Programming Languages and Techniques (CIS120)

Lecture 29

April 7, 2014

Exceptions

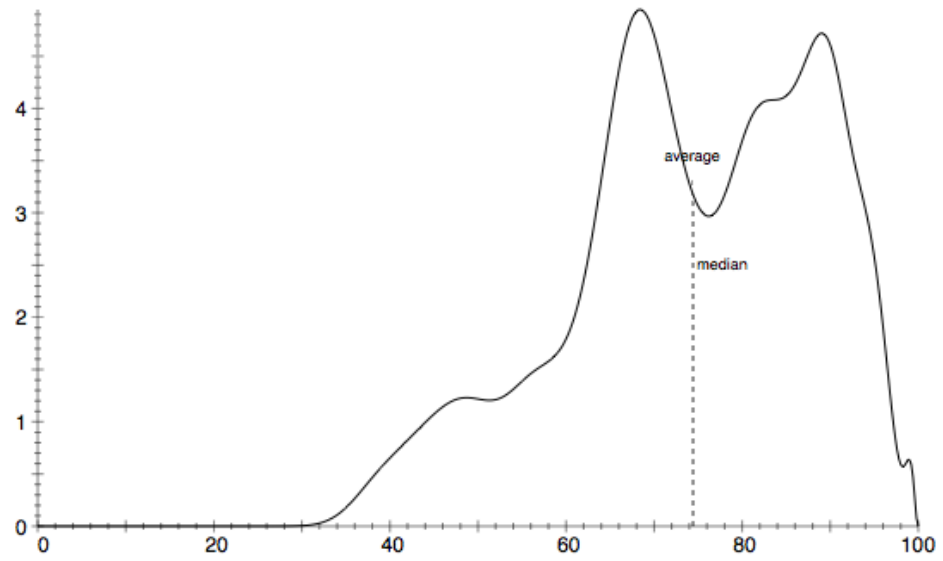
Announcements

- Read Chapter 28

- Homework 9 available later today, due Tuesday April 15th at midnight
  - Focus: working with Java's Collection and IO libraries

