Programming Languages and Techniques (CIS120)

Lecture 29

November 11, 2015

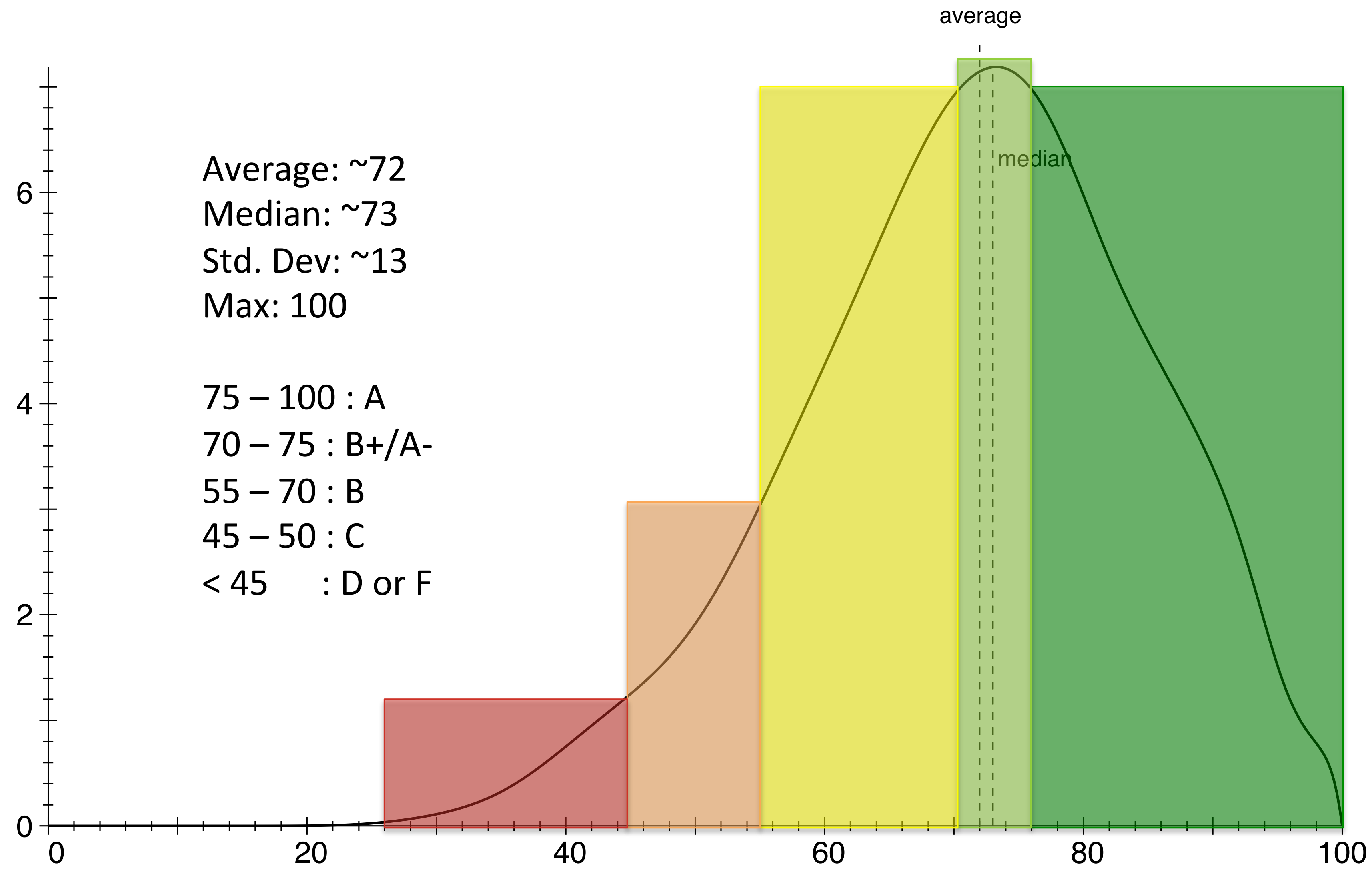
Overriding, Enums, Exceptions

Announcements

- HW07: PennPals is available
  - Due Tuesday: November 17th
  - Start Early!
  - Emphasizes: Java Collections, Design

- Midterm 2:
  - Solutions are up on the web site
  - You can examine your exams starting *tomorrow*
  - Levine 308

Midterm 2 Results



Average: ~72
Median: ~73
Std. Dev: ~13
Max: 100

75 – 100 : A
70 – 75 : B+/A-
55 – 70 : B
45 – 50 : C
< 45 : D or F

Poll

Have you started HW 07   PennPals?

1. No
2. I've thought about the ServerModel structure
3. I've got the basics working
4. I'm Done

Intentional Overriding

Abstract Classes

Abstract Classes

- Are like classes, but with some method implementations omitted.
  - They are instead declared *abstract*

- Non-abstract subclasses must provide an implementation of the missing methods

- Why? When there is a general algorithm whose implementation depends on the behavior of subtypes' implementation
  - E.g. removeAll defined in terms of remove in the Collections library

When To Override?

- Only override methods when the parent class is *designed* specifically to support such modifications:
  - If the library designer specifically describes the behavioral contract that the parent methods assume about overridden methods (e.g. equals, paintComponent)
  - If you're writing the code for both the parent and child class (and will maintain control of both parts as the software evolves) it might be OK to overrride.
  - Either way: document the design
  - Use the `@Override` annotation to mark intentional overriding

- Look for other means of achieving the desired outcome:
  - Use composition & delegation (i.e. wrapper objects) rather than overriding

How to prevent overriding

- By default, methods can be overridden in subclasses.
- The `final` modifier changes that.
- Final methods *cannot* be overridden in subclasses
  - Prevents subclasses from changing the “behavioral contract” between methods by overriding
  - `static final` methods cannot be hidden

- Similar, but not the same as final fields and local variables:
  - Act like the immutable name bindings in OCaml
  - Must be initialized (either by a static initializer or in the constructor) and cannot thereafter be modified.
  - `static final` fields are useful for defining constants (e.g. `Math.PI`)

Digression: Enumerations

See: ChatDemo.java

Enumerations (a.k.a. Enum Types)

- Java supports *enumerated* type constructors.
  - These are a bit like OCaml's datatypes.
- Example (from Chat HW) ServerError:

```
public enum ServerError {
    OKAY(200),
    INVALID_NAME(401),
    NO_SUCH_CHANNEL(402),
    NO_SUCH_USER(403),
    …
    // The integer associated with this enum value
   private final int value;
   ServerError(int value) {
        this.value = value;
   }
   public int getCode() {
        return value;
   }
}
```

Using Enums: Switch Cases

```
// Use of 'enum' from CommandParser.java (Chat HW)
CommandType t = …

switch (t) {
case CREATE : System.out.println("Got CREATE!"); break;
case MESG : System.out.println("Got MESG!"); break;
default: System.out.println("default");
}
```

- Multi-way branch, similar to OCaml's match
  - Works for: primitive data 'int', 'byte', 'char', *etc.*, Enum types , String
  - Not pattern matching! (Cannot bind subcomponents of an Enum)
- The `default` keyword specifies the "catch all" case.

What will be printed by the following program?

```java
Command.Type t = Command.Type.CREATE;

switch (t) {
case CREATE : System.out.println("Got CREATE!");
case MESG : System.out.println("Got MESG!");
case NICK : System.out.println("Got NICK!");
default: System.out.println("default");
}
```

1. Got CREATE!
2. Got MESG!
3. Got NICK!
4. default
5. something else

break

- GOTCHA: Must use explicit **break** to avoid "fallthrough"
  - without break, the code of the next branch will be run too!

- The program in the quiz prints all of the strings:

```
Got CREATE!
Got MESG!
Got NICK!
default
```

Enumerations

- Enum types are just a convenient way of defining a class along with methods of a class.
  - Enum types (implicitly) extend java.lang.Enum
  - They can contain constant data “properties”
  - As classes, they can have methods: e.g. to access a field
  - Intended to represent constant data values.

- Automatically generated static methods:
  - `valueOf` : converts a `String` to an Enum

    Command.Type c = Command.Type.valueOf ("CONNECT");
  - `values`: returns an `Array` of all the enumerated constants

    Command.Type[] varr = Command.Type.values();

Exceptions

Dealing with the unexpected

Why do methods “fail”?

- Some methods make requirements of their arguments
  - Input to max is a nonempty list, Item is non-null, more elements for 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 that doesn't exist
- Resources might be exhausted
  - Program uses all of the computer's disk space

- 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 must 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
  - e.g. in Java, we can create a special class like option
  - *Passes responsibility to caller, but caller forced to do the proper check*
- Use exceptions
  - Available both in OCaml and Java
  - Any caller (not just the immediate one) can handle the situation
  - 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

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

```java
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 without printing anything
2. Program prints “here in bar”, then stops
3. Program prints “here in bar”, then “here in foo”, then stops
4. Something else

Abstract Stack Machine

Workspace

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

Stack

Heap

Abstract Stack Machine

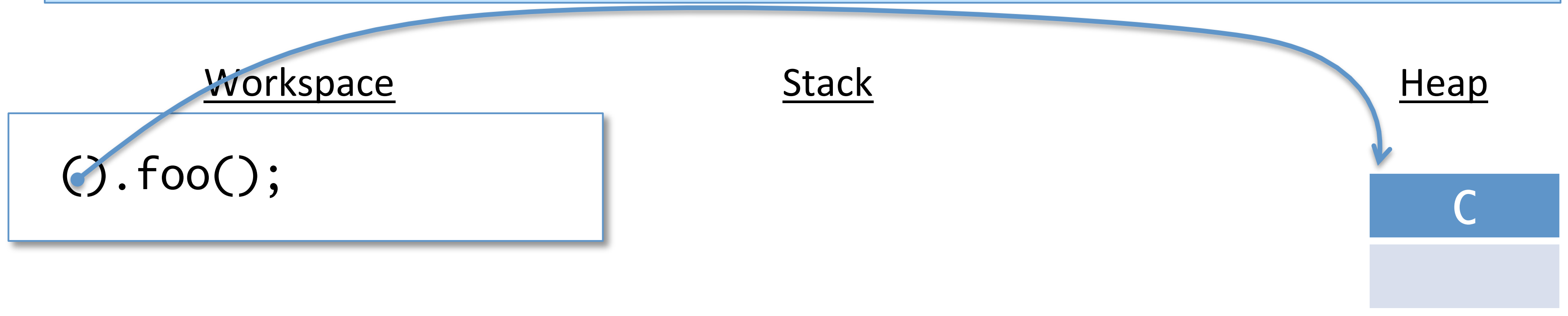
<u>Workspace</u>

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

<u>Stack</u>

<u>Heap</u>

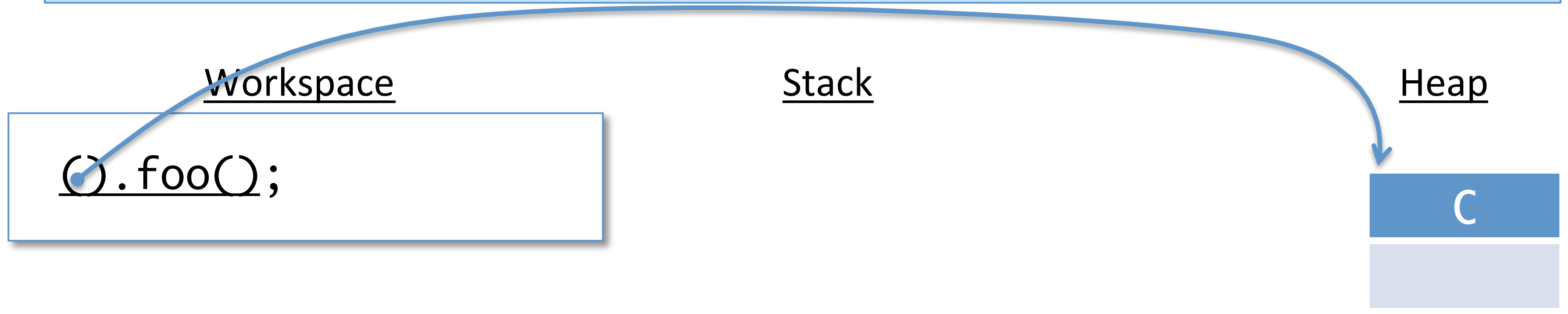
Abstract Stack Machine



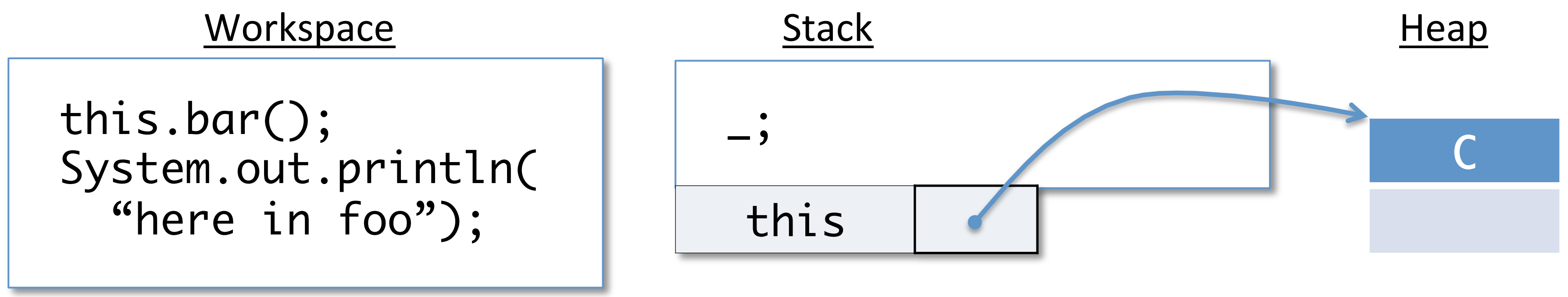
```
(•).foo();
```

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

Abstract Stack Machine



Abstract Stack Machine



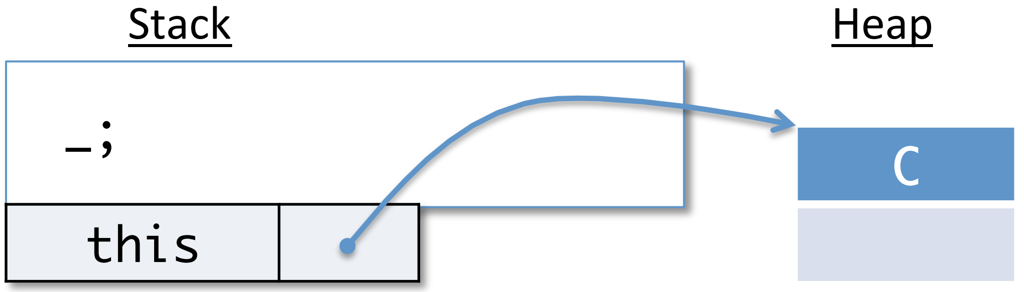
```
this.bar();
System.out.println(
  “here in foo”);
```

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.

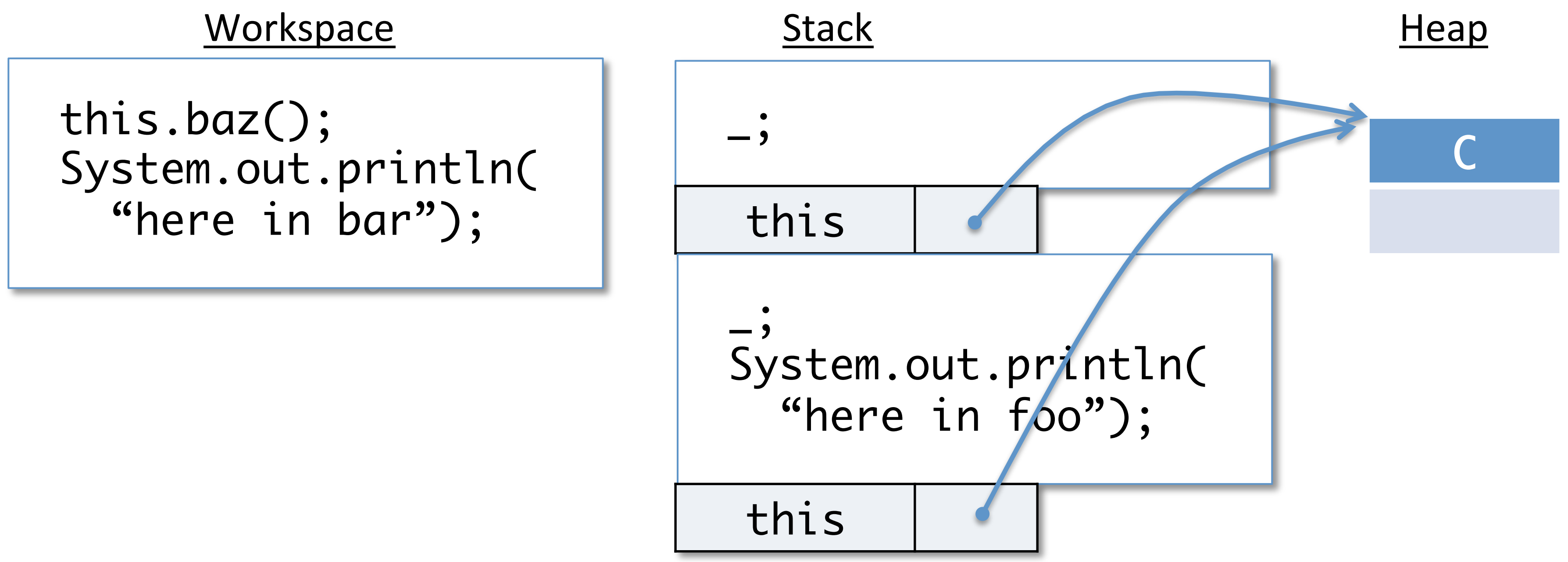
Abstract Stack Machine

Workspace

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



Abstract Stack Machine

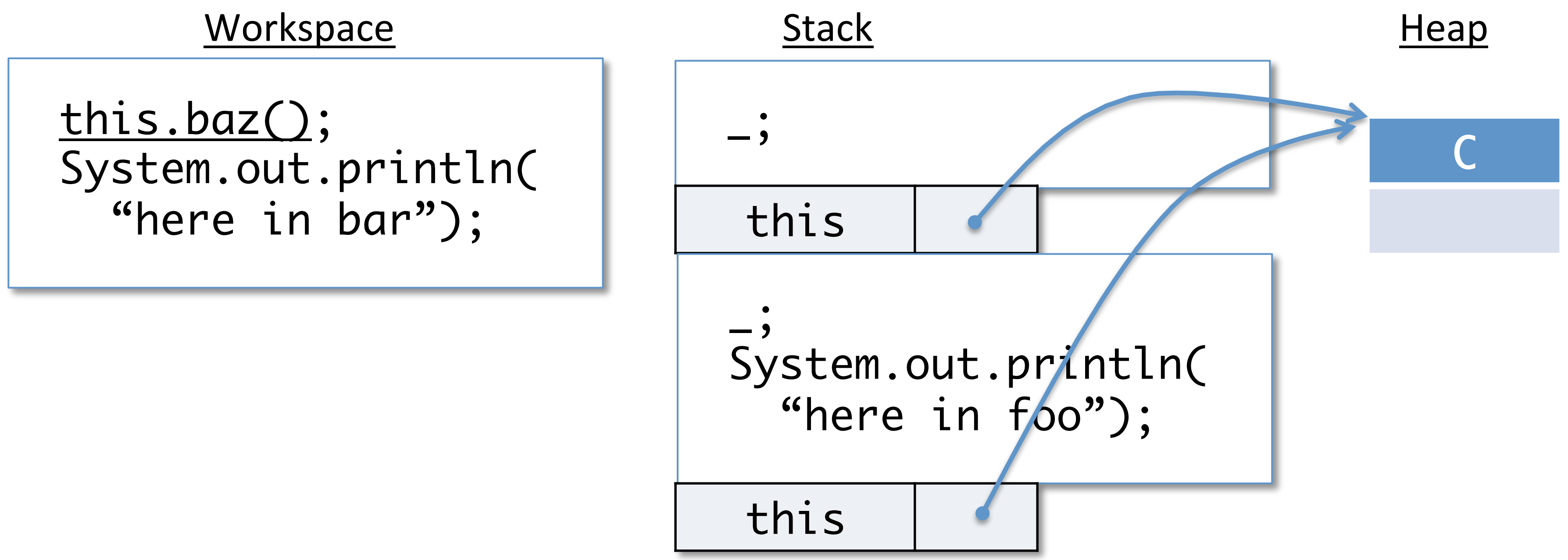


