Programming Languages and Techniques (CIS1200)

Lecture 29

Enumerations and Iteration

Chapter 25

Announcements (1)

- HW07: PennPals
  - Programming with Java Collections
  - Due Tuesday, November 12 at 11.59pm

- Midterm 2: Friday, November 15th
  - Similar to Midterm 1
  - Content: HW 4 – 6, Chapters 11-21 (Java Arrays) and Chapter 32 (Encapsulation) of lecture notes

Announcements (2)

- Midterm 2: Friday, November 15
  - Coverage: up to Monday, Oct. 28 (Chapters 11-21, 32)
  - During lecture (001 @ 10.15am, 002 @ noon)
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, closed notes
  - Review Material
    - old exams on the web site ("schedule" tab)
  - Review Session
    - TBA

Review: Overriding equals

Correct Implementation: Point

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        { return true; }
    if (obj == null)
        { return false; }
    if (getClass() != obj.getClass())
        { return false; }
    Point other = (Point) obj;
    if (x != other.x)
        { return false; }
    if (y != other.y)
        { return false; }
    return true;
}
```

Check whether obj is a Point

"*dynamic cast*" or "*type cast*"
"*downcast*" or "*coercion*"

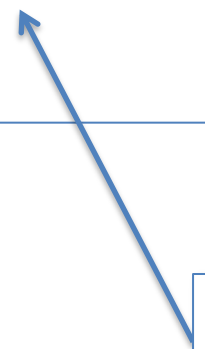
The class cast expression "(T)e" is a runtime test of the dynamic class of of e.
If T is not a subtype of the dynamic class, then a ClassCastException is thrown.
The static type of the expression "(T)e" is T.

Compatibility with compareTo

- For classes that implement the Comparable<E> interface, the equals and compareTo methods should agree:
  - o.compareTo(p) == 0 exactly when o.equals(p)

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass())
      return false;
    Point point = (Point) o;
    return (this.compareTo(point) == 0);

    }
```



Implement equals by using compareTo.

Overriding Equality in Practice

- This is all a bit complicated!
- Fortunately, some tools (e.g. IntelliJ) can autogenerate equality methods of the kind we developed.
  - Just need to specify which fields should be taken into account.
  - (and you should know why some comparisons use == and some use .equals)

Iterating over collections

iterators, while, for, for-each loops

Iterator and Iterable

```
interface Iterator<E> {
    boolean hasNext();
    E next();
    default void delete();   // optional
    default void forEachRemaining(..); // optional
}
```

```
interface Iterable<E> {
    Iterator<E> iterator();
}
```

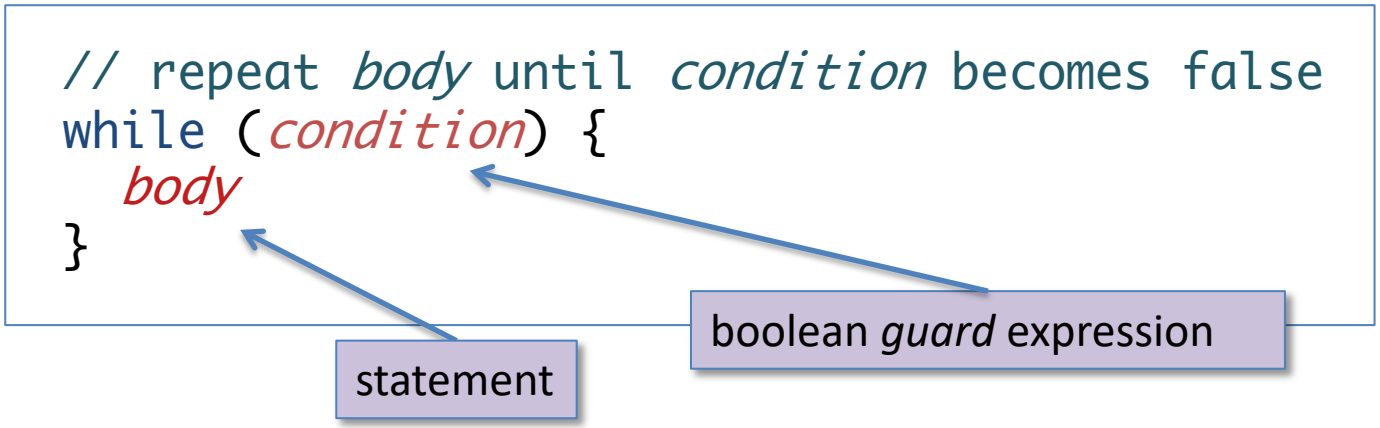
```
interface Collection<E> extends Iterable<E> …
```

Challenge: given a List<Book> how would you add each book's data to a catalogue using an iterator?

While Loops

syntax:

```
// repeat body until condition becomes false
while (condition) {
   body
}
```



example:

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
    Book book = iter.next();
    catalogue.addInfo(book);
    numBooks = numBooks+1;
}
```

For Loops

syntax:

```
for (init-stmt; condition; next-stmt) {
  body
}
```

equivalent while loop:

```
init-stmt;
while (condition) {
  body
  next-stmt;
}
```

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
for (Iterator<Book> iter = shelf.iterator();
     iter.hasNext();) {
   Book book = iter.next();
   catalogue.addInfo(book);
   numBooks = numBooks+1;
}
```

For-each Loops

syntax:

```
// repeat body for each element in collection
for (type var : coll) {
  body
}
```

element type E

Array of E or instance of Iterable<E>

example:

```
List<Book> shelf = …  // create a list of books

// iterate through the elements on a shelf
for (Book book : shelf) {
    catalogue.addInfo(book);
    numBooks = numBooks+1;
}
```

For-each Loops (cont'd)

Another example:

```
int[] arr = …  // create an array of ints

// count the non-null elements of an array
for (int elt : arr) {
    if (elt != 0) cnt = cnt+1;
}
```

For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).

29: What is printed by iteratorExample()?

0

sumElts = 0 numElts = 0

0%

sumElts = 3 numElts = 2

0%

sumElts = 10 numElts = 3

0%

NullPointerException

0%

something else

0%

Iterator example

```
public static void iteratorExample() {
    List<Integer> nums = new LinkedList<>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    Iterator<Integer> iter =
        nums.iterator();
    while (iter.hasNext()) {
      Integer v = iter.next();
      sumElts = sumElts + v;
      numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

What is printed by iteratorExample()?

1. sumElts = 0 numElts = 0
2. sumElts = 3 numElts = 2
3. sumElts = 10 numElts = 3
4. NullPointerException
5. Something else

Answer: 3

29: What is printed by nextNextExample()?

0

sumElts = 0 numElts = 0

0%

sumElts = 3 numElts = 2

0%

sumElts = 8 numElts = 2

0%

NullPointerException

0%

something else

0%

Another Iterator example

```
public static void nextNextExample() {
    List<Integer> nums = new LinkedList<>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int sumElts = 0;
    int numElts = 0;
    Iterator<Integer> iter =
        nums.iterator();
    while (iter.hasNext()) {
      Integer v = iter.next();
      sumElts = sumElts + v;
      v = iter.next();
      numElts = numElts + v;
    }
    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

What is printed by nextNextExample()?

1. sumElts = 0 numElts = 0
2. sumElts = 3 numElts = 2
3. sumElts = 8 numElts = 2
4. NullPointerException
5. Something else

Answer: 5 NoSuchElementException

For-each version

```
public static void forEachExample() {
    List<Integer> nums = new LinkedList<>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    for (Integer v : nums) {
      sumElts = sumElts + v;
      numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

Enumerations

Enumerations (a.k.a. Enum Types)

- Java supports *enumerated* type constructors
  - Intended to represent constant data values

```
enum CommandType {
    CREATE, INVITE, JOIN, KICK, LEAVE, MESG, NICK
}
```

- Intuitively similar to a simple usage of OCaml datatypes
  - …but each language provides extra bells and whistles that the other does not

Enums with data

```
public enum ServerResponse {
    OKAY(200),
    INVALID_NAME(401),
    NO_SUCH_CHANNEL(402),
    NO_SUCH_USER(403),
    USER_NOT_IN_CHANNEL(404),
    USER_NOT_OWNER(406),
    JOIN_PRIVATE_CHANNEL(407),
    INVITE_TO_PUBLIC_CHANNEL(408),
    NAME_ALREADY_IN_USE(500),
    CHANNEL_ALREADY_EXISTS(501);

    // The integer associated with this enum value
    private final int value;

    ServerResponse(int value) {
        this.value = value;
    }

    public int getCode() {
        return value;
    }
}
```

Elements of the enum can be declared along with “parameters”

When the object representing each element is created, the associated parameters are passed to the constructor method.

Enums are Classes

- Enums are a convenient way of defining a class along with some standard static methods

valueOf : converts a `String` to an Enum

```
CommandType c = CommandType.valueOf("CREATE");
```

values: returns an array of all the enumerated constants

```
CommandType[] varr = CommandType.values();
```

- Implicitly extend class `java.lang.Enum`
- Can include specialized constructors, fields and methods, as in ServerResponse

Using Enums: Switch

```
// Use of 'enum'
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.*, plus enum types and String
  - Not as powerful as OCaml pattern matching! (Yet!*)
- The `default` keyword specifies a "catch all" (wildcard) case
- Must indicate end of each case using break or return

*ML-style pattern matching that binds variables, etc., has been a "preview feature" of recent versions of Java, and is slowly being integrated into the main design.

What will be printed by the following program?

0

Got CREATE!

0%

Got MESG!

0%

Got NICK!

0%

default

0%

something else

0%

What will be printed by the following program?

```
CommandType t = CommandType.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

Answer: 5 something else!

break

- GOTCHA: By default, each branch will "fall through" into the next, so that code actually prints:

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

- Use an explicit `break` statement to avoid fall-through:

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

Alternative Option – switch Expressions

- Introduced in Java 14
- No need for **break** statements to prevent fall through
- Read more here - https://docs.oracle.com/en/java/javase/14/language/switch-expressions.html

```
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");
}
```

Alternative Option – switch Expressions

- (Similar to OCaml pattern matching), these are *expressions* and evaluate to a single value
- (Similar to OCaml pattern matching), these must be exhaustive

```
int x = switch (t) {
  case CREATE -> 5;
  case MESG   -> 10;
  case NICK   -> 15;
  default     -> 20;
}
```

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 / refute the hypothesis.
   - *Don't* make random changes!
   - You should try to predict the effects of your experiment
   - Re-run test cases
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 the stack trace
  - They give you a lot of information

- If you are using and IDE (like IntelliJ), it is worth taking a little time to learn how to use the debugger!
  - create “breakpoints” that stop the program and let you inspect the state of the abstract machine

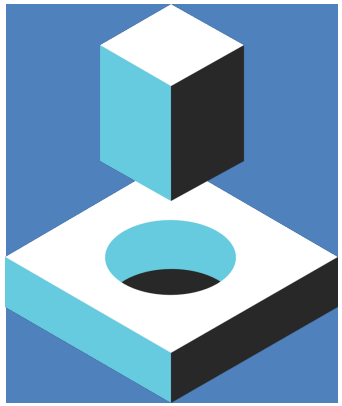
- Simple print statements are also very effective!
  - Confirm or disprove hypothesis
  - Can sometimes be easier to control than the debugger
  - e.g.: The code reached "HERE!" (or not)

Exceptions

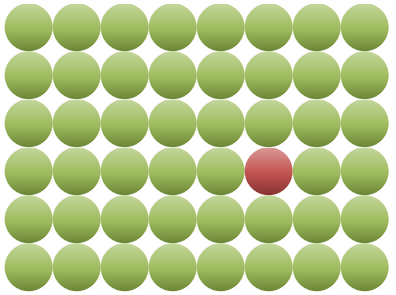
Dealing with the unexpected

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



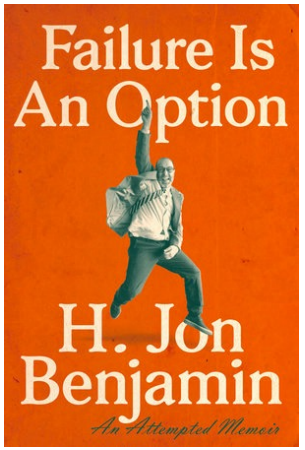
Error 404
Page Not Found!



- 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);
        }
        …
}
```

30: What happens if we do (new C()).foo(). ? The program does...

