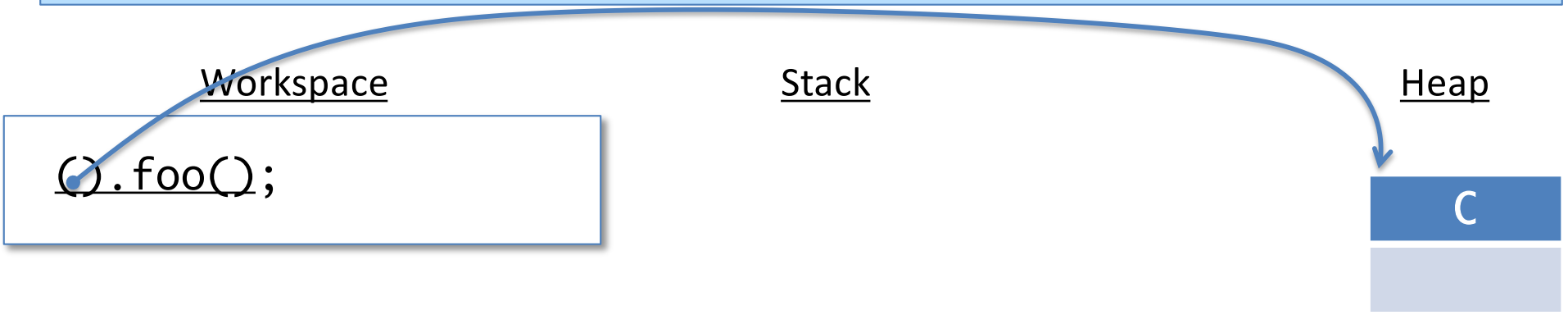
Heap

Abstract Stack Machine

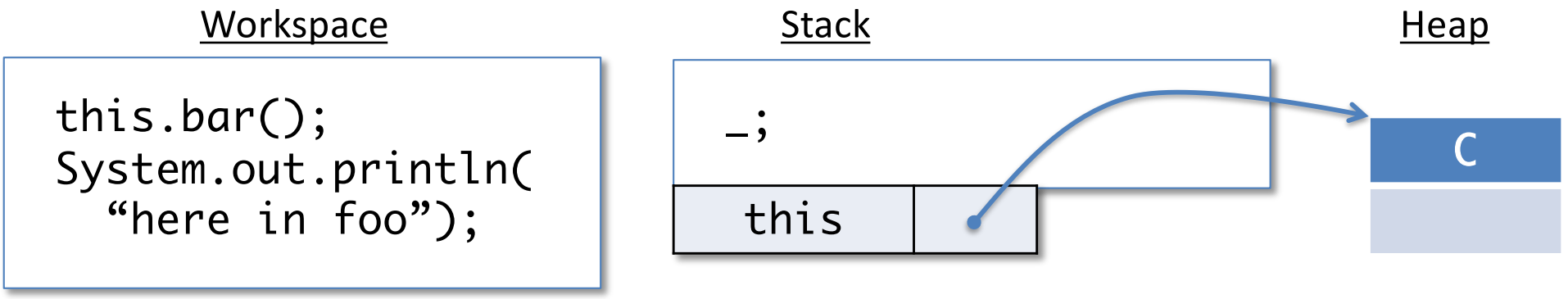


Allocate a new instance of C in the heap.

Abstract Stack Machine



Abstract Stack Machine



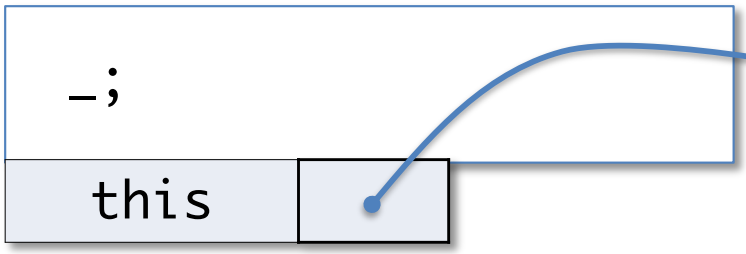
Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack.

Abstract Stack Machine

Workspace

```
this.bar();
System.out.println(
  “here in foo”);
```

Stack



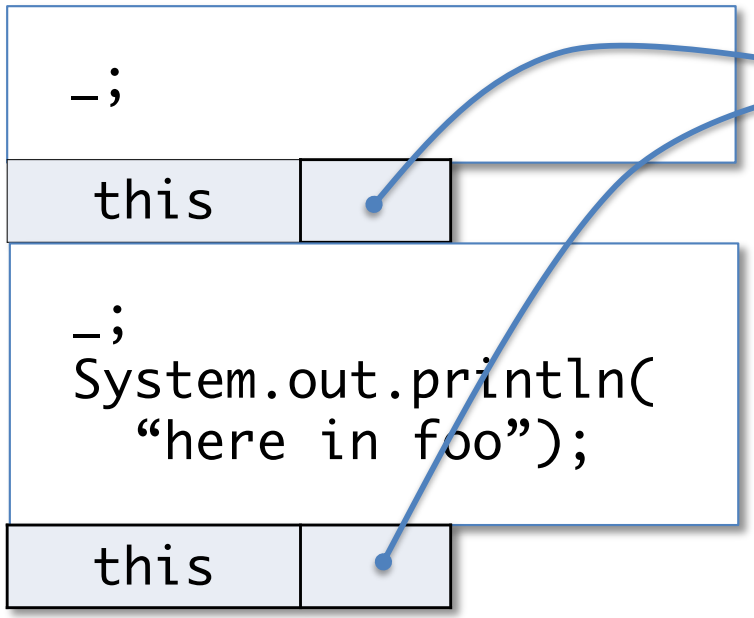
Heap

Abstract Stack Machine

Workspace

```
try {
    baz();
} catch (Exception e)
{ System.out.println
  (“caught”); }
System.out.println(
  “here in bar”);
```