```
this.baz();
System.out.println(
  “here in bar”);
```

```
_;
```

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

Abstract Stack Machine

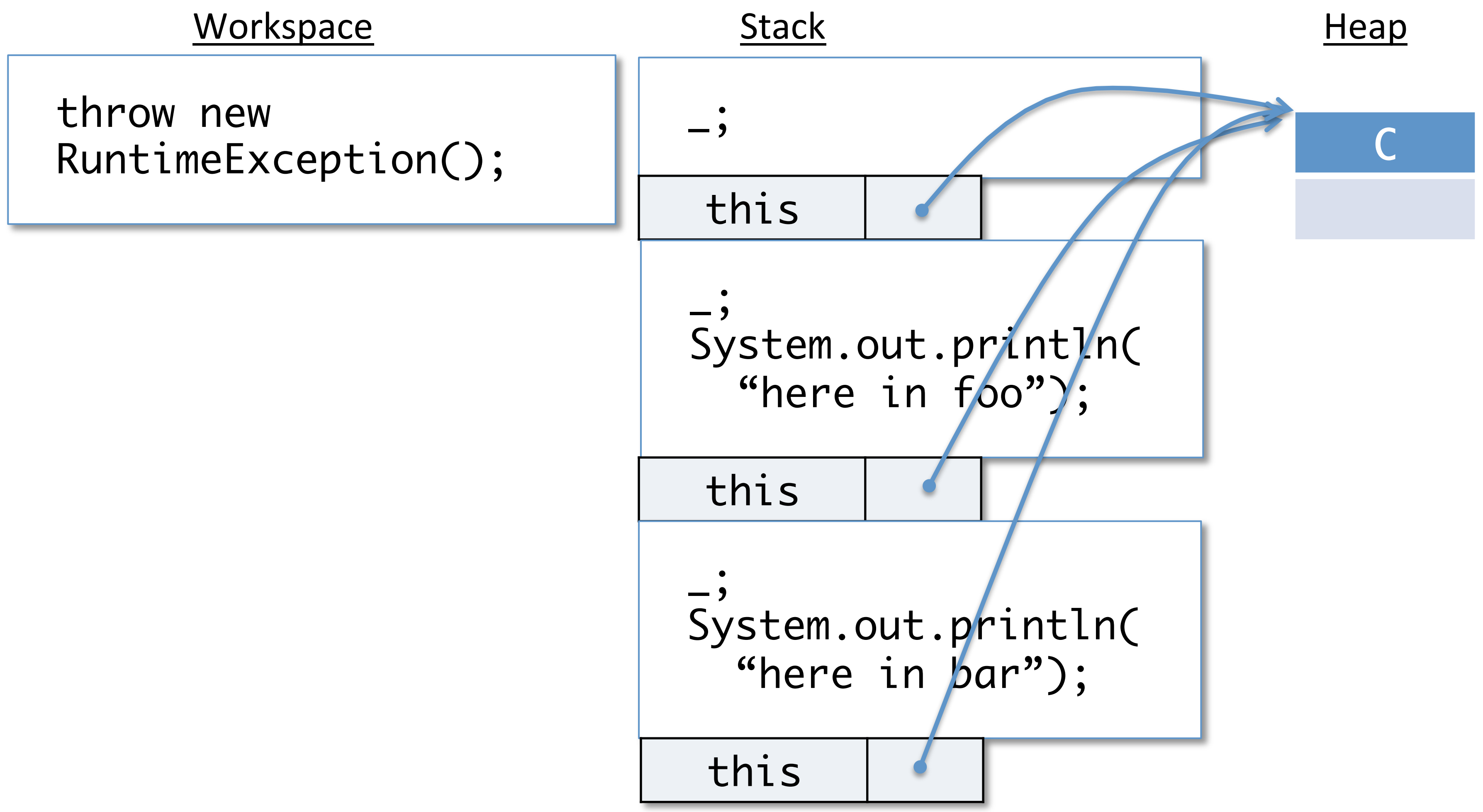


```
this.baz();
System.out.println(
  “here in bar”);
```

```
_;
```

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

Abstract Stack Machine

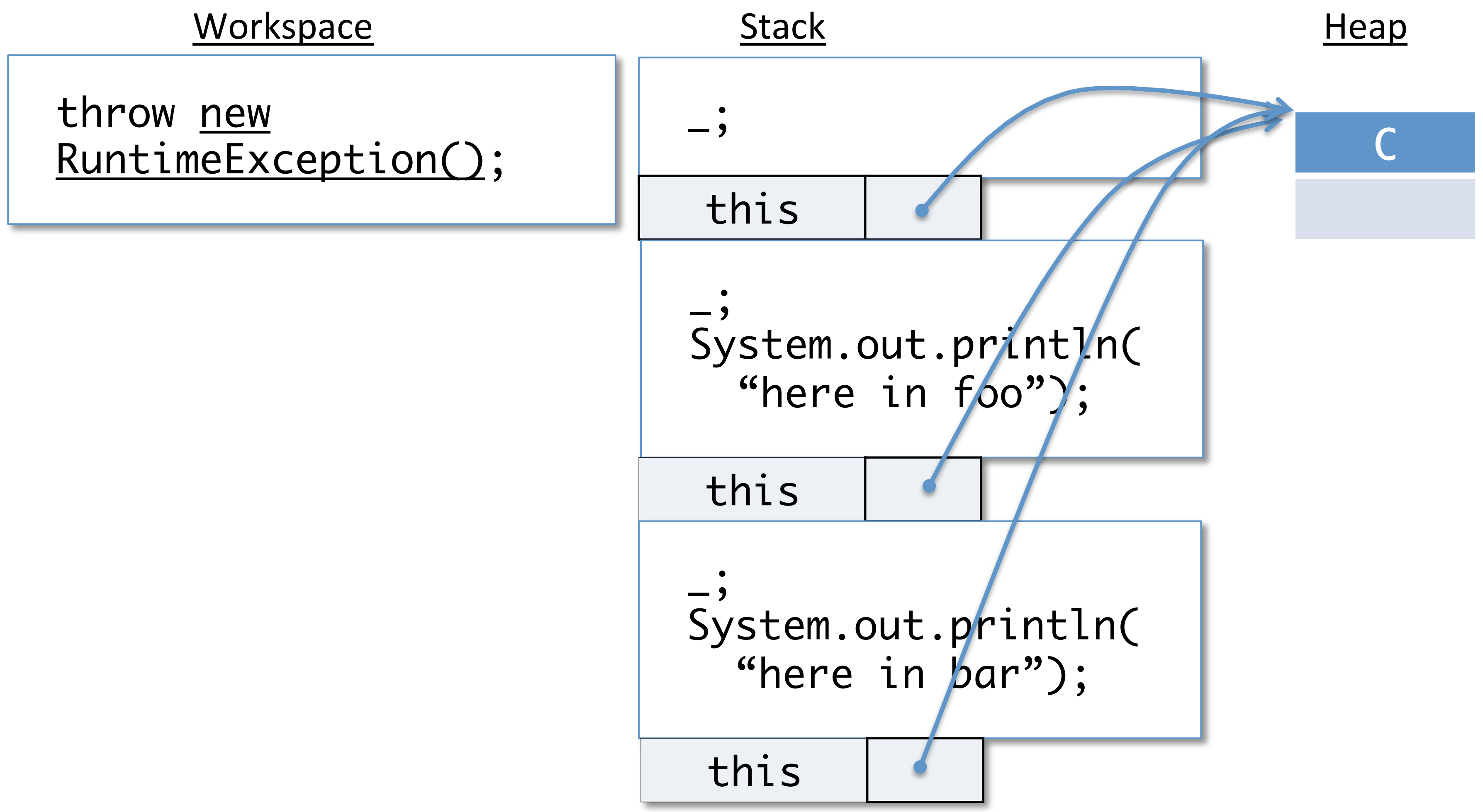


```
throw new
RuntimeException();
```

```
_;
System.out.println(
  "here in foo");
```

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

Abstract Stack Machine



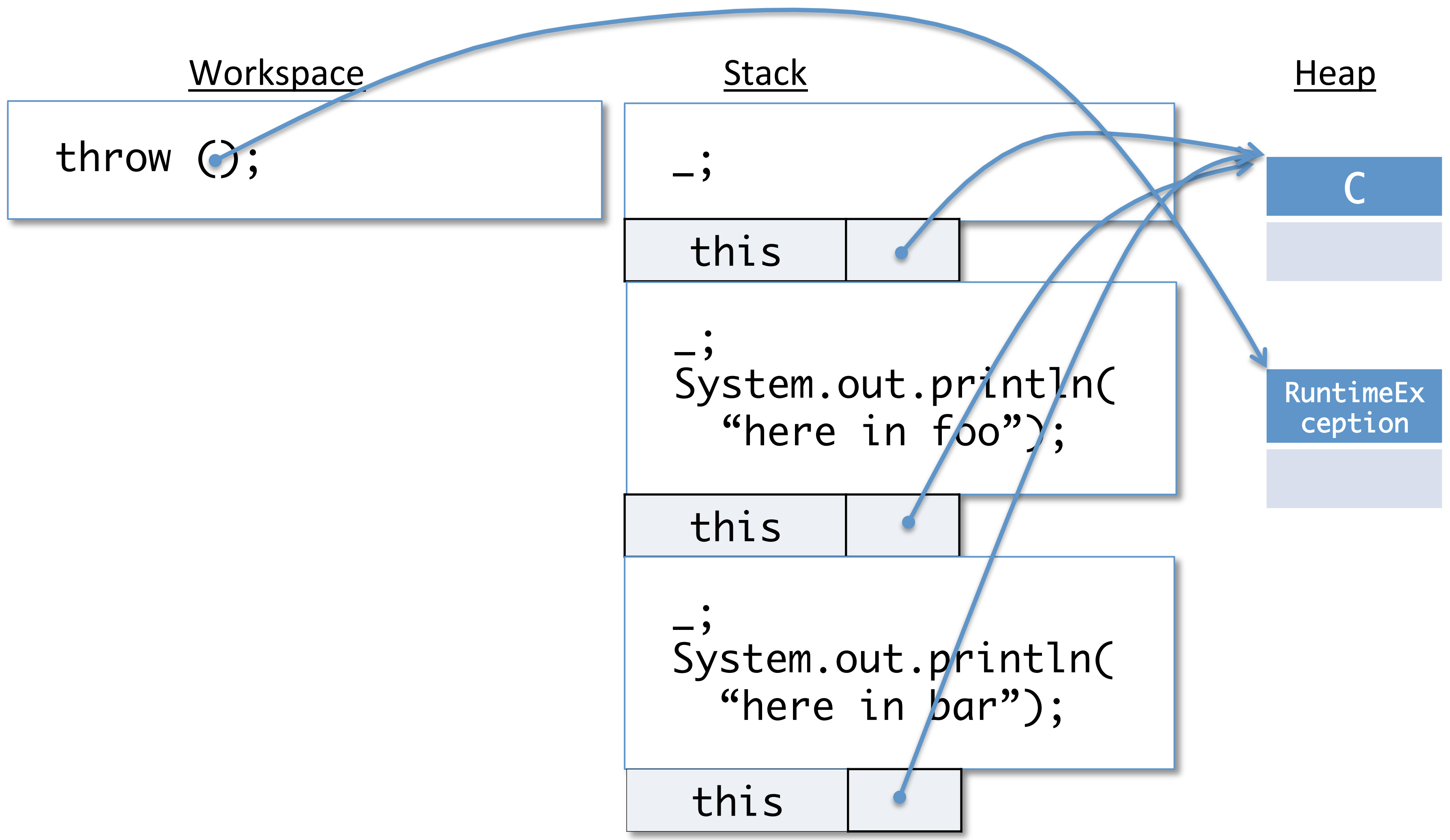
```
throw new
RuntimeException();
```