0

Program stops without printing anything

0%

Program prints “here in bar”, then stops

0%

Program prints “here in bar”, then “here in foo”, then stops

0%

Something else

0%

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 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

Answer: 4* (*well… depends on whether you count stderr as "printing")

Abstract Stack Machine

Workspace

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

Stack

Heap

Abstract Stack Machine

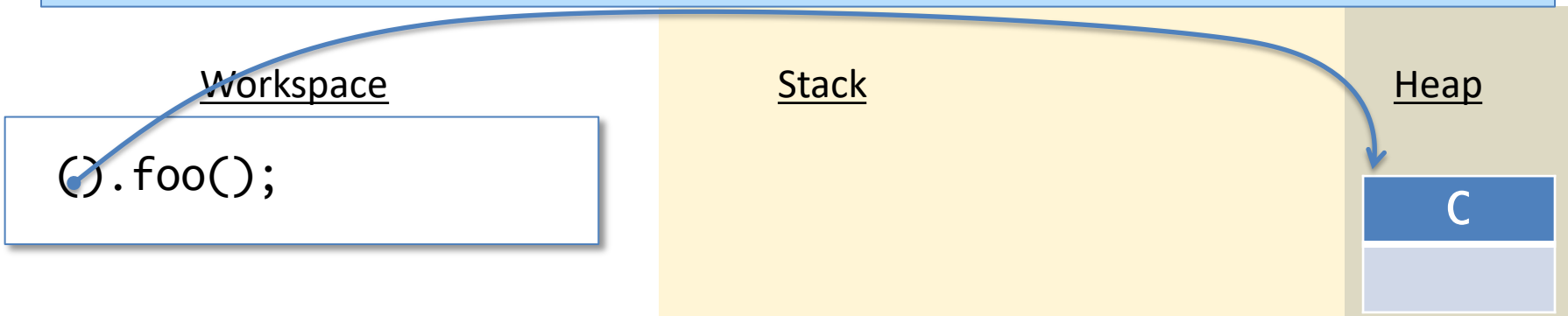
Workspace

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

Stack

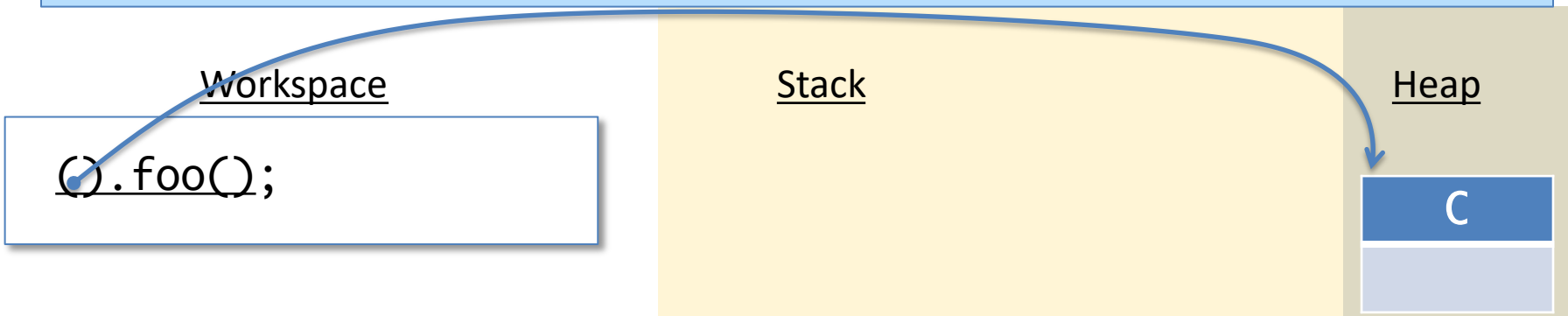
Heap

Abstract Stack Machine



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

Abstract Stack Machine

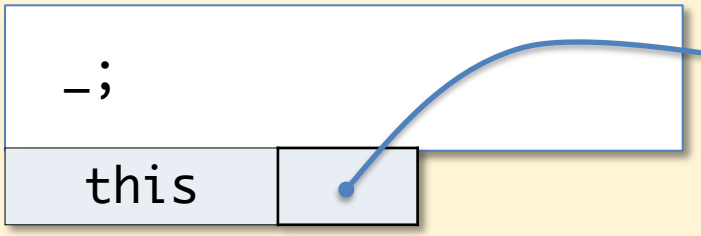


Abstract Stack Machine

Workspace

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

Stack



Heap

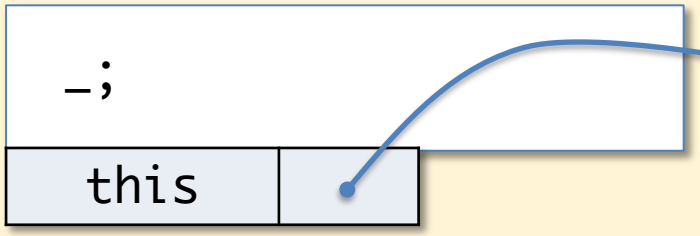
Save a copy of the current workspace in the stack, leaving a “hole”, written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack. Use the dynamic class to look up the method body from the class table.

Abstract Stack Machine

Workspace

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

Stack



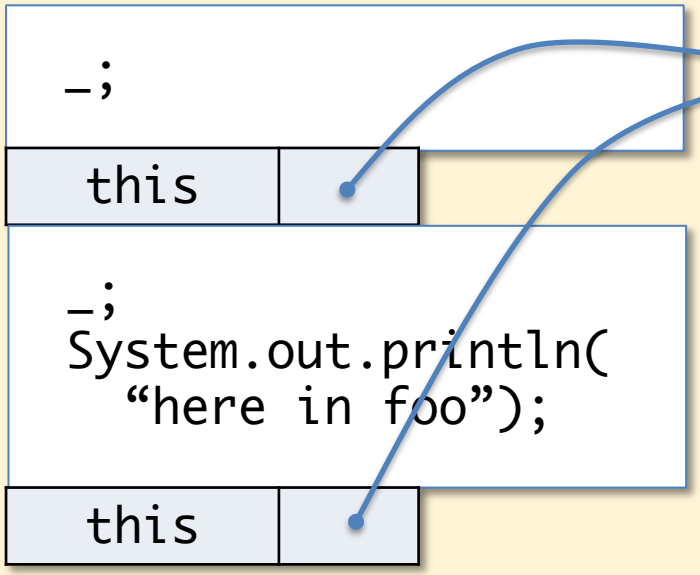
Heap

Abstract Stack Machine

Workspace

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

Stack



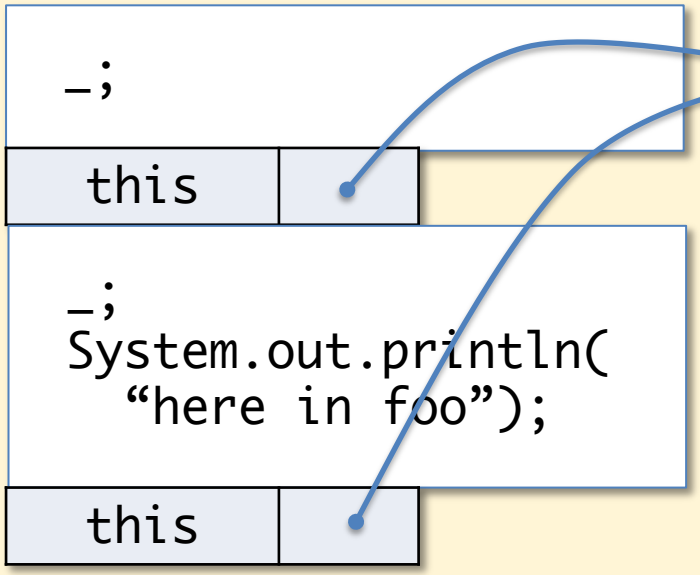
Heap

Abstract Stack Machine

Workspace

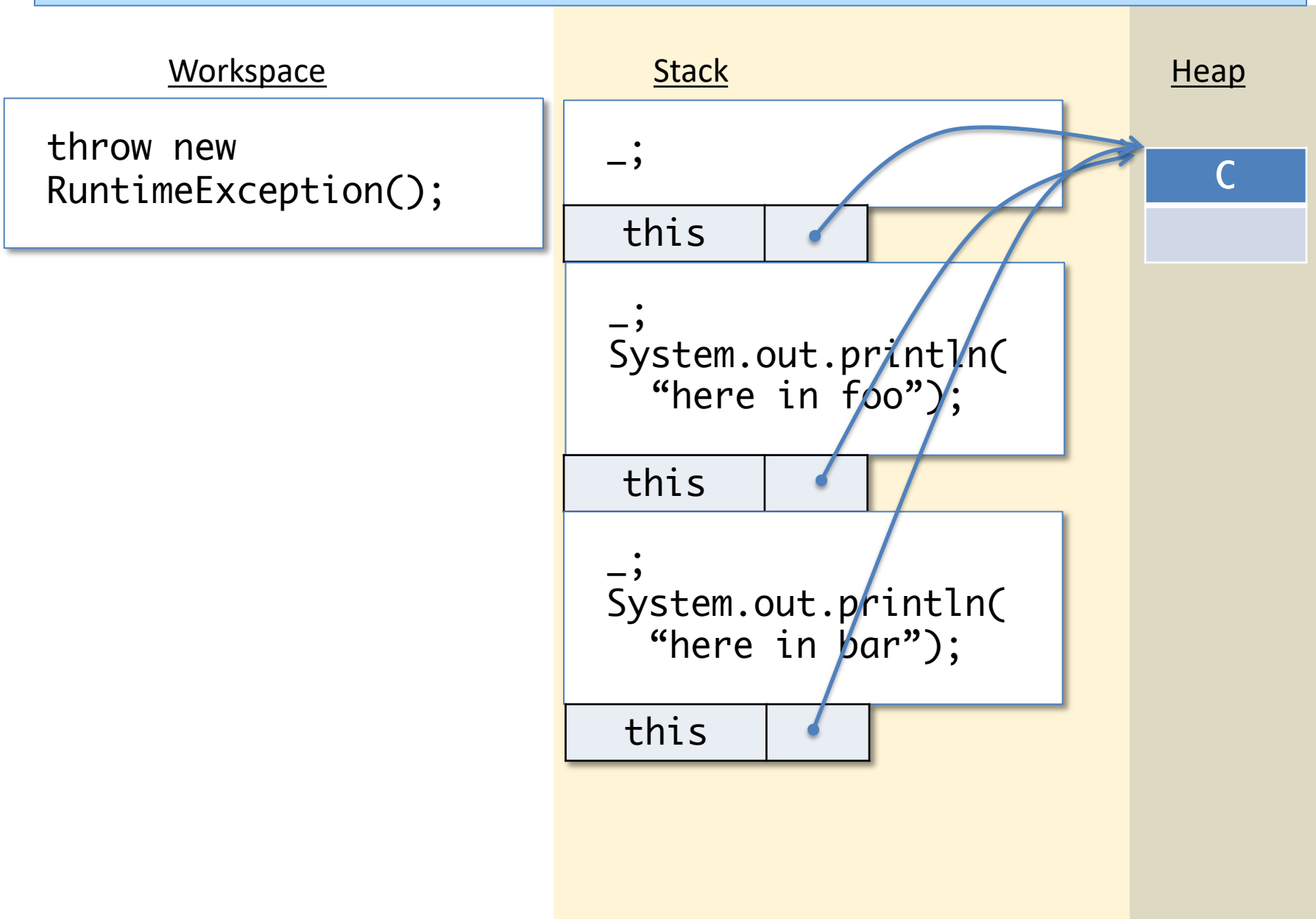
```
this.baz();
System.out.println(
  “here in bar”);
```

