# Programming Languages and Techniques (CIS120)

Lecture 29

Mar 28, 2012

## Exceptions II

# Announcements

- Midterm 2 is Friday, Mar 30th
  - Location is across campus: **FAGN AUD**
  - Review session: **TONIGHT** 8-10PM in Levine 101
  - Lab this week is review (bring questions!)

- HW09 will be available next Monday, Apr2.

# Exceptions

Dealing with the unexpected.

# Exceptions

- An exception is an *object* representing abnormal conditions.
  - 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 method.
  - The exception propagates up the invocation stack until it either reaches the top and 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

# Realistic Example

```
void loadImage (String fileName) {
  try {
      Picture p = new Picture(fileName);     // could fail

      // ... code to display the new picture in the window
      // executes only if the picture is successfully created.

   } catch (IOException ex) {

      // Use the GUI to send an error message to the user
      // using a dialog window
      JOptionPane.showMessageDialog(
          frame,                    // parent of dialog window
                                    // error message to display
          "Cannot load file\n" + ex.getMessage(),
          "Alert",                          // title of dialog
          JOptionPane.ERROR_MESSAGE    // type of dialog
         );
  }
```

# Simplfied 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 Exception();
    }
}
```

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

# Abstract Stack Machine

Workspace | Stack | Heap

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

# Abstract Stack Machine

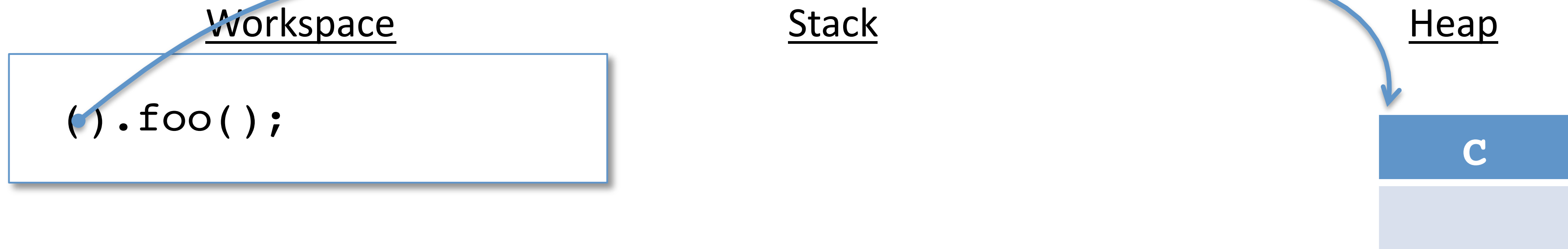
Workspace