```
_;
```

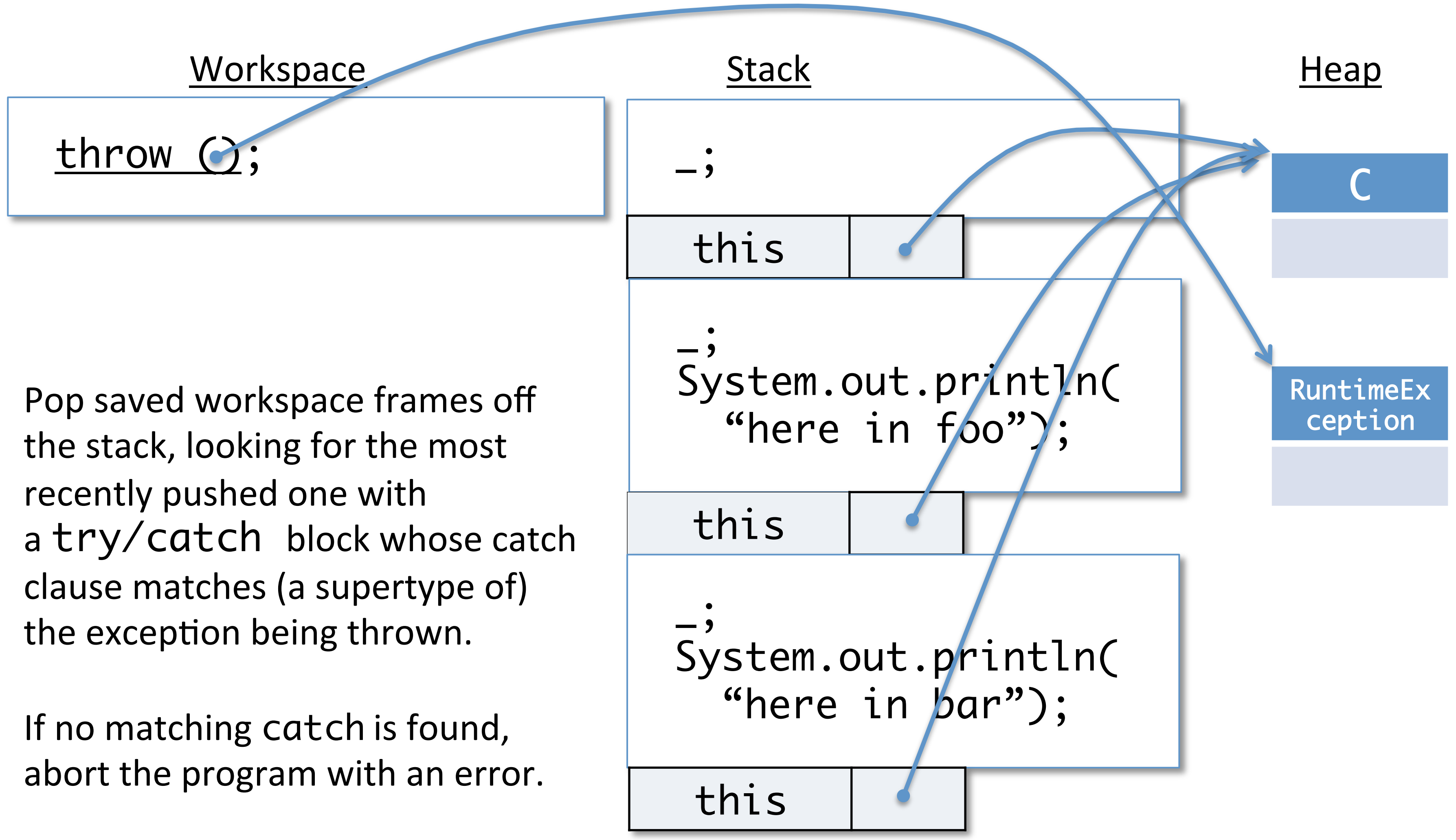
```
_;
System.out.println(
  "here in foo");
```

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

Abstract Stack Machine



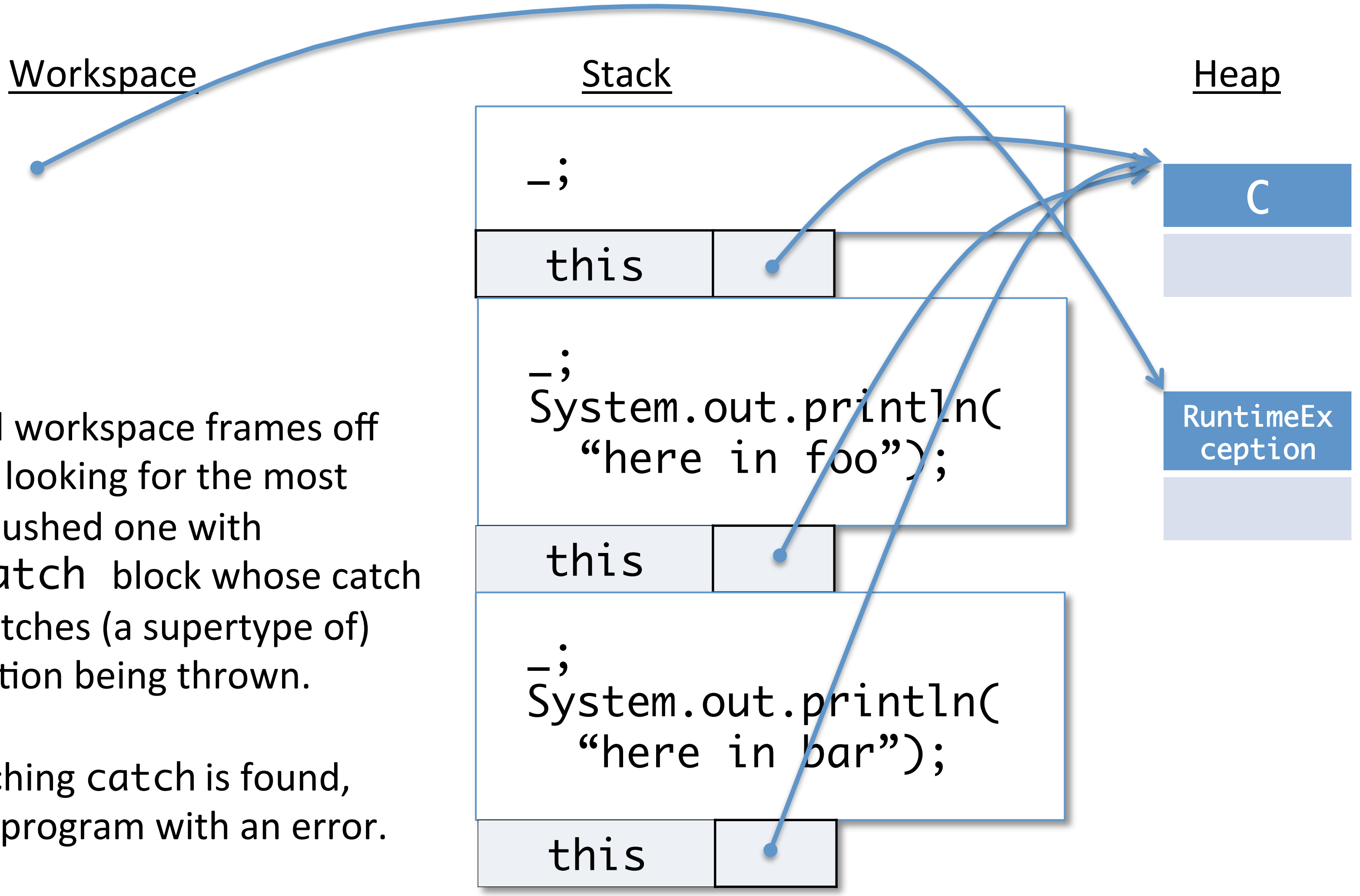
Abstract Stack Machine



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.

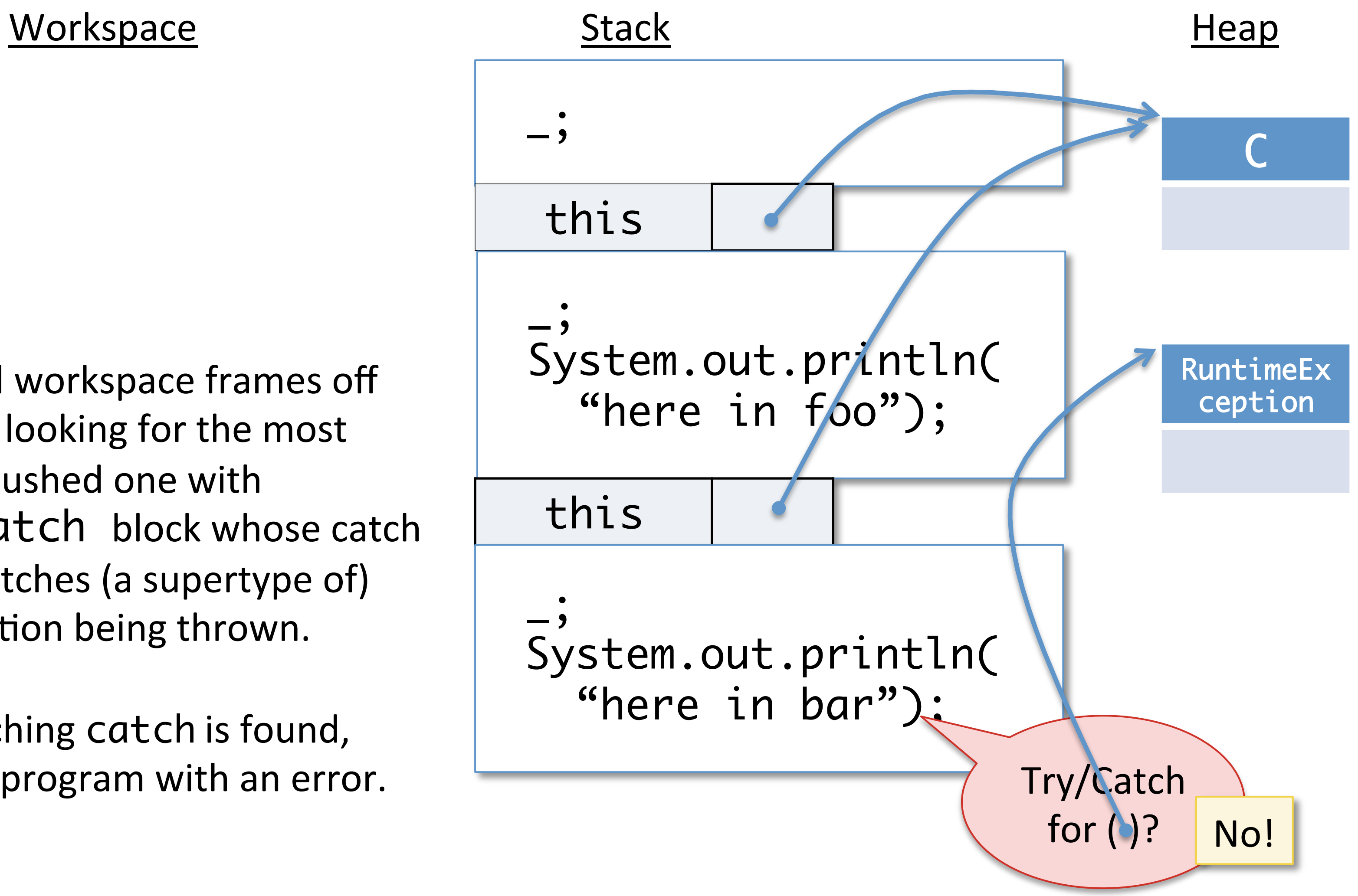
Abstract Stack Machine



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.

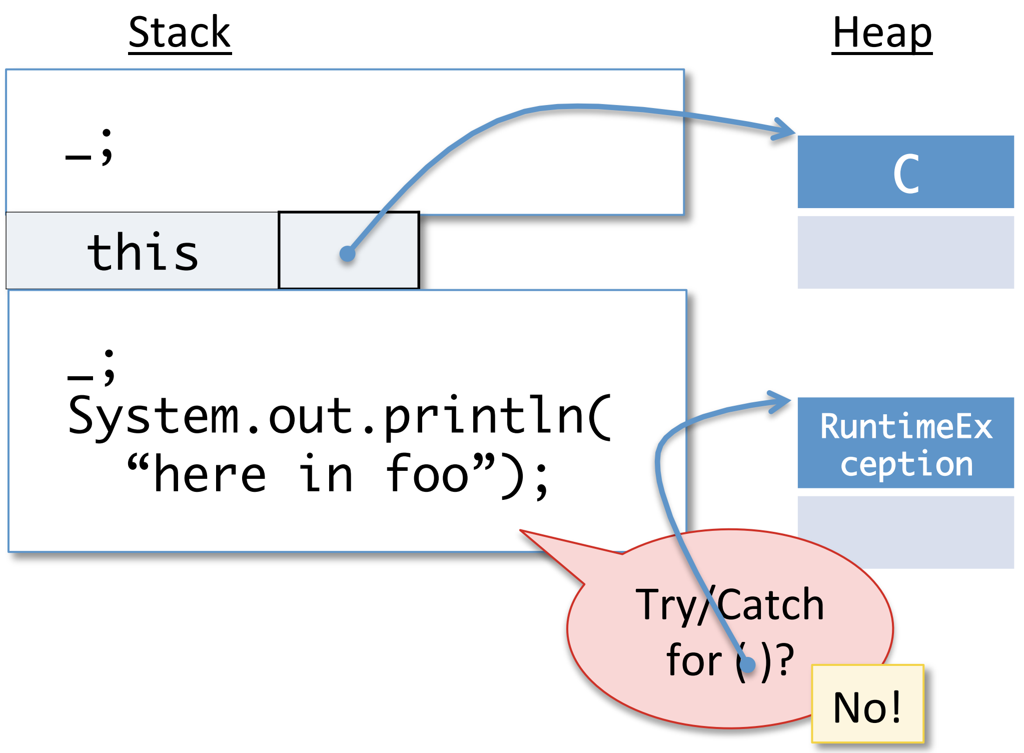
Abstract Stack Machine



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

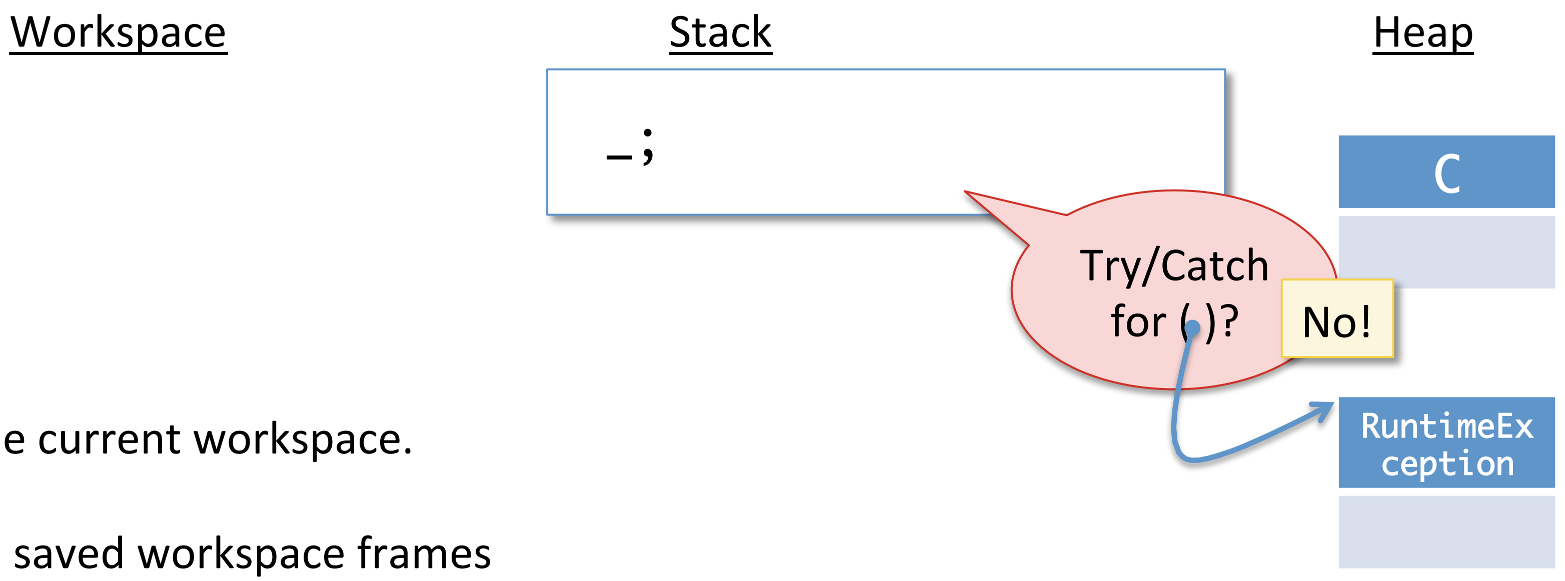


Discard the current workspace.

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



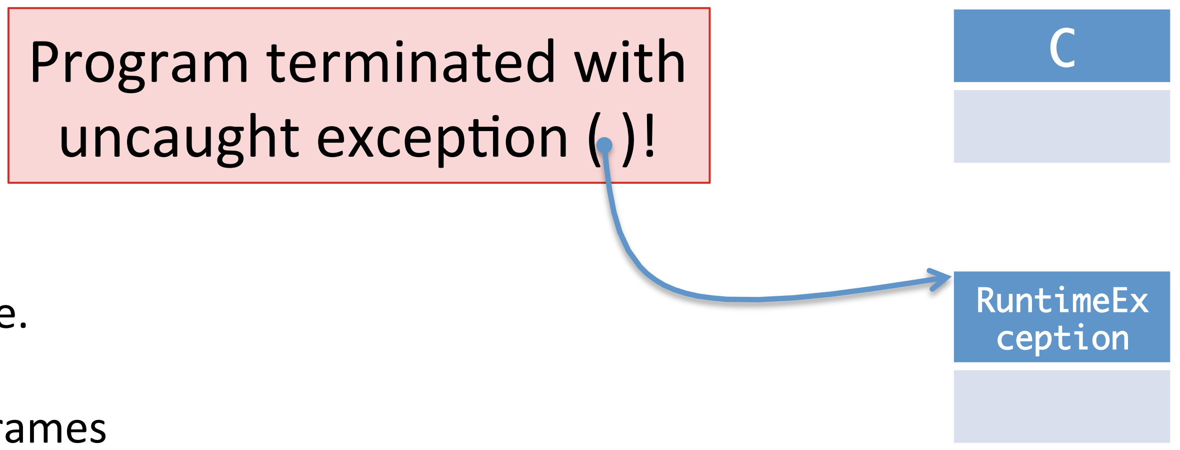
Discard the current workspace.

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



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

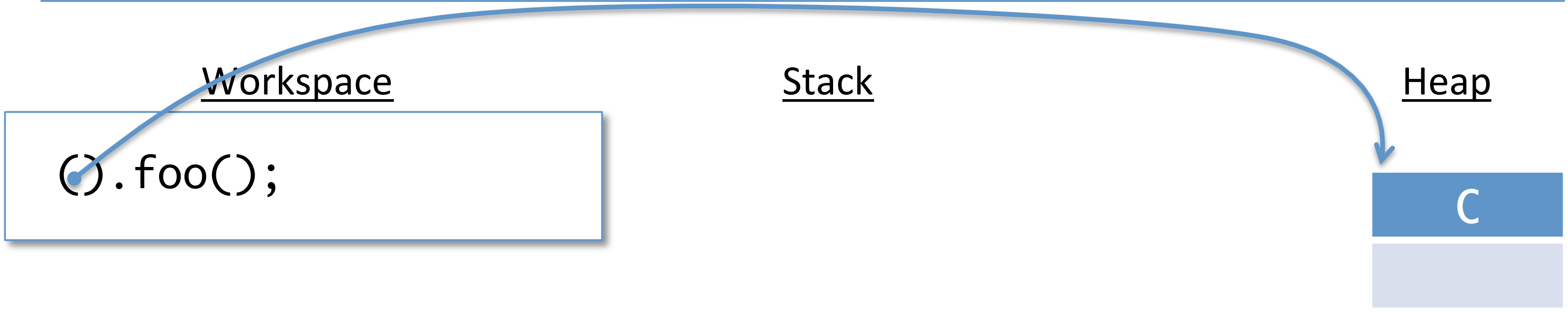
Workspace

Stack

Heap

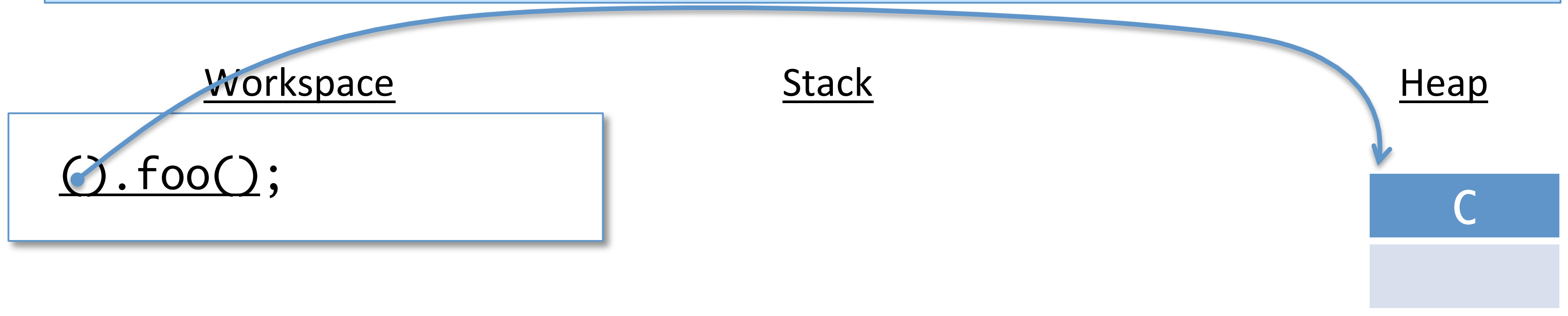
```
(new C()).foo();
```

Abstract Stack Machine



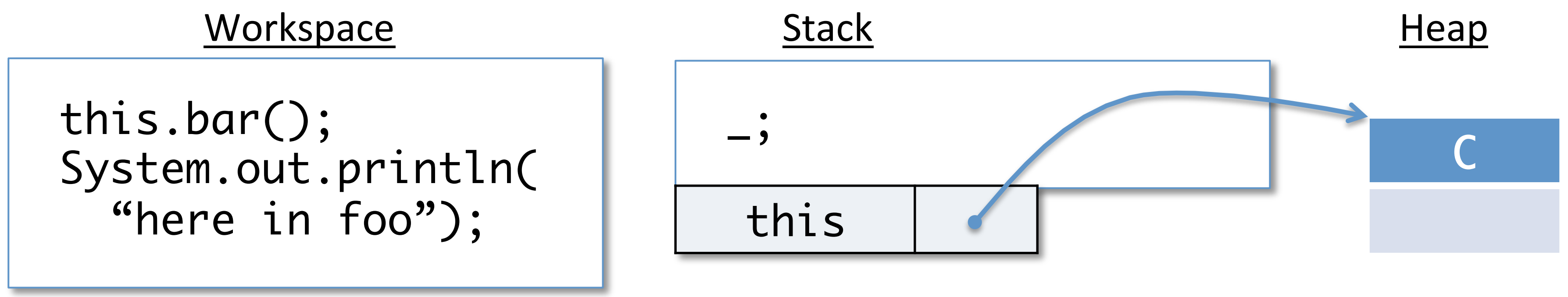
Allocate a new instance of C in the heap.

Abstract Stack Machine