Stack

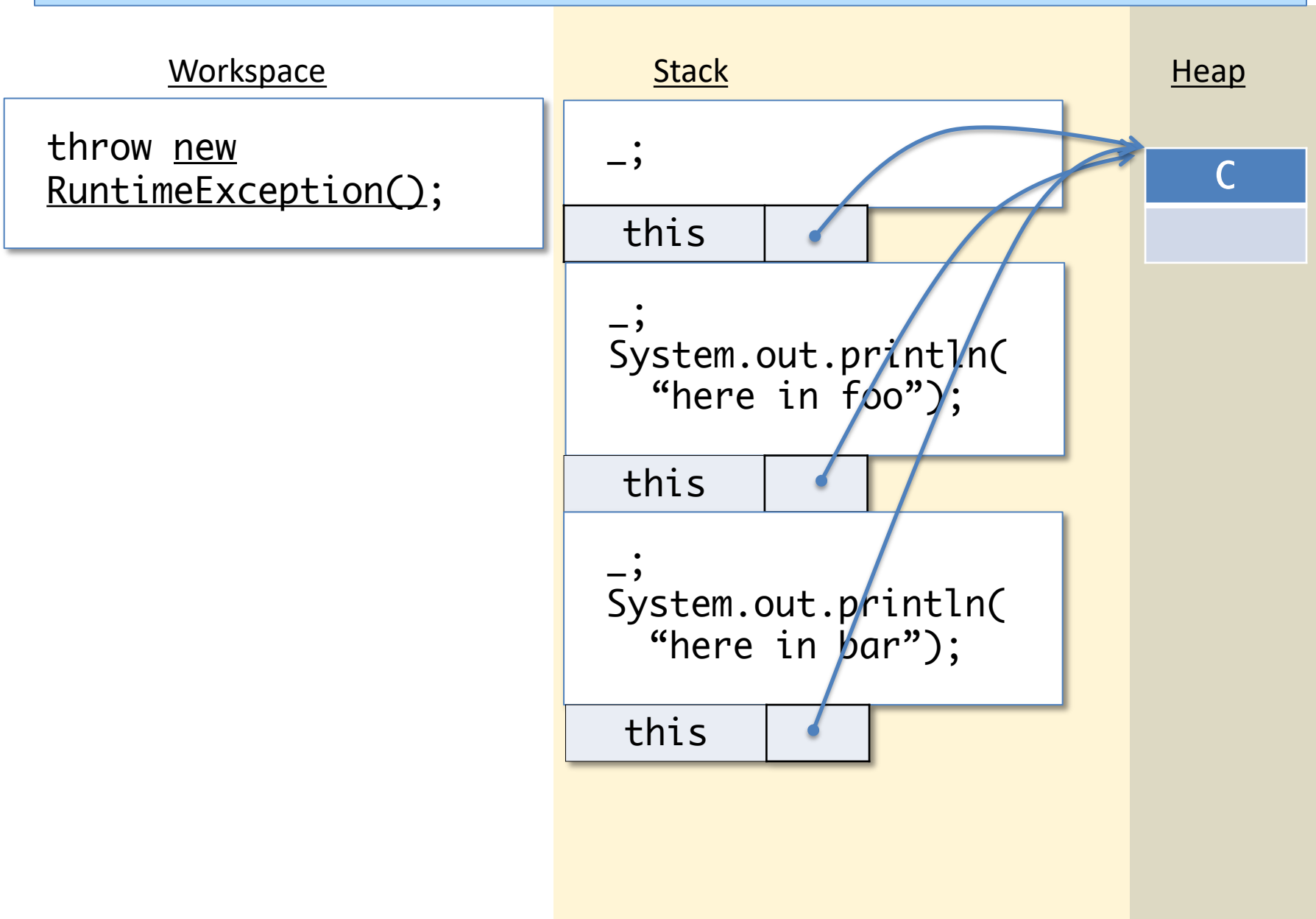


Heap

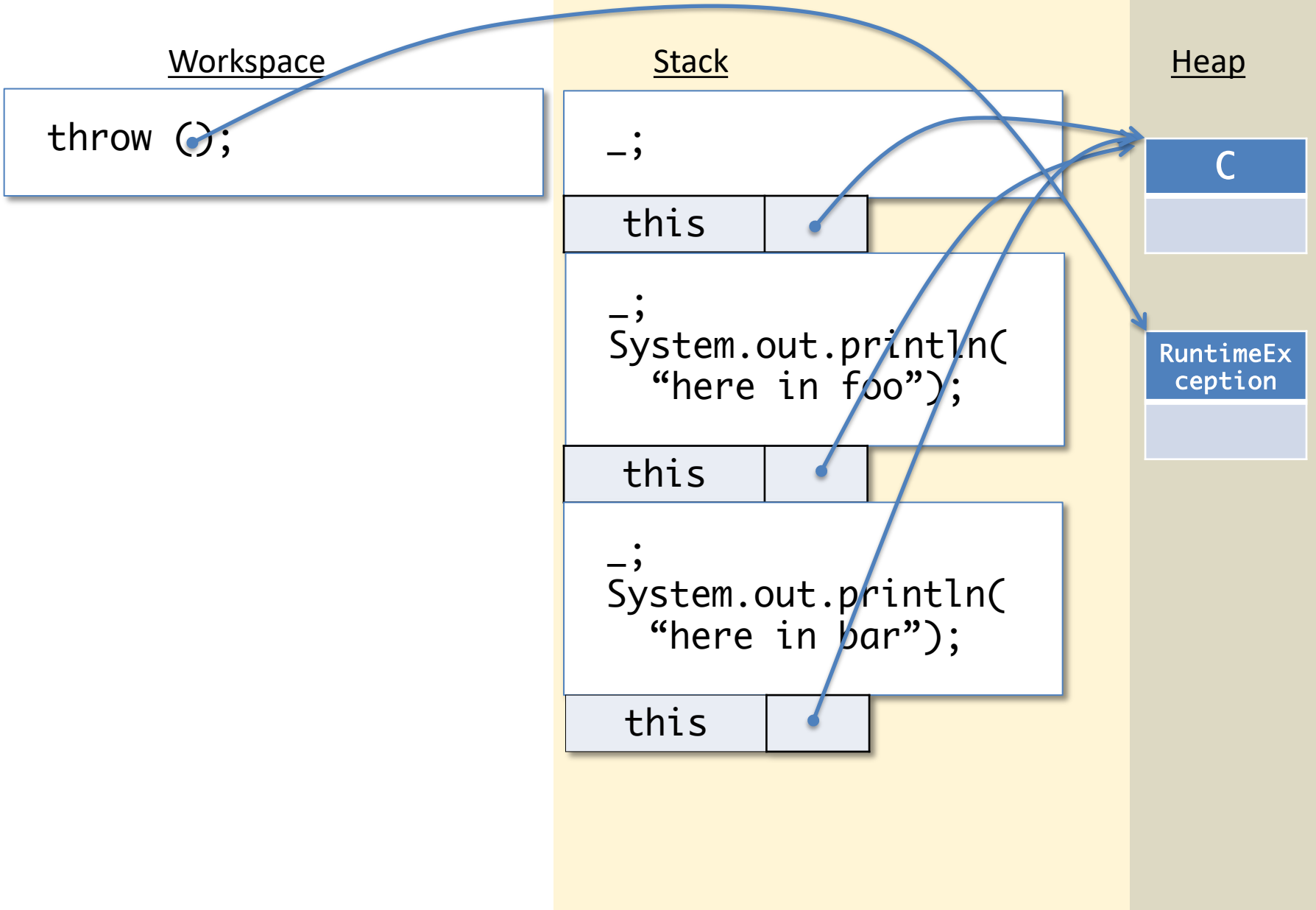
Abstract Stack Machine



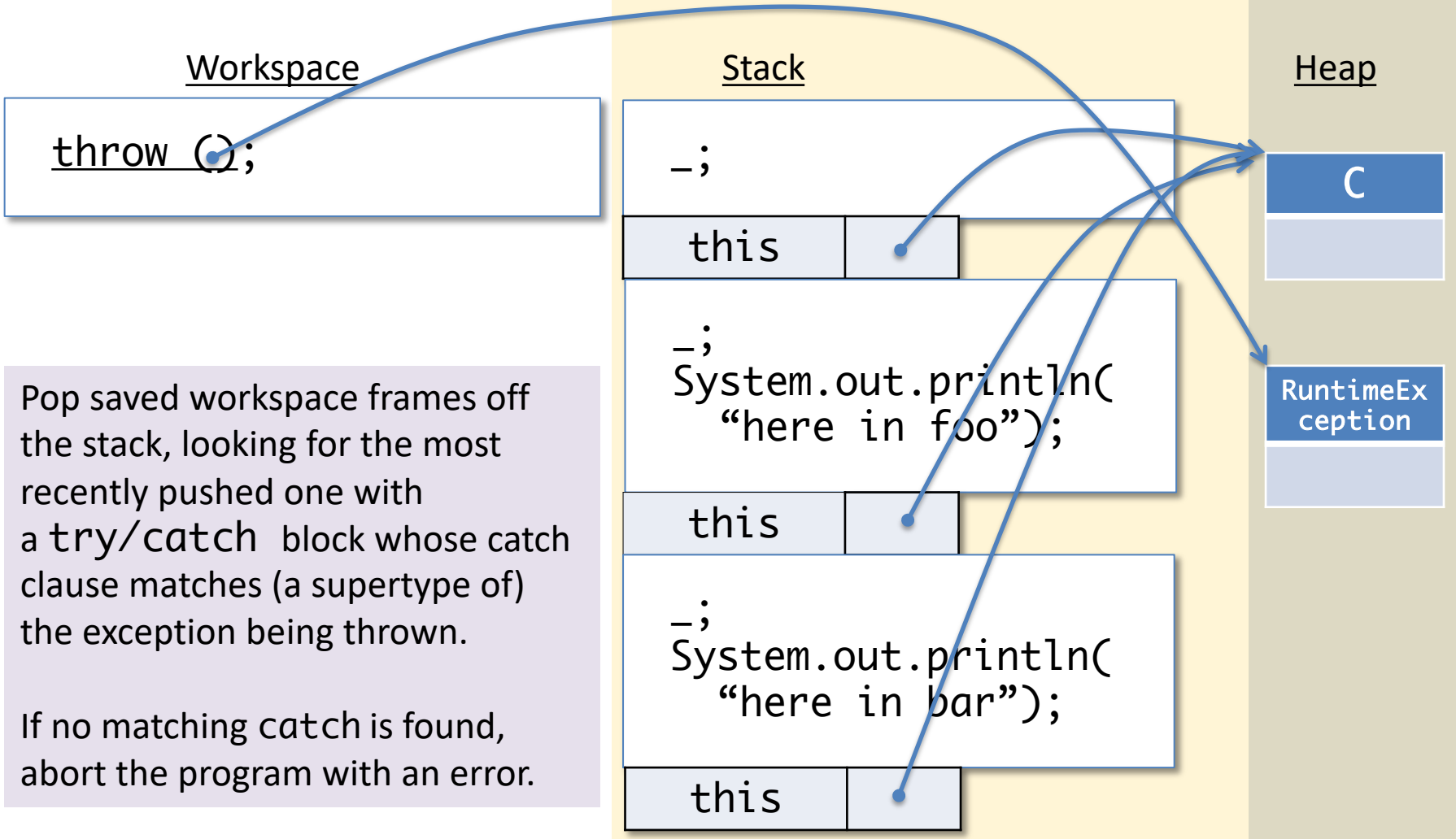
Abstract Stack Machine



Abstract Stack Machine



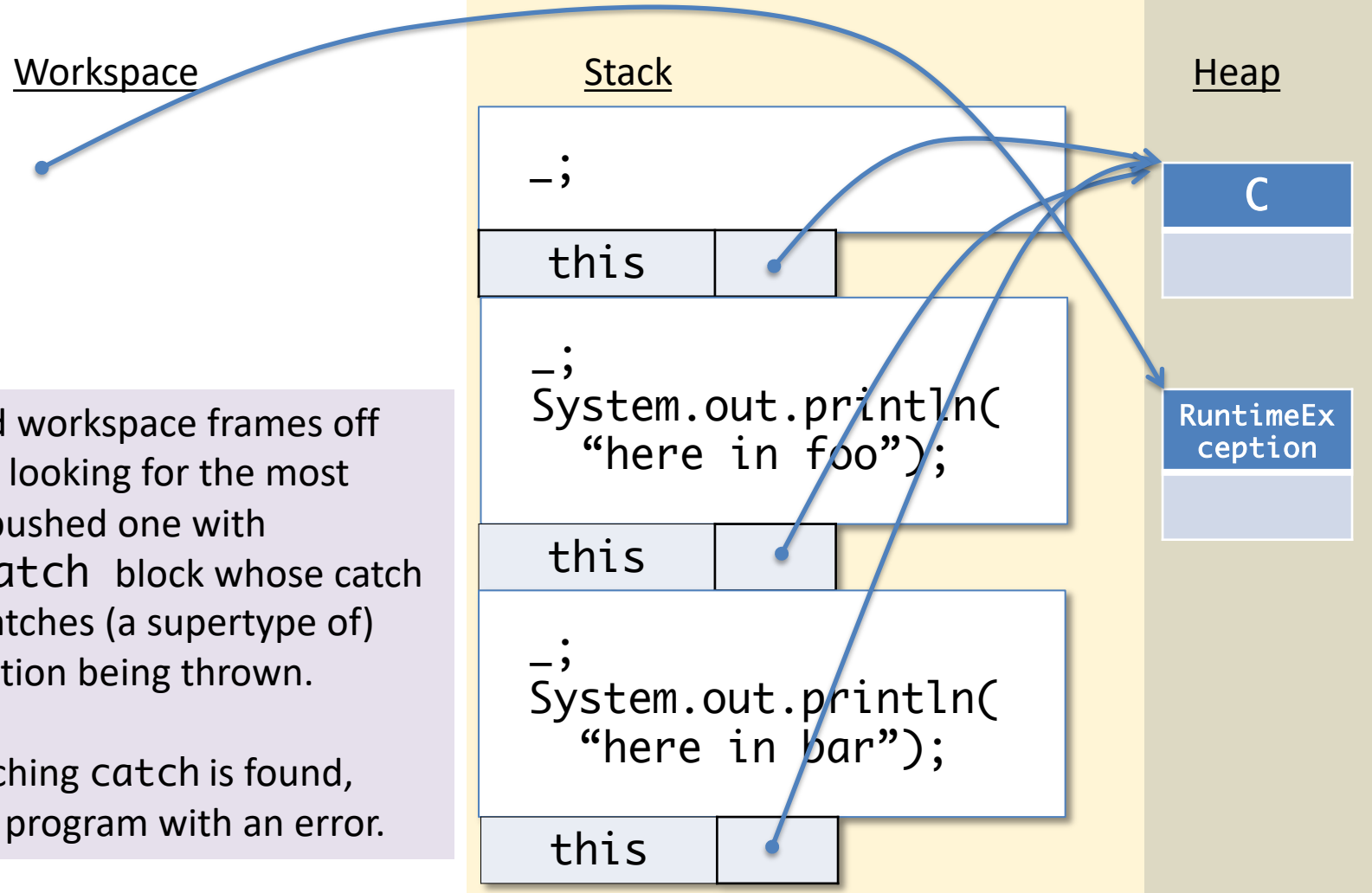
Abstract Stack Machine



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.

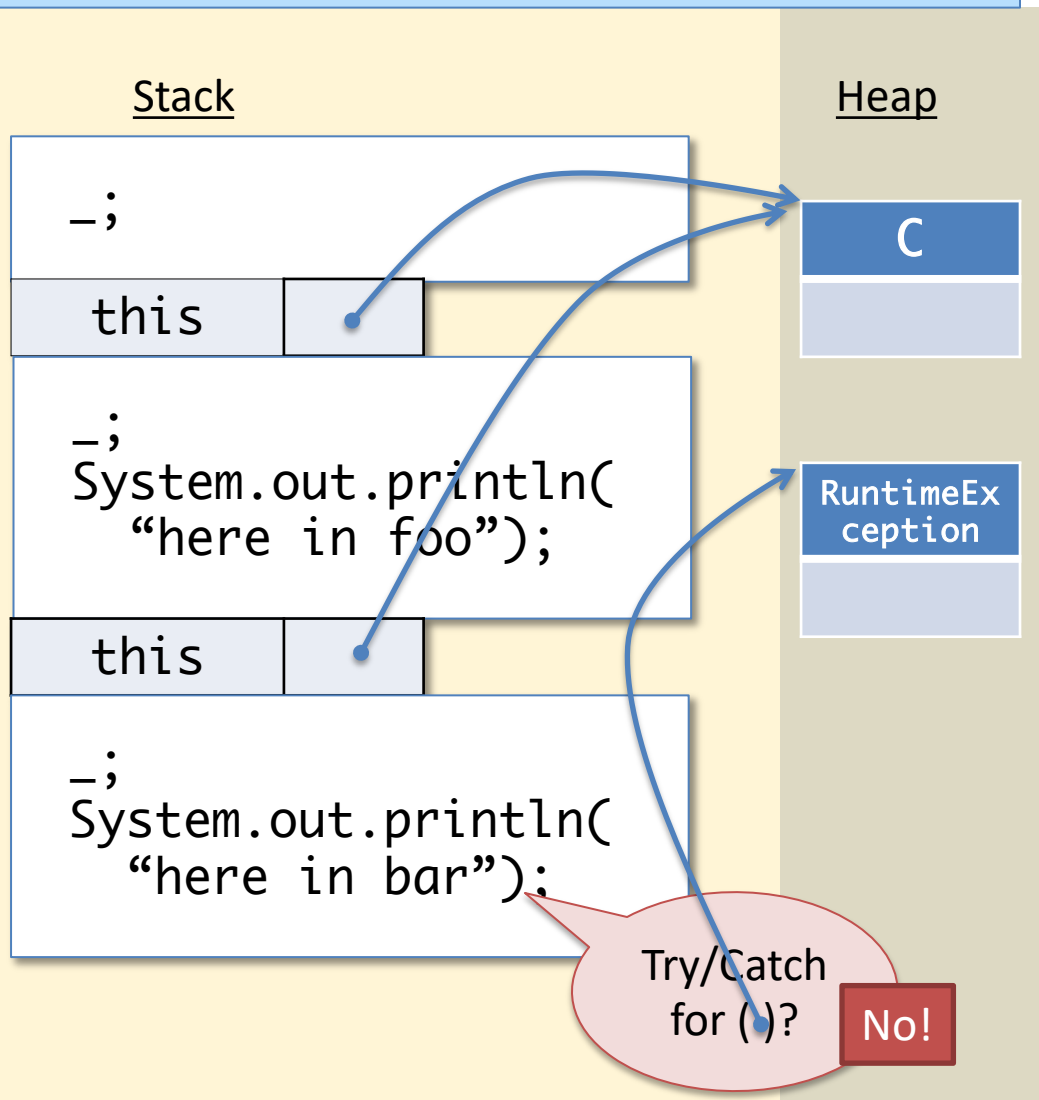