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

Stack

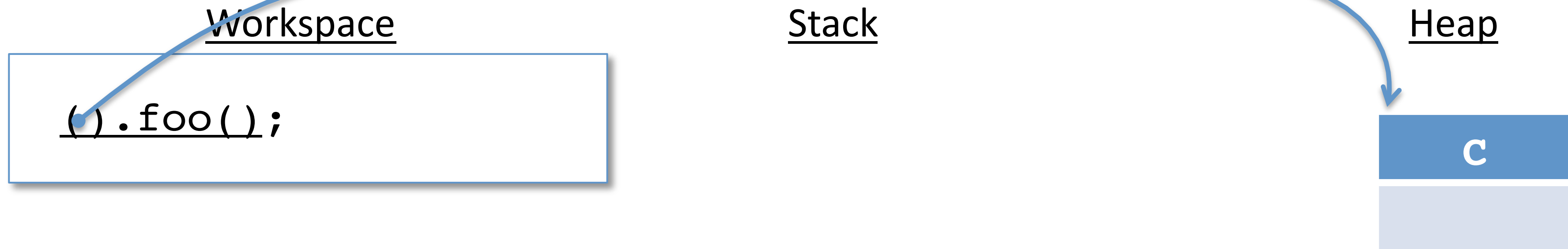
Heap

# Abstract Stack Machine

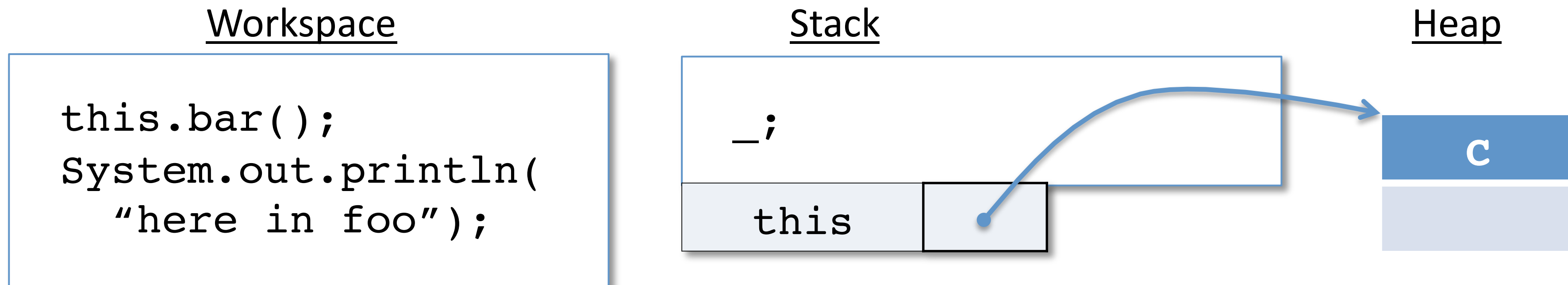



Allocate a new instance of C in the heap. (Skipping details of trivial Constructor.)

# Abstract Stack Machine



# Abstract Stack Machine



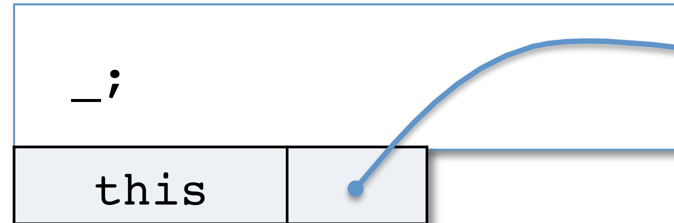

Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack. Use the dynamic class to lookup the method body from the class table.

# Abstract Stack Machine

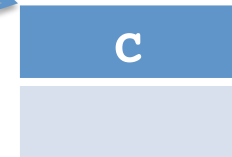
Workspace

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

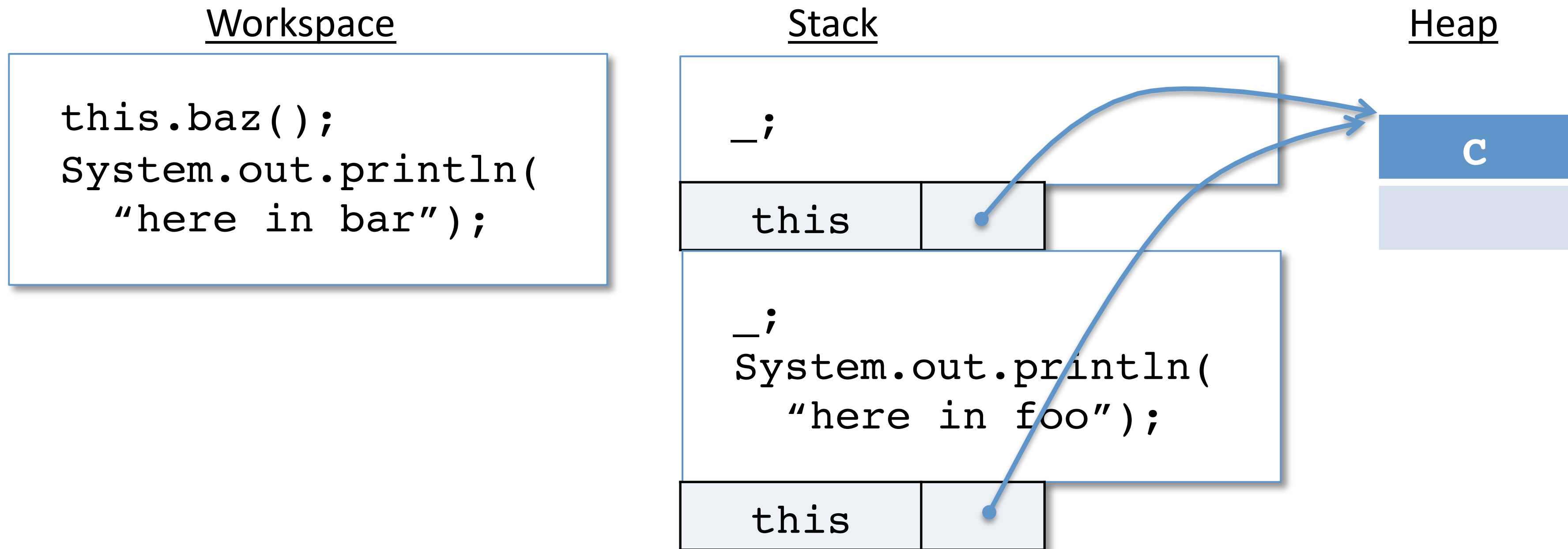
Stack



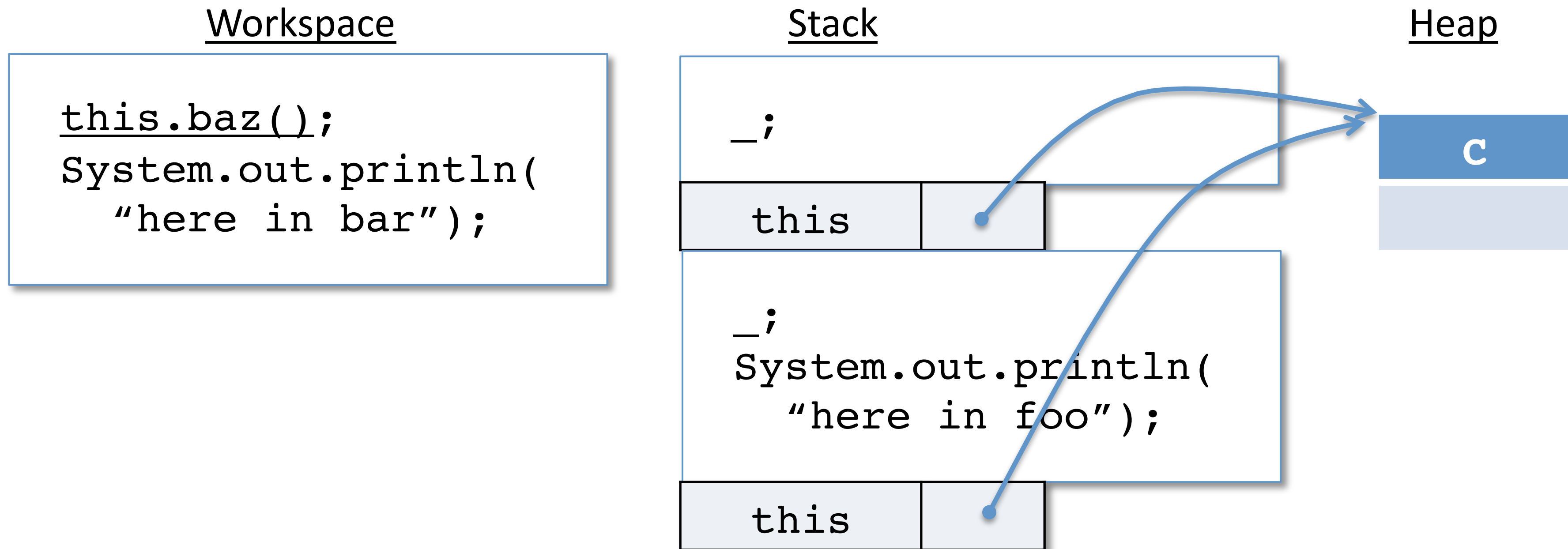

Heap



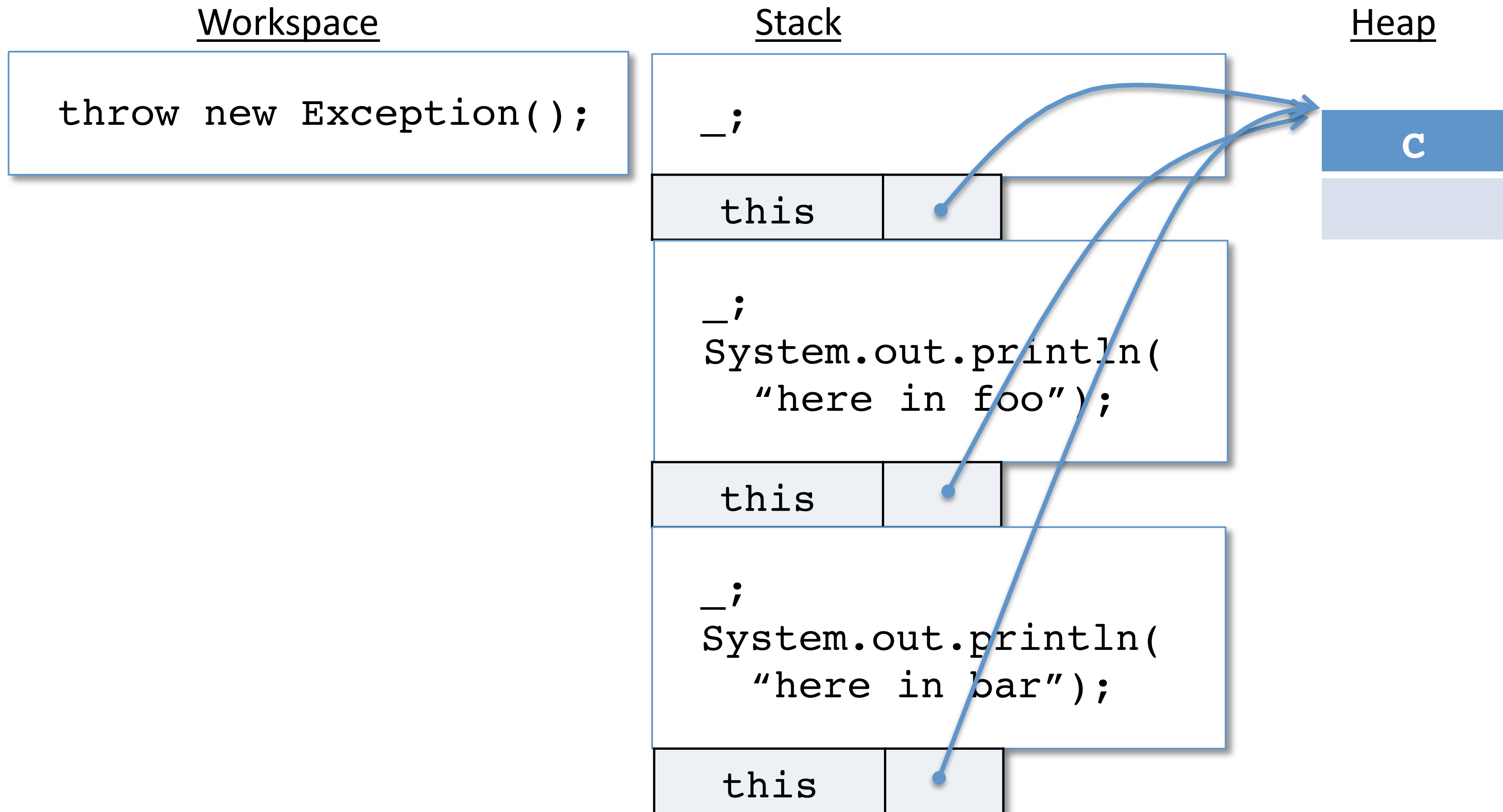
# Abstract Stack Machine



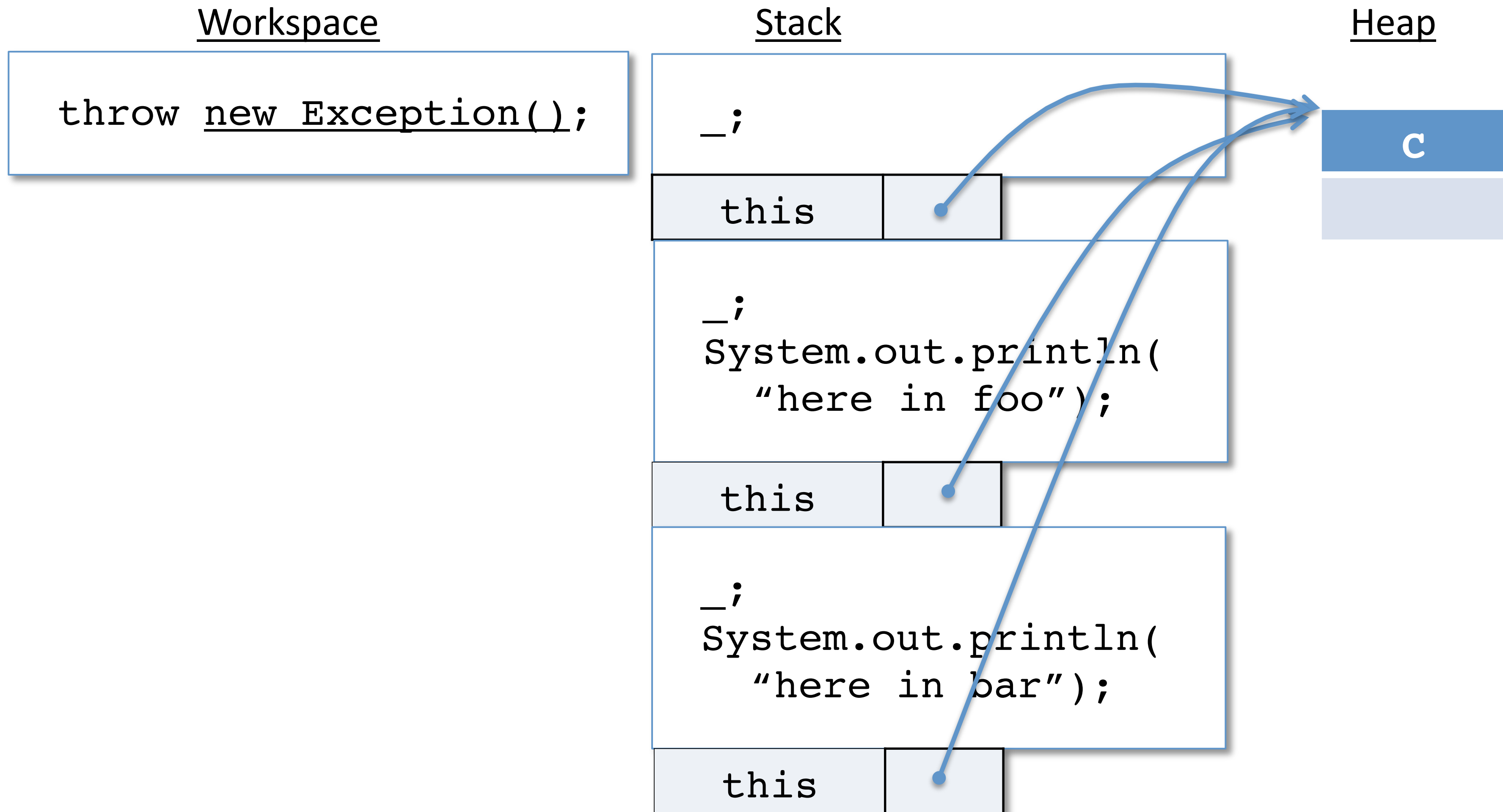
# Abstract Stack Machine



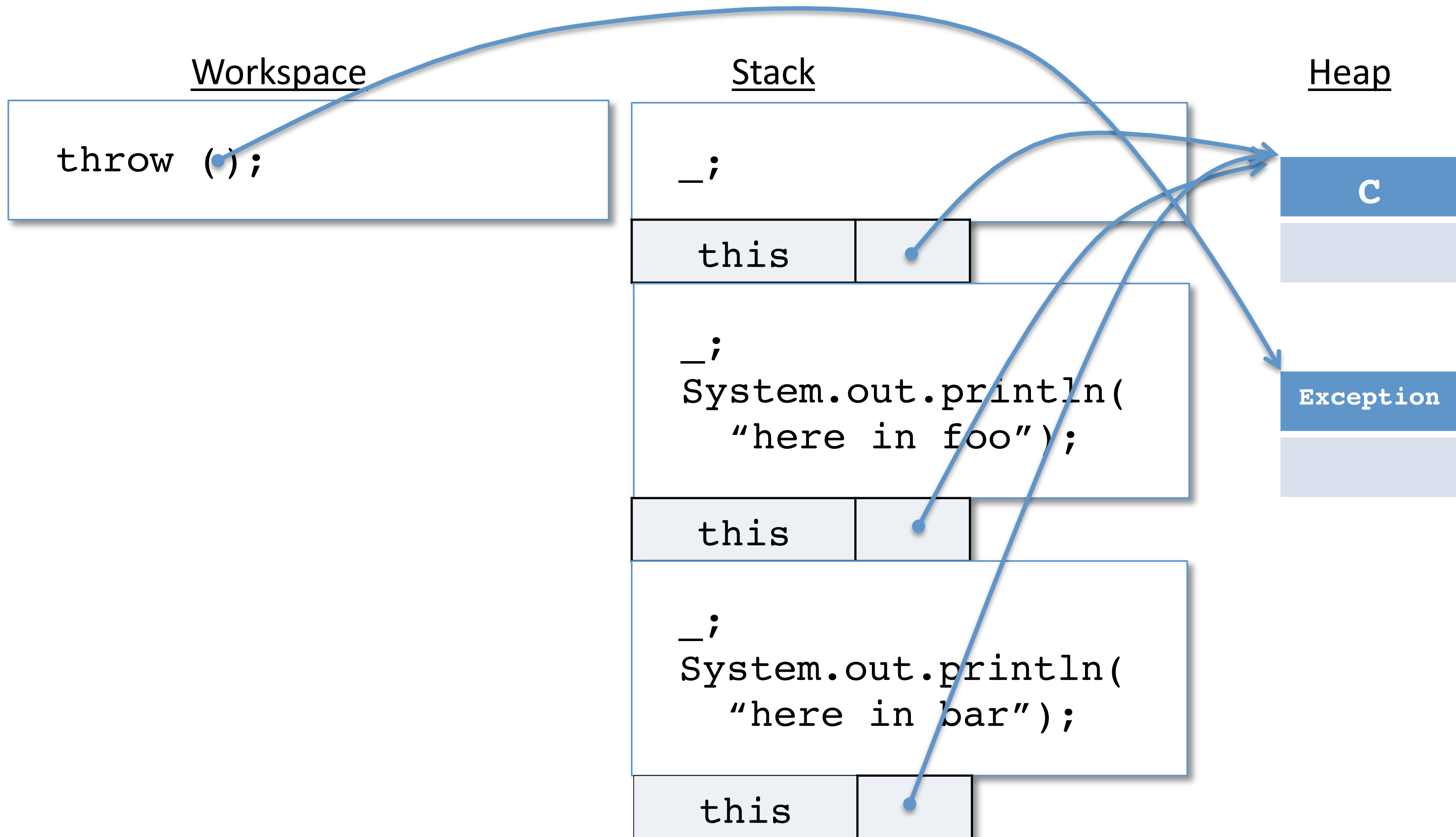
# Abstract Stack Machine



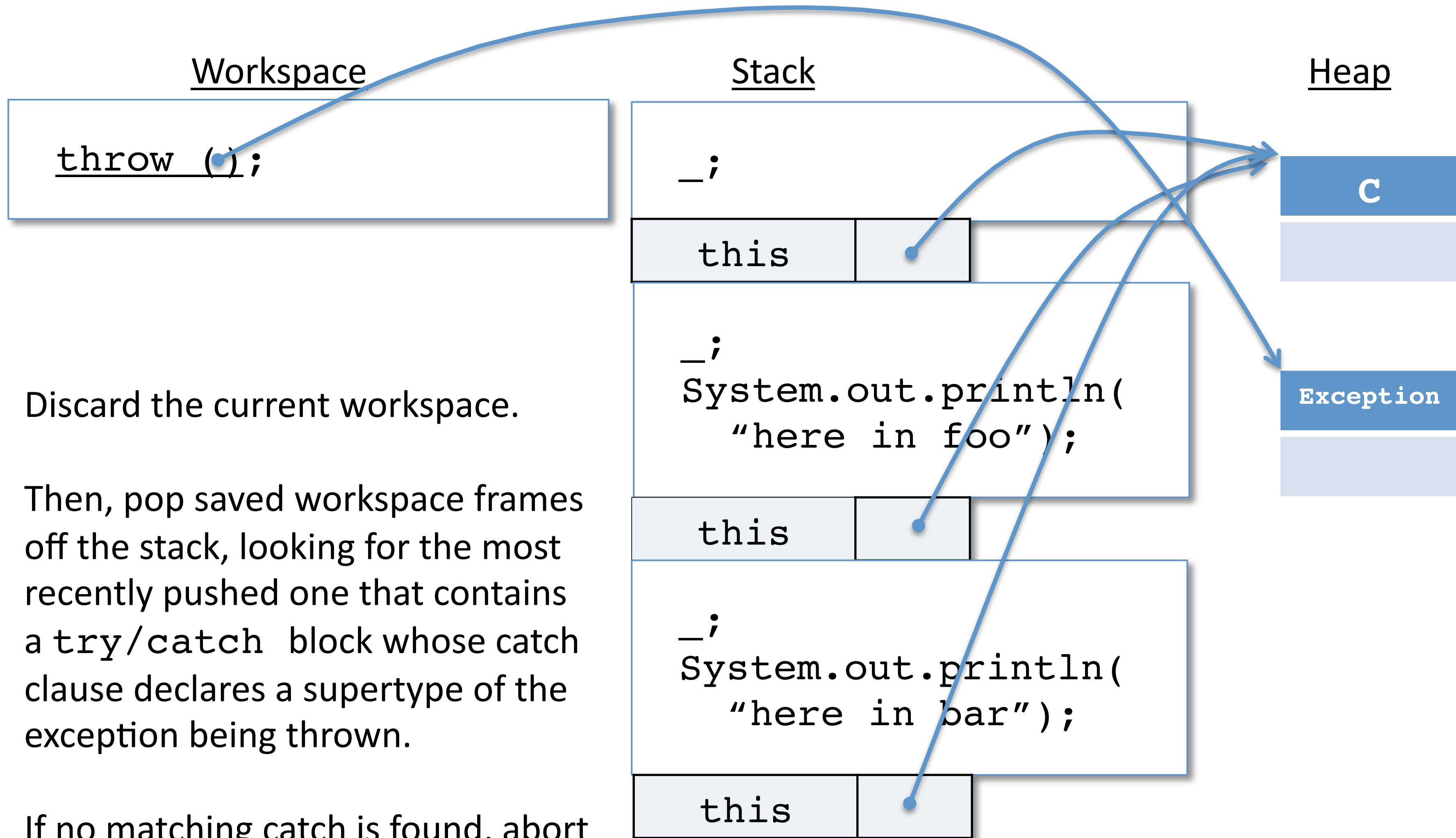
# Abstract Stack Machine



# Abstract Stack Machine



# Abstract Stack Machine

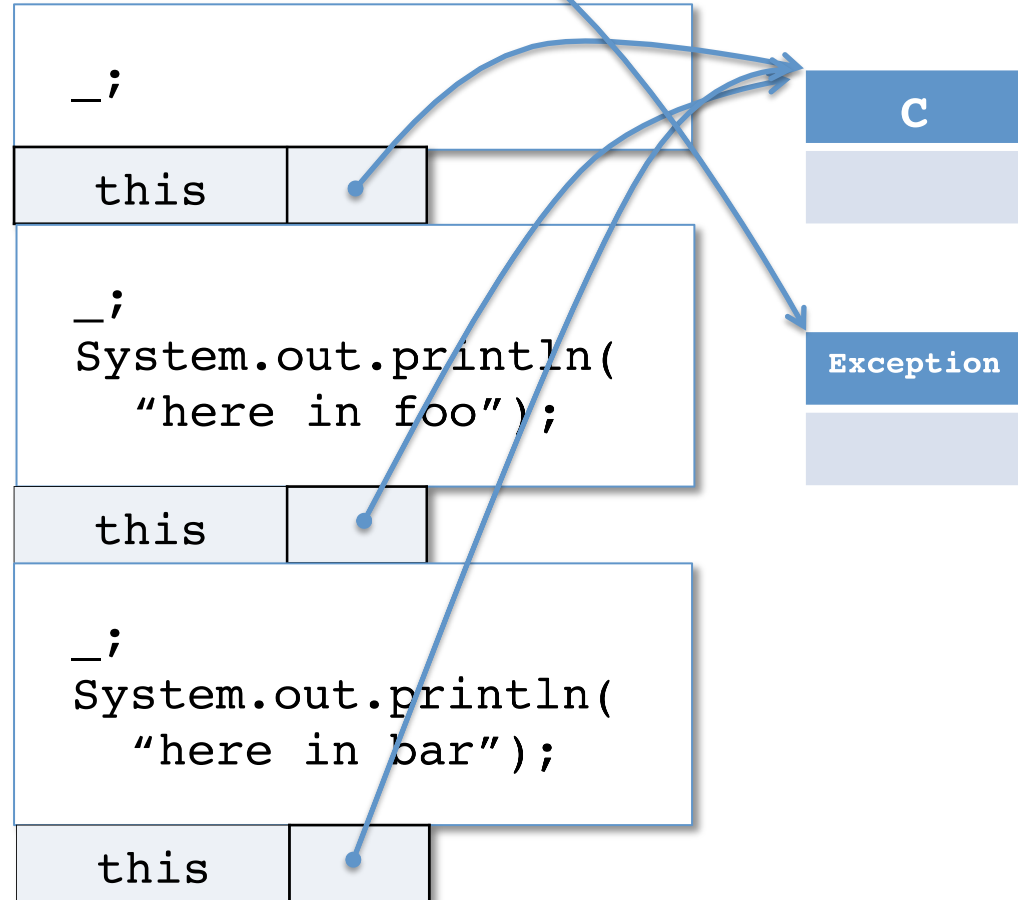



Discard the current workspace.

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




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.

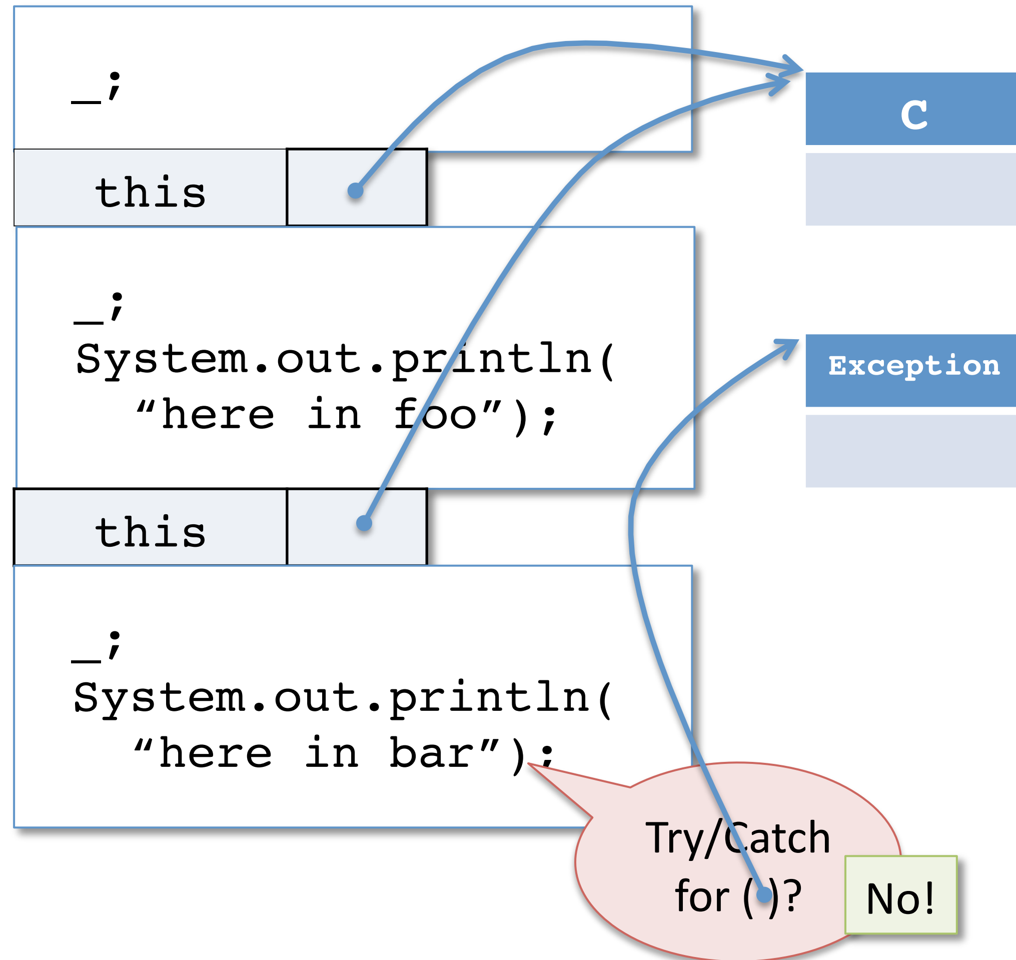
# Abstract Stack Machine

Workspace

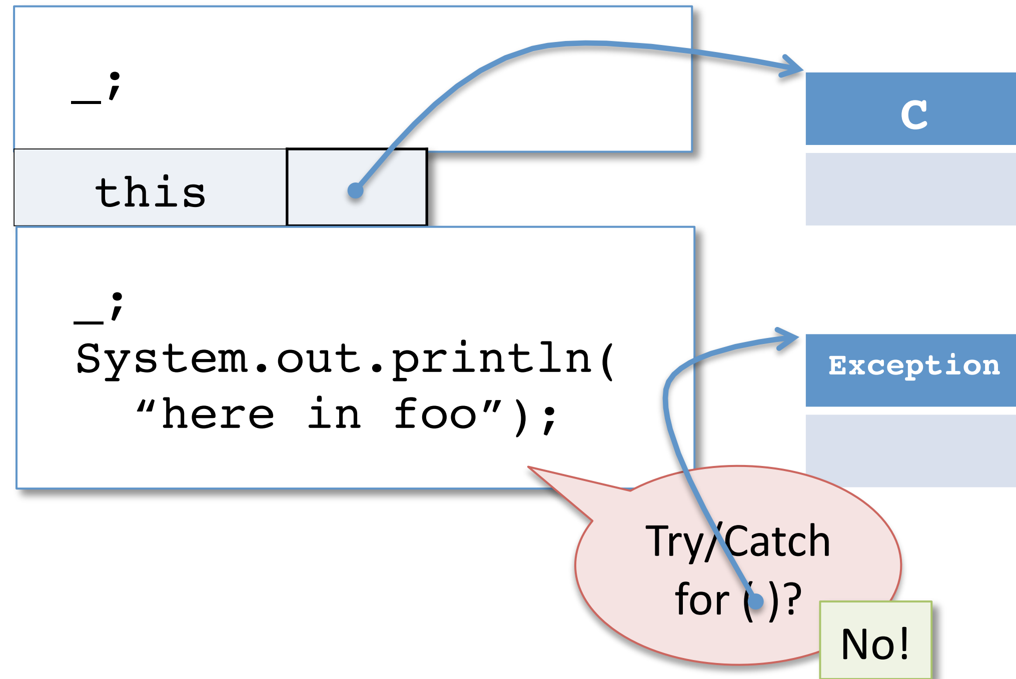
Discard the current workspace.

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

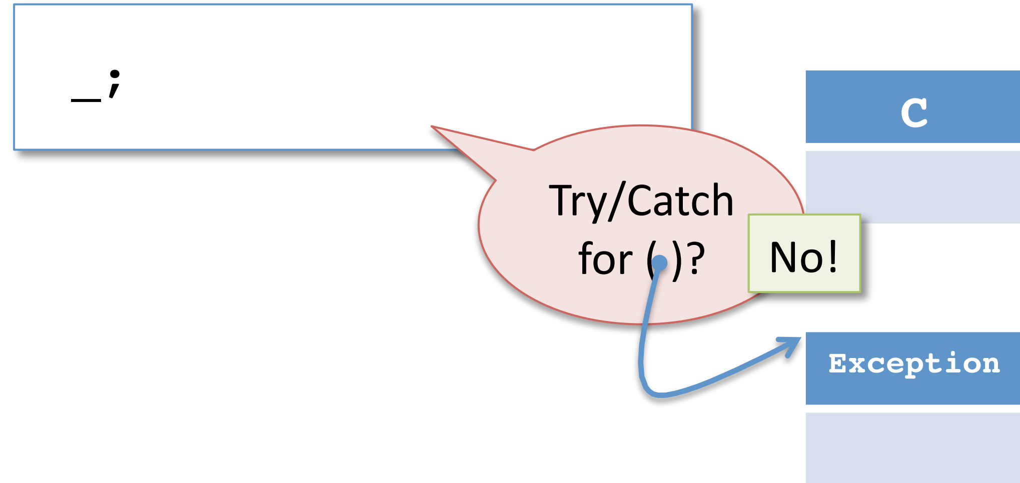



Discard the current workspace.

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




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 ( )!

C

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 Exception();
    }
}
```