```
().foo();
```

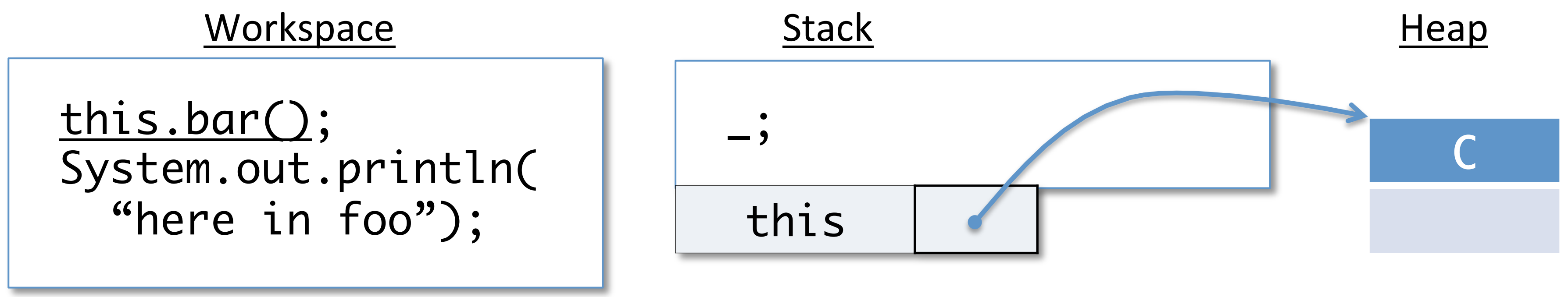
Abstract Stack Machine



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

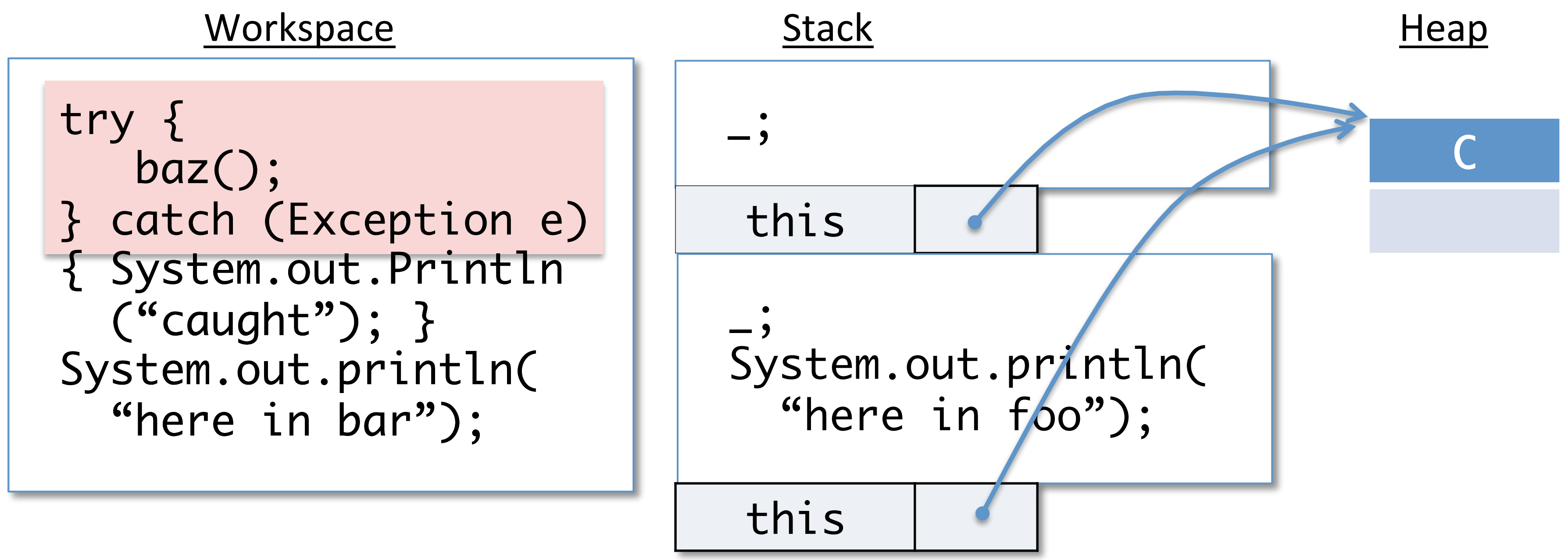
Save a copy of the current workspace in the stack, leaving a “hole”, written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack.

Abstract Stack Machine



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

Abstract Stack Machine



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

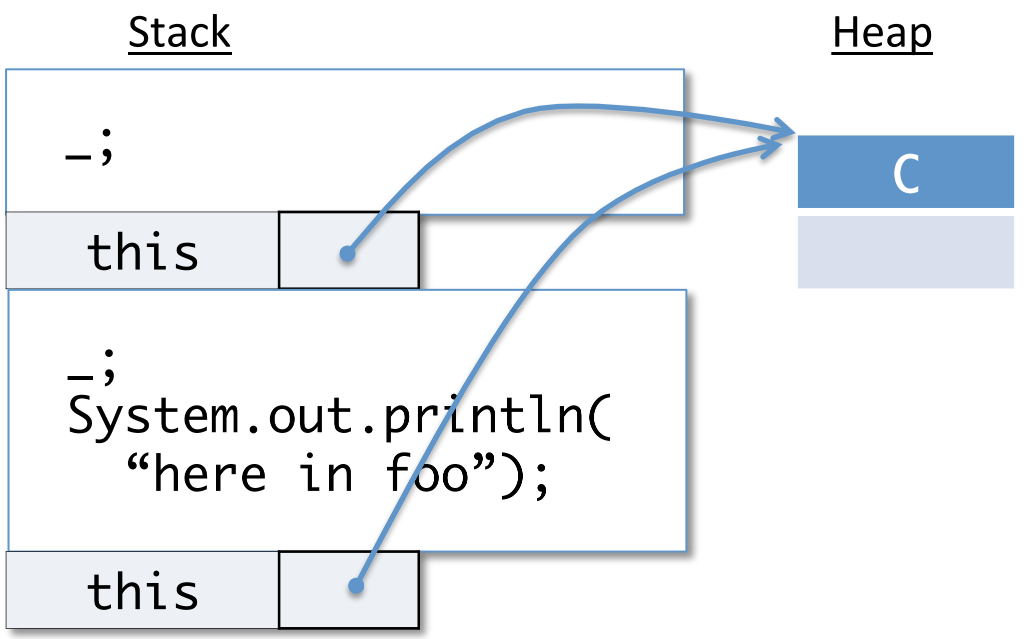
```
_;
```