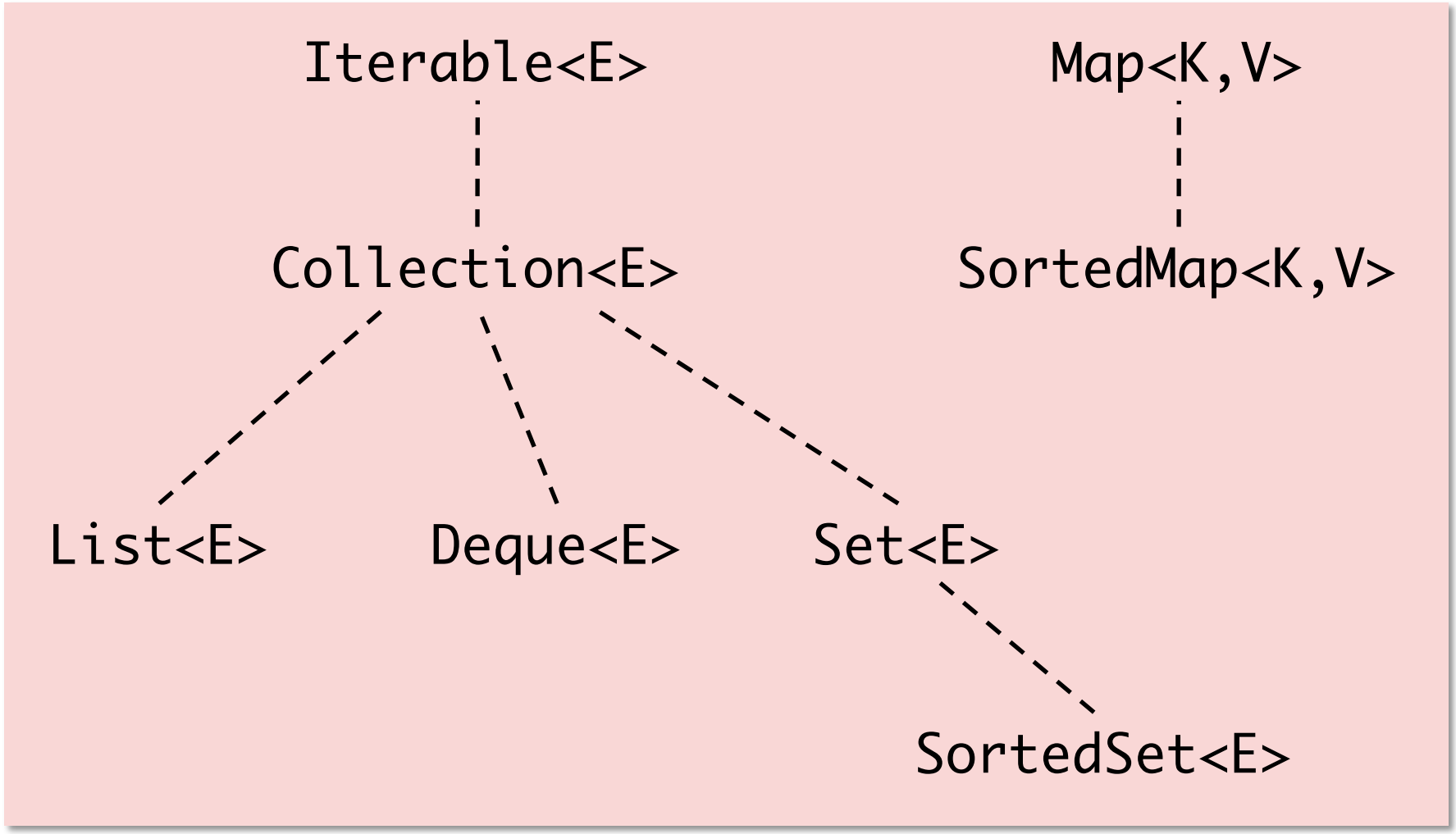
Midterm 2

- Grades available online
- Stats:
  - median: 75
  - stddev: 14.31
  - max: 99



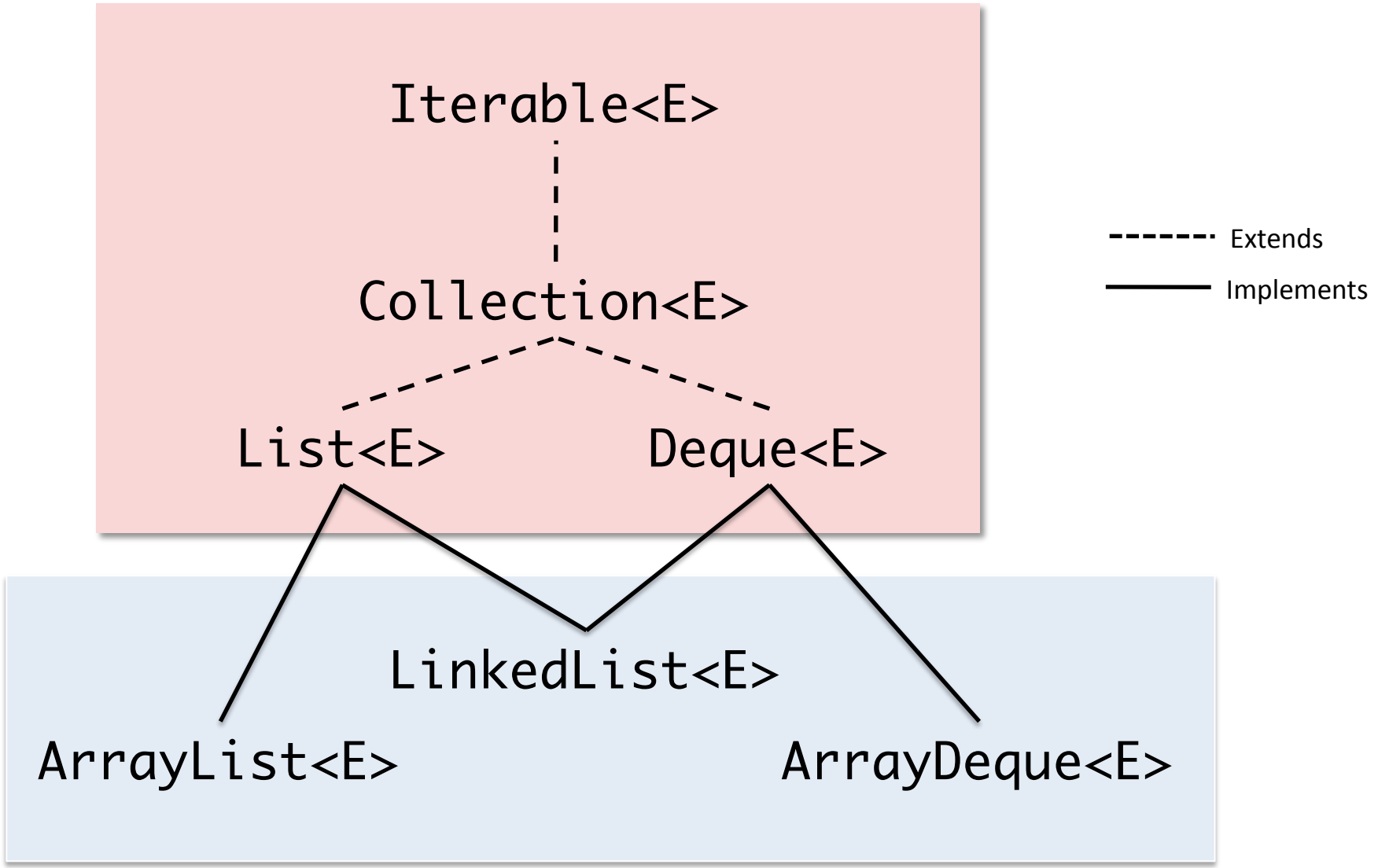
- Grade breakdown
  - > 80 A
  - 62-79 B
  - 46-61 C
  - < 45 D
- Make-up exam tomorrow, will be able to view exams with Ms. Laura Fox in Levine 308 starting Wed.