If no matching `catch` is found, abort the program with an error.

Abstract Stack Machine

Workspace

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.

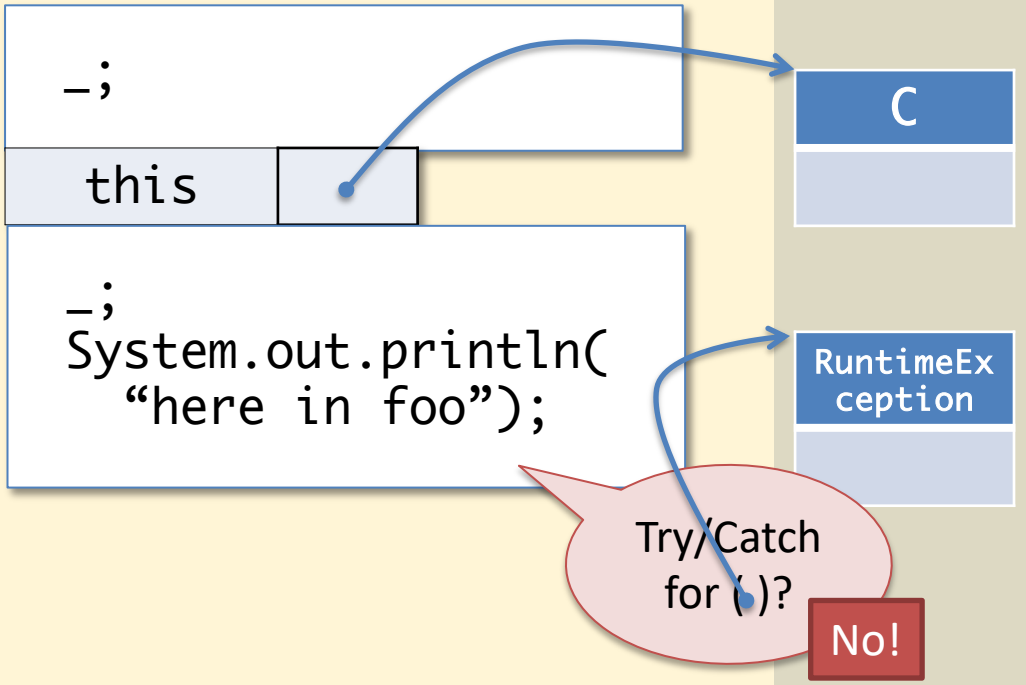


Abstract Stack Machine

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.

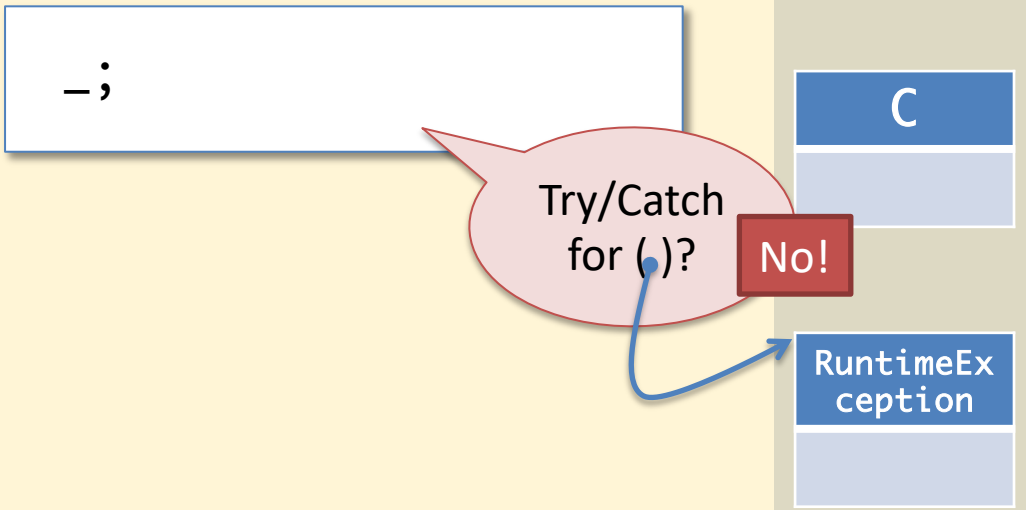
If no matching catch is found, abort the program with an error.