```
_;
System.out.println(
  "here in foo");
```

Abstract Stack Machine

Workspace

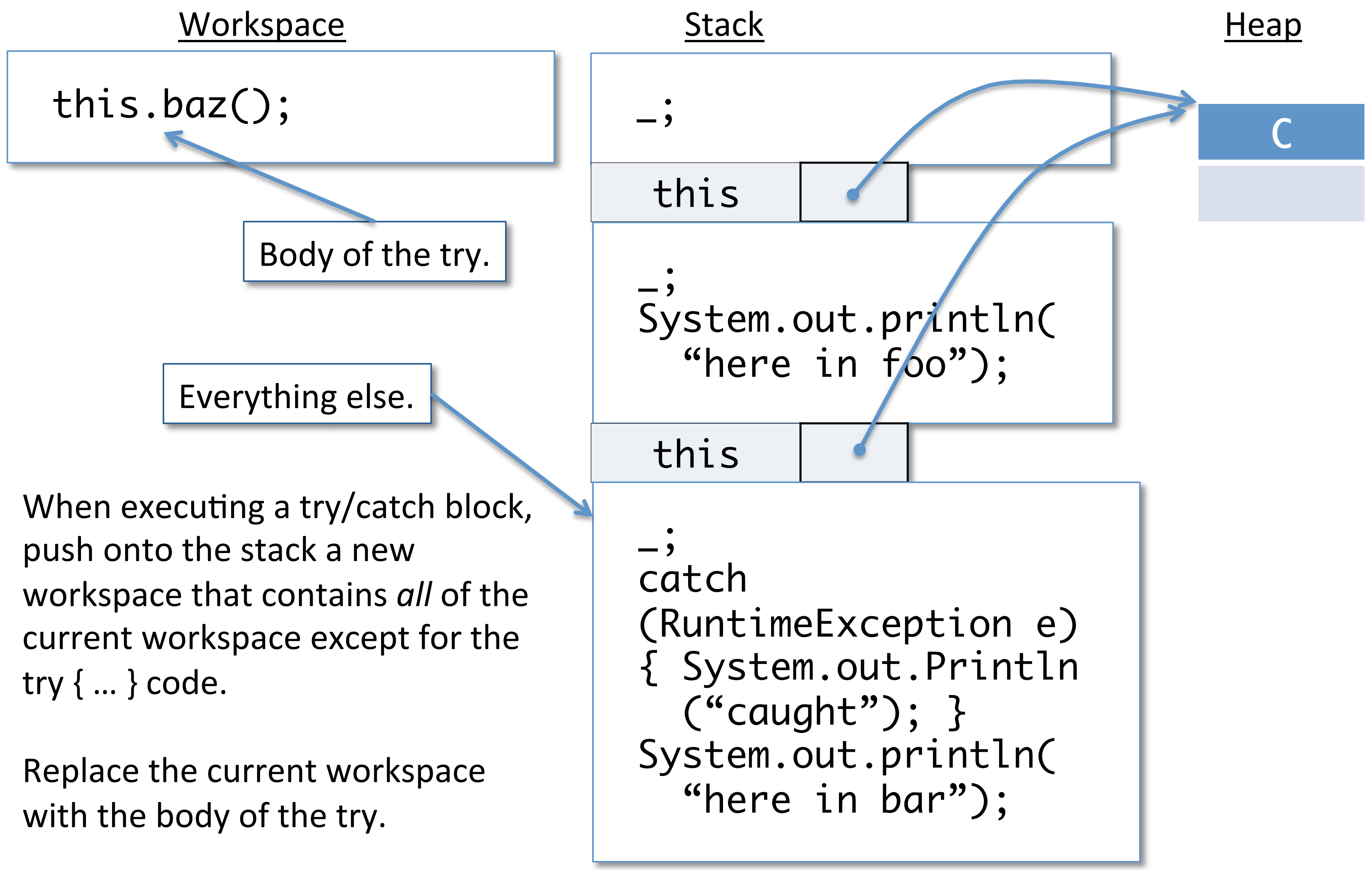
```
try {
   baz();
} catch (Exception e)
{ System.out.Println
  ("caught"); }
System.out.println(
  "here in bar");
```



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



Workspace:

```
this.baz();
```

Stack:

```
_;
```

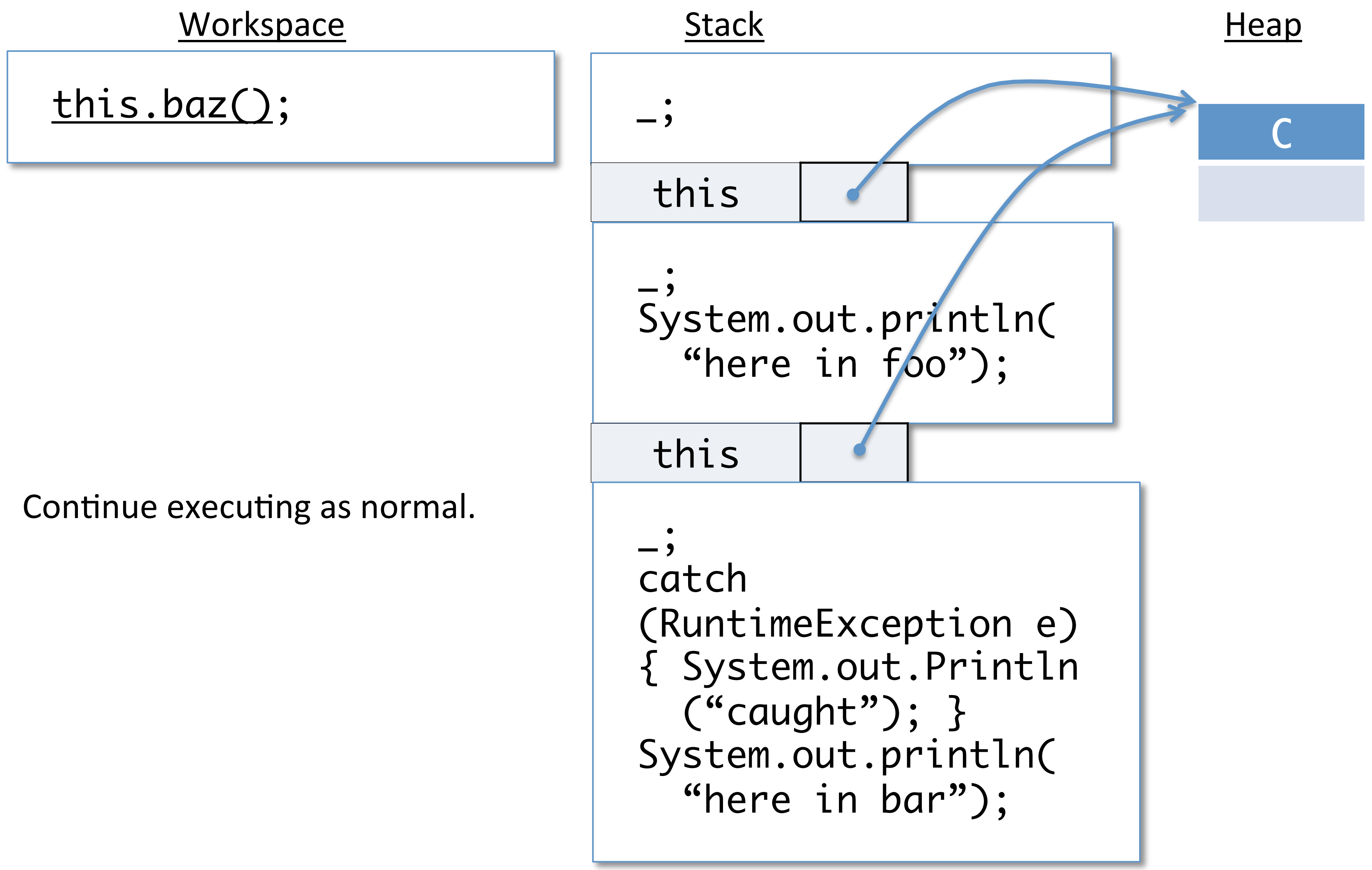
```
_;
System.out.println(
  “here in foo”);
```

```
_;
catch
(RuntimeException 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



Workspace:

```
this.baz();
```

Stack:

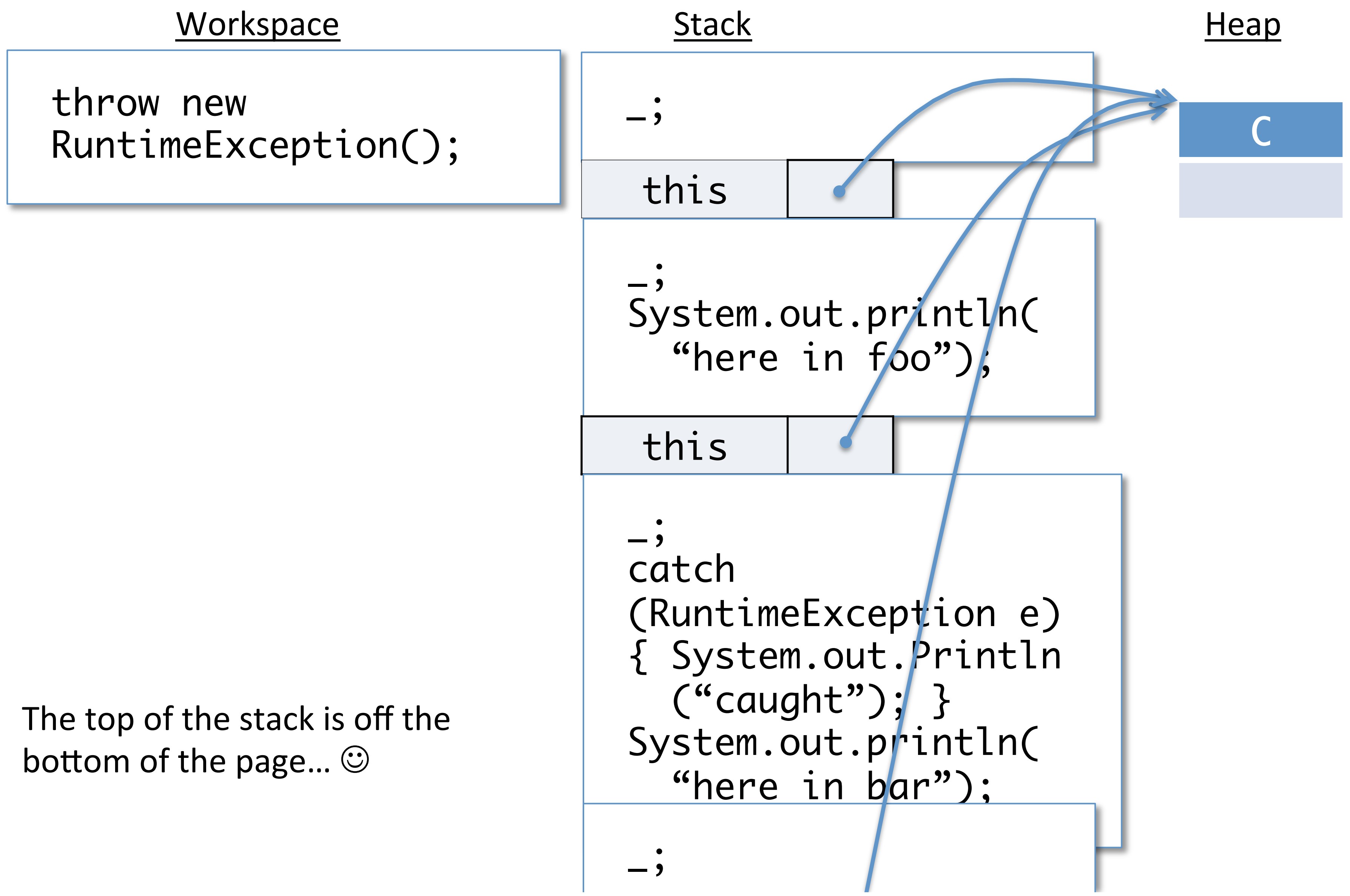
```
_;
```

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

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

Continue executing as normal.