Stack

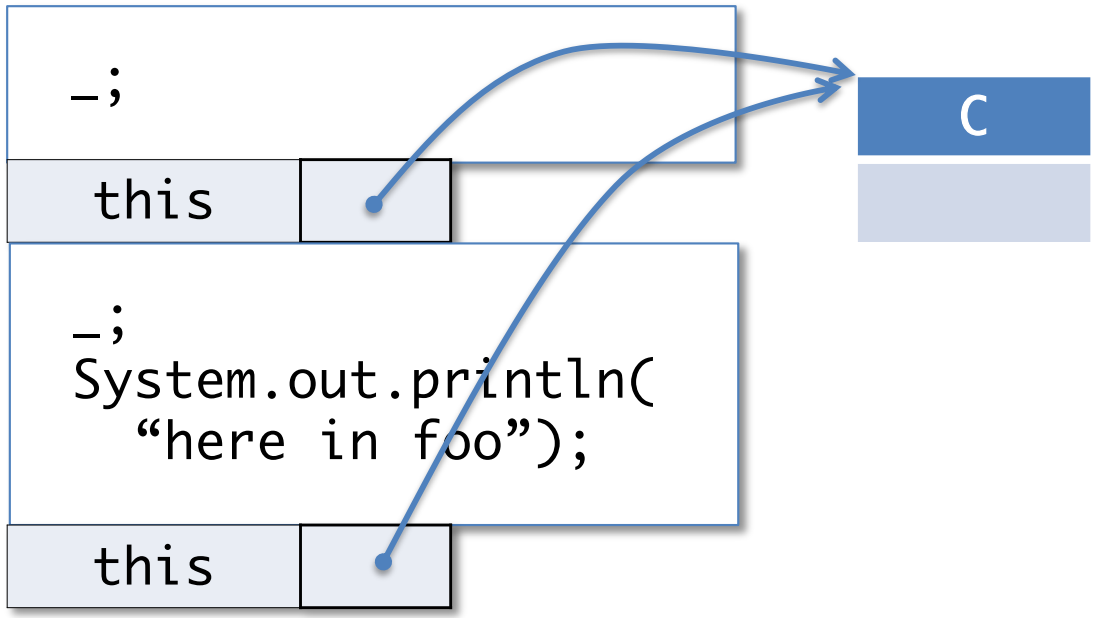


Heap

Abstract Stack Machine

Workspace

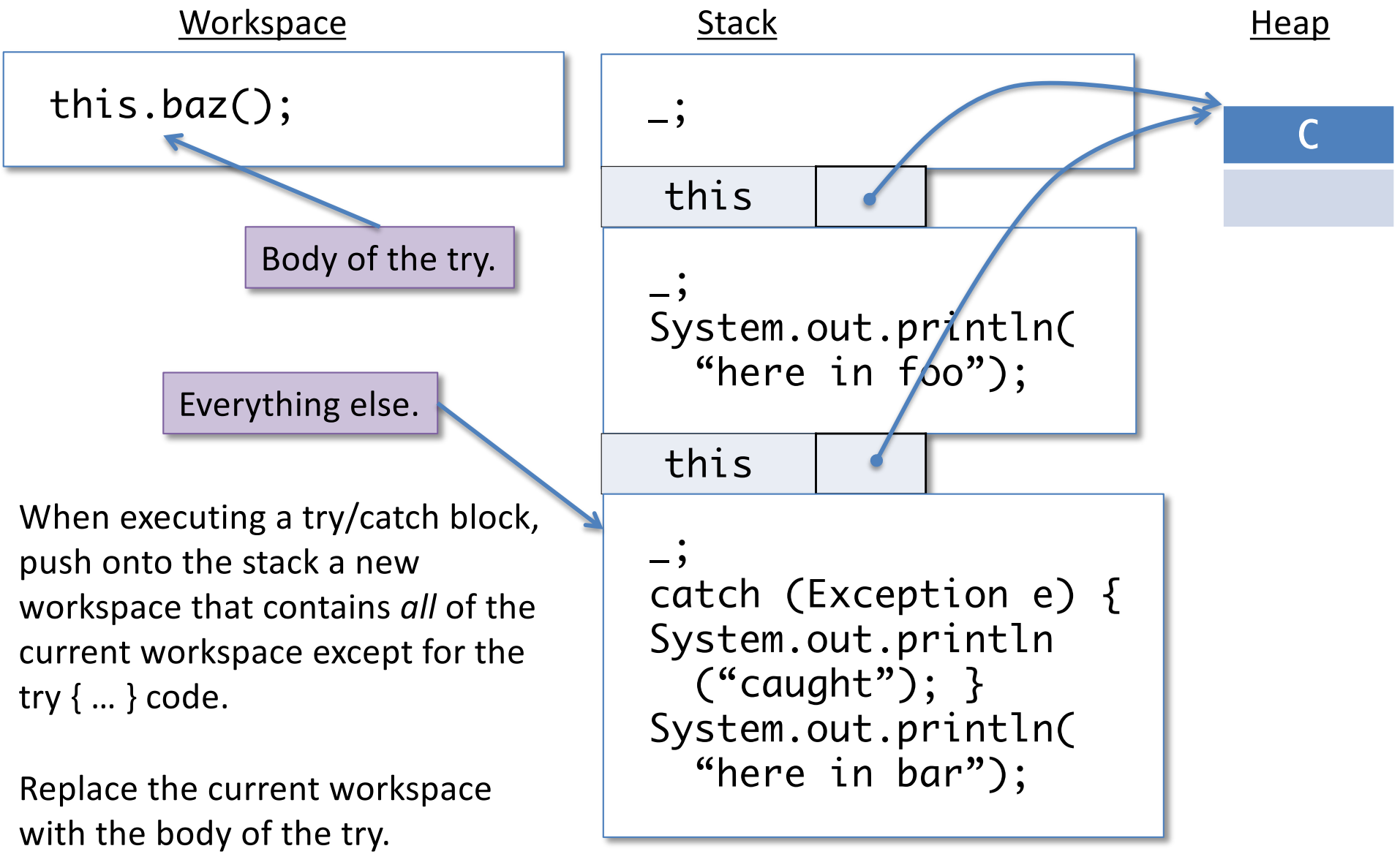
```
try {
   baz();
} catch (Exception e)
{ System.out.println
   ("caught"); }
System.out.println(
   "here in bar");
```



When executing a try/catch block, push onto the stack a new workspace that contains *all* of the current workspace except for the try { … } code.

Replace the current workspace with the body of the try.

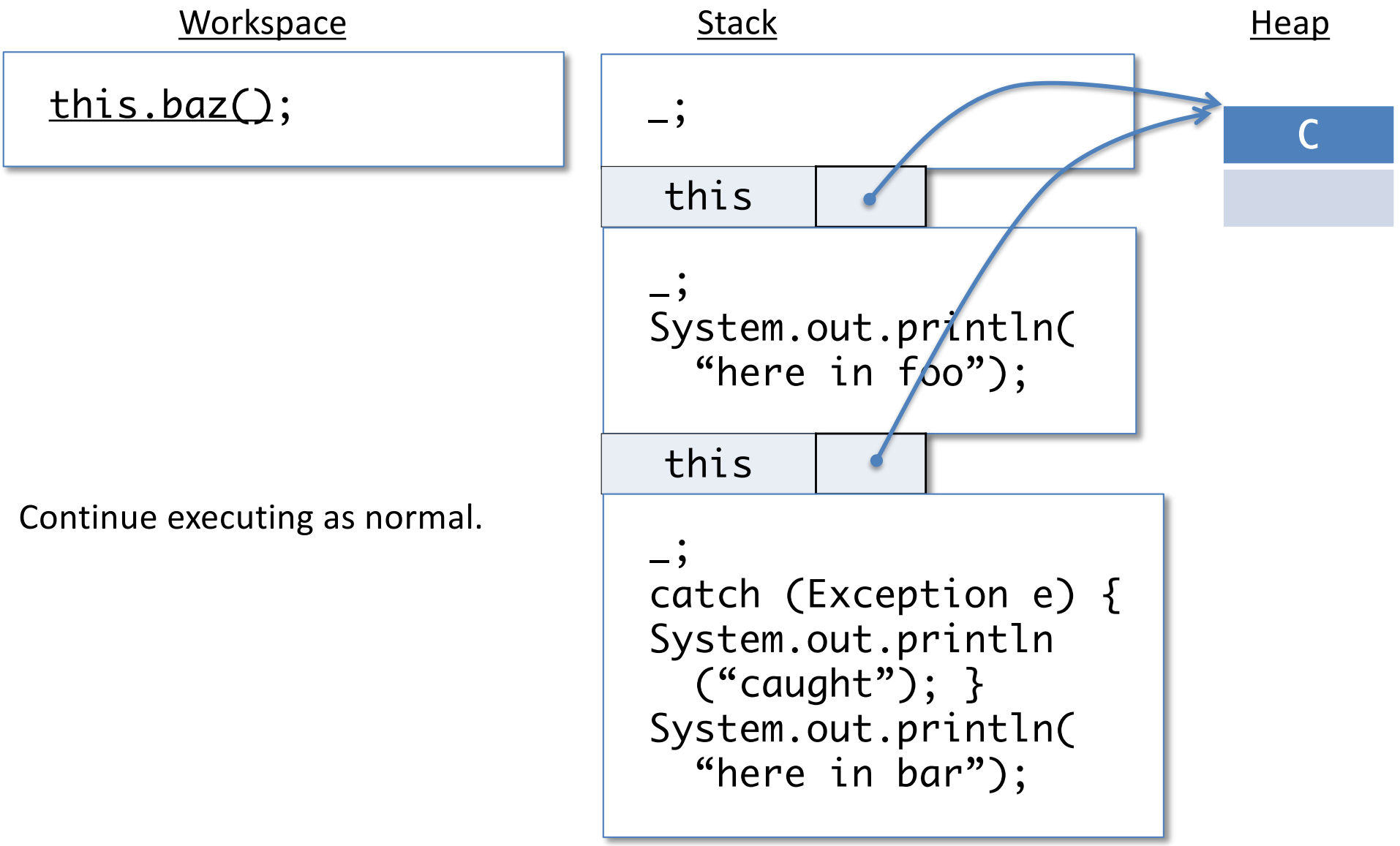
Abstract Stack Machine



When executing a try/catch block, push onto the stack a new workspace that contains *all* of the current workspace except for the try { … } code.

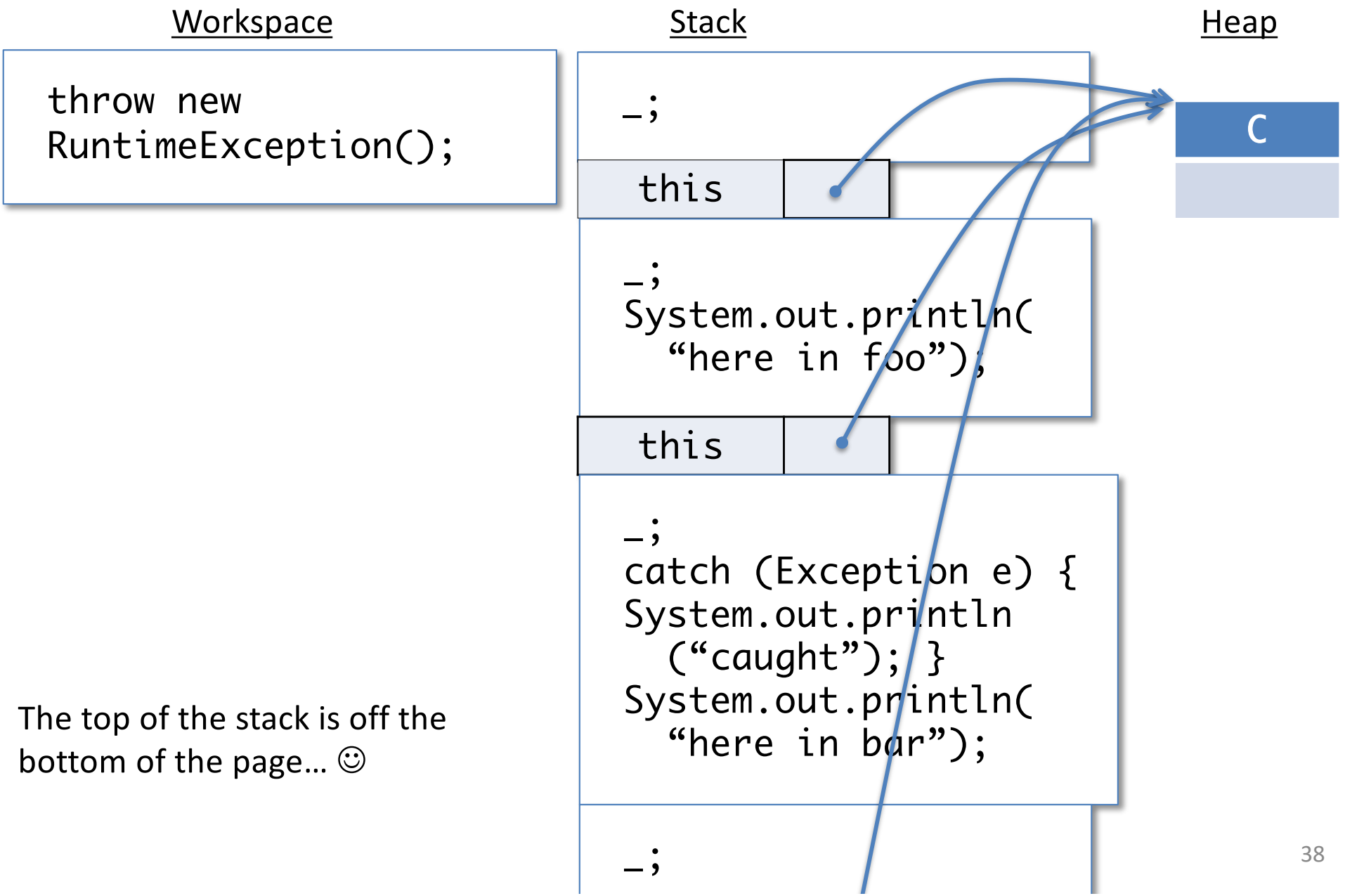
Replace the current workspace with the body of the try.