Abstract Stack Machine

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

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

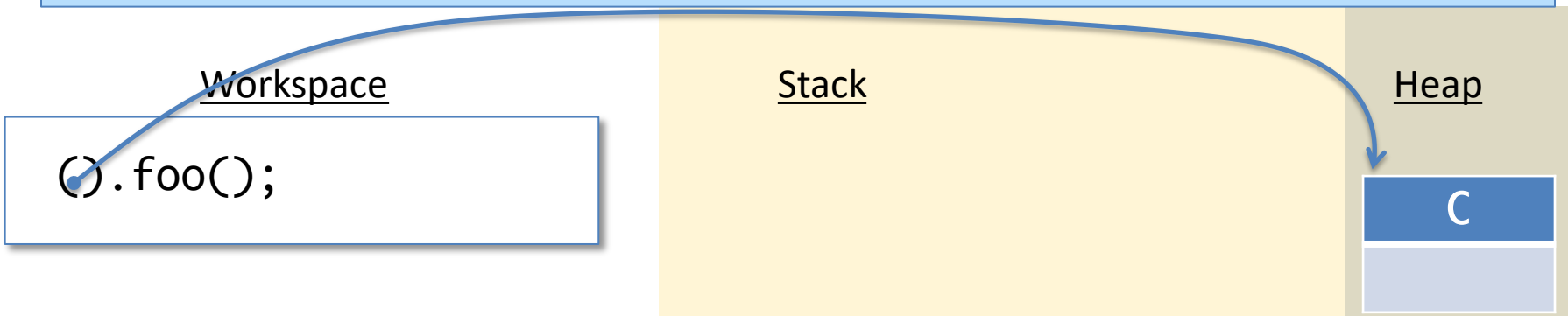
Workspace

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

Stack

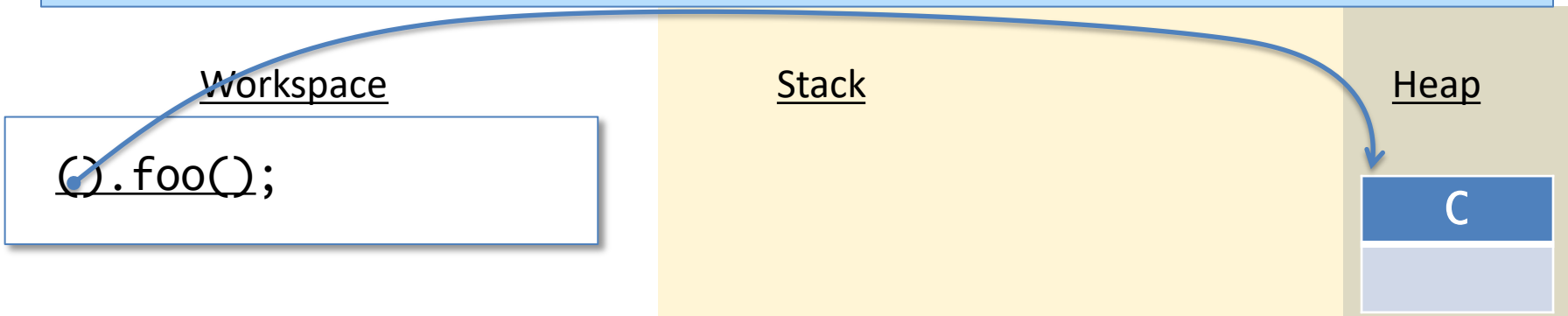
Heap

Abstract Stack Machine



Allocate a new instance of C in the heap.

Abstract Stack Machine

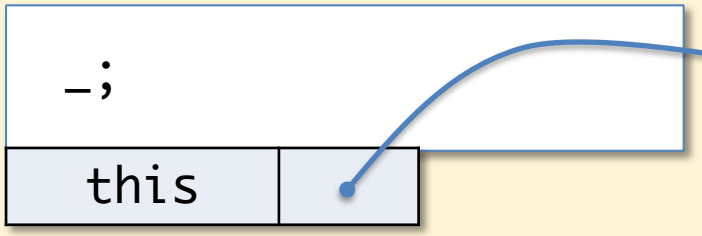


Abstract Stack Machine

Workspace

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

Stack



Heap

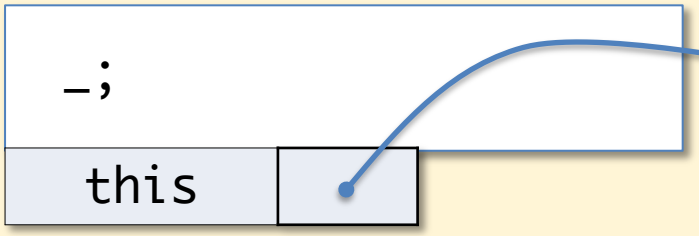
Save a copy of the current workspace in the stack, leaving a “hole”, written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack.