Abstract Stack Machine



```
throw new
RuntimeException();
```

```
_;
```

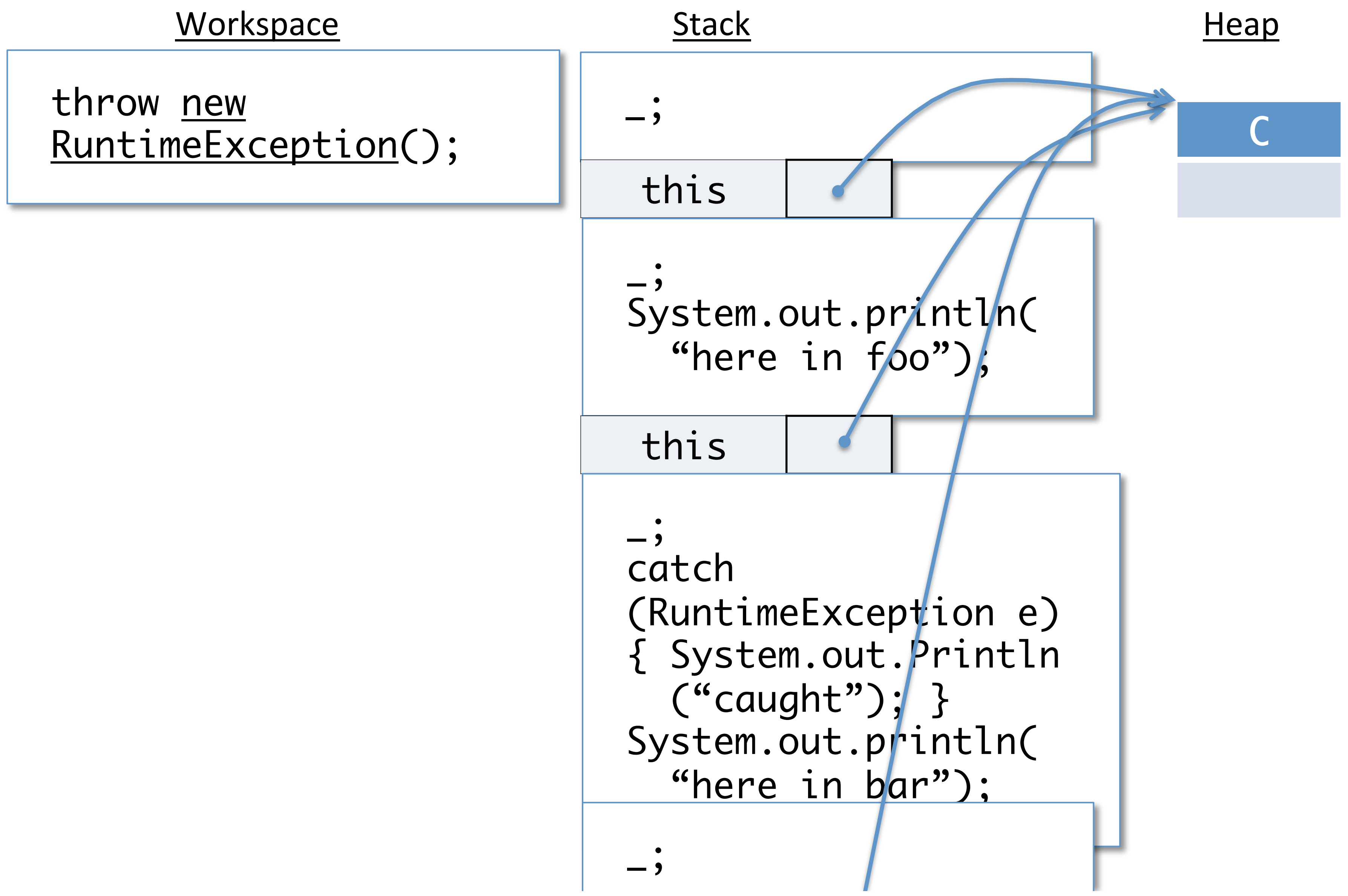
```
_;
System.out.println(
  “here in foo”);
```

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

```
_;
```

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

Abstract Stack Machine



```
throw new
RuntimeException();
```

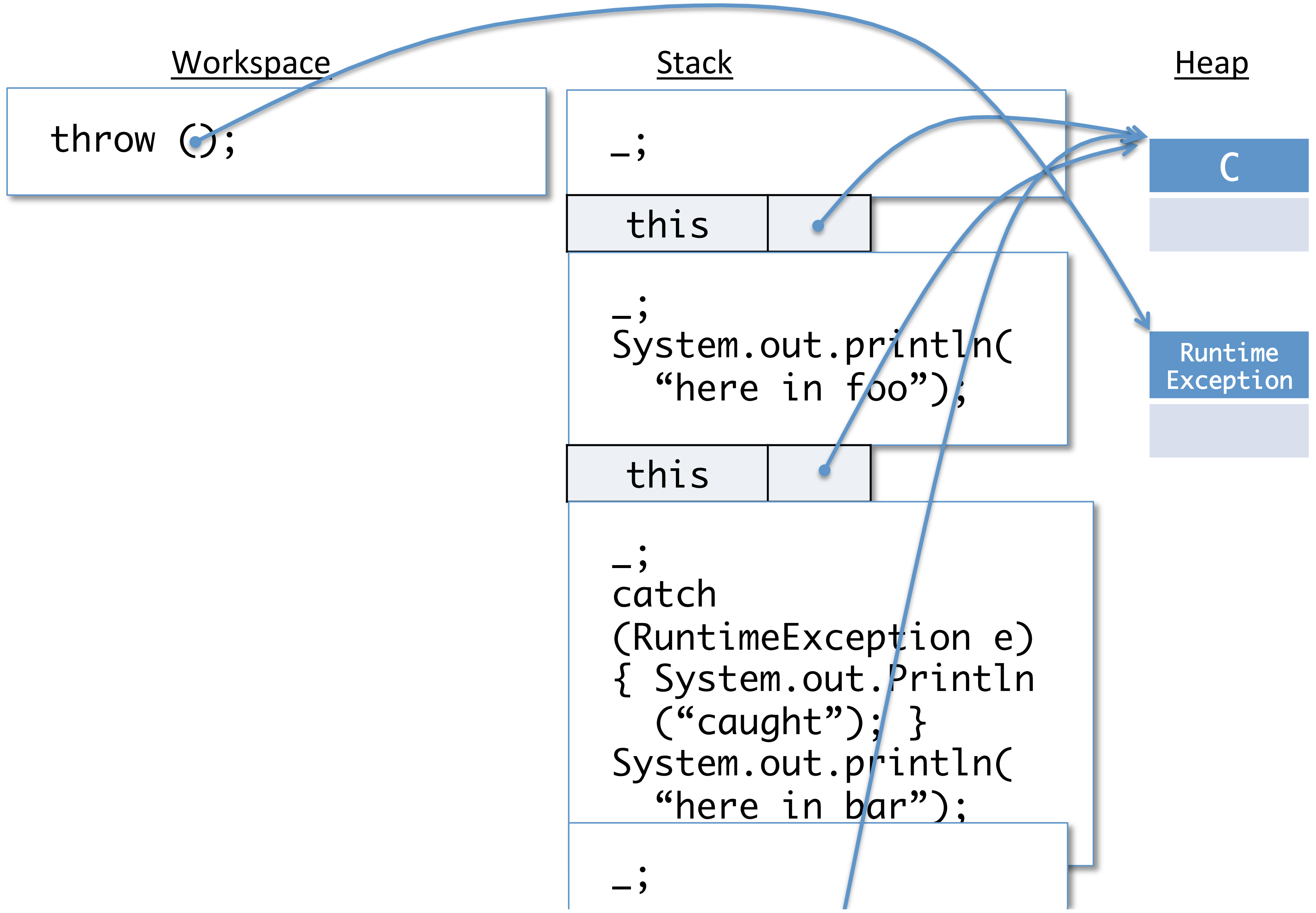
```
_;
```

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

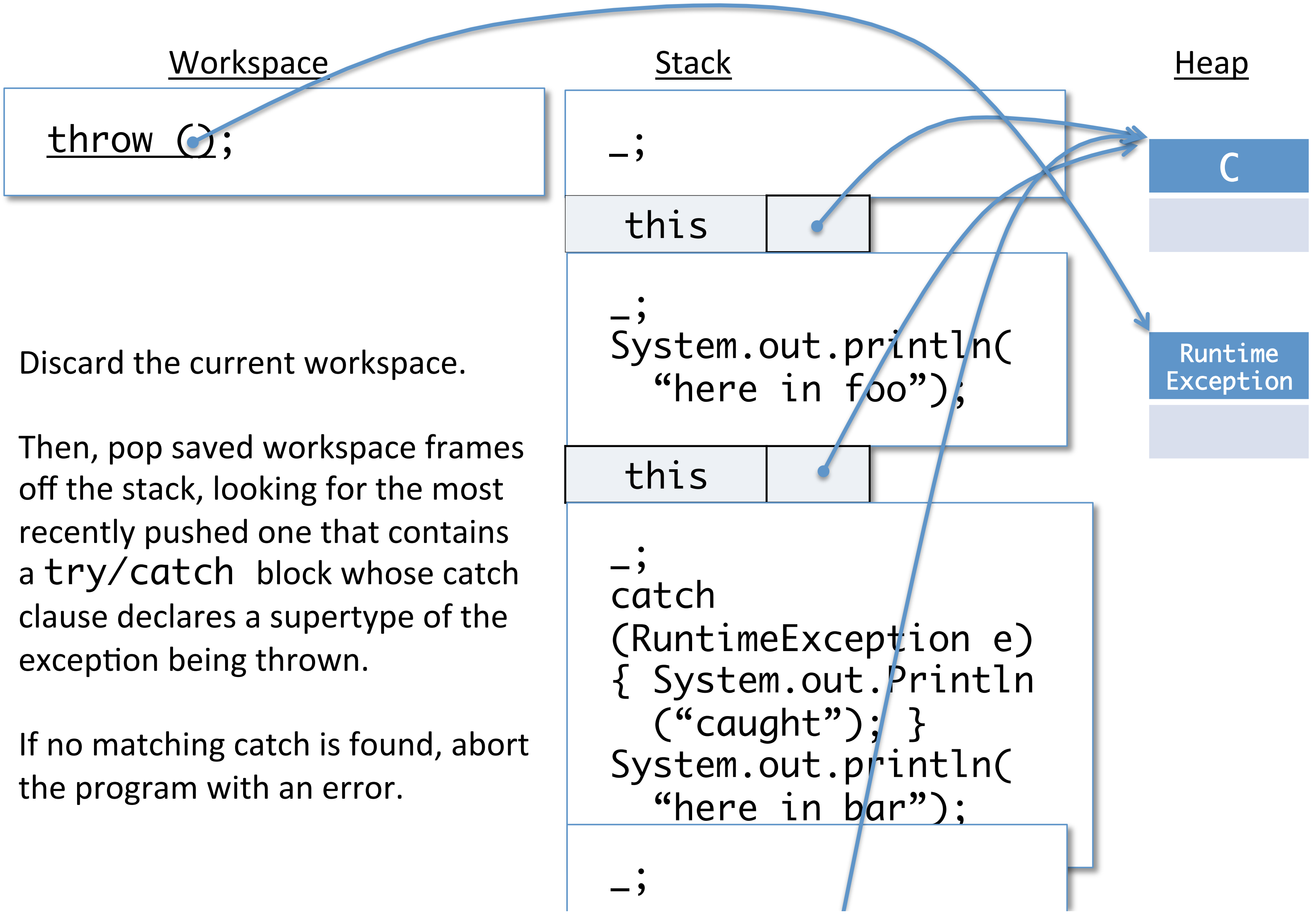
```
_;
catch
(RuntimeException e)
{ System.out.Println
  (“caught”); }
System.out.println(
  “here in bar”);
```

```
_;
```

Abstract Stack Machine



Abstract Stack Machine



```
throw ( );
```

```
_;
```

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

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

```
_;
```

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.

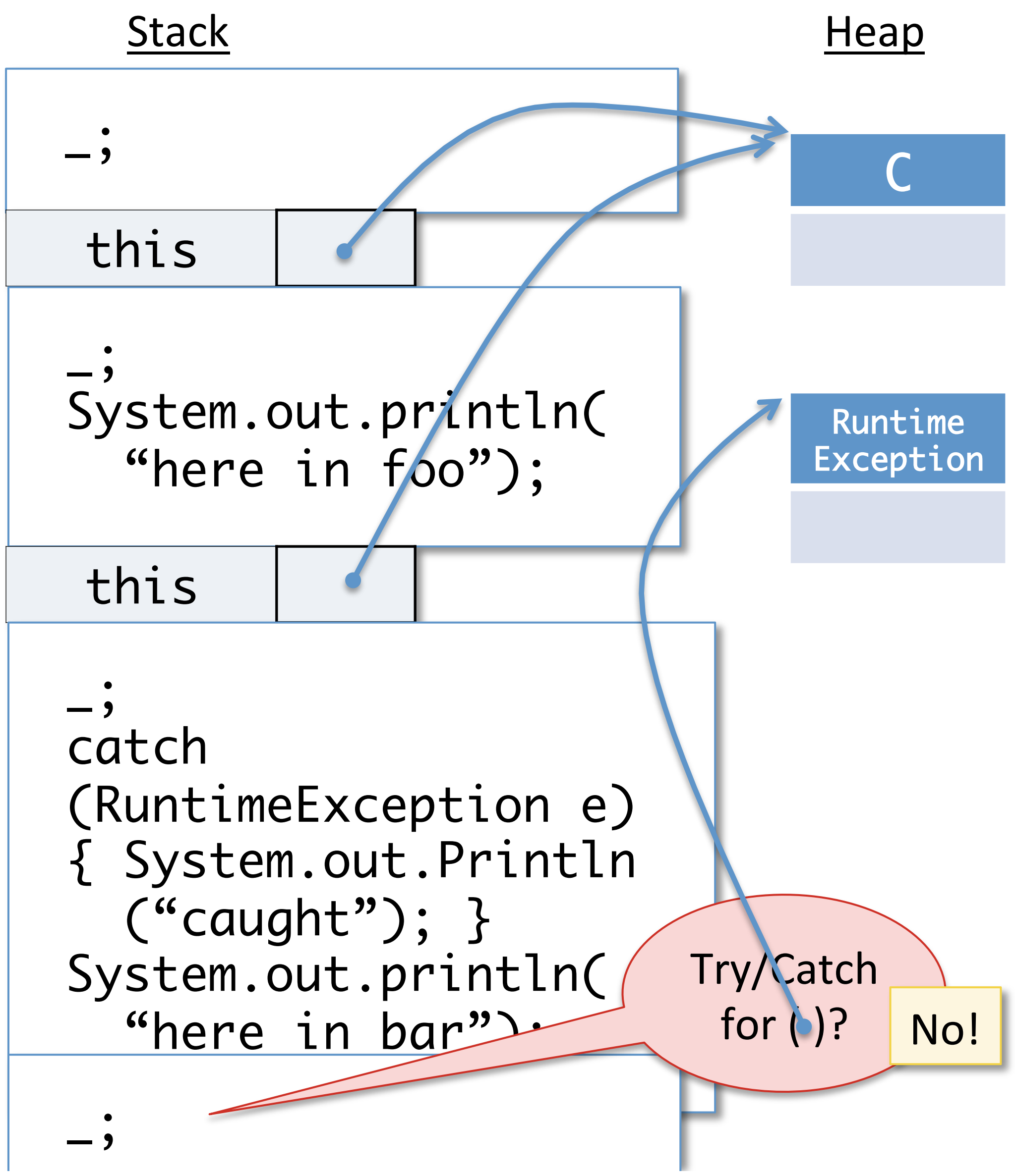
Abstract Stack Machine

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.