Abstract Stack Machine



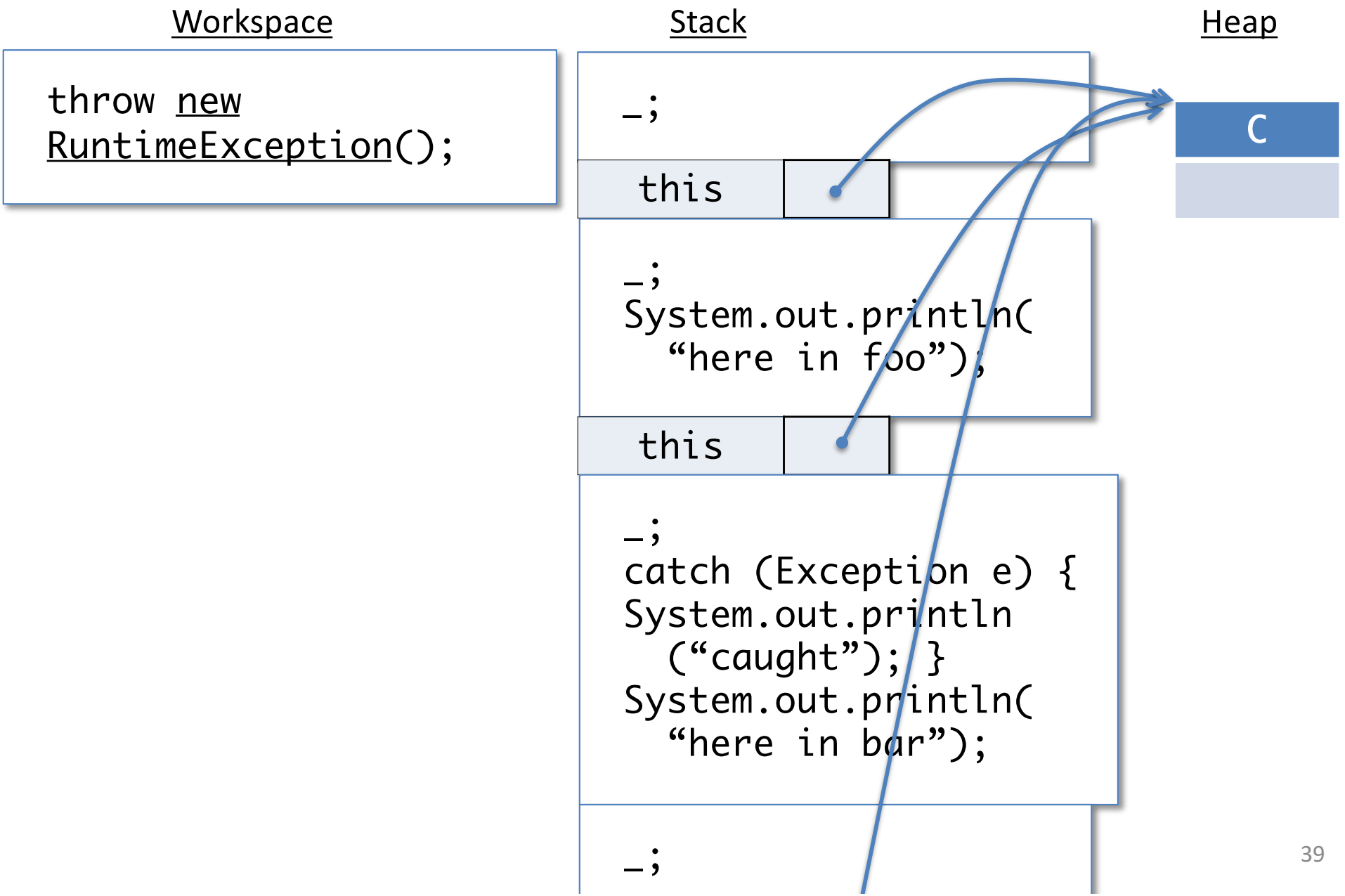
Continue executing as normal.