Abstract Stack Machine

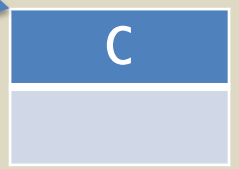
Workspace

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

Stack



Heap

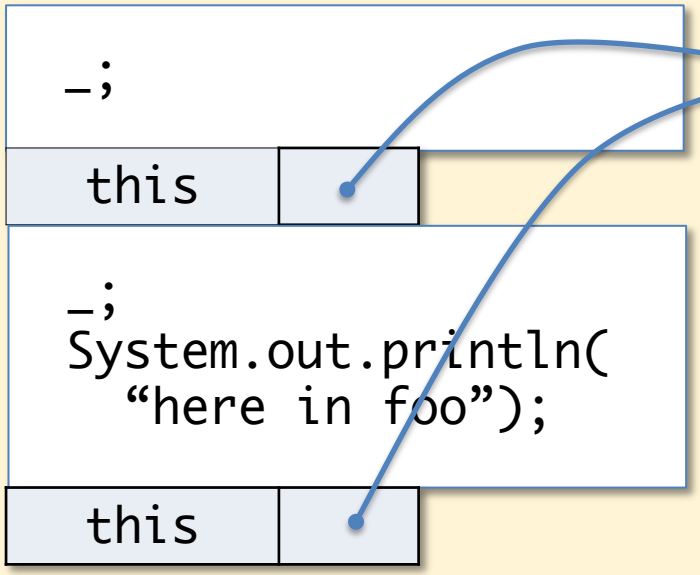


Abstract Stack Machine

Workspace

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

Stack



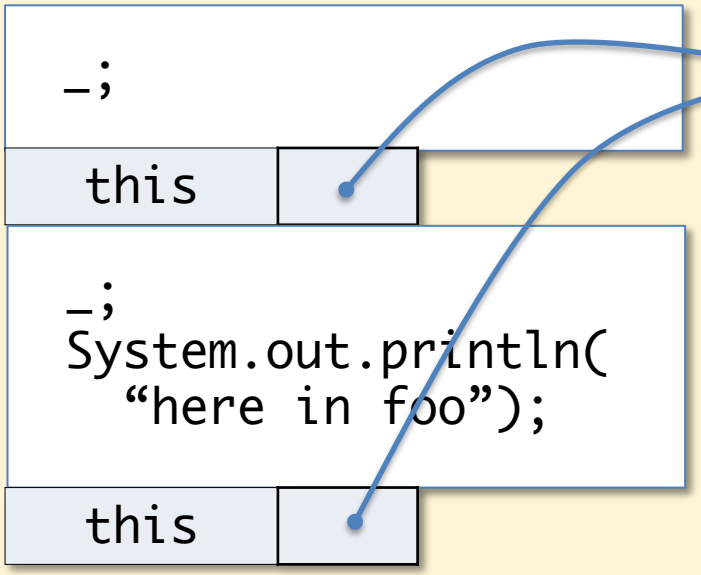
Heap

Abstract Stack Machine

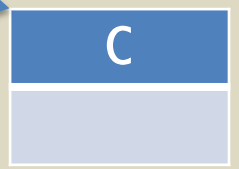
Workspace

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

Stack



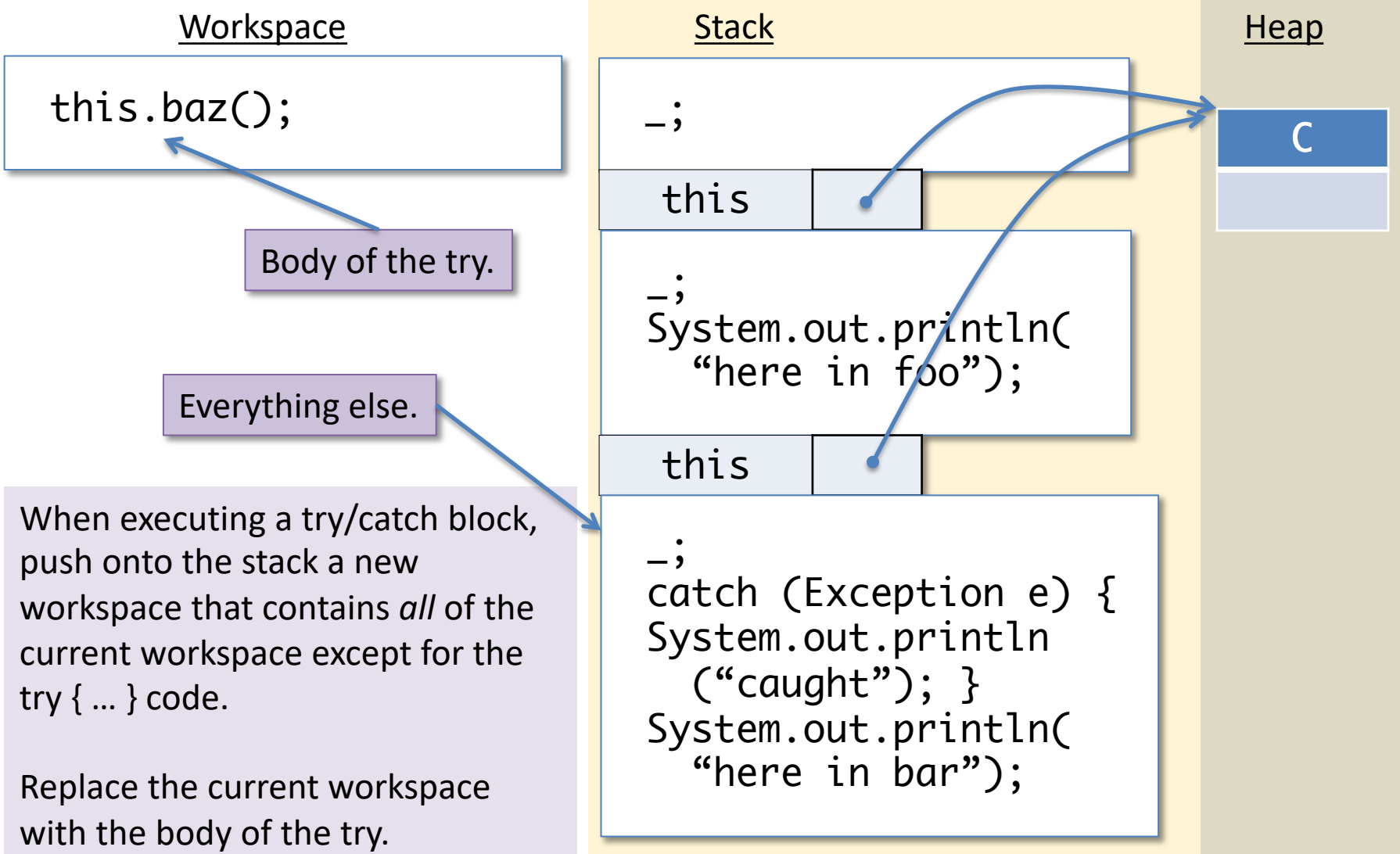
Heap



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

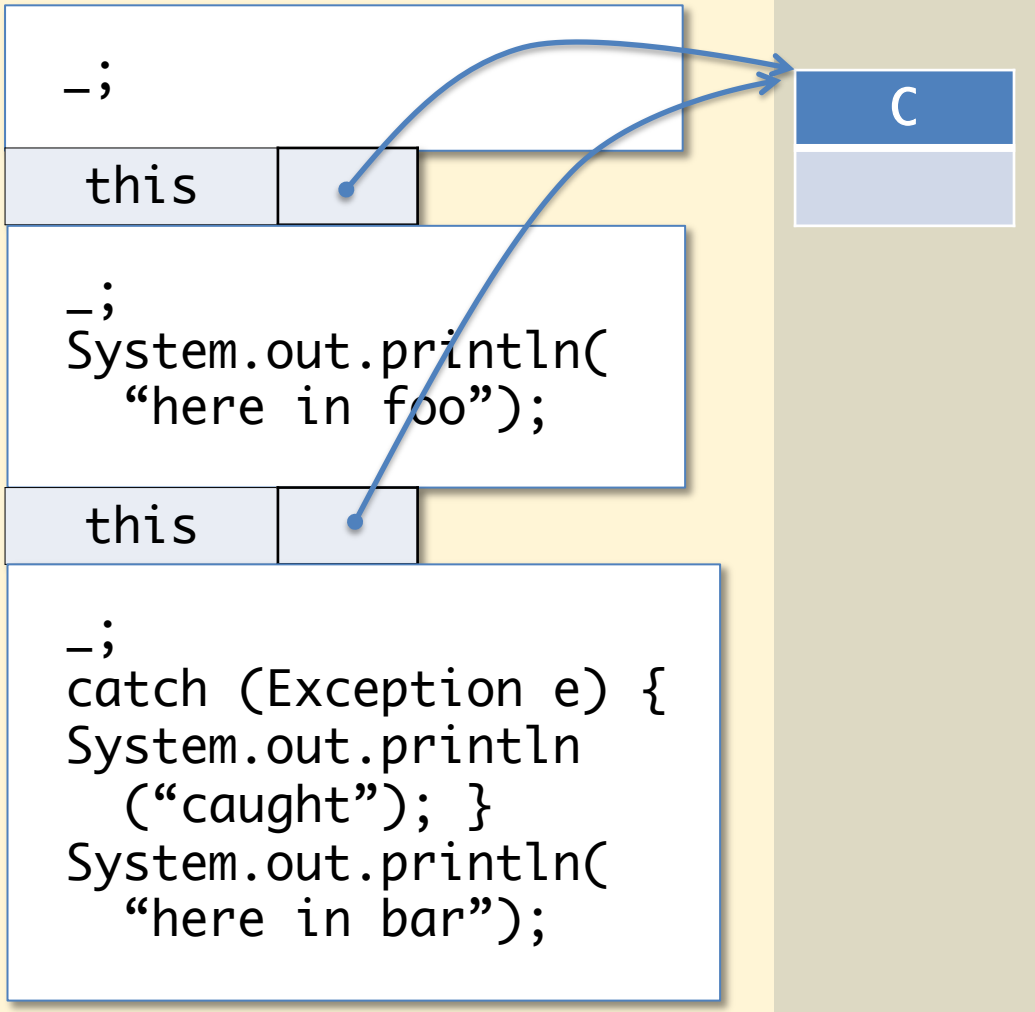


Abstract Stack Machine

Workspace

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

Stack



Heap

Continue executing as normal.