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

# Abstract Stack Machine

Workspace

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

Stack

Heap

# Abstract Stack Machine

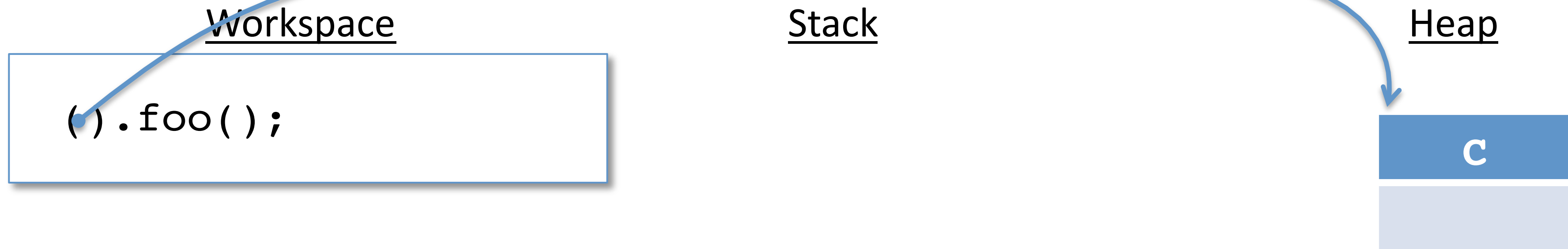
Workspace

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

Stack

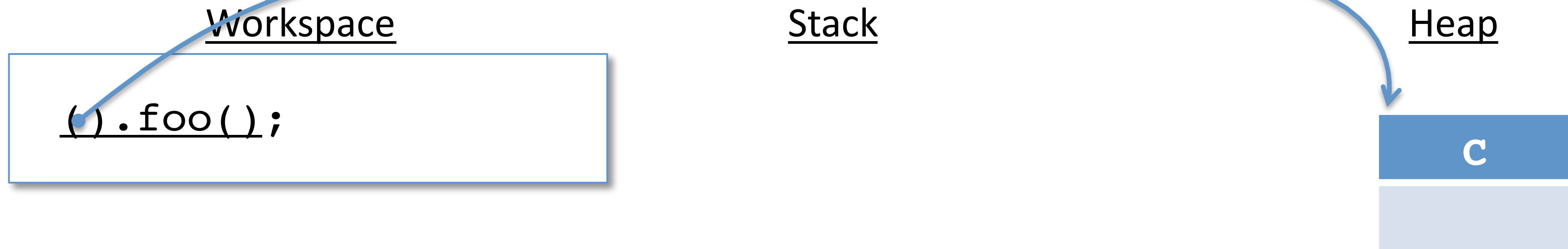
Heap

# Abstract Stack Machine

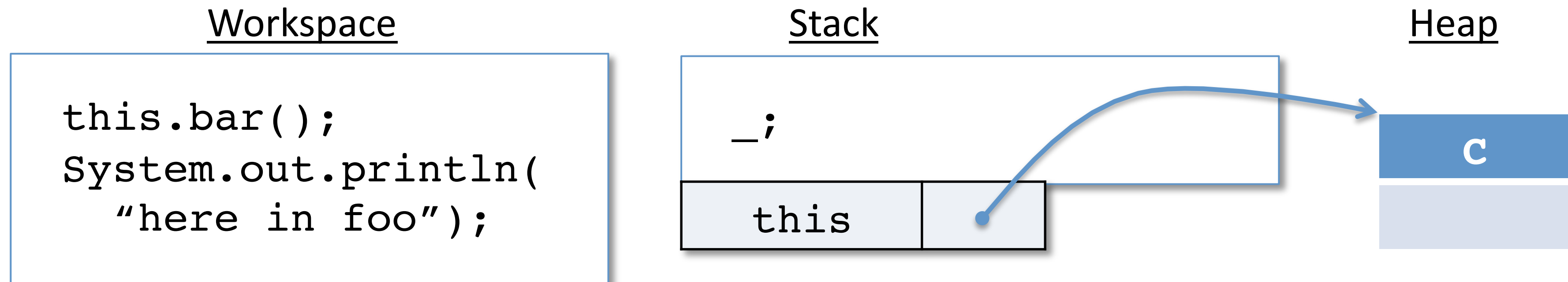



Allocate a new instance of C in the heap.