Interfaces* of the Collections Library



*not all of them!

Sequences


Iterable<E>
Collection<E>
List<E>
Deque<E>
LinkedList<E>
ArrayList<E>
ArrayDeque<E>
Extends
Implements

Iterator and Iterable

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

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

Iterator example

```
List<Integer> nums = new LinkedList<Integer>();
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;
}
```

What happens at end of loop?

1. sumElts and numElts are both 0
2. sumElts is 3 and numElts is 2
3. sumElts is 10 and numElts is 3
4. NullPointerException
5. Something else

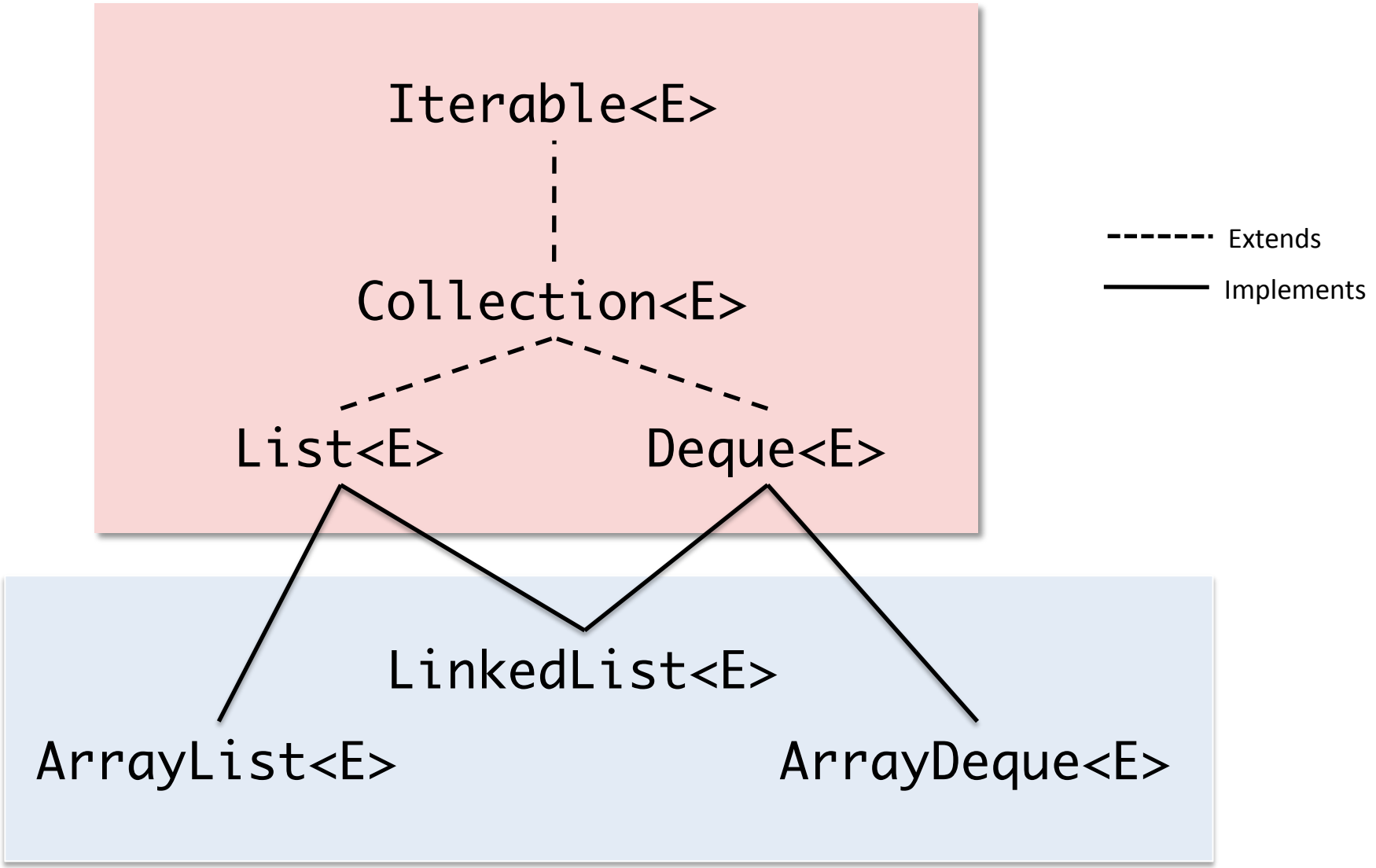
For-each version

```
List<Integer> nums = new LinkedList<Integer>();
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;
}
```

Subtyping and Generics

Subtyping and Generics



Subtyping and Generics

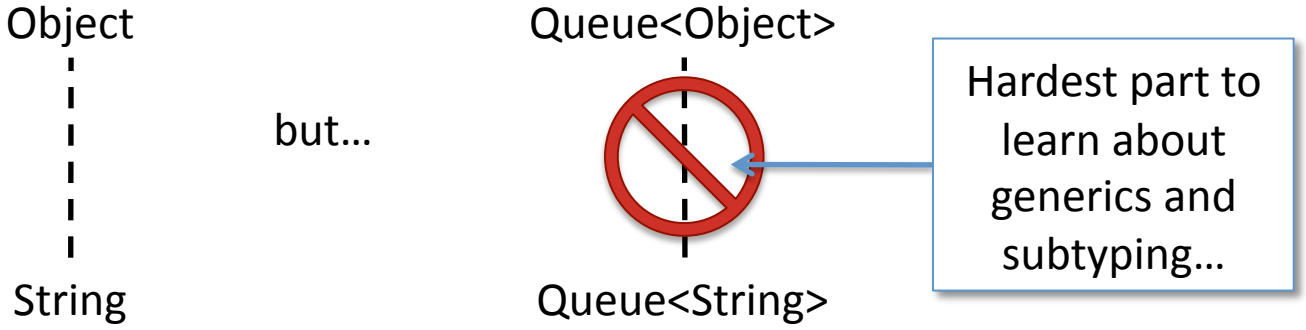
```
Queue<String> qs = new QueueImpl<String>();
Queue<Object> qo = qs;

qo.enq(new Object());
String s = qs.deq();
```

```
Ok?  Sure!
Ok?  Let's see…

Ok?  I guess
Ok?  Noooo!
```

- Java generics are *invariant*:
  - Subtyping of *arguments* to generic types does not imply subtyping between the instantiations:



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*
  - *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 HW 7

```
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
         );
  }
```