Abstract Stack Machine

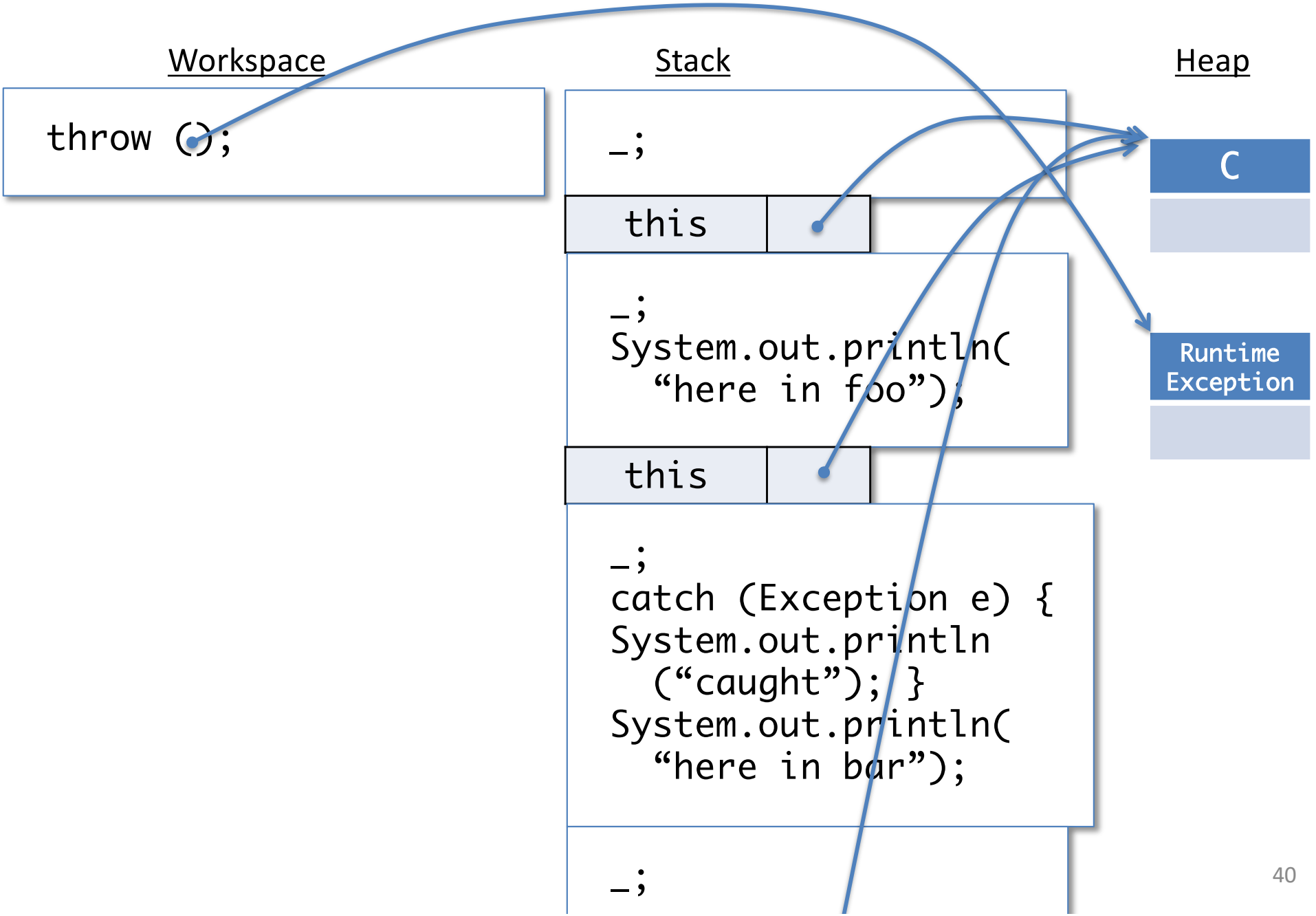


The top of the stack is off the bottom of the page… ☺

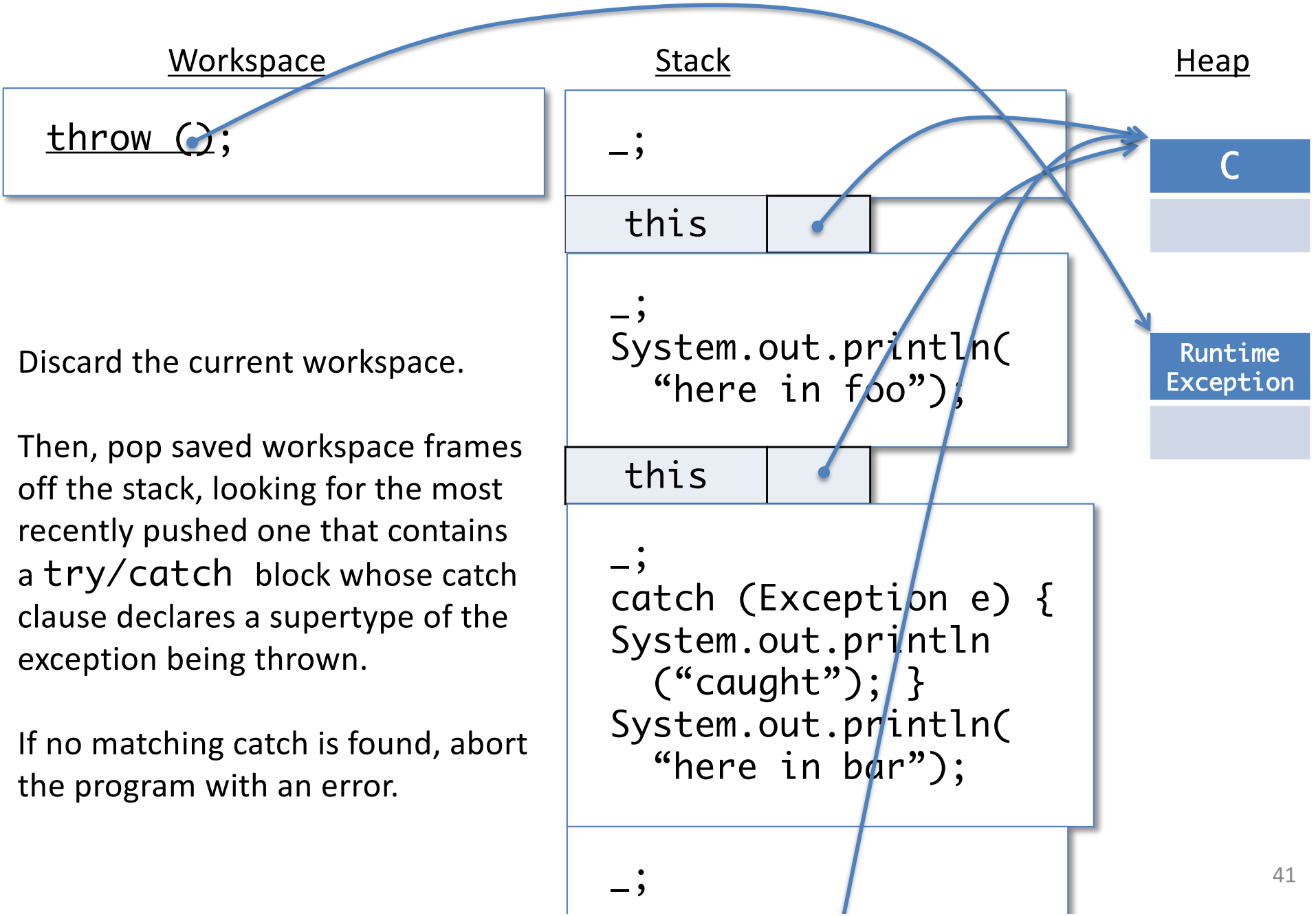
Abstract Stack Machine



Abstract Stack Machine



Abstract Stack Machine



Discard the current workspace.

Then, pop saved workspace frames off the stack, looking for the most recently pushed one that contains a `try/catch` block whose catch clause declares a supertype of the exception being thrown.

If no matching catch is found, abort the program with an error.

Abstract Stack Machine

Workspace

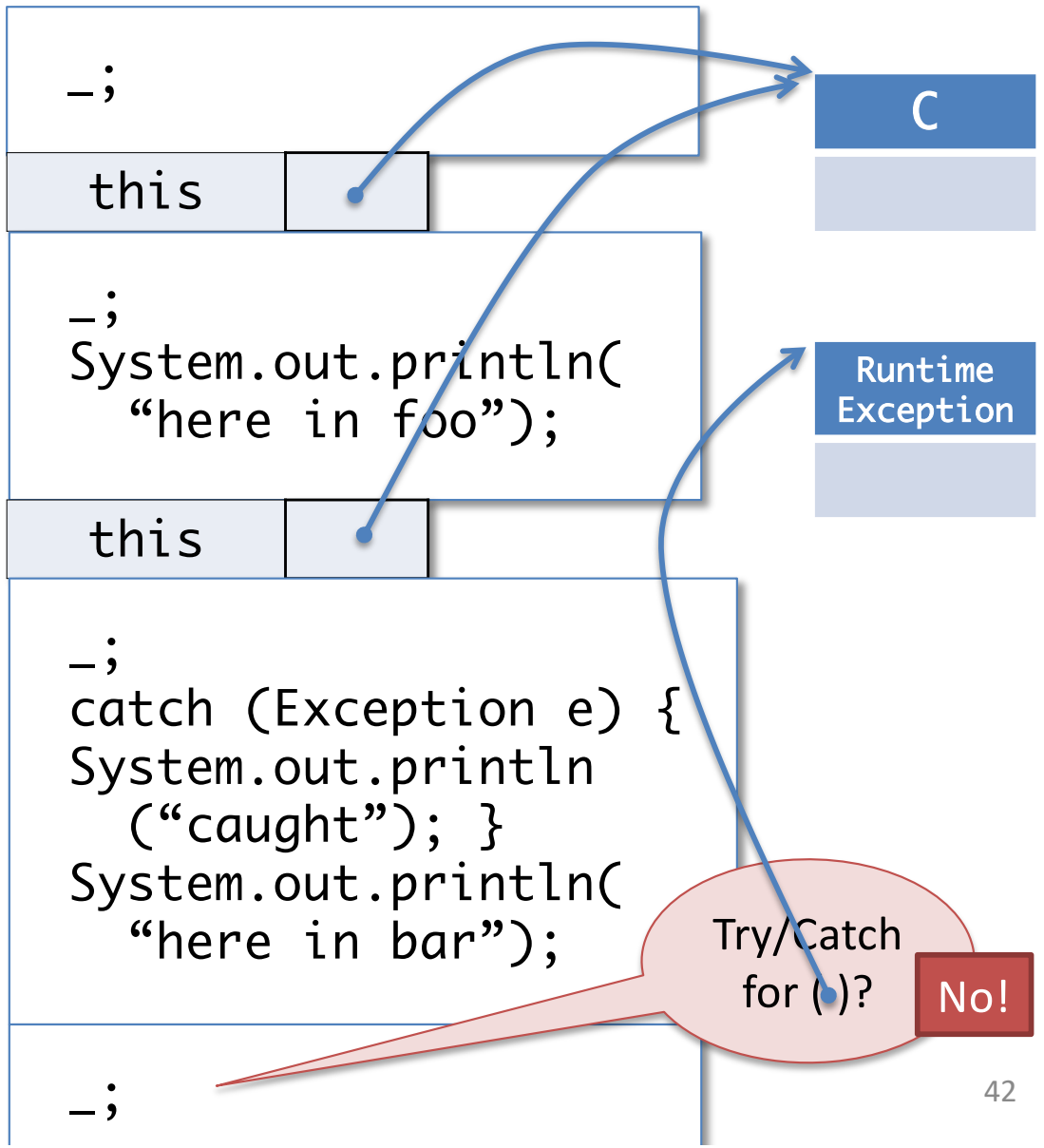
Stack

Heap

Discard the current workspace.

Then, pop saved workspace frames off the stack, looking for the most recently pushed one that contains a `try/catch` block whose catch clause declares a supertype of the exception being thrown.

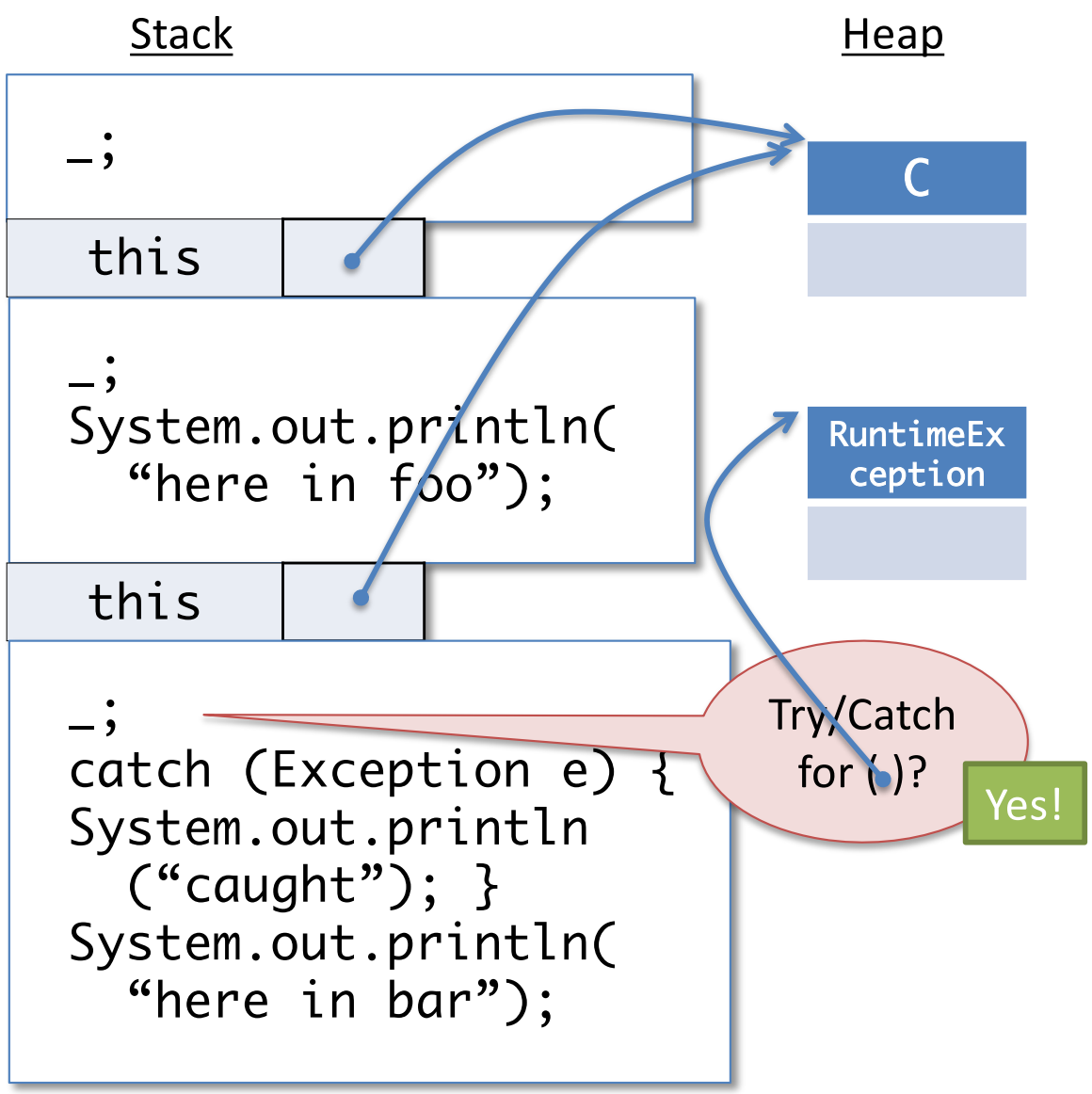
If no matching catch is found, abort the program with an error.