# Abstract Stack Machine



# Abstract Stack Machine



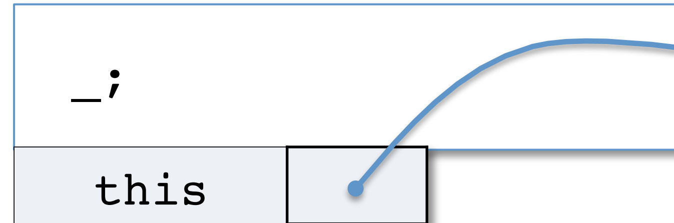

Save a copy of the current workspace in the stack, leaving a “hole”, written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack.

# Abstract Stack Machine

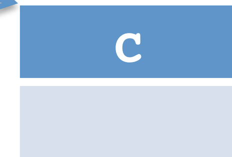
Workspace

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

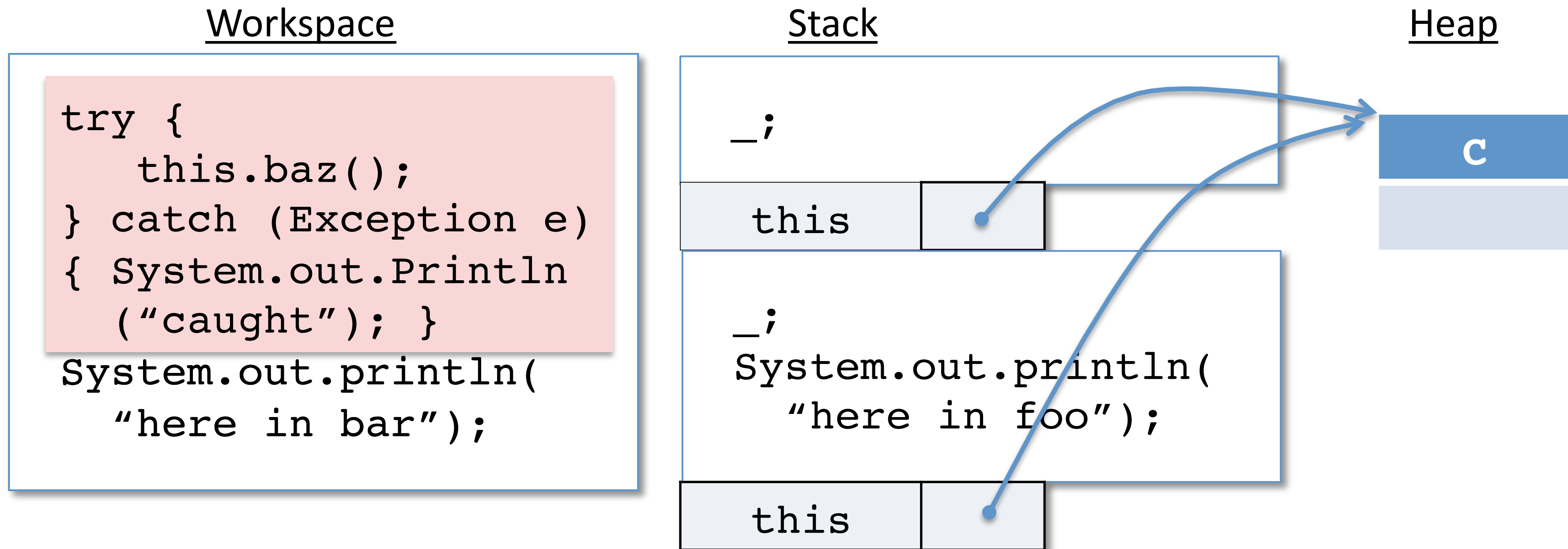
Stack



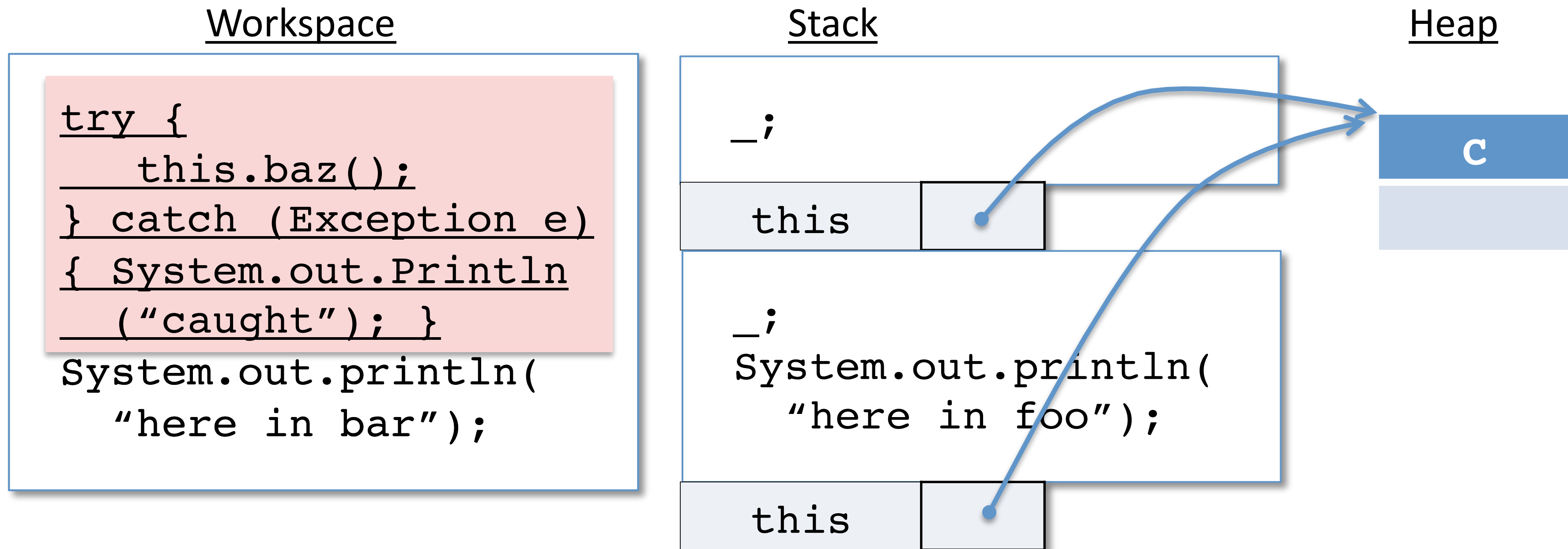

Heap



# Abstract Stack Machine



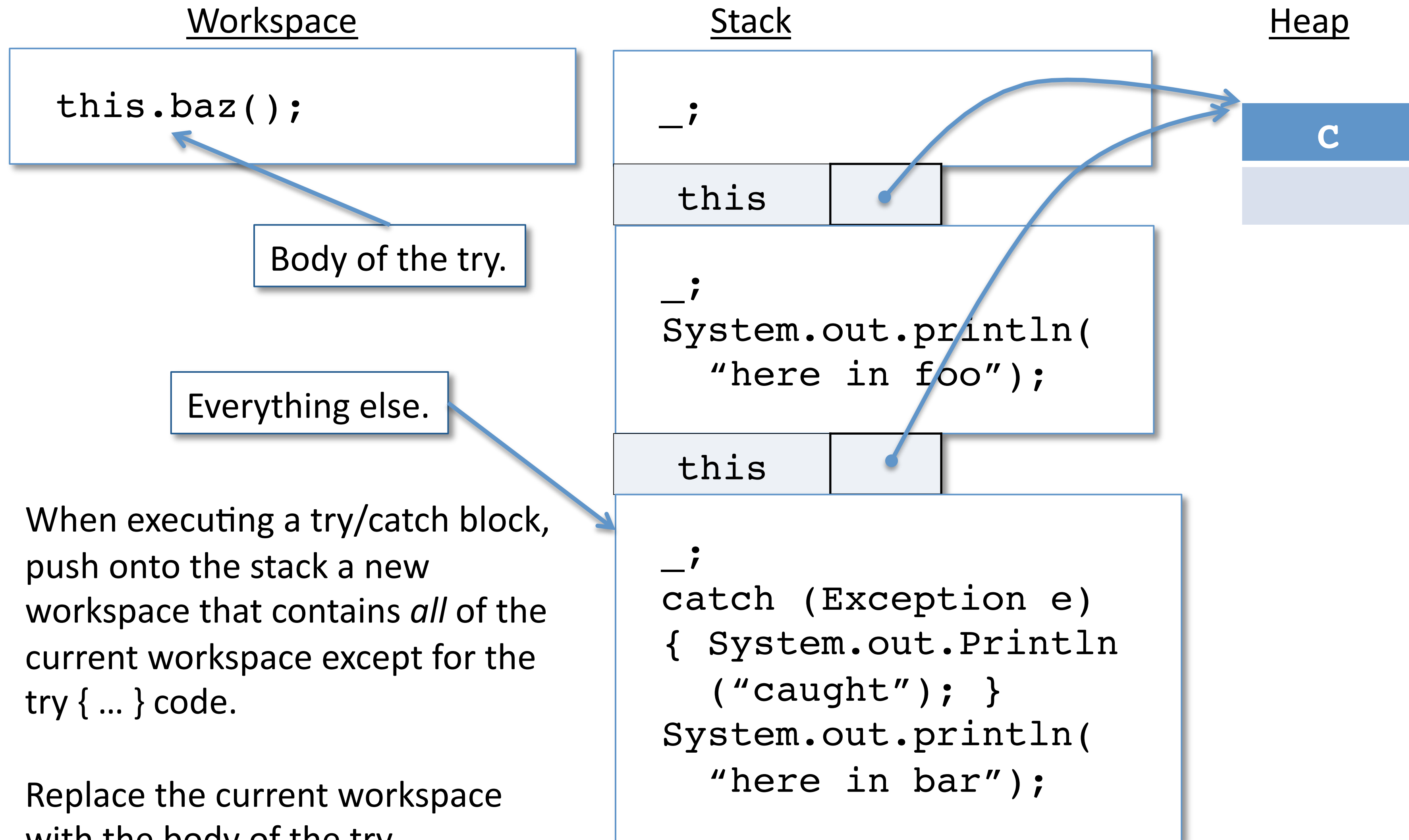
# Abstract Stack Machine



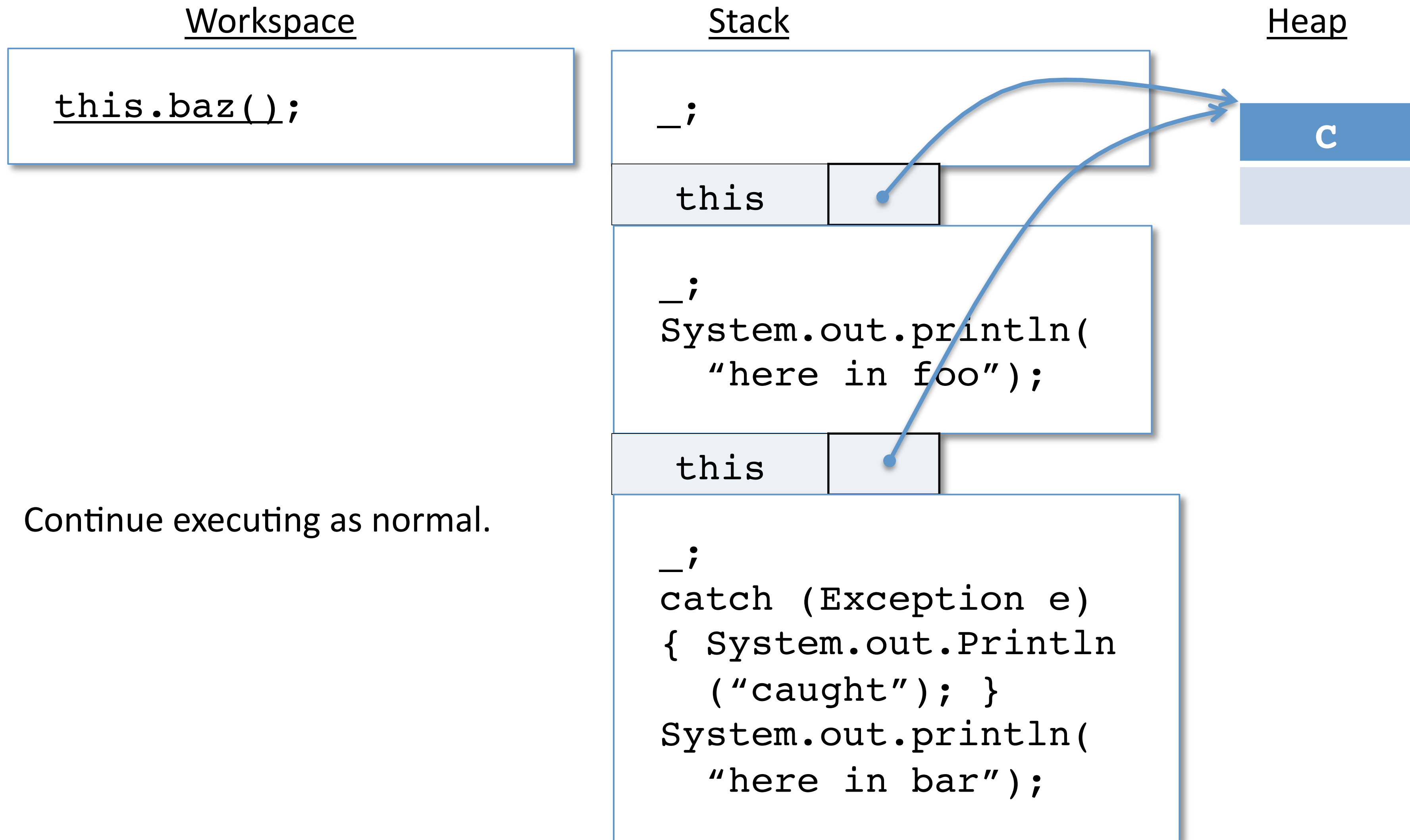

When executing a try/catch block, push onto the stack a new workspace that contains *all* of the current workspace except for the try { … } code.