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 Exception();
    }
}
```

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

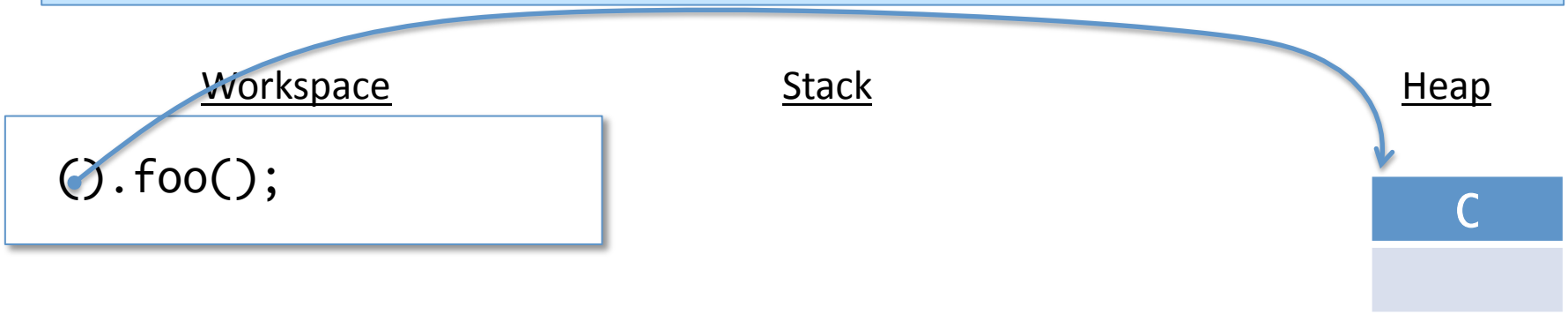
Workspace

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

Stack