Abstract Stack Machine

When a matching catch block is found, add a new binding to the stack for the exception variable declared in the catch. Then replace
the workspace with catch body and the rest of the saved workspace.

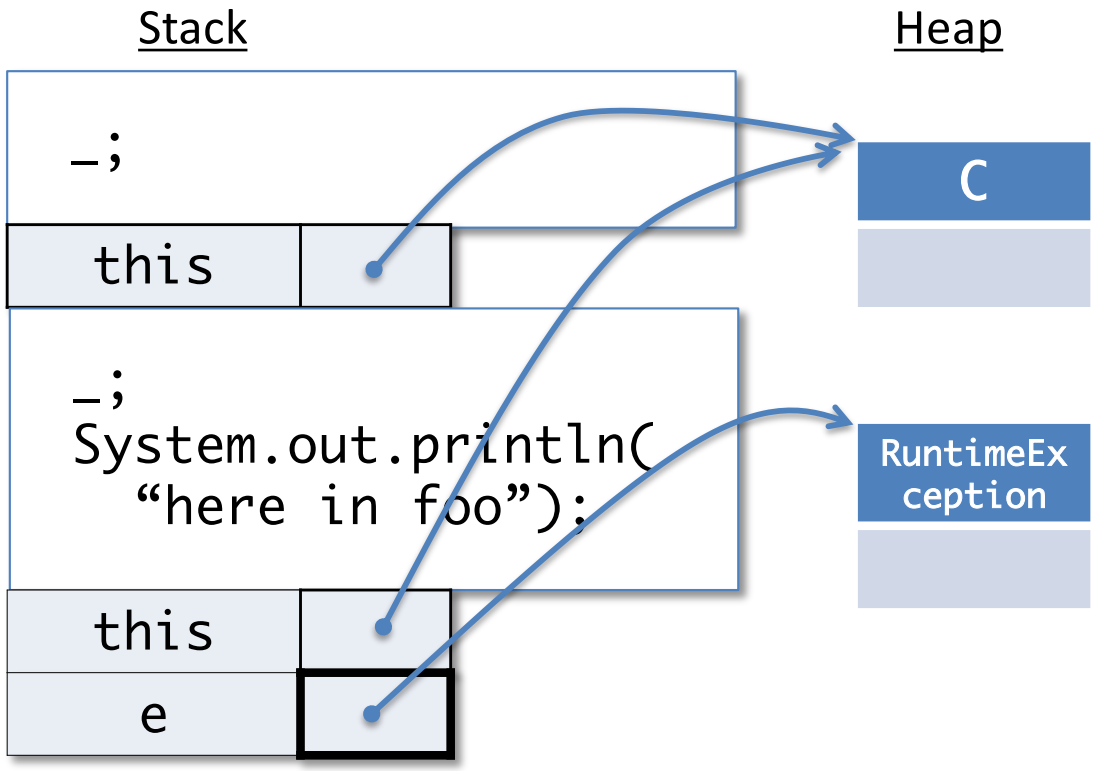
Continue executing as usual.



Abstract Stack Machine

Workspace

```
{ System.out.println
  (“caught”); }
System.out.println(
  “here in bar”);
```



When a matching catch block is found, add a new binding to the stack for the exception variable declared in the catch. Then replace
the workspace with catch body and the rest of the saved workspace.

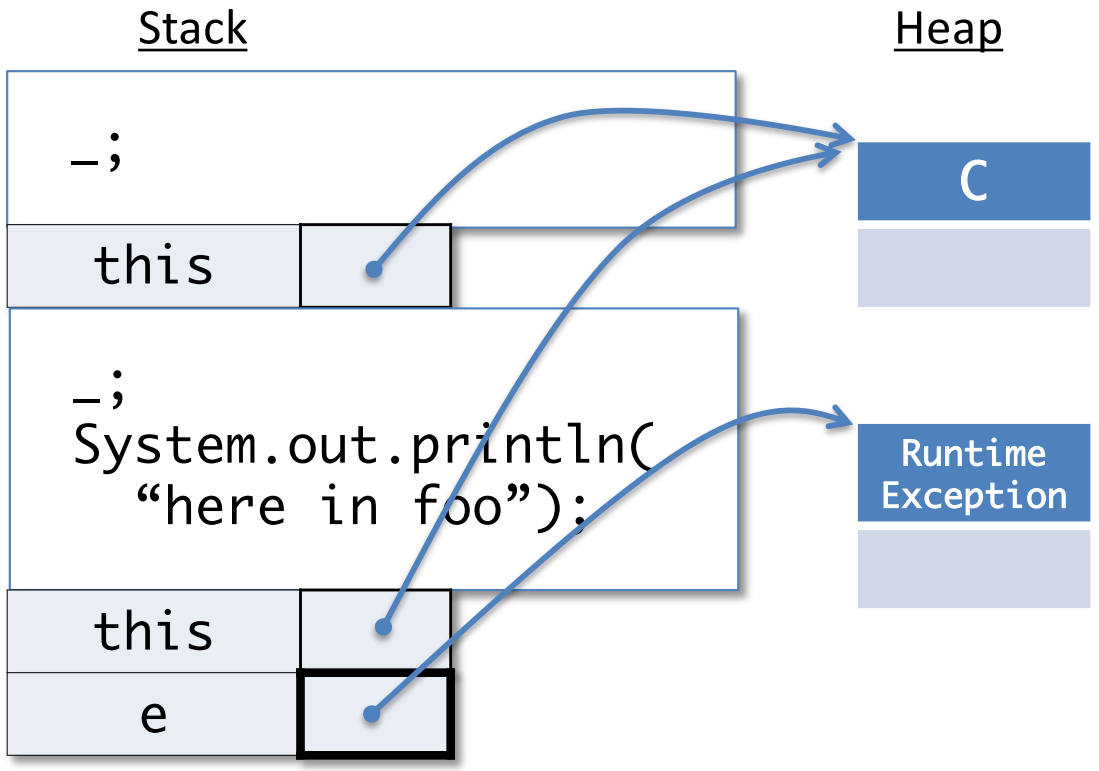
Continue executing as usual.

Abstract Stack Machine

Workspace

```
{ System.out.println
  (“caught”); }
System.out.println(
  “here in bar”);
```

Continue executing as usual.

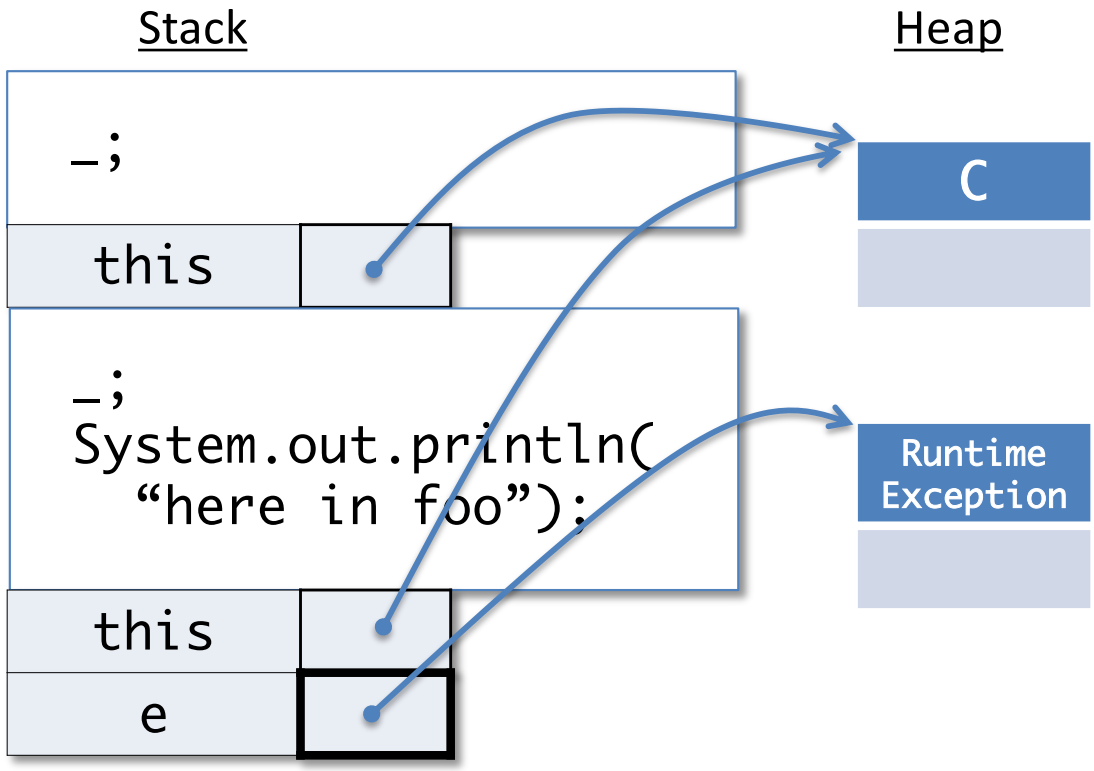


Abstract Stack Machine

Workspace

```
{ ; }
System.out.println(
  “here in bar”);
```

Continue executing as usual.