Replace the current workspace with the body of the try.

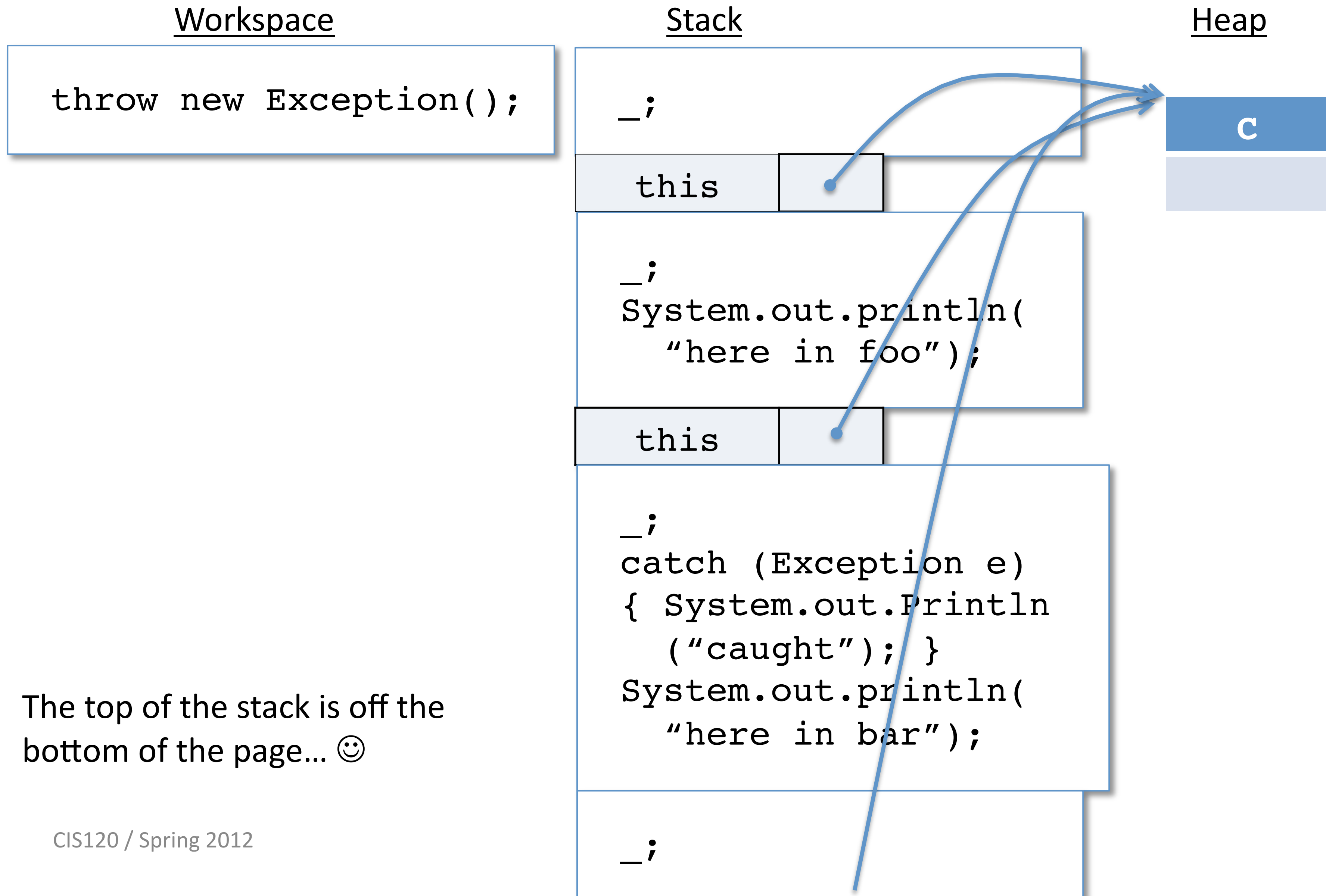
# Abstract Stack Machine



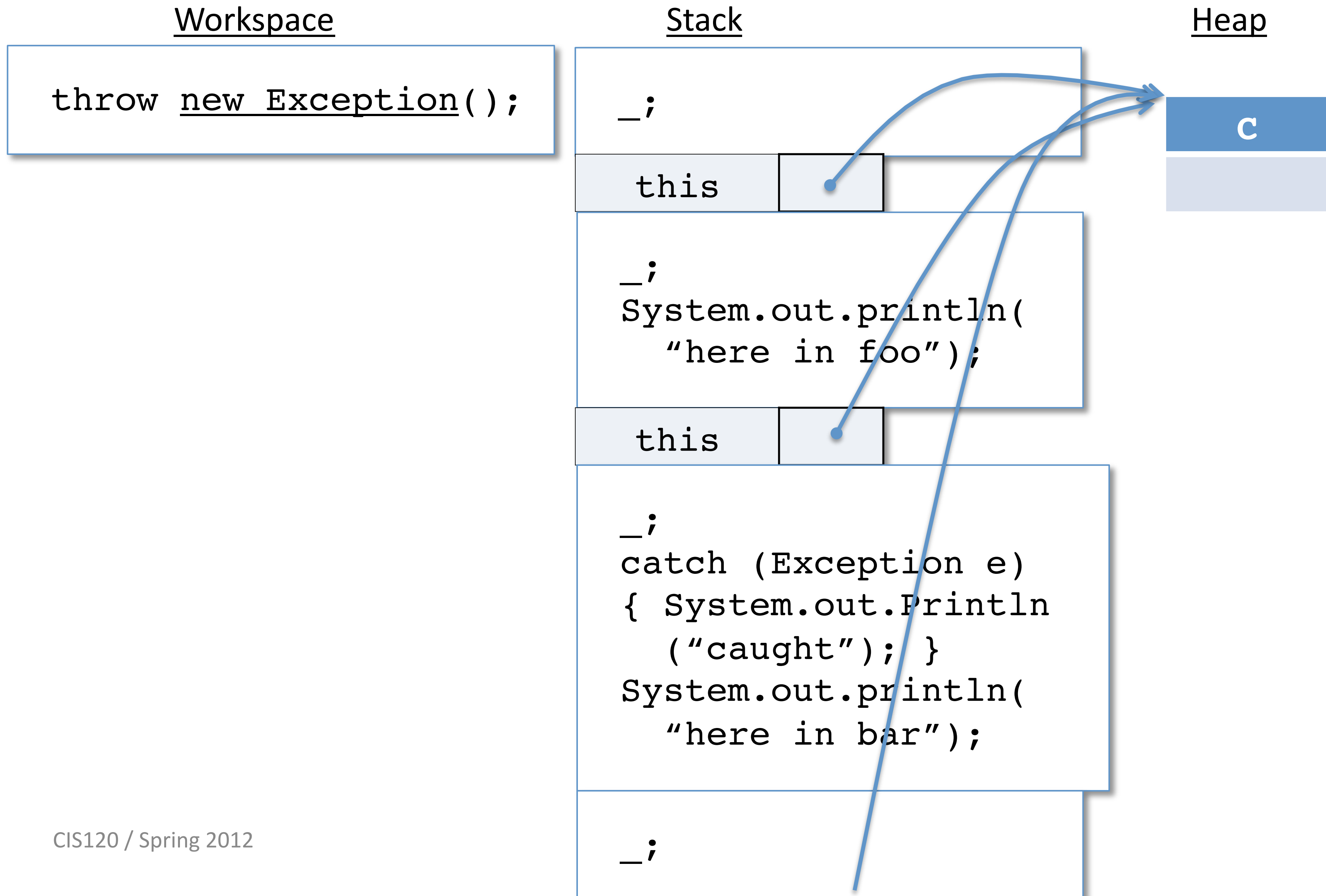
# Abstract Stack Machine



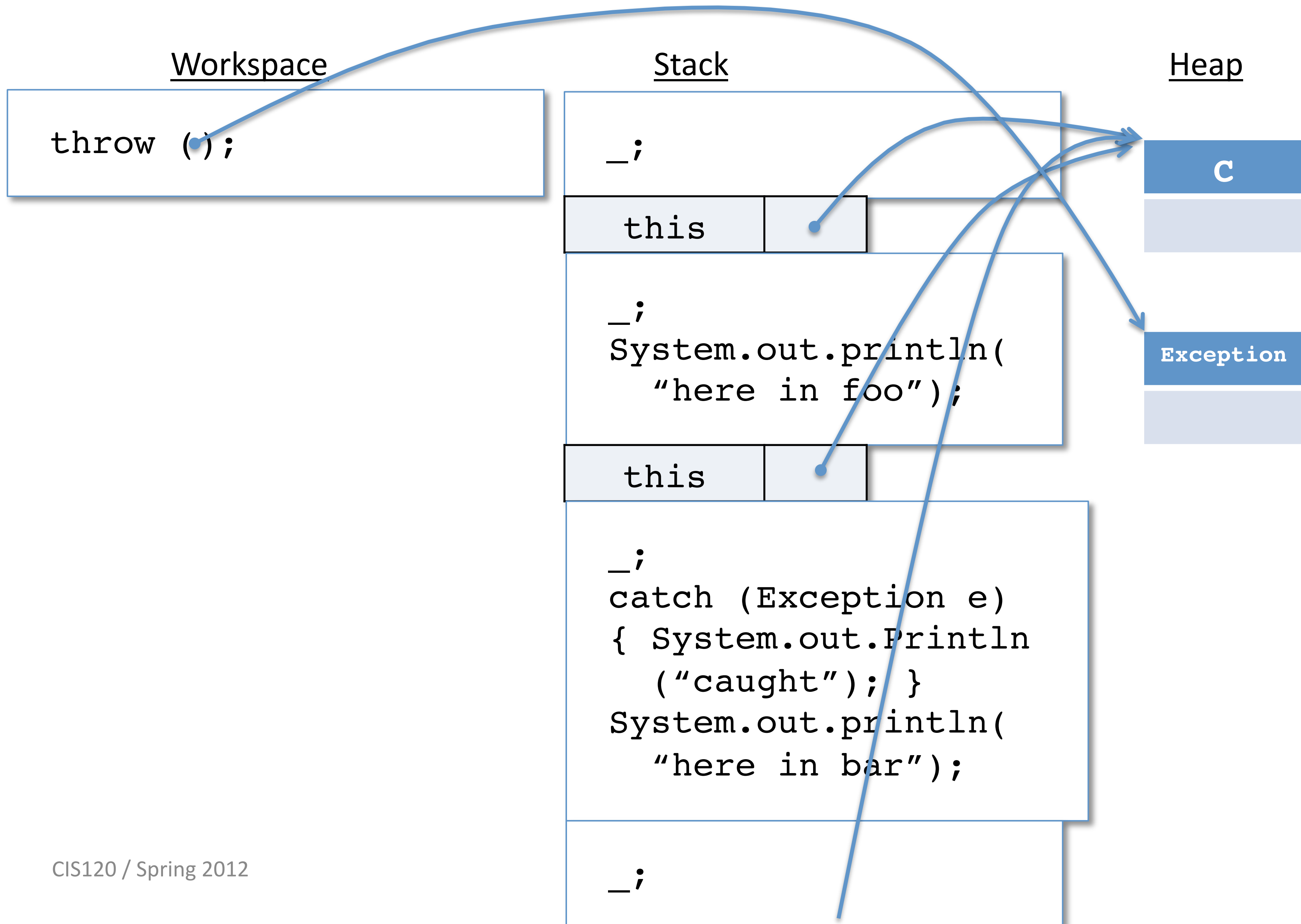
# Abstract Stack Machine



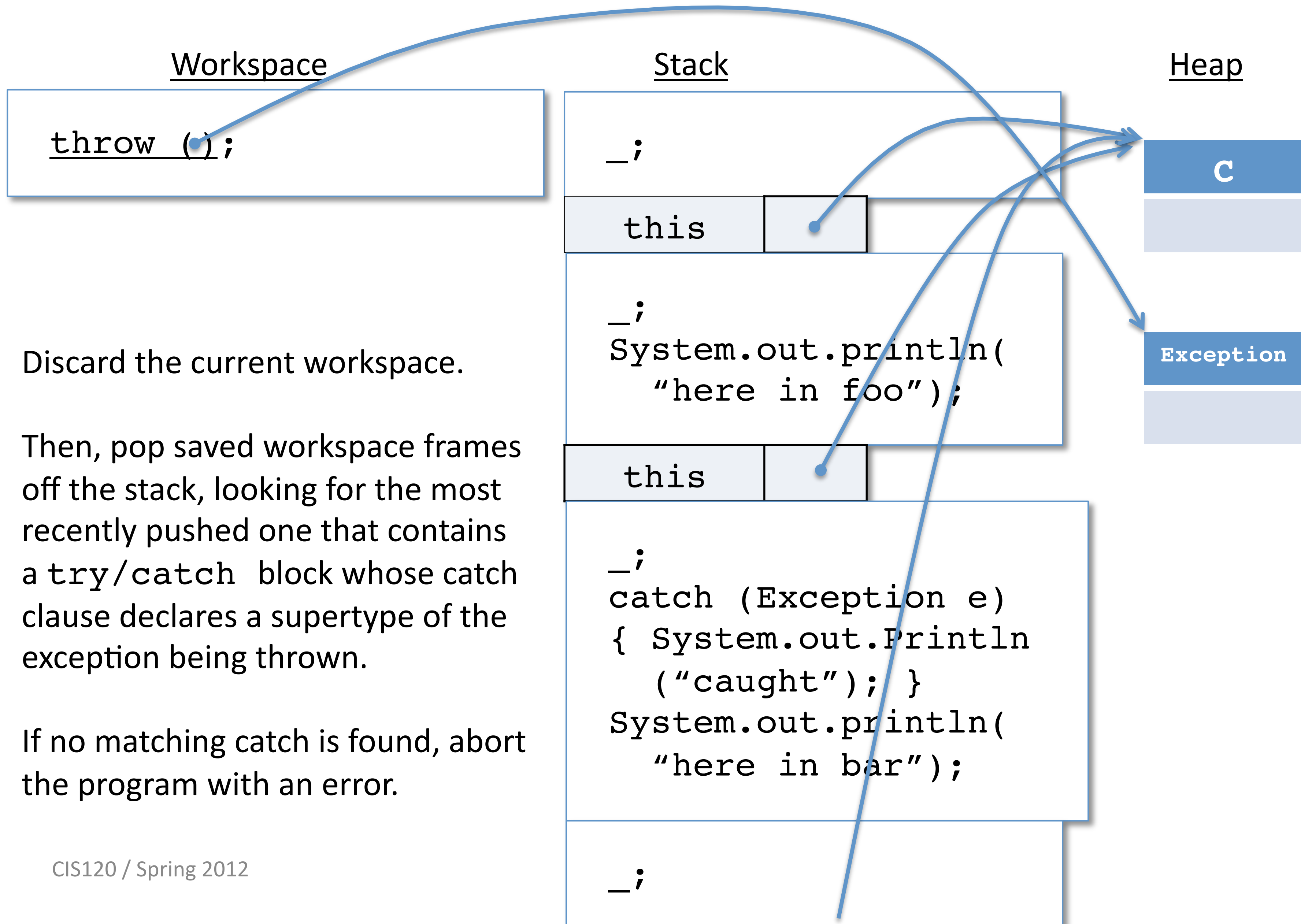
# Abstract Stack Machine



# Abstract Stack Machine



# Abstract Stack Machine



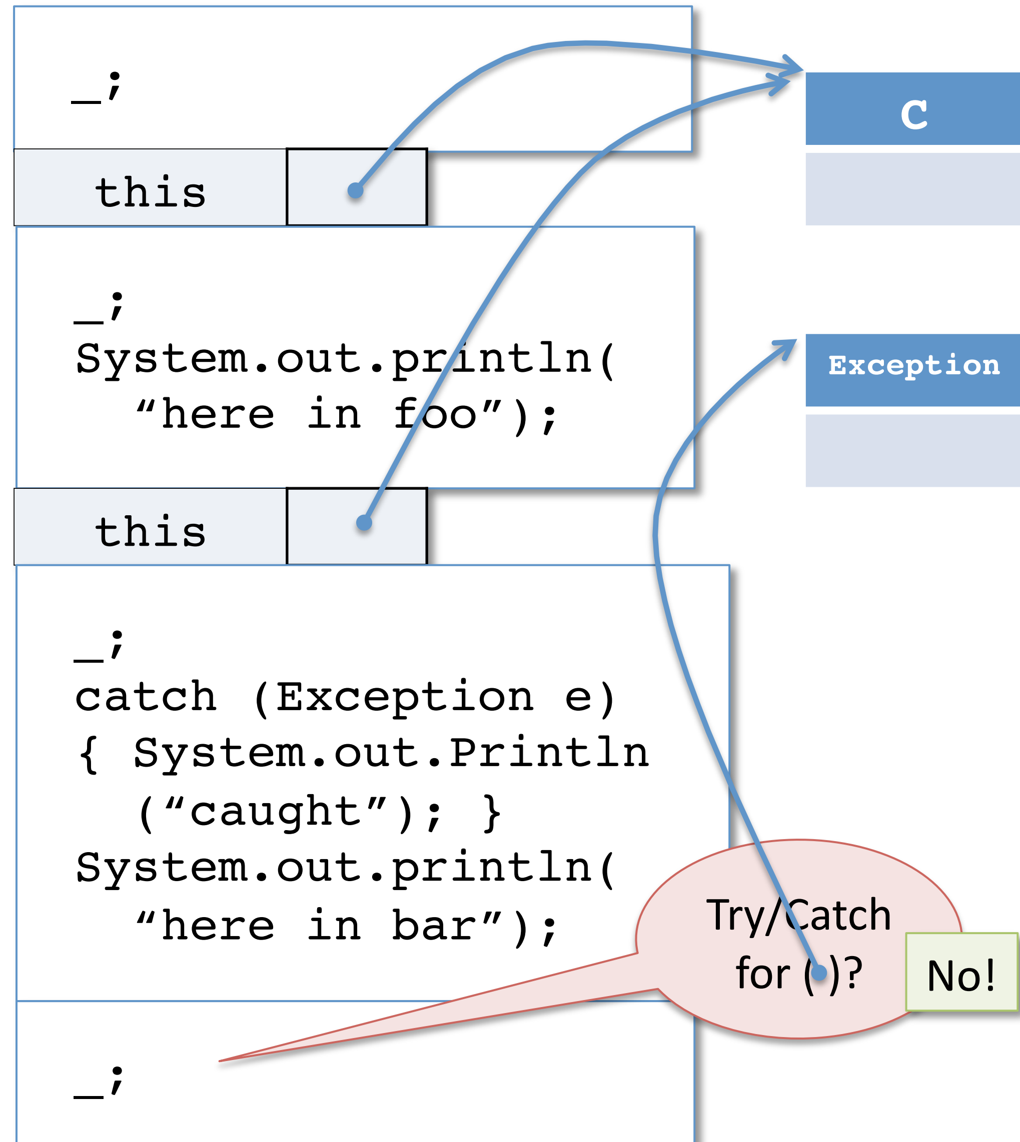