Abstract Stack Machine

Workspace

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

Stack

```
_;
```

this

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

this

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

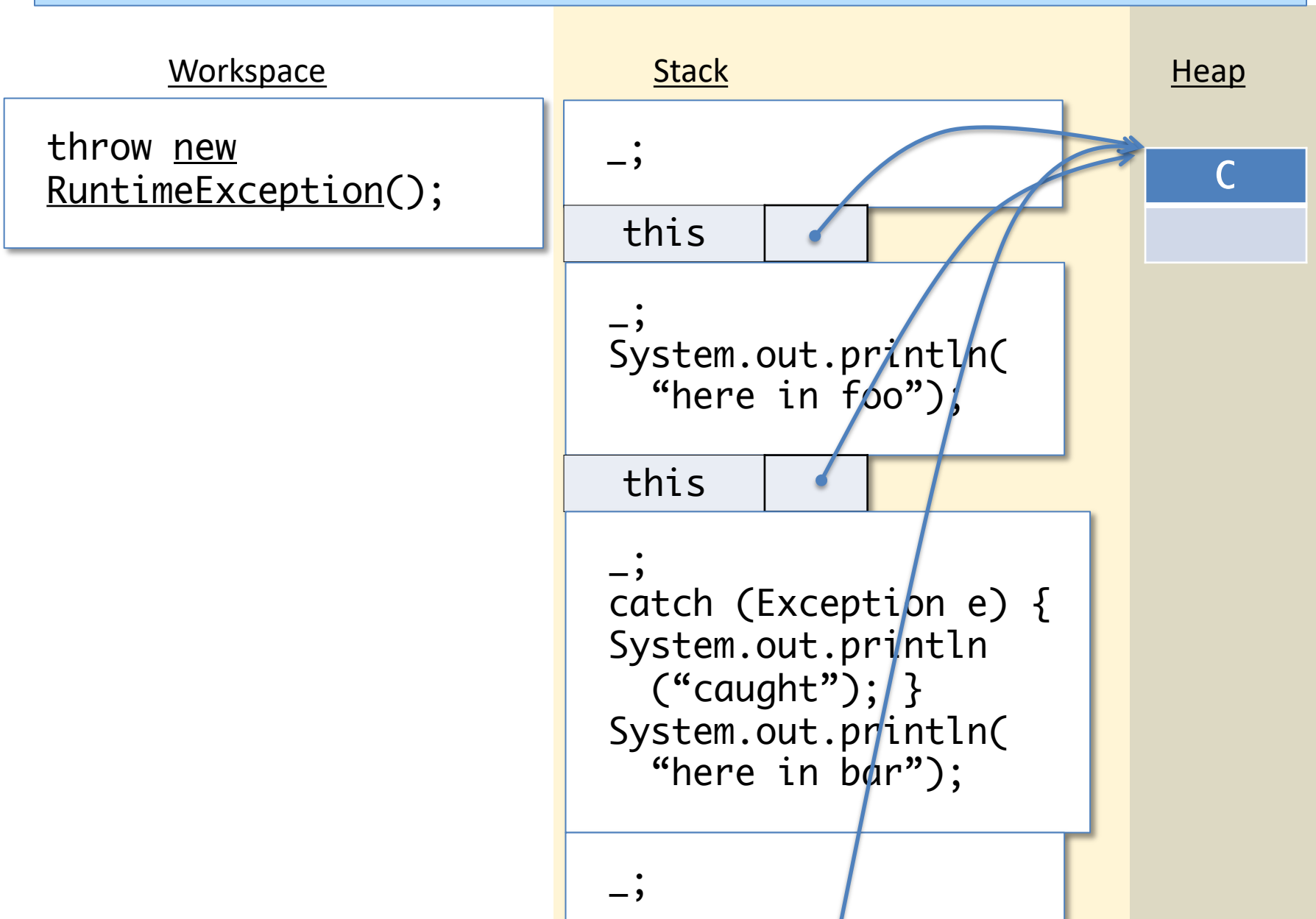
```
_;
```

Heap

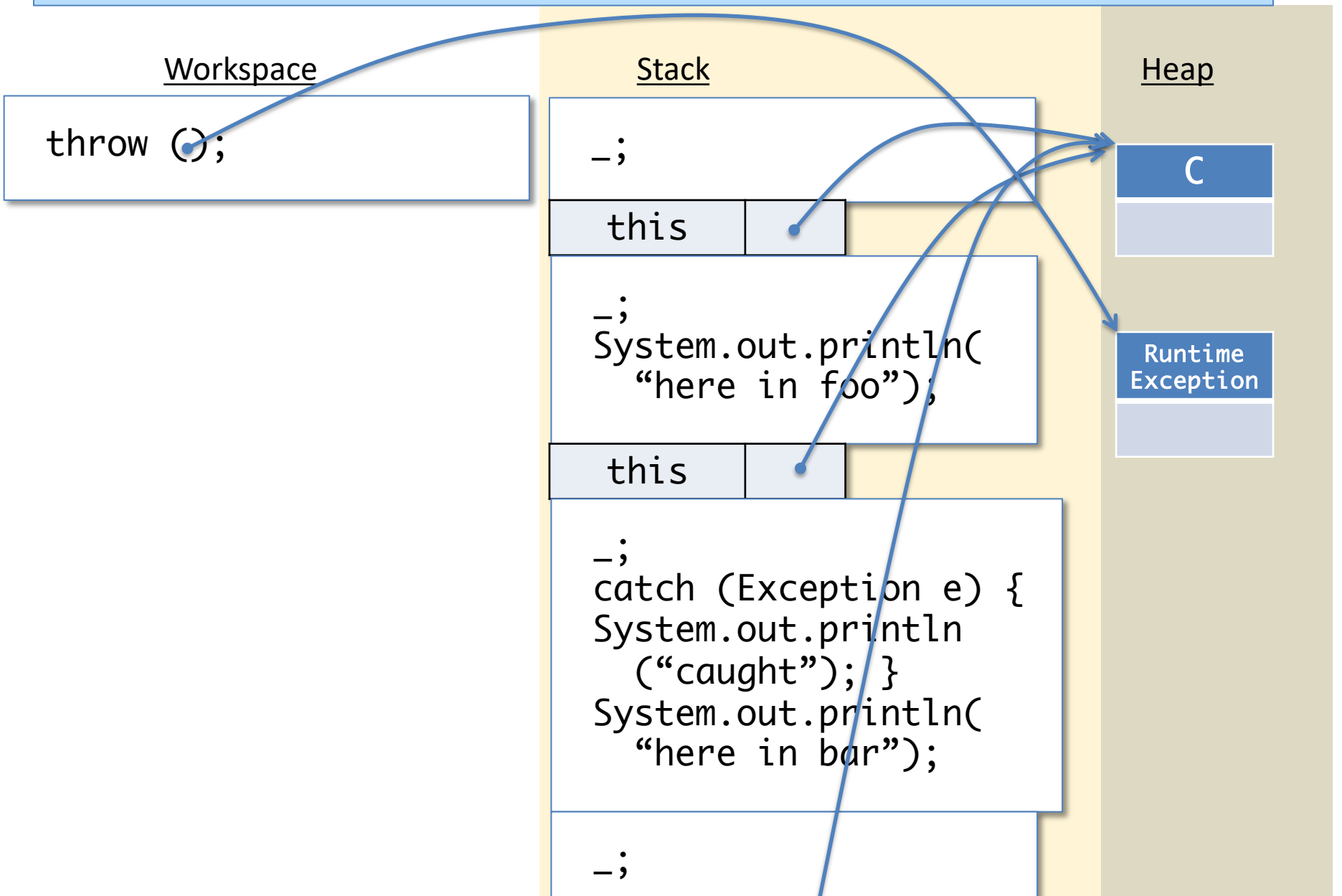
C

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

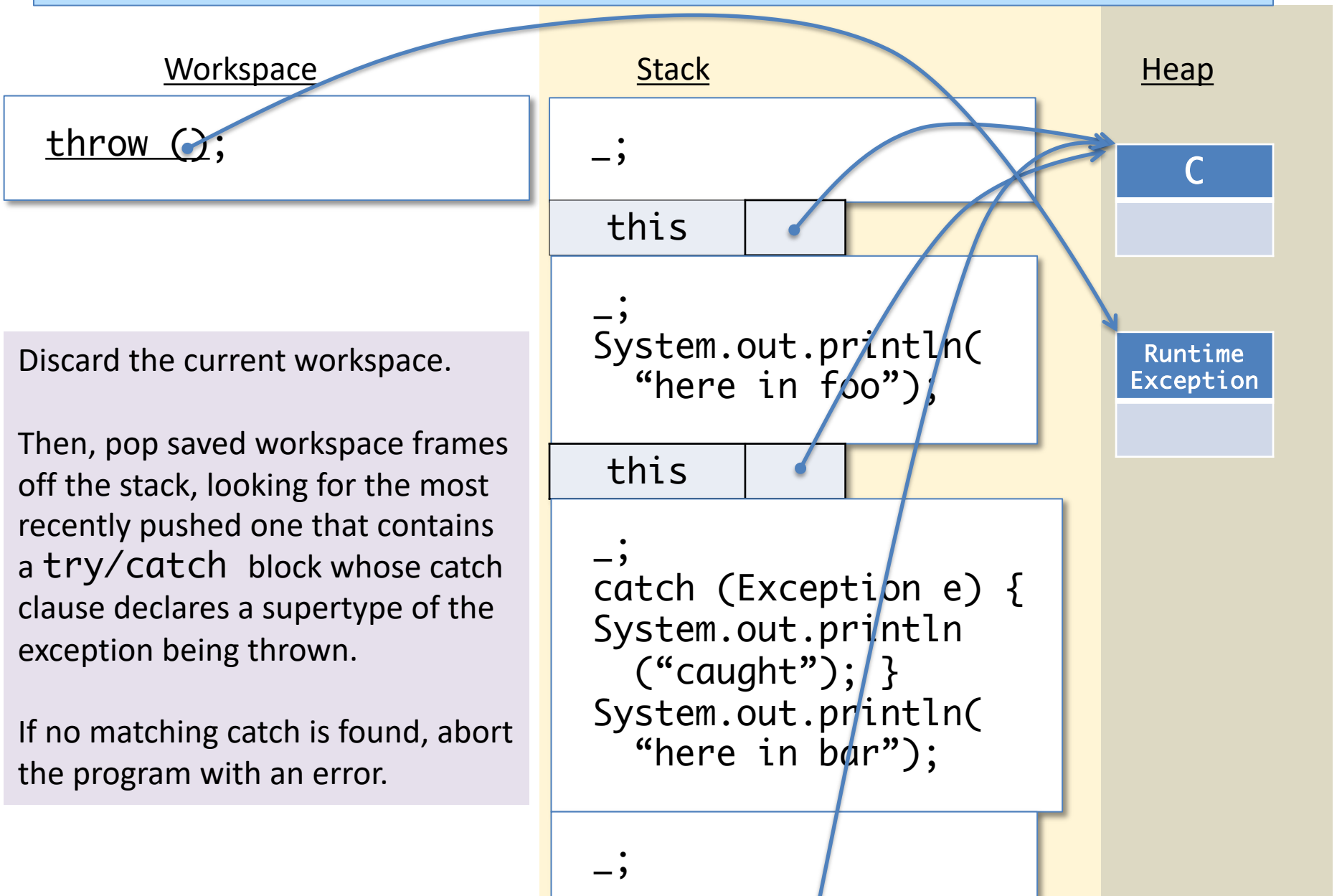
Abstract Stack Machine



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.

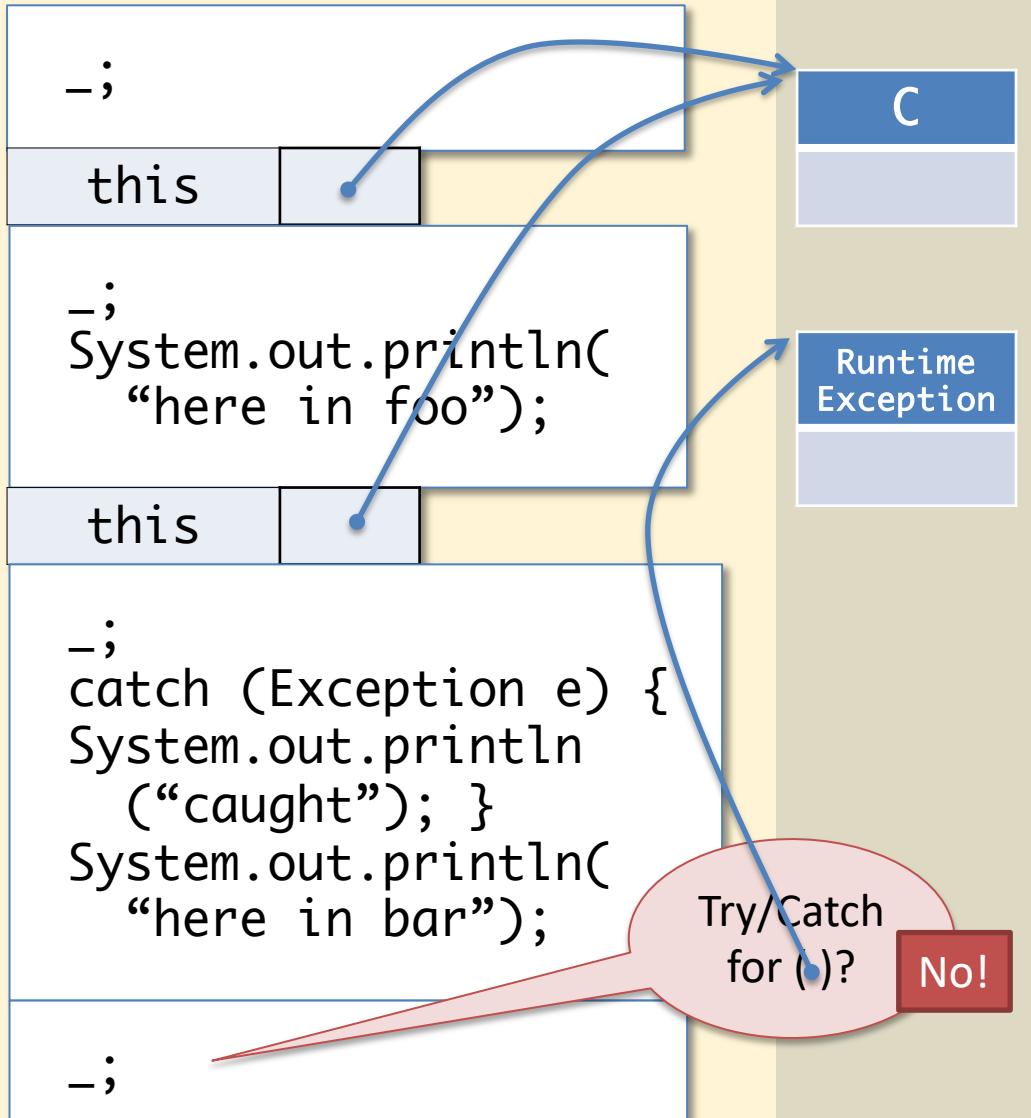
If no matching catch is found, abort the program with an error.

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

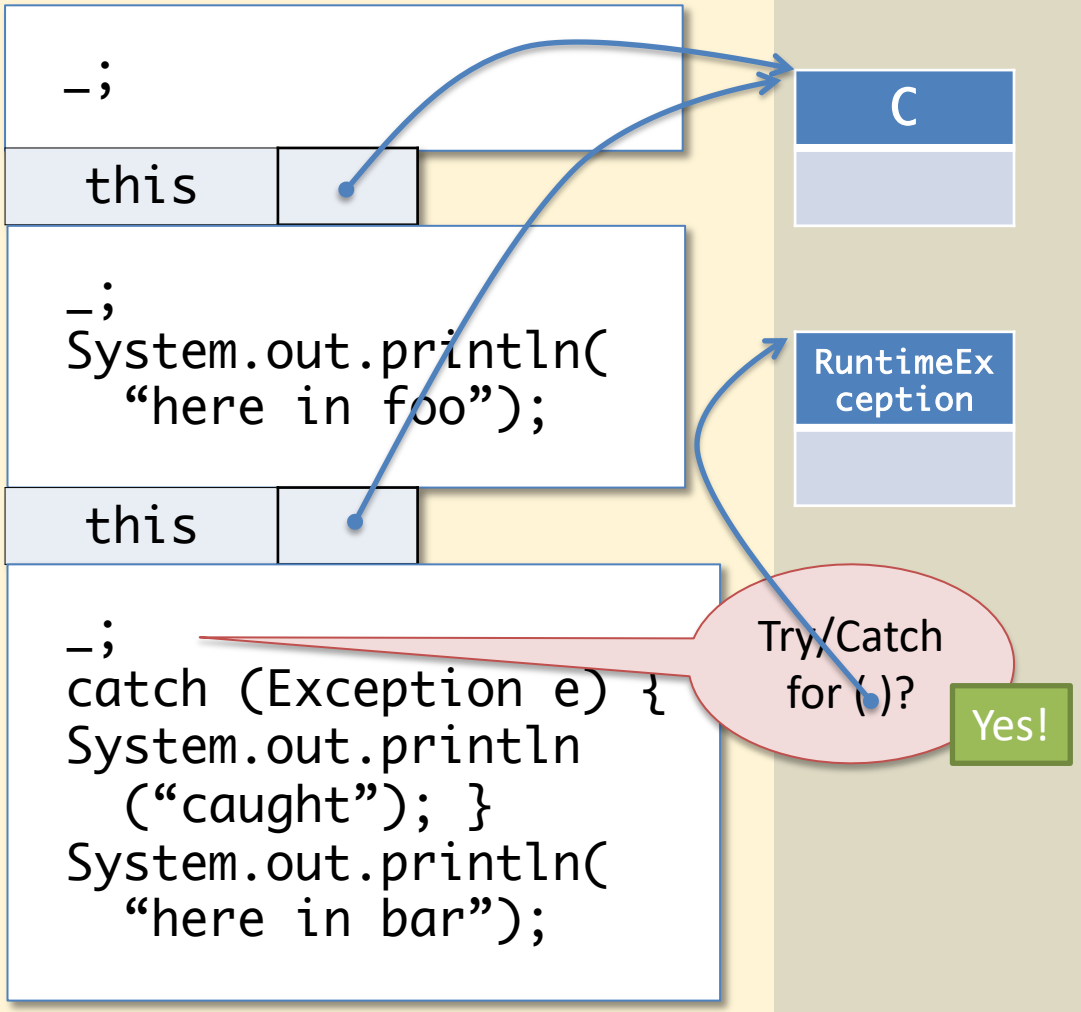
Workspace

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.



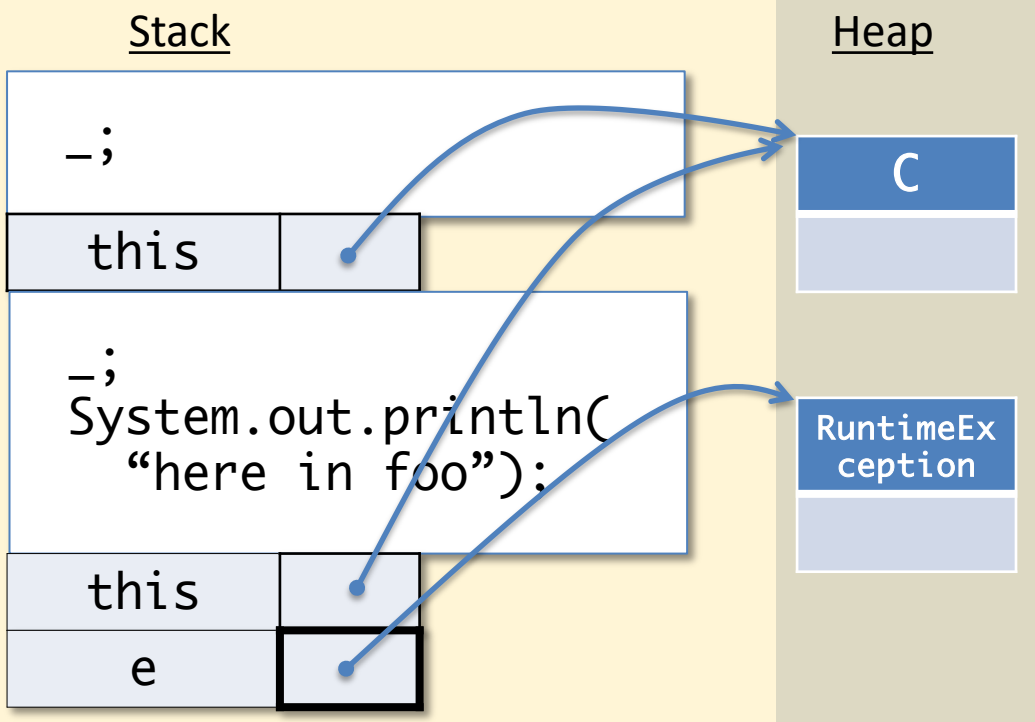
Abstract Stack Machine

Workspace

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

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.