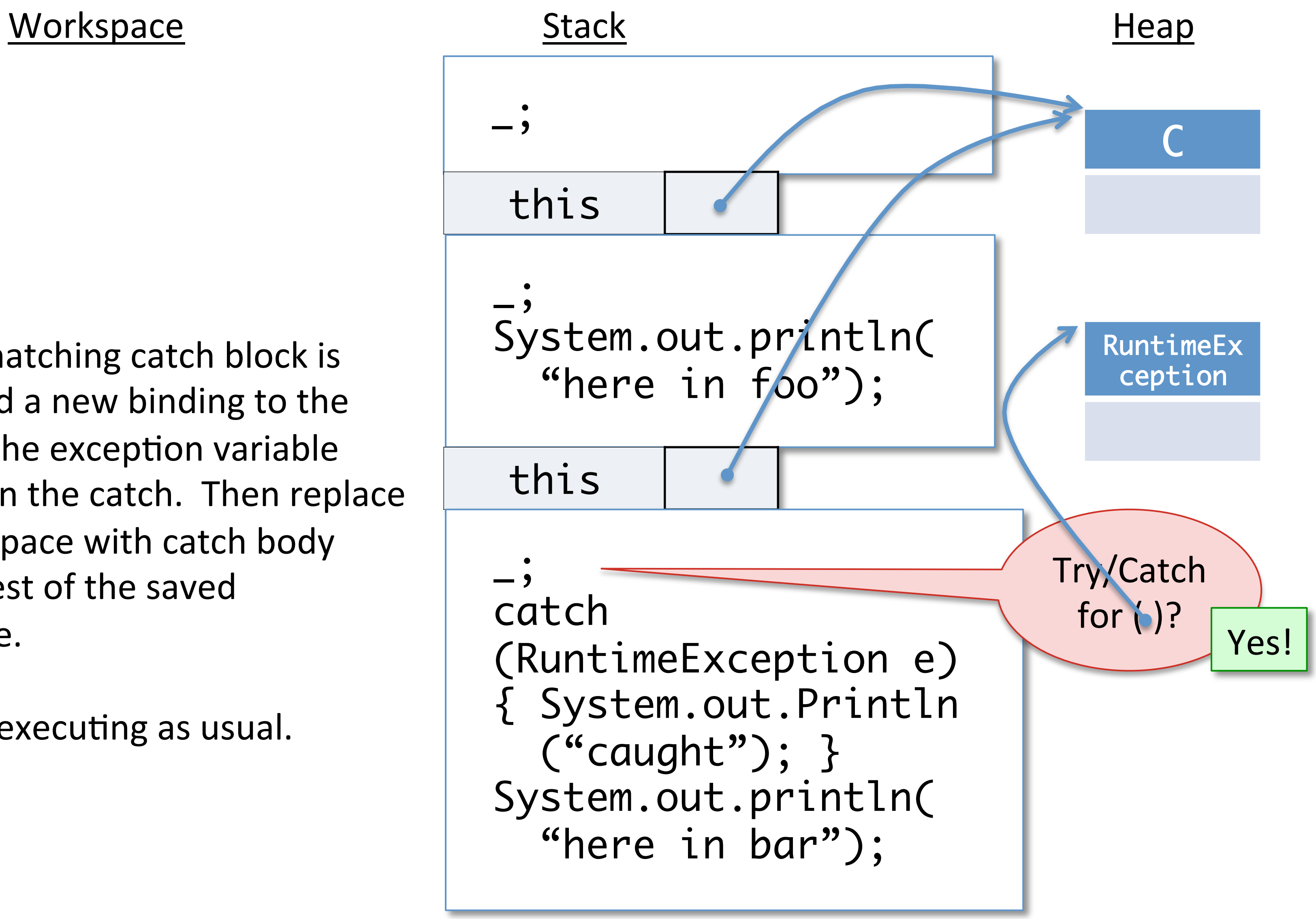
If no matching catch is found, abort the program with an error.



Abstract Stack Machine

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.

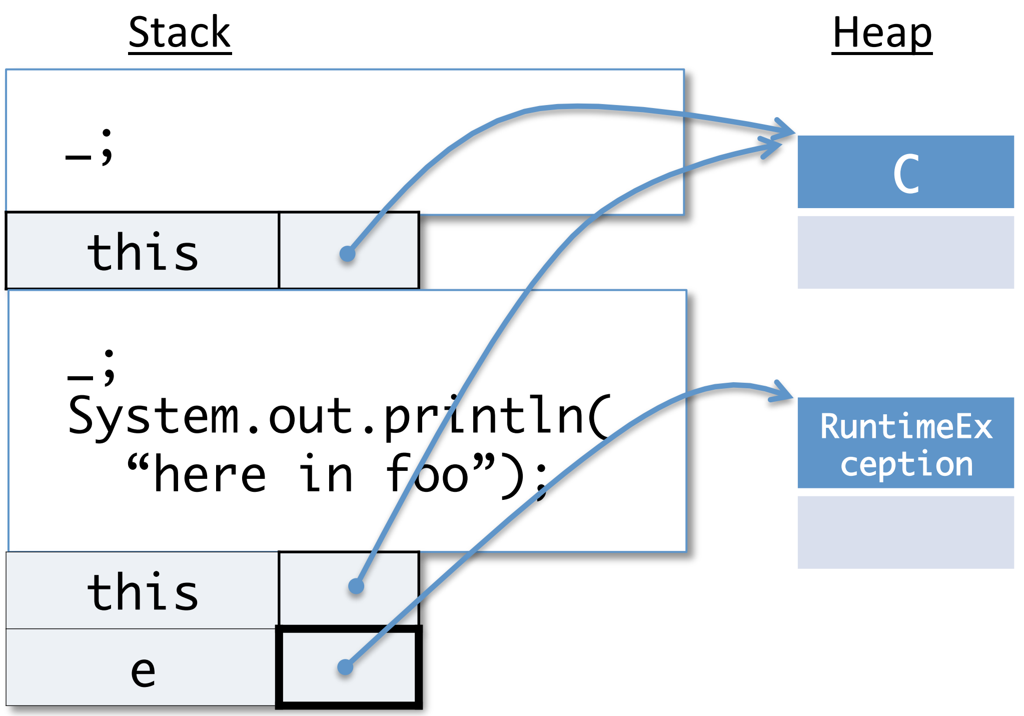
Continue executing as usual.



Abstract Stack Machine

Workspace

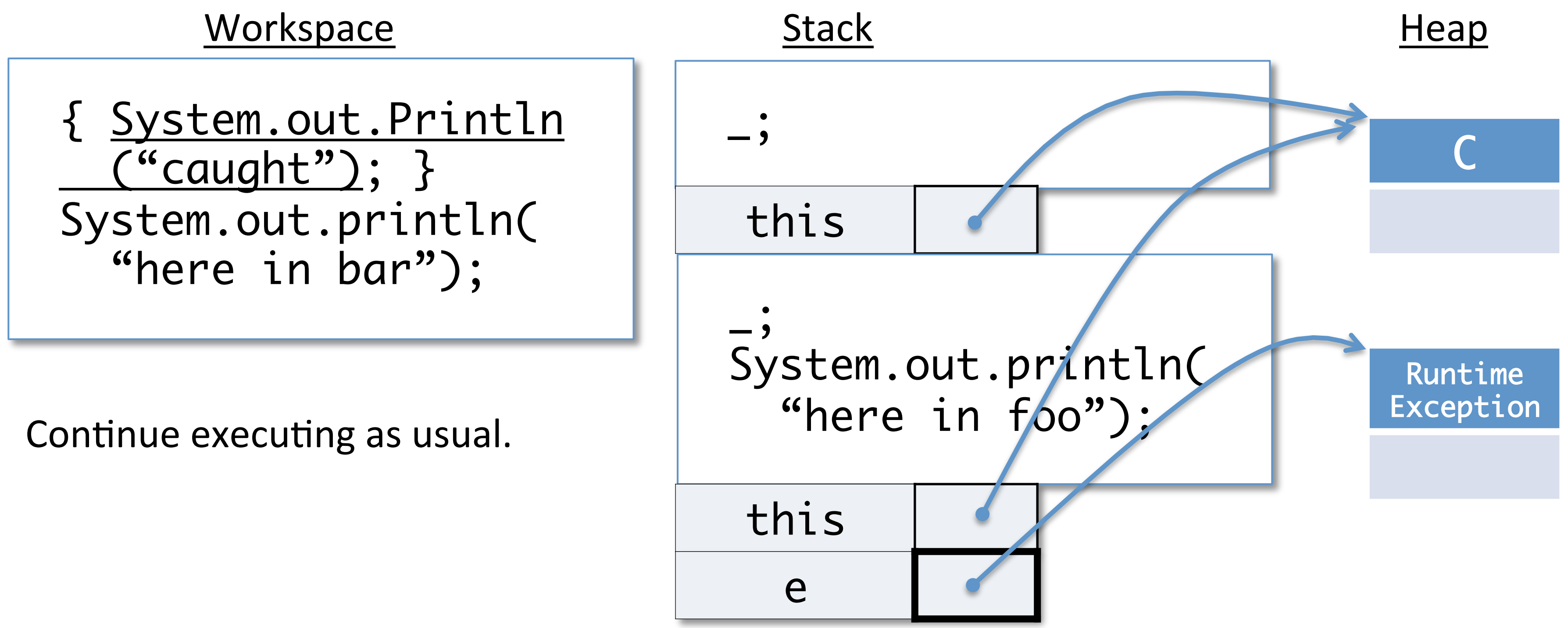
```
{ System.out.Println
  ("caught"); }
System.out.println(
  "here in bar");
```



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



Continue executing as usual.

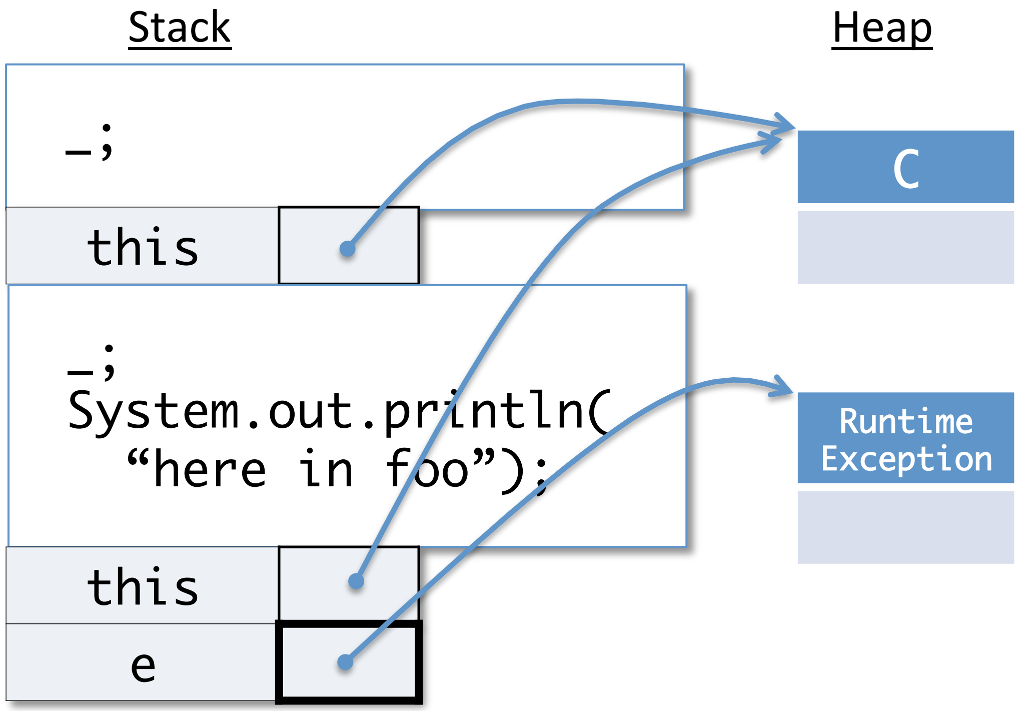
Abstract Stack Machine

Workspace

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

Continue executing as usual.

Console
caught

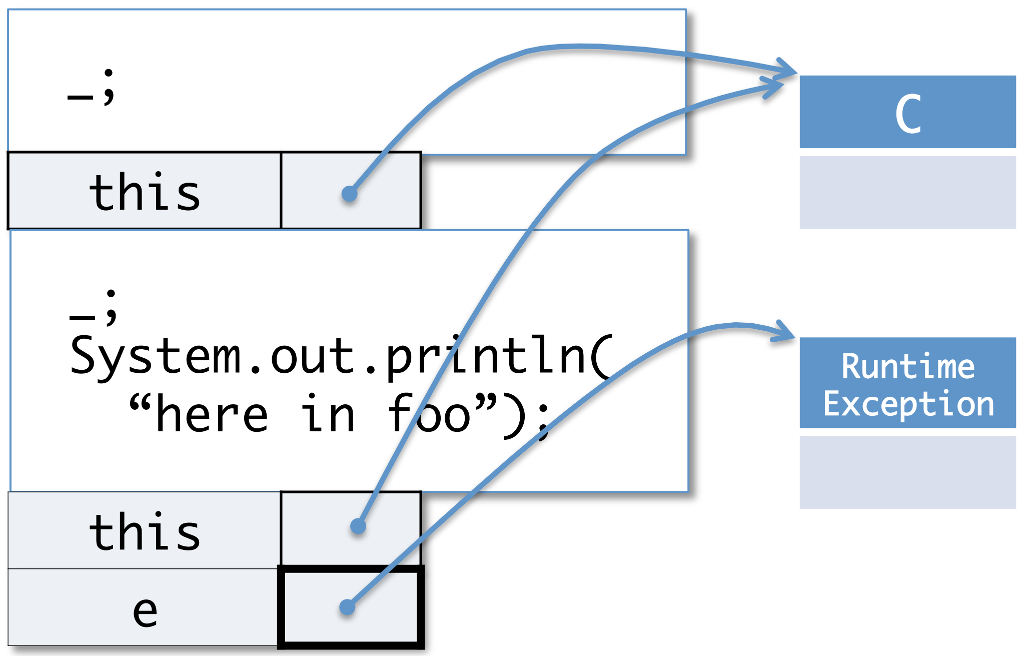


Abstract Stack Machine

Workspace

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

Stack Heap

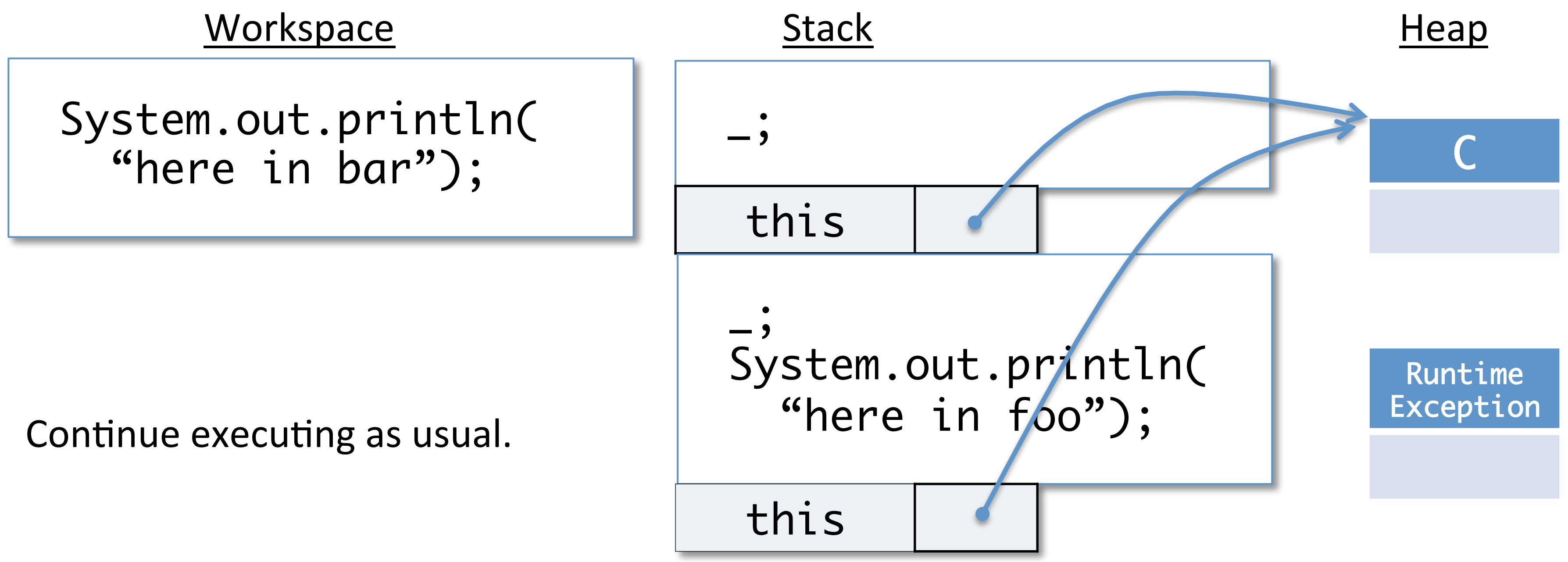


We’re sweeping a few details about lexical scoping of variables under the rug – the scope of e is just the body of the catch, so when that is done, e must be popped from the stack.