# Abstract Stack Machine

Workspace

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.

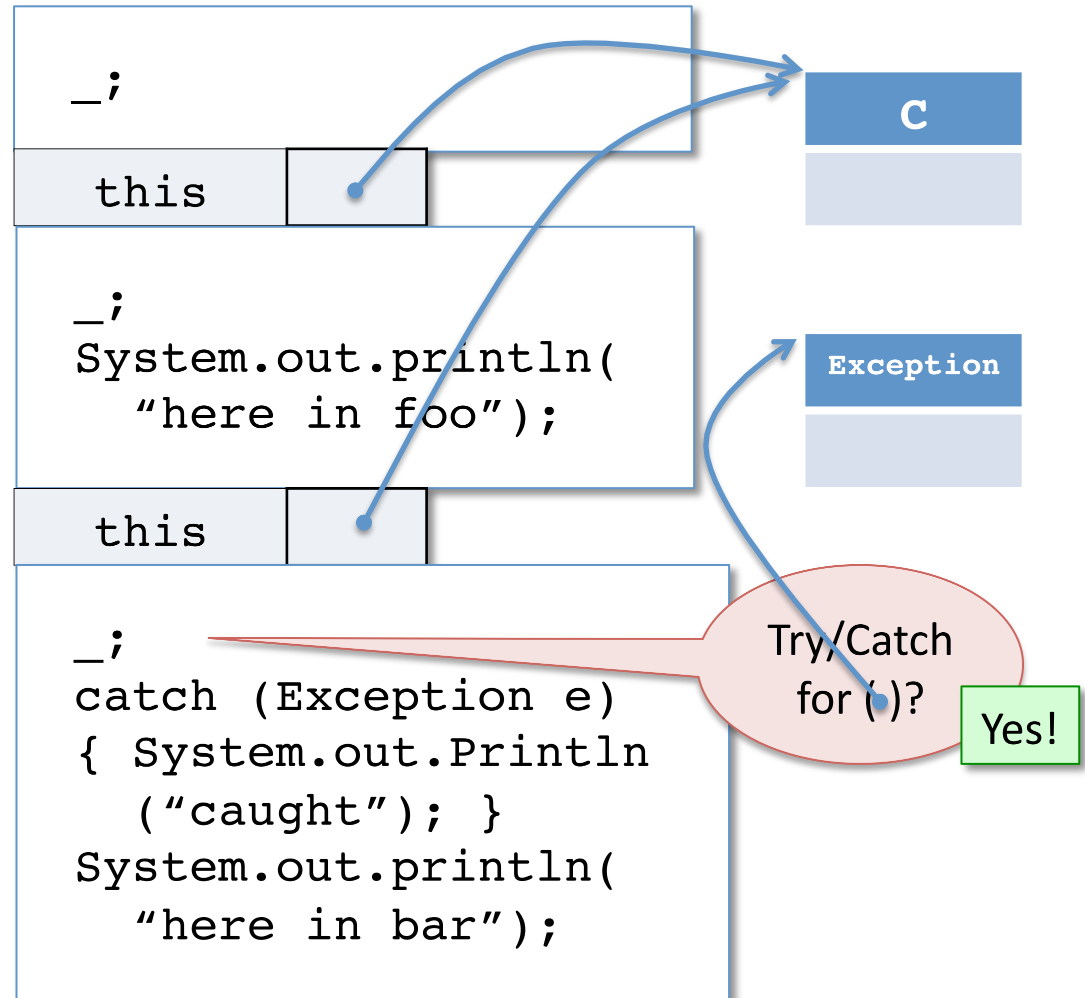


# Abstract Stack Machine

Workspace

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.



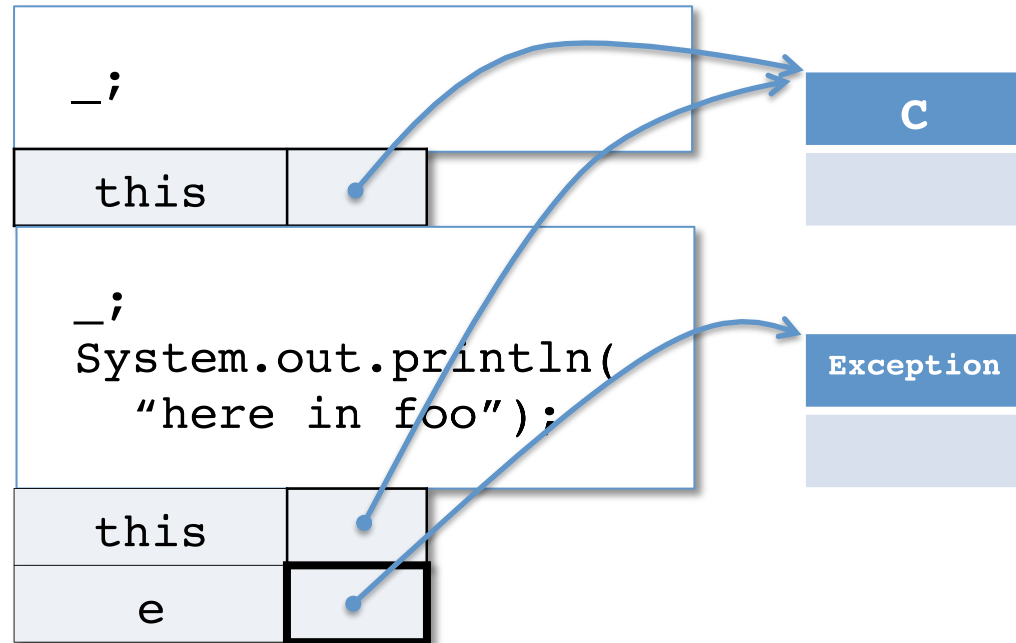
# Abstract Stack Machine

Workspace

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

Stack

Heap




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.