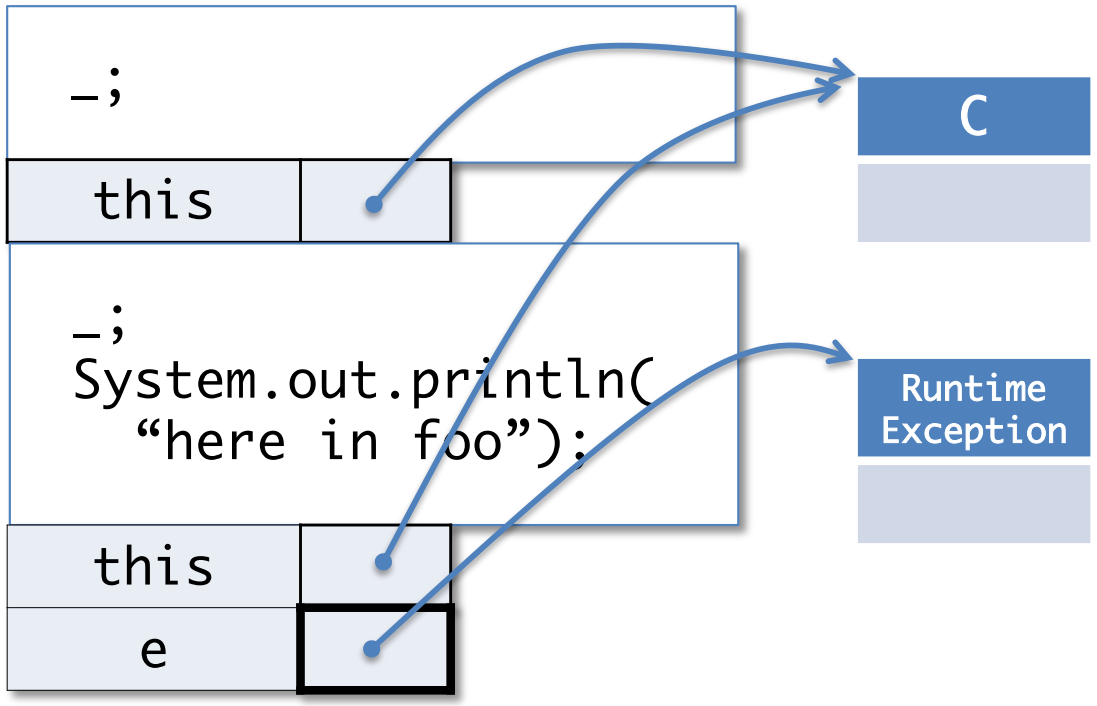
Console

caught

Abstract Stack Machine

Workspace

```
{ ; }
System.out.println(
  “here in bar”);
```



We’re sweeping a few details about lexical scoping of variables under the rug – the scope of e is just the body of the catch, so when that is done, e must be popped from the stack.

Console

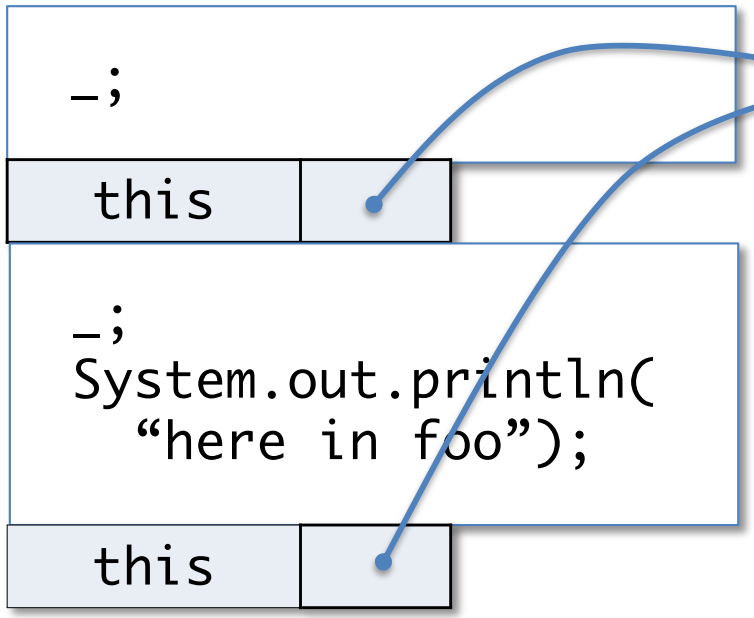
```
caught
```

Abstract Stack Machine

Workspace

```
System.out.println(
  “here in bar”);
```

Stack



Heap

Continue executing as usual.

Console

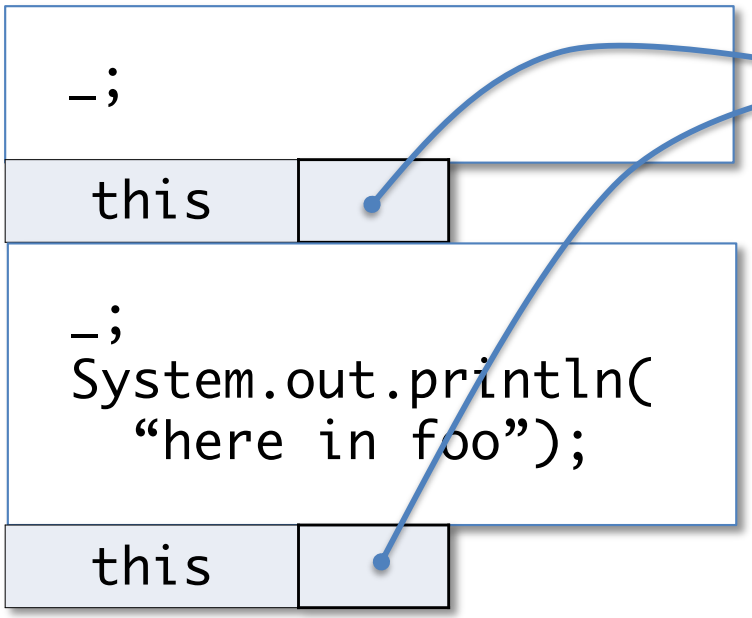
```
caught
```

Abstract Stack Machine

Workspace

```
System.out.println(
  "here in bar");
```

Stack



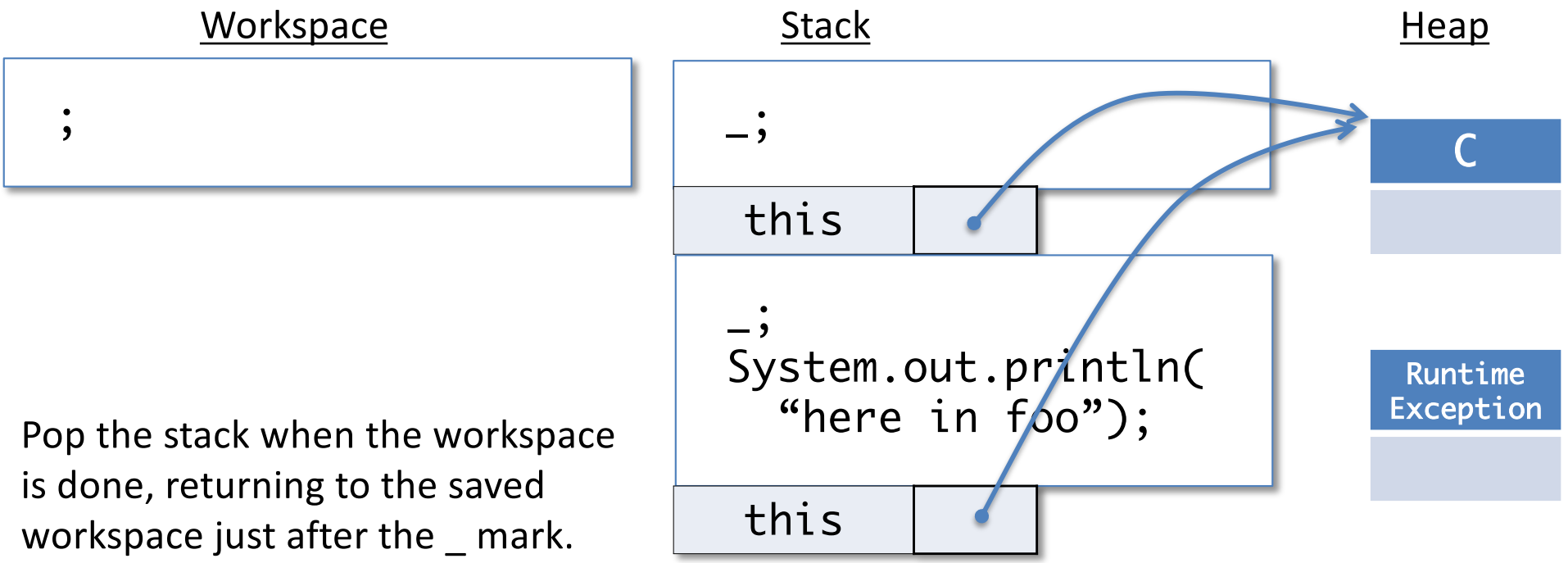
Heap

Continue executing as usual.