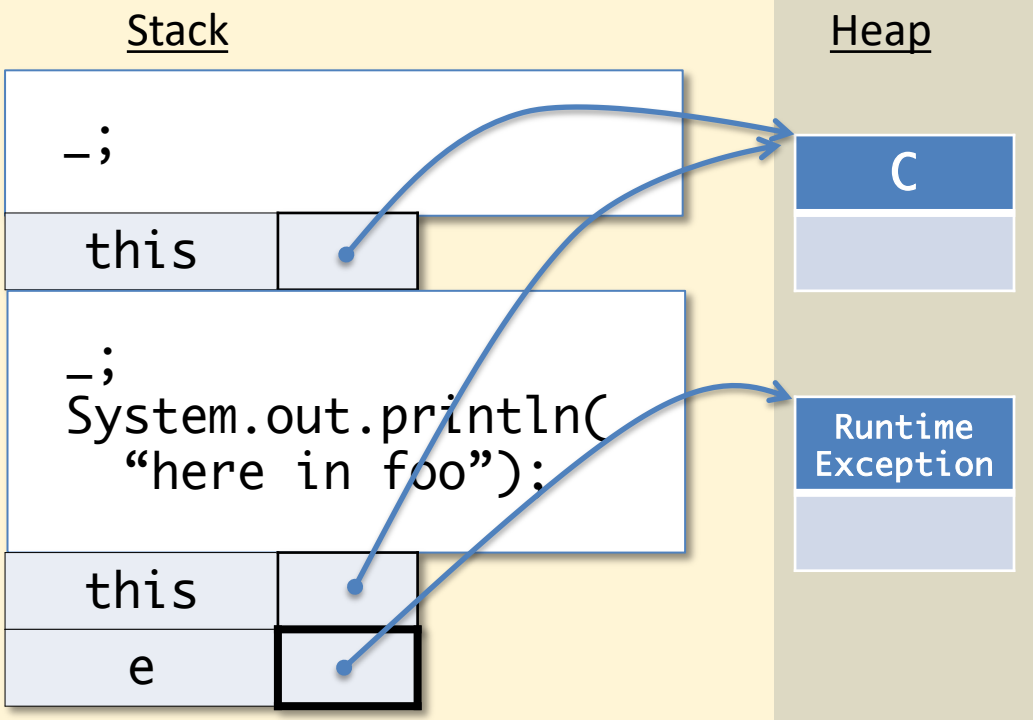


Abstract Stack Machine

Workspace

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

Continue executing as usual.



Abstract Stack Machine

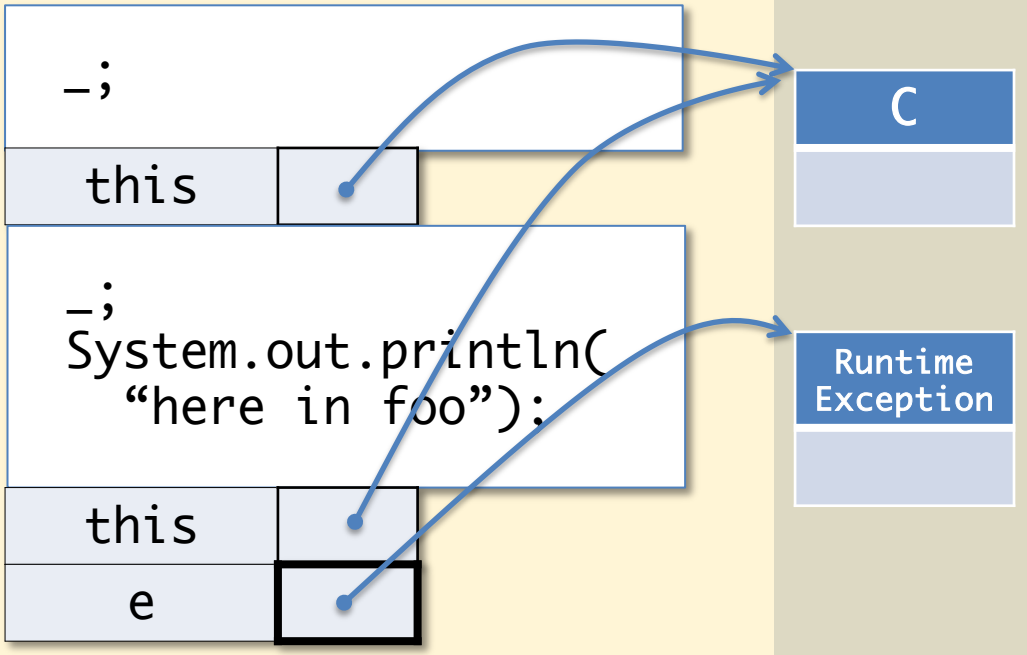
Workspace

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

Continue executing as usual.

Console

caught



Abstract Stack Machine

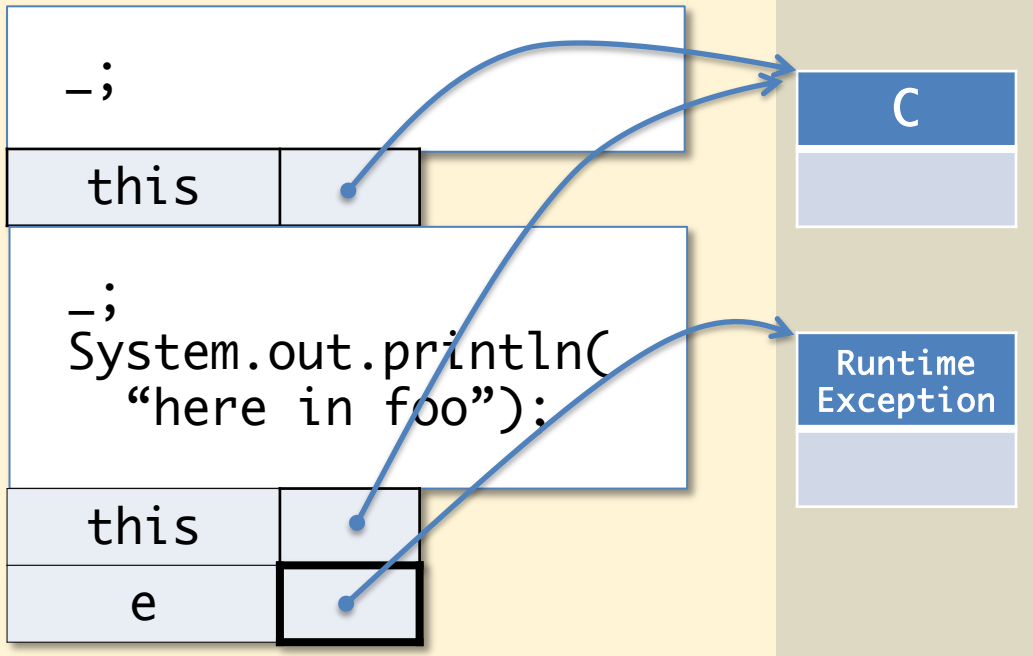
Workspace

```
{ ; }
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 too.

Console

caught



Abstract Stack Machine

Workspace

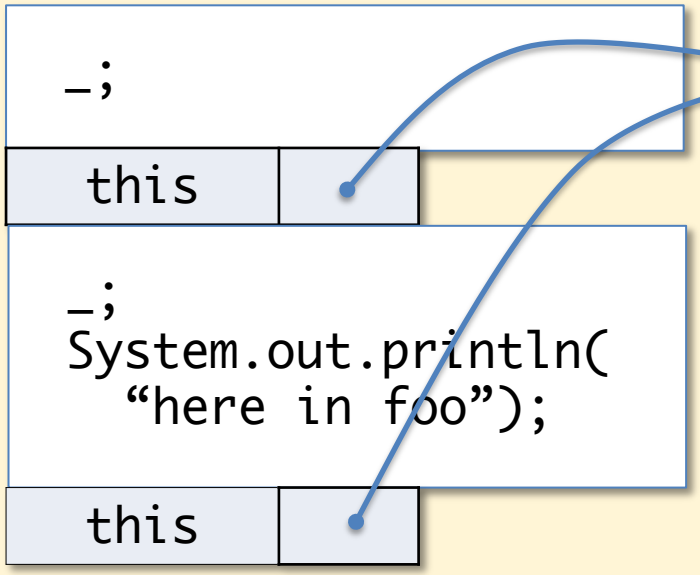
```
System.out.println(
  “here in bar”);
```

Continue executing as usual.

Console

caught

Stack



Heap

Abstract Stack Machine

Workspace

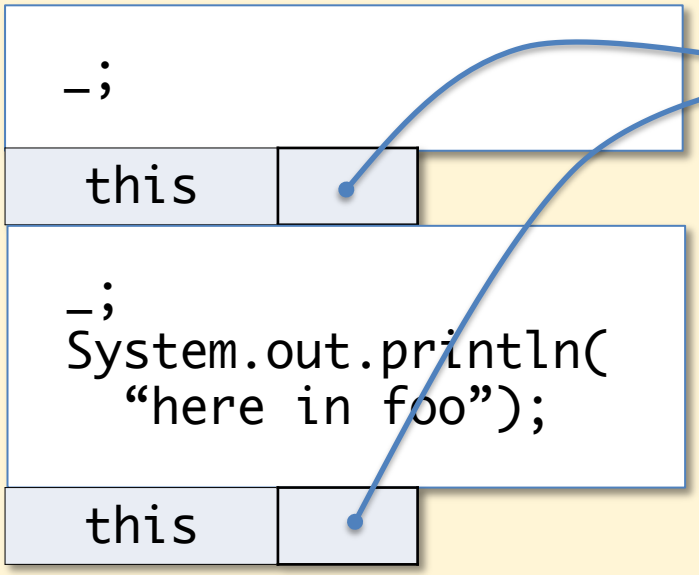
```
System.out.println(
  "here in bar");
```

Continue executing as usual.

Console

caught

Stack



Heap

Abstract Stack Machine

Workspace

```
;
```

Stack

```
_;
```

this

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

this

Heap

C

Runtime Exception

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

Console

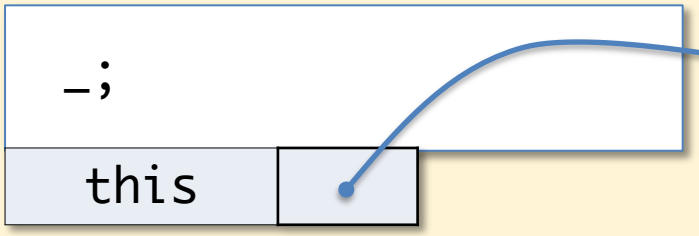
```
caught
here in bar
```

Abstract Stack Machine

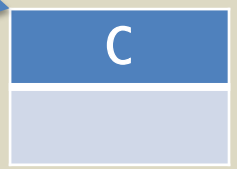
Workspace

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

Stack



Heap



Continue executing as usual.

Console

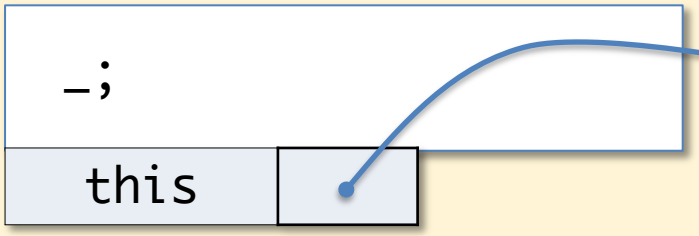
```
caught
here in bar
```

Abstract Stack Machine

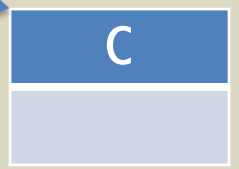
Workspace

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

Stack



Heap



Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine

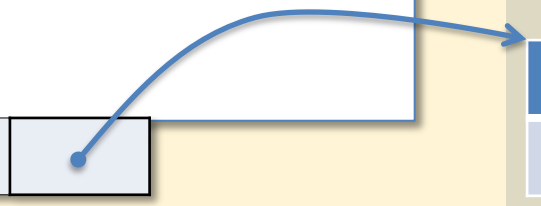
Workspace

```
;
```

Stack

```
_;
```

this



Heap

C

Runtime Exception

Continue executing as usual.

Console

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

Abstract Stack Machine

Workspace

Stack

Heap

C

Runtime Exception

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 a given "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!