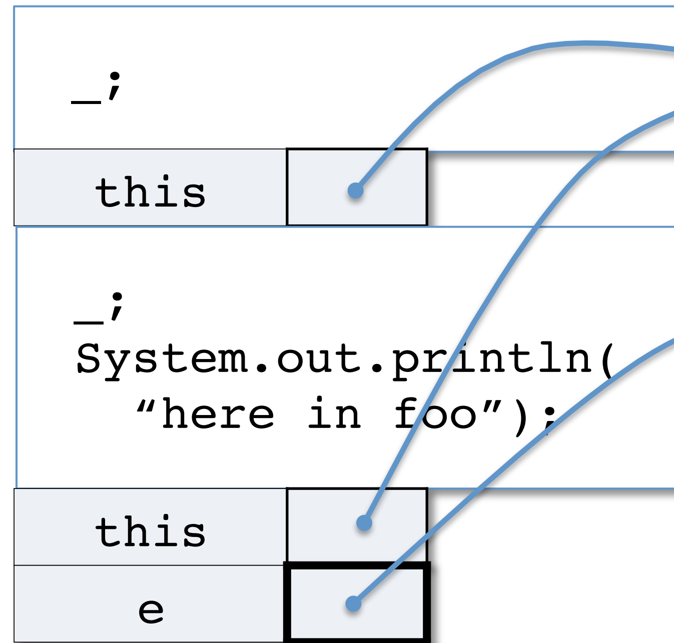
# Abstract Stack Machine

Workspace

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

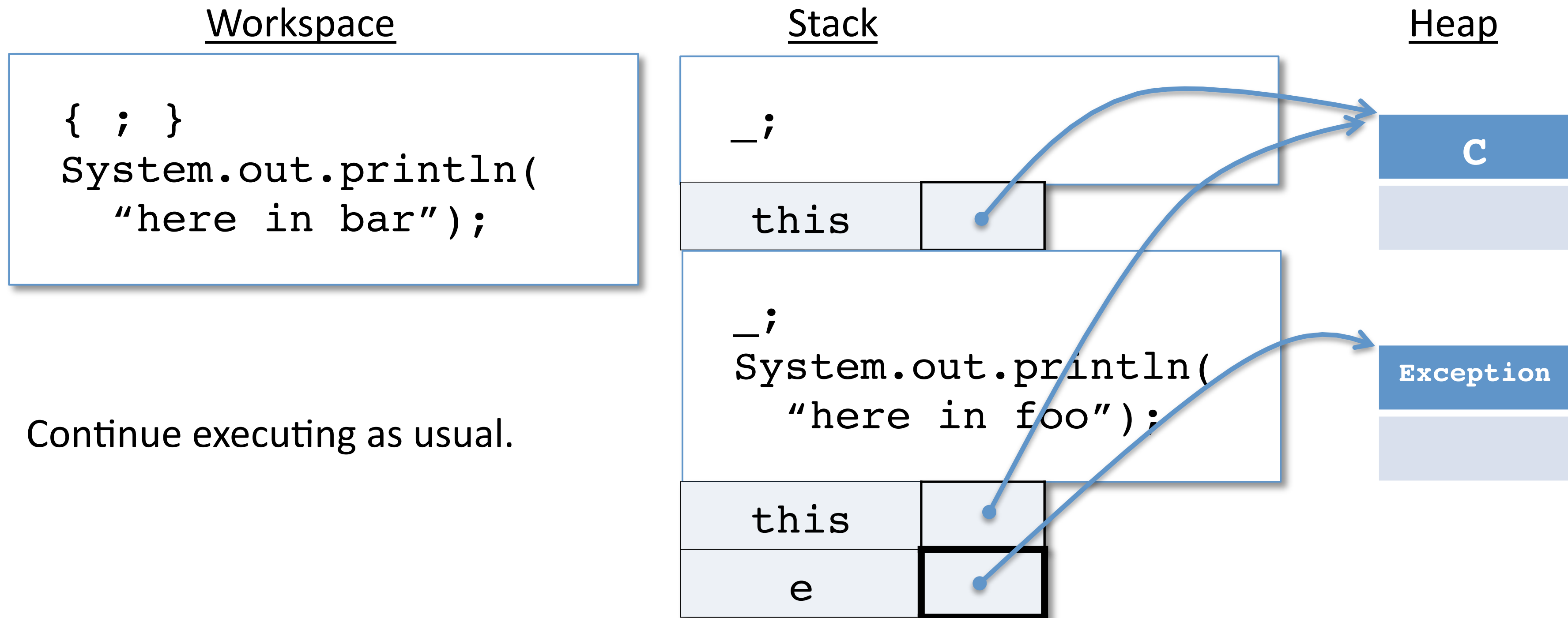
Stack

Heap




Continue executing as usual.

# Abstract Stack Machine




Continue executing as usual.

Console

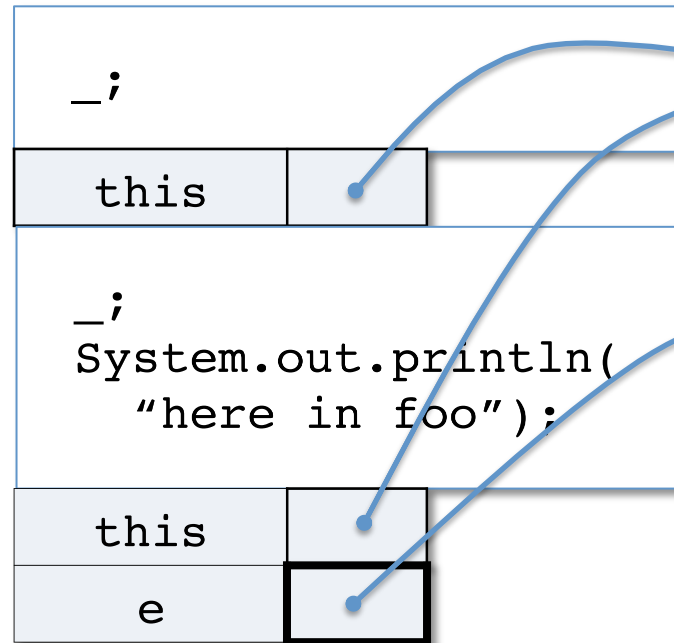
```
caught
```

# Abstract Stack Machine

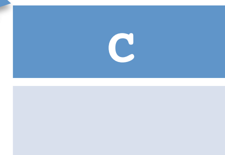
Workspace

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

Stack




Heap

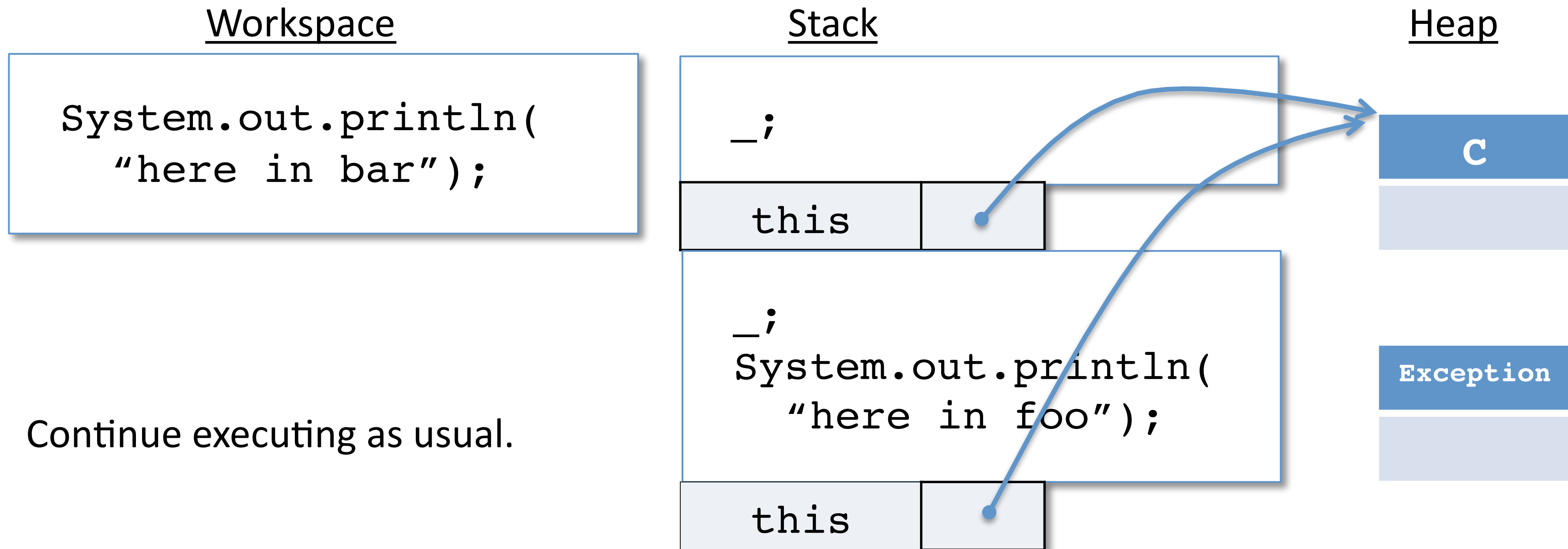


We're sweeping a few details about lexical scoping of variables under the rug – the scope of e is just the body of the catch, so when that is done, e must be popped from the stack.

Console

```
caught
```

# Abstract Stack Machine

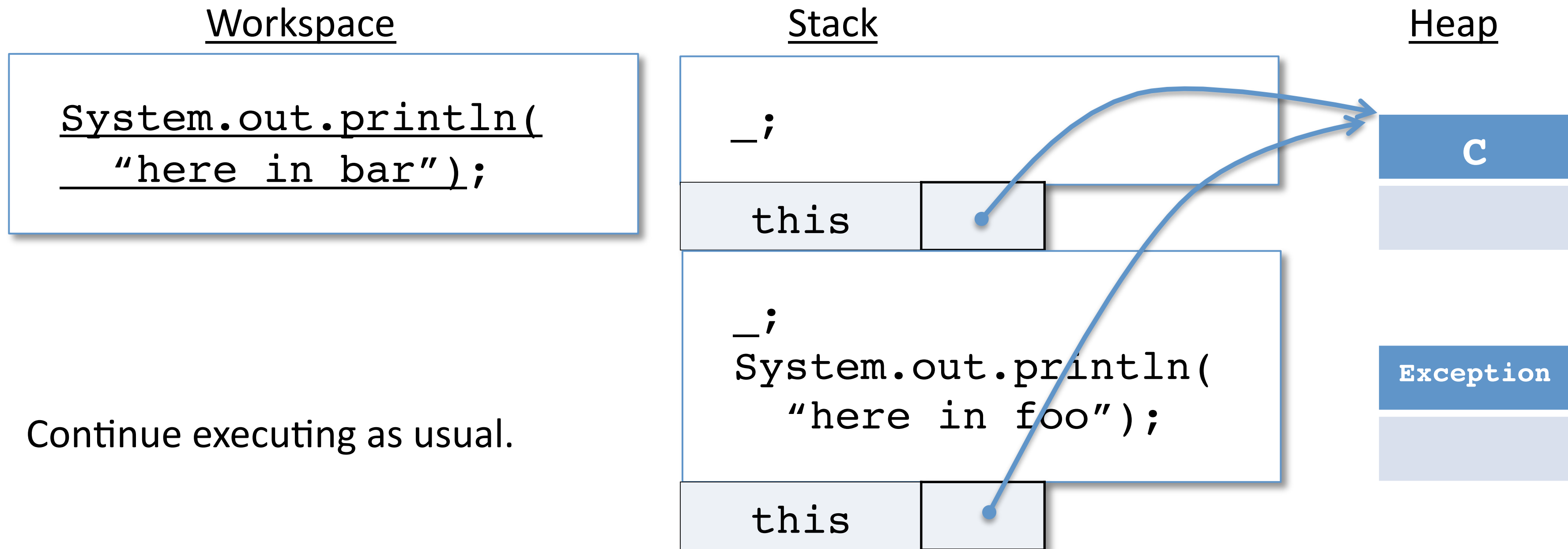



Continue executing as usual.

Console

caught

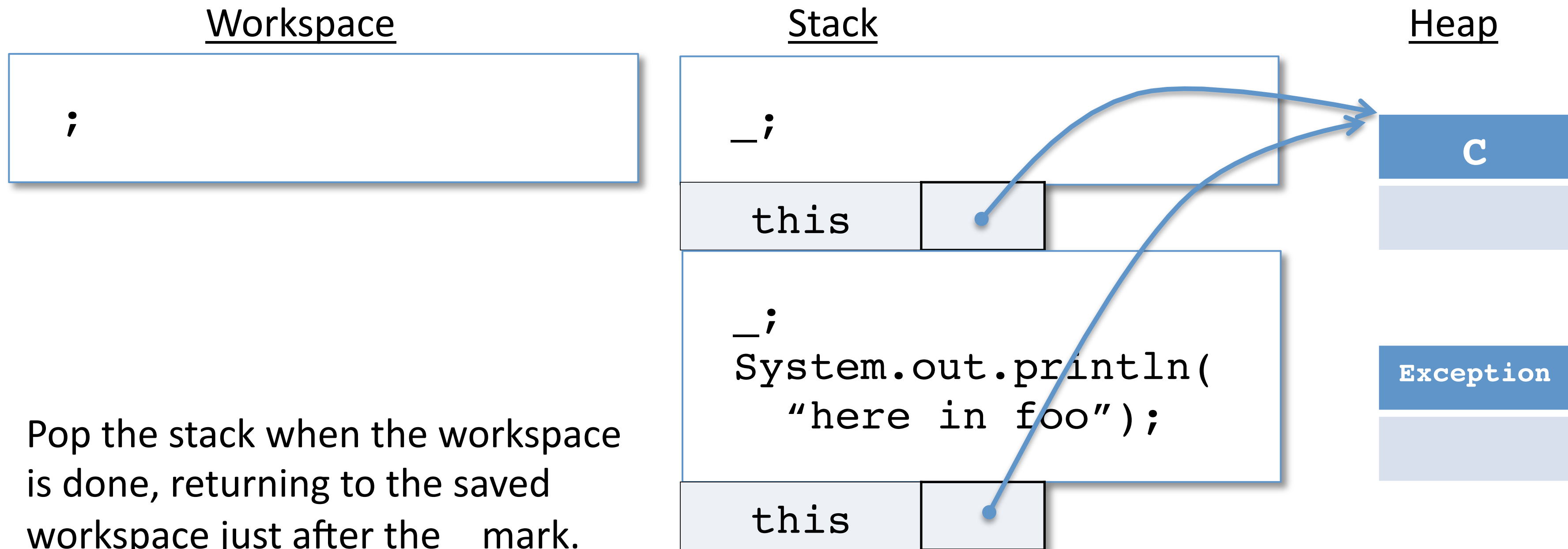
# Abstract Stack Machine




Continue executing as usual.