Console

```
caught
```

Abstract Stack Machine



Pop the stack when the workspace is done, returning to the saved workspace just after the _ mark.

Console

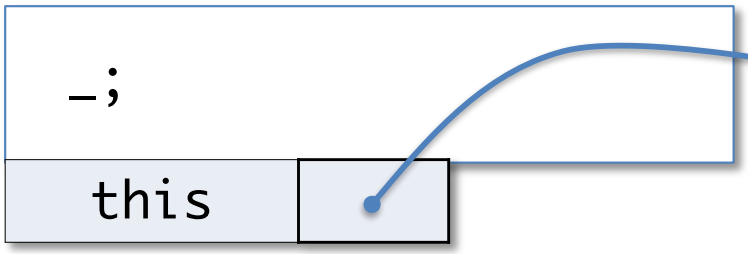
```
caught
here in bar
```

Abstract Stack Machine

Workspace

```
System.out.println(
  “here in foo”);
```

Stack



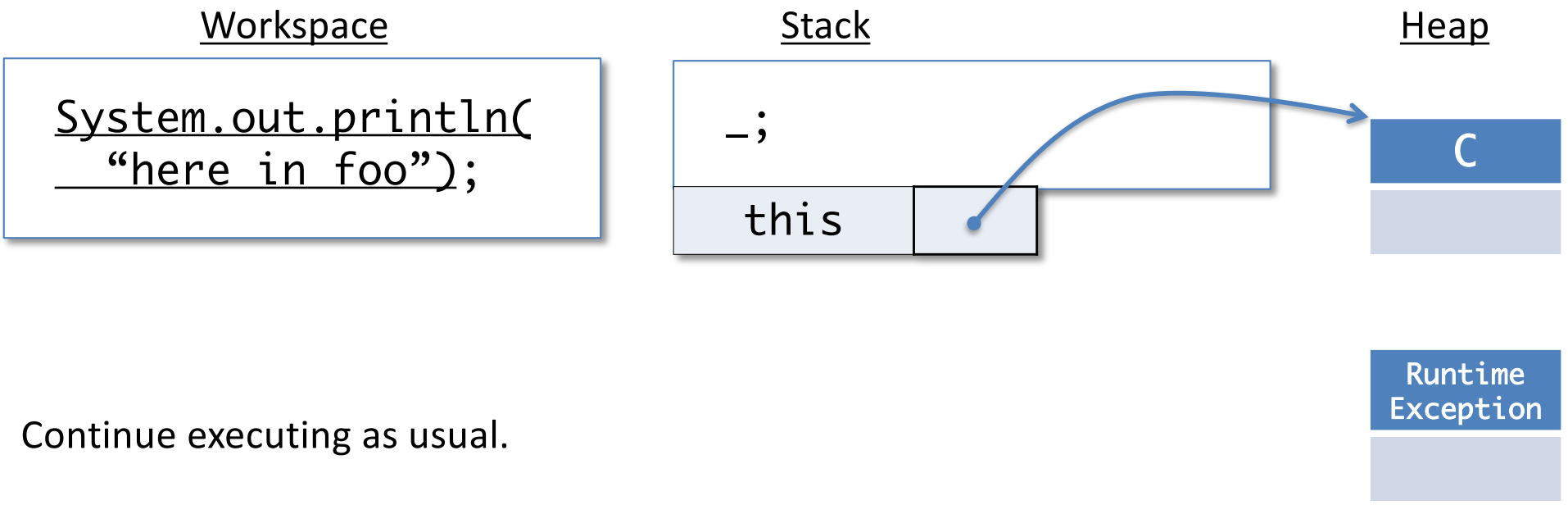
Heap

Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine

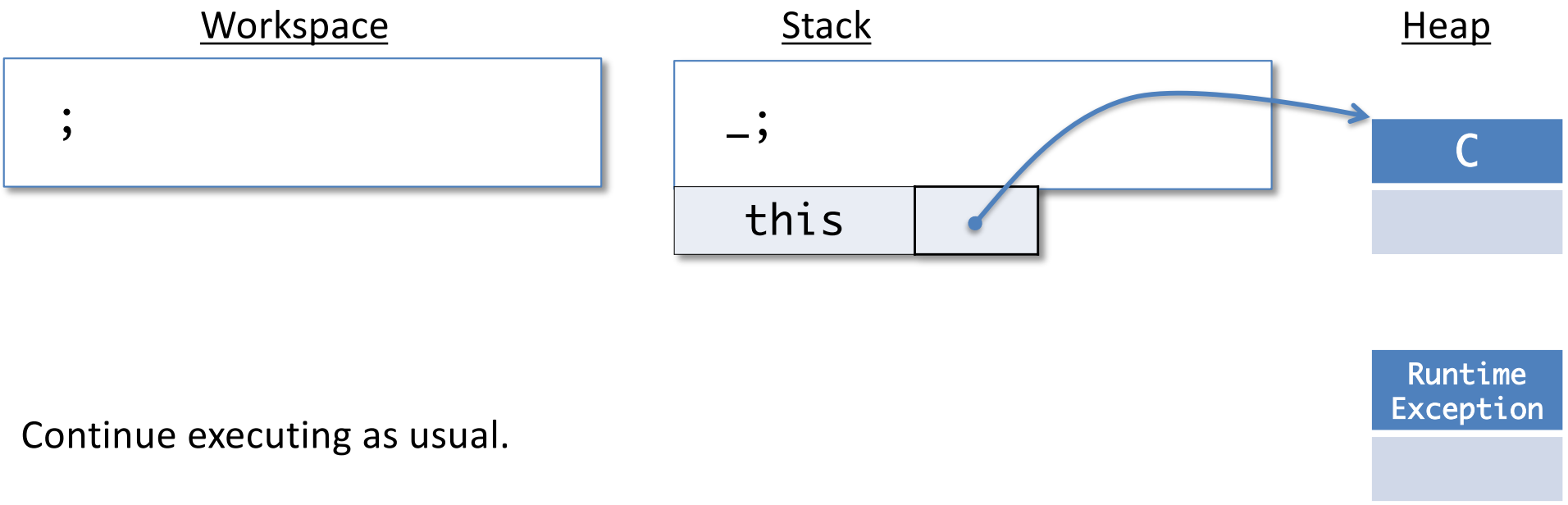


Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine



Continue executing as usual.

Console

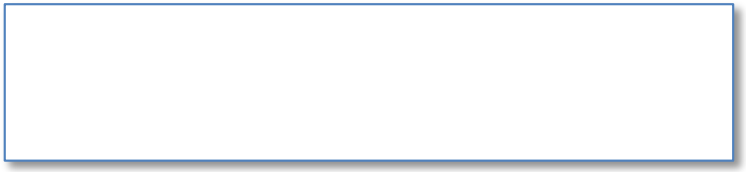
```
caught
here in bar
here in foo
```

Abstract Stack Machine

Workspace

Stack

Heap



Program terminated normally.

Console

```
caught
here in bar
here in foo
```

When No Exception is Thrown

- If no exception is thrown while executing the body of a try {…} block, evaluation *skips* the corresponding catch block.
  - i.e. if you ever reach a workspace where "catch" is the statement to run, just skip it:

Workspace

```
catch
(RuntimeException e)
{ System.out.Println
  ("caught"); }
System.out.println(
  "here in bar");
```