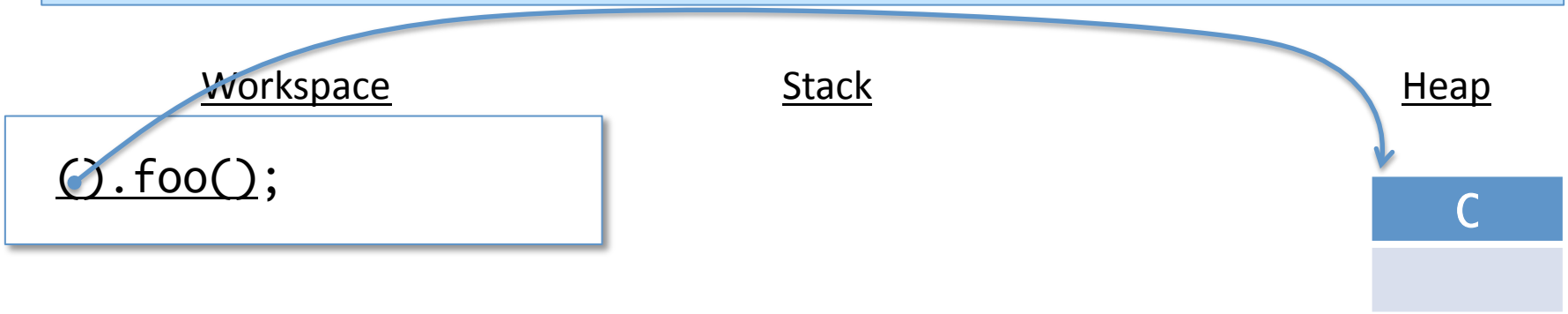
Heap

Abstract Stack Machine

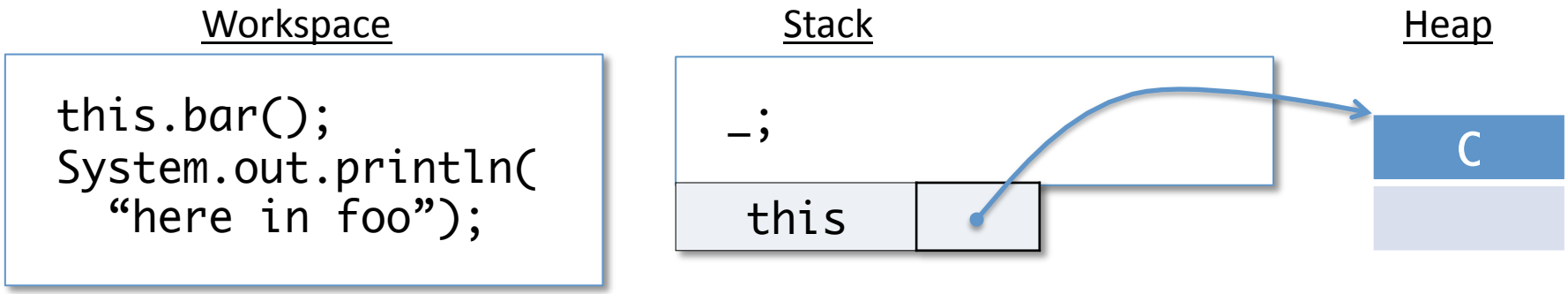


Allocate a new instance of C in the heap. (Skipping details of trivial constructor for C.)

Abstract Stack Machine

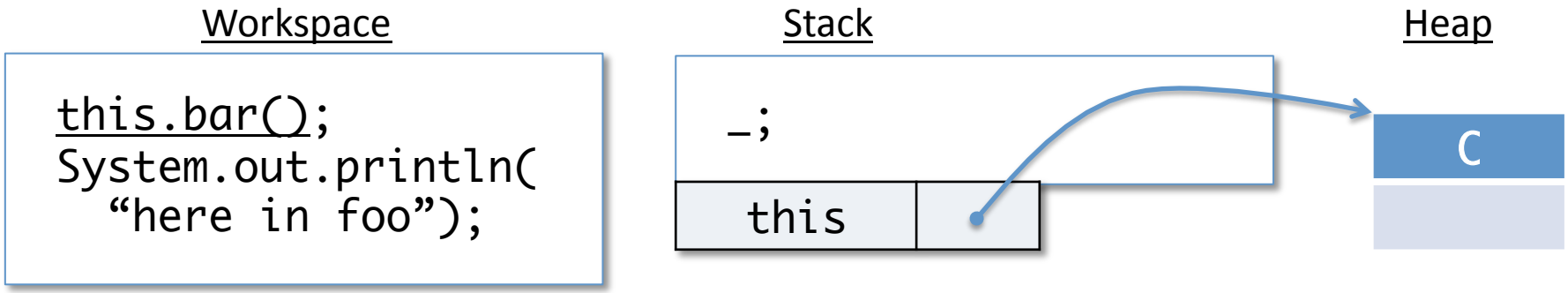