Console
caught

# Abstract Stack Machine

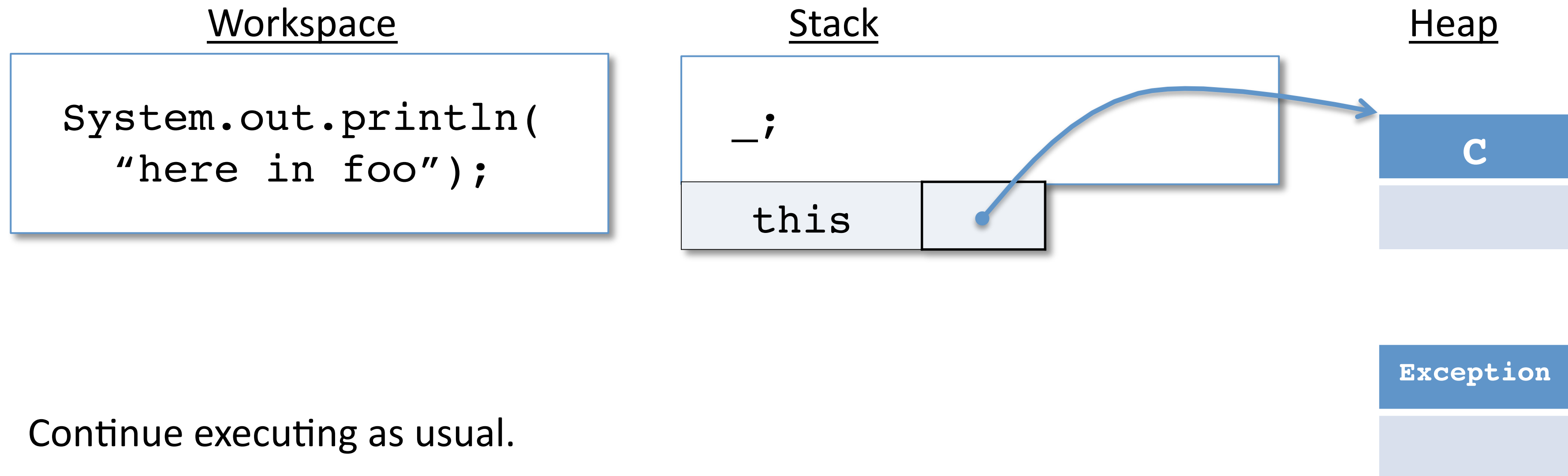



Pop the stack when the workspace is done, returning to the saved workspace just after the _ mark.

Console

```
caught
here in bar
```

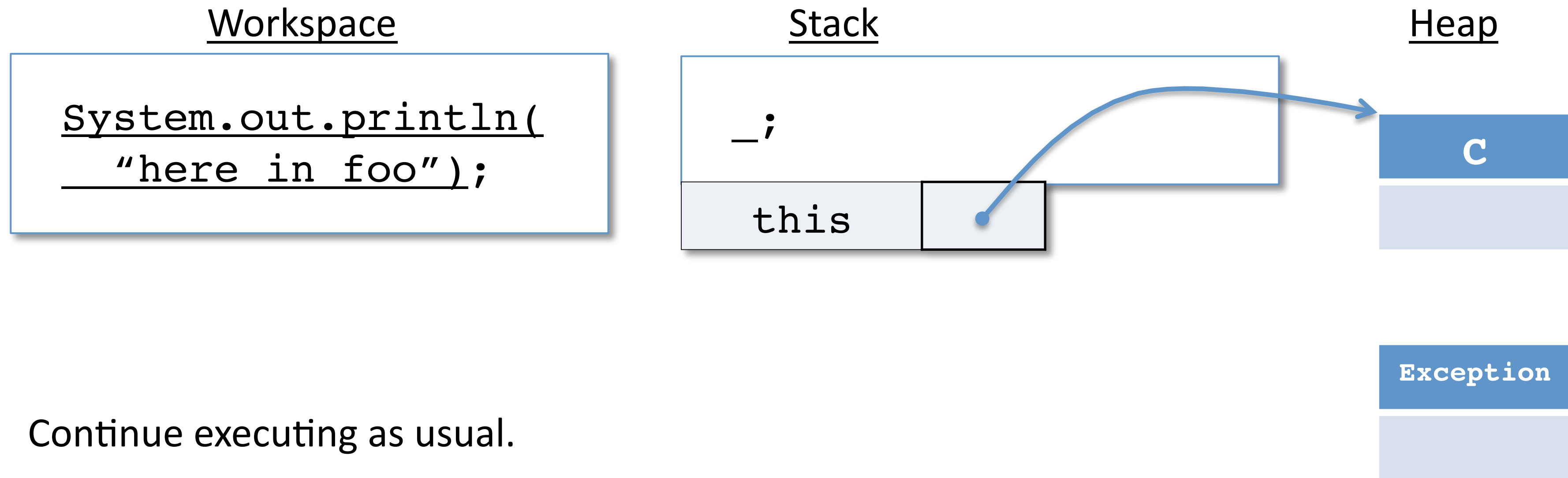
# Abstract Stack Machine




Continue executing as usual.

Console
```
caught
here in bar
```

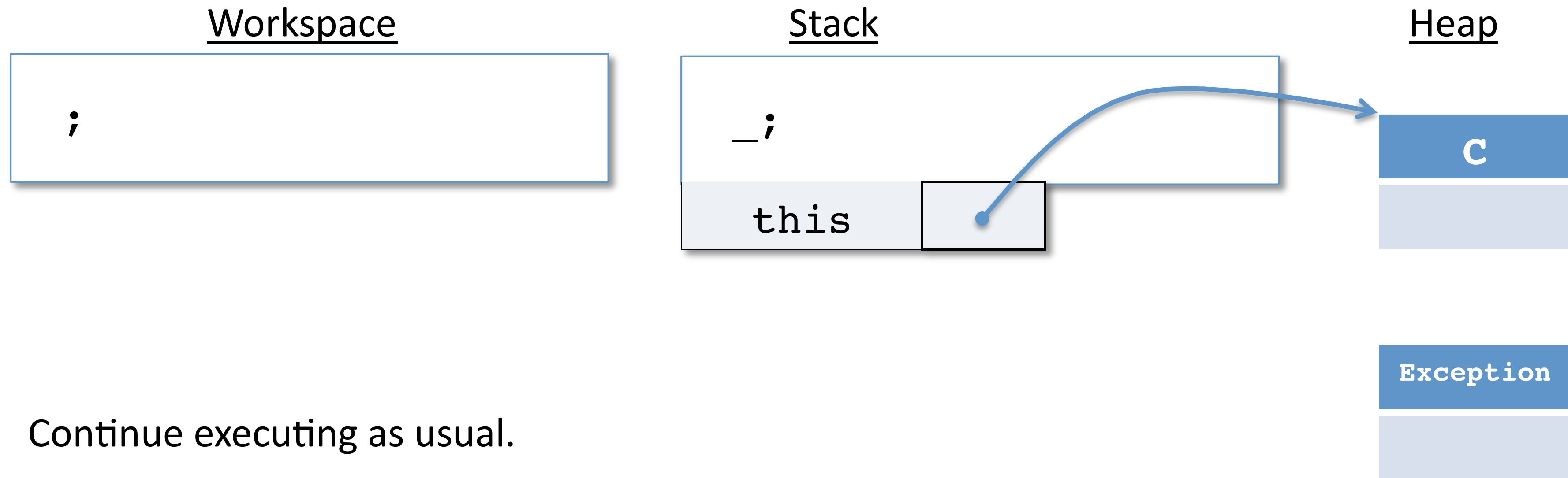
# Abstract Stack Machine




Continue executing as usual.

Console
```
caught
here in bar
```

# Abstract Stack Machine

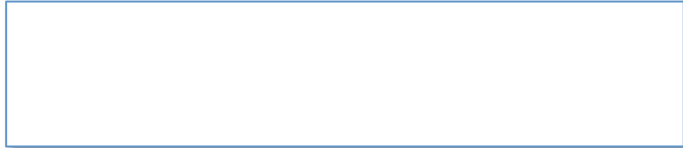



Continue executing as usual.

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

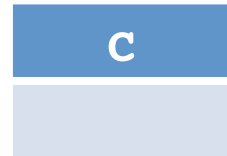
# Abstract Stack Machine

Workspace



Stack

Heap



Program terminated normally.

Console

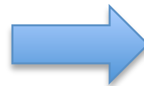
```
caught
here in bar
here in foo
```

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



Workspace

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

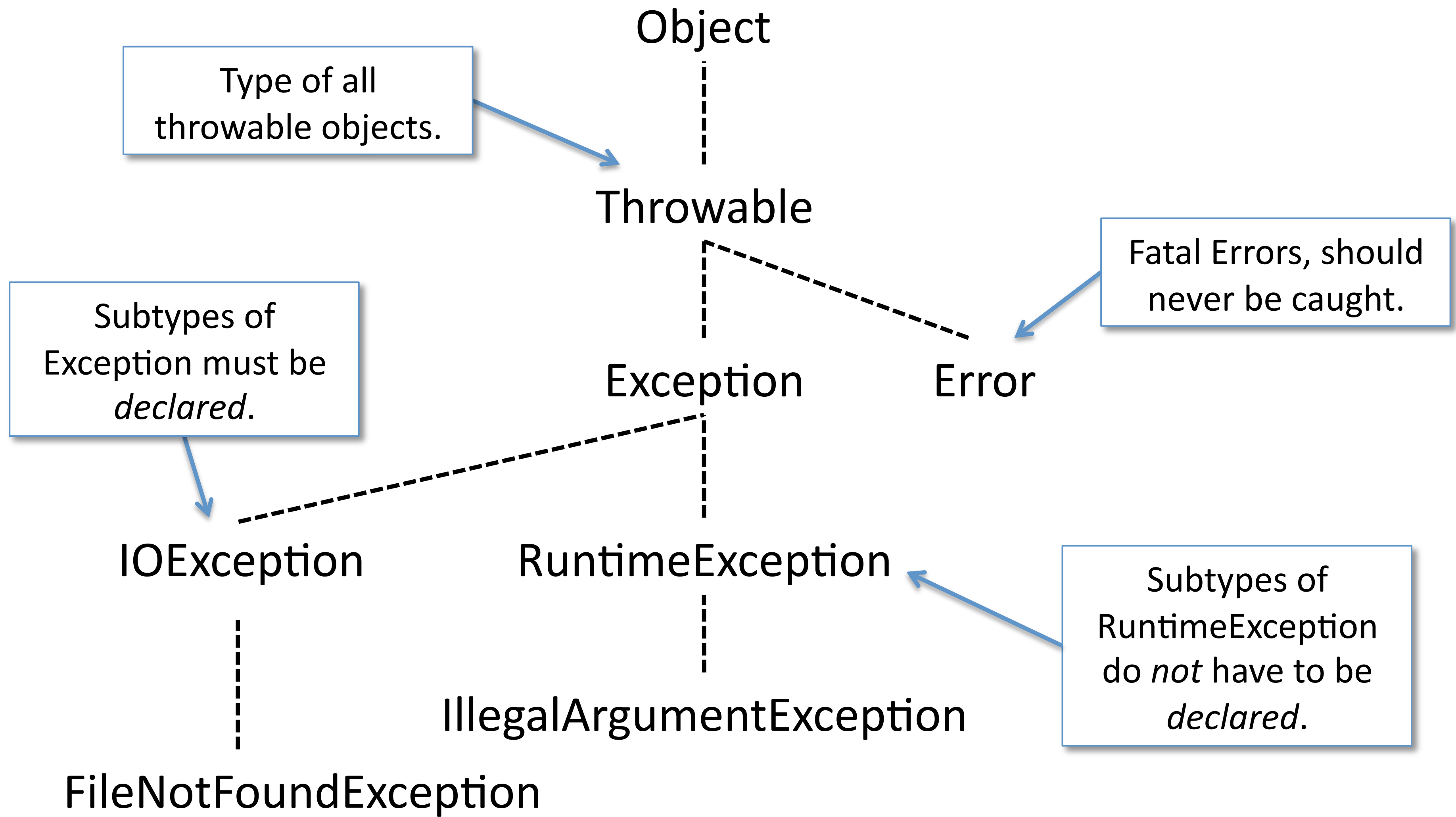
# Catching Exceptions

- 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!

# 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, 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
  - 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 good compromise?
  - Declared exceptions for libraries, where the documentation and usage enforcement are critical
  - Undeclared for client-exceptions to facilitate more flexible code

# 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 when doing so can convey useful information to possible callers that can handle the exception.
- It is often sensible to catch one exception and re-throw a different (more meaningful) kind of exception.
  - e.g. when implementing WordScanner, we caught IOException and threw 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
  - i.e. Don't do: try {…} catch (Exception e) {…}<br>instead do: try {…} catch (IOException e) {…}

# Finally

- A "finally" clause of a try/catch/finally statement *always* gets run, regardless of whether there is no exception, a propagated exception, a caught exception, or even if the method returns from inside the try.
- "Finally" is most 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) {…}
    }
  }
}
```