Workspace

```
System.out.println(
  "here in bar");
```

Catching Exceptions

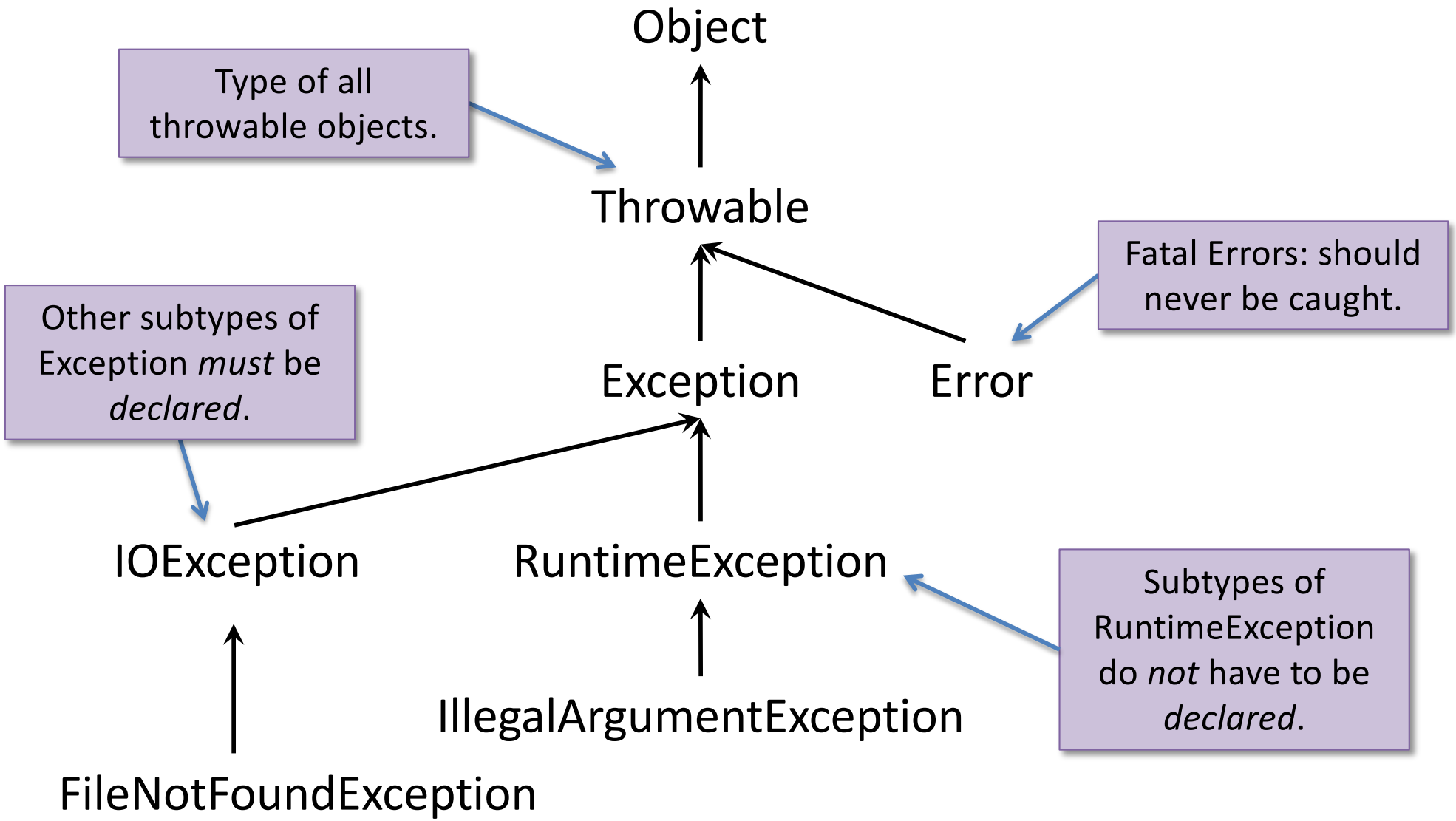
- There can be more than one "catch" clause associated with each "try"
  - Matched in order, according to the *dynamic* class of the exception thrown
  - Helps refine error handling

```
try {
        …    // do something with the IO library
} catch (FileNotFoundException e) {
        …    // handle an absent file
} catch (IOException e) {
        …    // handle other kinds of IO errors.
}
```

- Good style: be as specific as possible about the exceptions you're handling.
  - Avoid `catch (Exception e) {…}` it's usually too generic!

Informative Exception Handling

Exception Class Hierarchy



Checked (Declared) Exceptions

- Exceptions that are subtypes of `Exception` but not `RuntimeException` are called *checked* or *declared*.
- A method that might throw a checked exception must declare it using a "throws" clause in the method type.
- The method might raise a checked exception either by:
  - directly throwing such an exception

```
public void maybeDoIt (String file) throws AnException {
        if (…) throw new AnException();  // directly throw
        …
```

  - or by calling another method that might itself throw a checked exception

```
public void doSomeIO (String file) throws IOException {
        Reader r = new FileReader(file);  // might throw
        …
```

Unchecked (Undeclared) Exceptions

- Subclasses of RuntimeException *do not* need to be declared via "throws"
  - even if the method does not explicitly handle them.
- Many "pervasive" types of errors cause RuntimeExceptions
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException

```
public void mightFail (String file) {
        if (file.equals("dictionary.txt")) {
            // file could be null!
        …
```

- The original intent was that such exceptions represent disastrous conditions from which it was impossible to sensibly recover…

Declared vs. Undeclared?

- Tradeoffs in the software design process:
- *Declared*: better documentation
  - forces callers to acknowledge that the exception exists
- *Undeclared*: fewer static guarantees (compiler can help less)
  - but, much easier to refactor code
- In practice: test-driven development encourages "fail early/fail often" model of code design and lots of code refactoring, so "undeclared" exceptions are prevalent.

- A reasonable compromise:
  - Use declared exceptions for libraries, where the documentation and usage enforcement are critical
  - Use undeclared exceptions in client code to facilitate more flexible development

Checked vs. Unchecked Exceptions

Which methods need a "throws" clause?

Note: IllegalArgumentException is a subtype of RuntimeException.

IOException is not.

1) all of them
2) none of them
3) m and n
4) n only
5) n, r, and s
6) n, q, and s
7) m, p, and s
8) something else

Answer:
n, q and s should say throws IOException

```
public class ExceptionQuiz {
    public void m(Object x) {
      if (x == null)
          throw new IllegalArgumentException();
    }
    public void n(Object y) {
      if (y == null) throw new IOException();
    }
    public void p() {
      m(null);
    }
    public void q() {
      n(null);
    }
    public void r() {
      try { n(null); } catch (IOException e) {}
    }
    public void s() {
      n(new Object());
    }
}
```

Finally

```
try {
    ...
} catch (Exn1 e1) {
   ...
} catch (Exn2 e2) {
   ...
} finally {
   ...
}
```

- A `finally` clause of a try/catch/finally statement *always* gets run, regardless of whether there is no exception, a propagated exception, or a caught exception.

Using Finally

- `Finally` is often used for releasing resources that might have been held/created by the `try` block:

```
public void doSomeIO (String file) {
  FileReader r = null;
  try {
    r = new FileReader(file);
    … // do some IO
  } catch (FileNotFoundException e) {
    … // handle the absent file
  } catch (IOException e) {
    … // handle other IO problems
  } finally {
    if (r != null) {    // don't forget null check!
      try { r.close(); } catch (IOException e) {…}
    }
  }
}
```

Using Finally

```
class C {
      public void foo() {
        this.bar();
        System.out.println("here in foo");
      }
      public void bar() {
        try {
          this.baz();
        } catch (Exception e) {
             System.out.println("caught");
        } finally { System.out.println("finally"); }
        System.out.println("here in bar");
      }
      public void baz() {
        throw new RuntimeException();
      }
}
```

What happens if we do `(new C()).foo()` ?

Answer: 4

1. Program prints only "finally"
2. Program prints "here in bar", then "here in foo", then "finally"
3. Program prints "finally", then "caught", then "here in foo"
4. Program prints "caught", then "finally", then "here in bar", then "here in foo"

Using Finally

```
class C {
      public void foo() {
        this.bar();
        System.out.println("here in foo");
      }
      public void bar() {
        try {
          this.baz();
        } catch (Exception e) {
             System.out.println("caught");
        } finally { System.out.println("finally"); }
        System.out.println("here in bar");
      }
      public void baz() {
        throw new RuntimeException();
      }
    }
```

Good Style for Exceptions

- In Java, exceptions should be used to capture *exceptional circumstances*
  - Try/catch/throw incur performance costs and complicate reasoning about the program, don't use them when better solutions exist

- *Re-use existing exception types* when they are meaningful to the situation
  - e.g. use NoSuchElementException when implementing a container

- Define your own subclasses of Exception if doing so can convey useful information to possible callers that can handle the exception.

Good Style for Exceptions

- It is often sensible to catch one exception and re-throw a different (more meaningful) kind of exception.
  - e.g. when implementing `WordScanner` (in upcoming lectures), we catch `IOException` and throw `NoSuchElementException` in the `next` method.

- Catch exceptions as near to the source of failure as makes sense
  - i.e. where you have the information to deal with the exception

- Catch exceptions with as much precision as you can

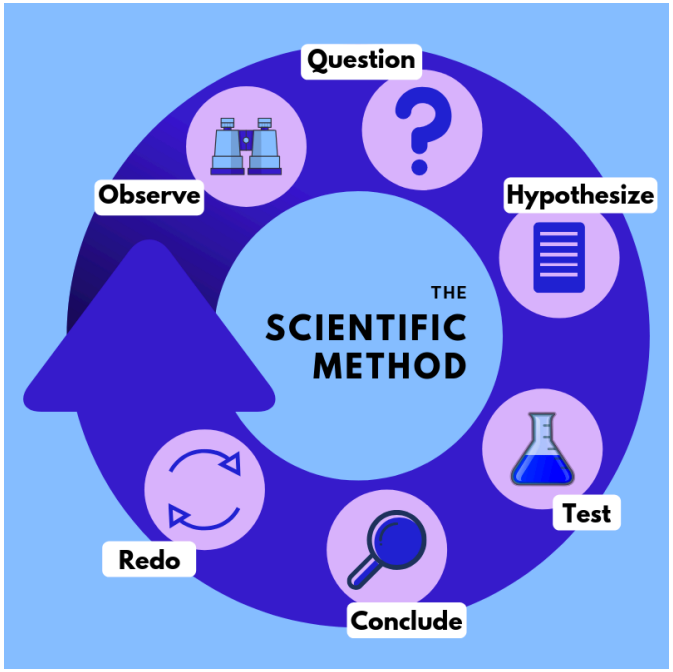
BAD: `try {…} catch (Exception e) {…}`

BETTER: `try {…} catch (IOException e) {…}`

Some Advice on Debugging

Use the Scientific Method

1. Make an observation / ask a question
   - One of my test cases fails!
   - Which assertion? What exception? What is the stack trace?
2. Formulate a hypothesis
   - Could I have passed null as bar to foo.munge(bar)?
3. Conduct an experiment
   - Modify the program to try to confirm or refute the hypothesis.
   - *Don't* make random changes!
   - Predict the outcome of your experiment
   - Re-run test cases, or execute the program
4. Analyze the results
   - Did the modified code behave as expected?
5. Draw conclusions / Report results
   - Create a new test case (if appropriate)



Observing Behavior

- Understand exceptions and their stack traces
  - They give you a lot of information
- If you are using Eclipse, it is worth taking a little time to learn how to use the debugger!
  - See Piazza for a Quick Start tutorial
- Simple print statements are also very effective!
  - Confirm or disprove hypothesis
  - e.g.: The code reached "HERE!" (or not)