Console

```
caught
```

Abstract Stack Machine

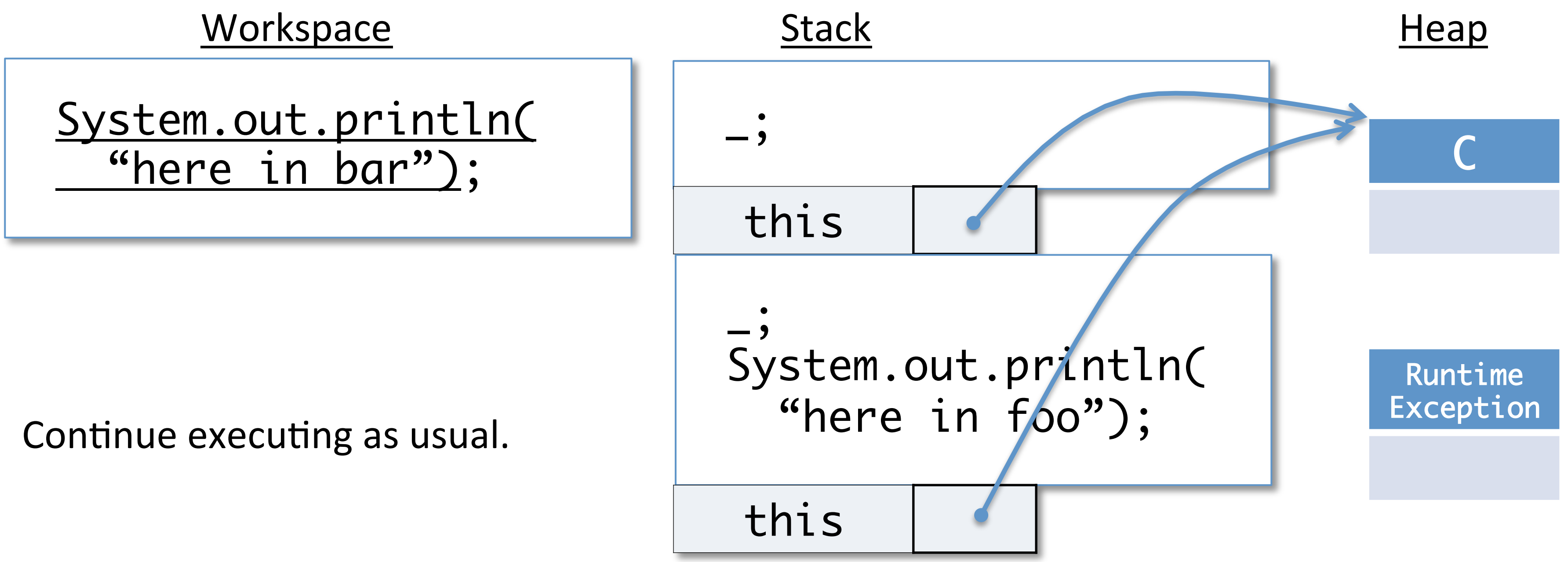

Workspace
System.out.println("here in bar");
Stack
_;
this
_;
System.out.println("here in foo");
this
Heap
C
Runtime Exception

Continue executing as usual.

Console

```
caught
```

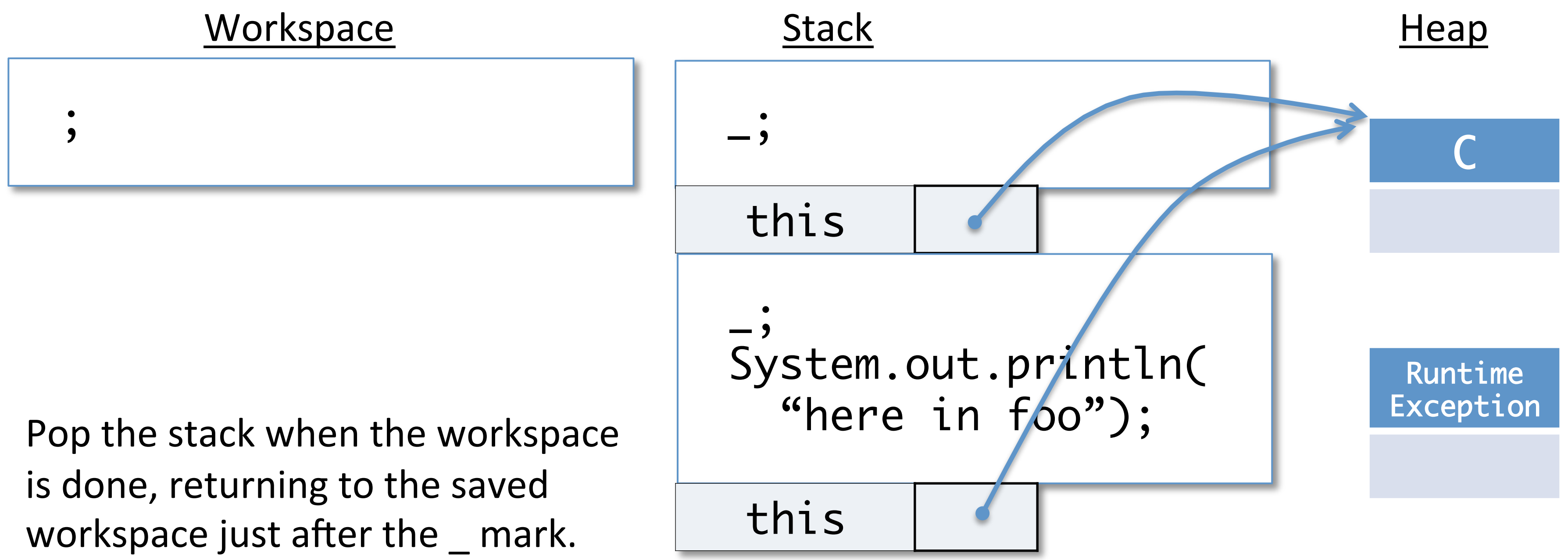
Abstract Stack Machine


Workspace
Stack
Heap
System.out.println("here in bar");
_;
this
_;
System.out.println("here in foo");
this
C
Runtime Exception

Continue executing as usual.

Console
caught

Abstract Stack Machine



```
;
```

```
_;
```

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

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

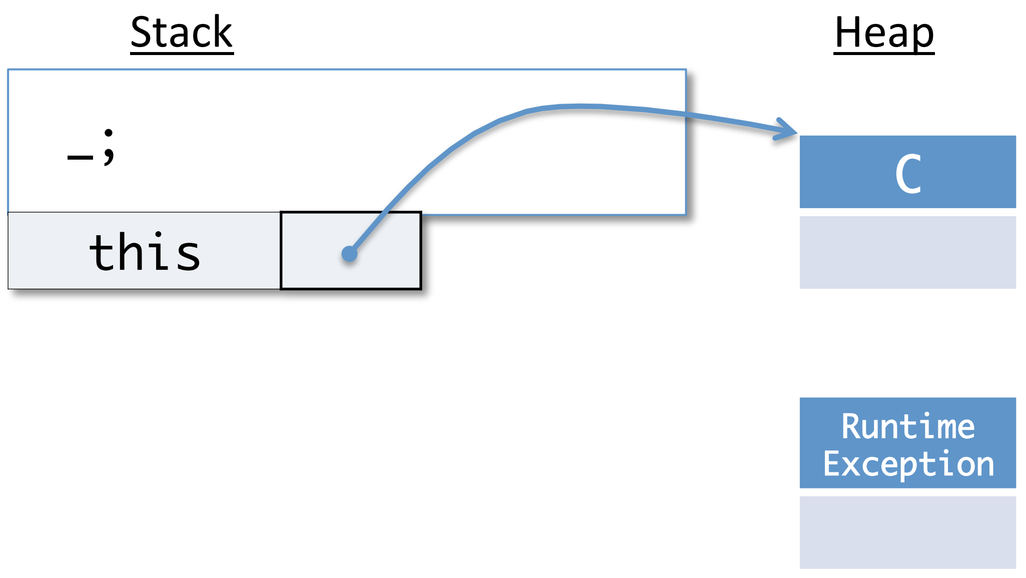
Console

```
caught
here in bar
```

Abstract Stack Machine

Workspace

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

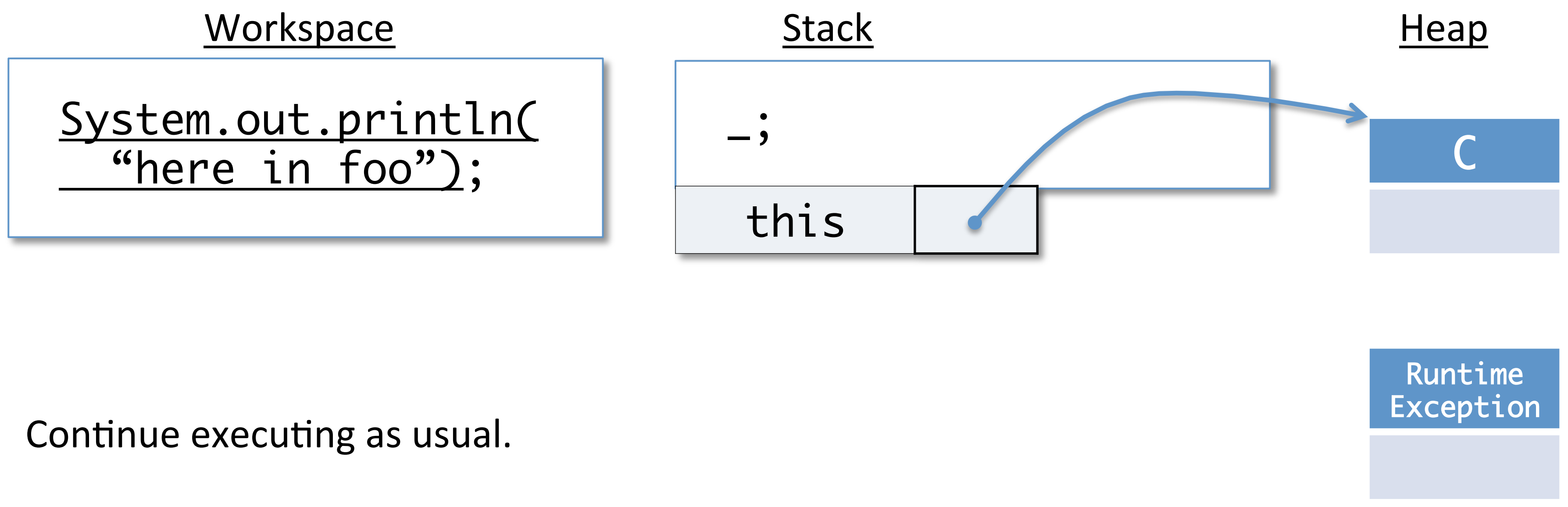


Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine



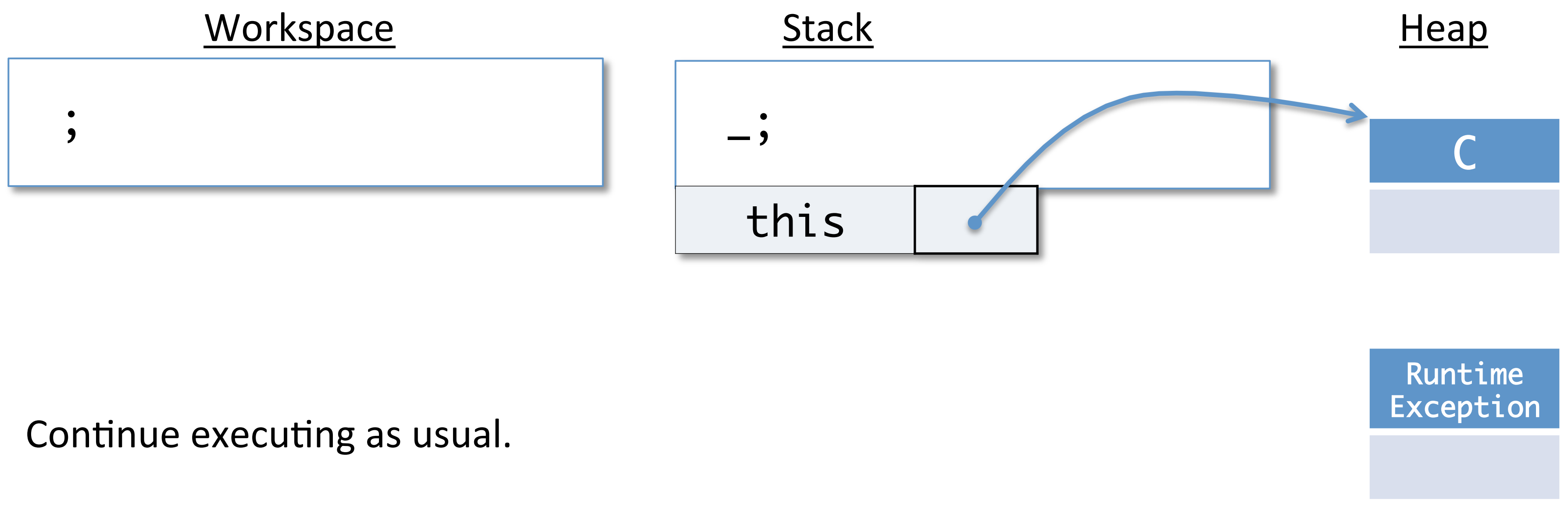
```
System.out.println(
  "here in foo");
```

Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine



Continue executing as usual.

Console

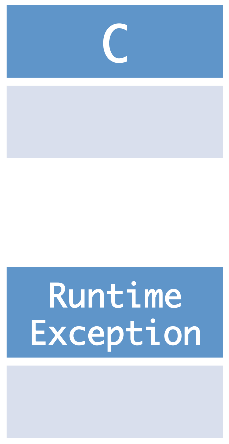
```
caught
here in bar
here in foo
```

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!