Abstract Stack Machine

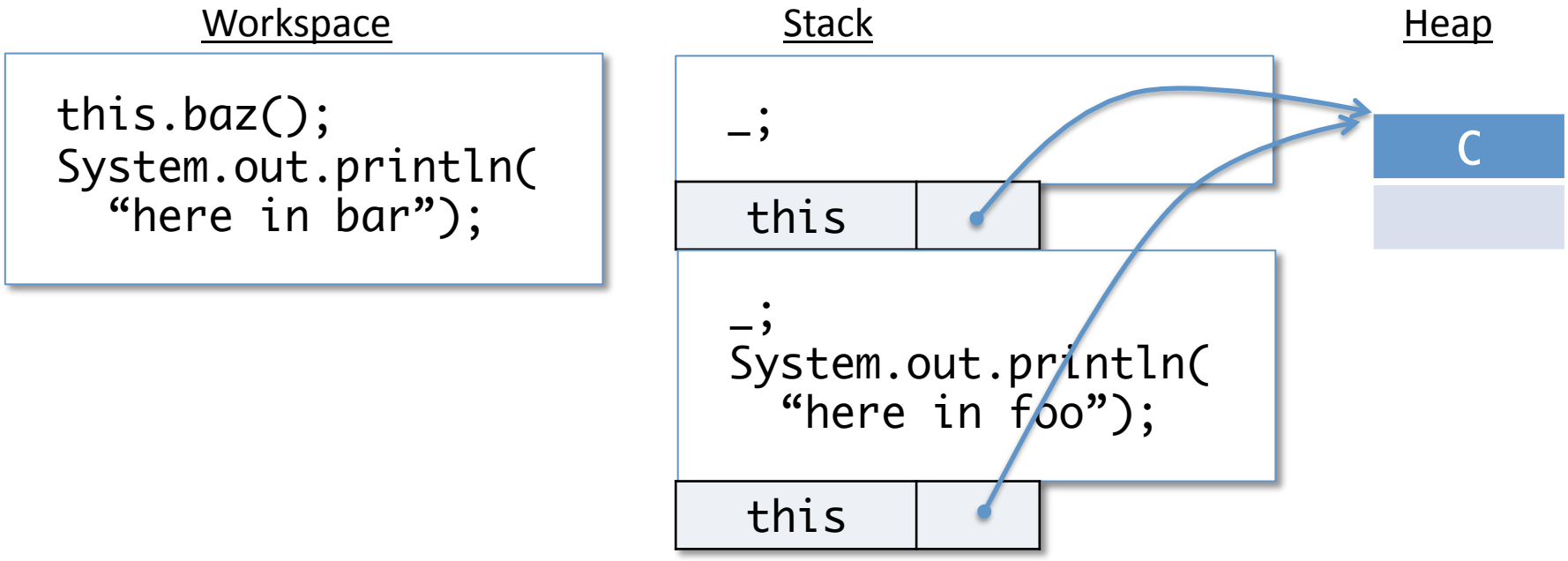


Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack. Use the dynamic class to lookup the method body from the class table.

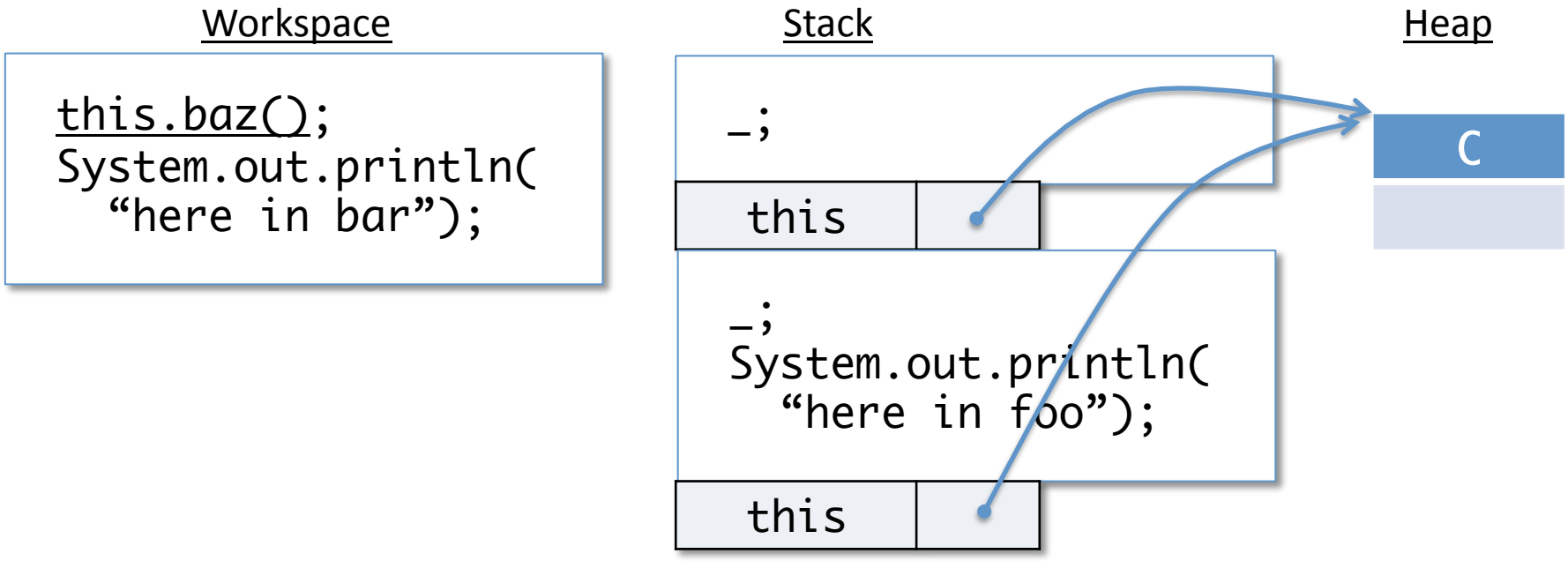
Abstract Stack Machine



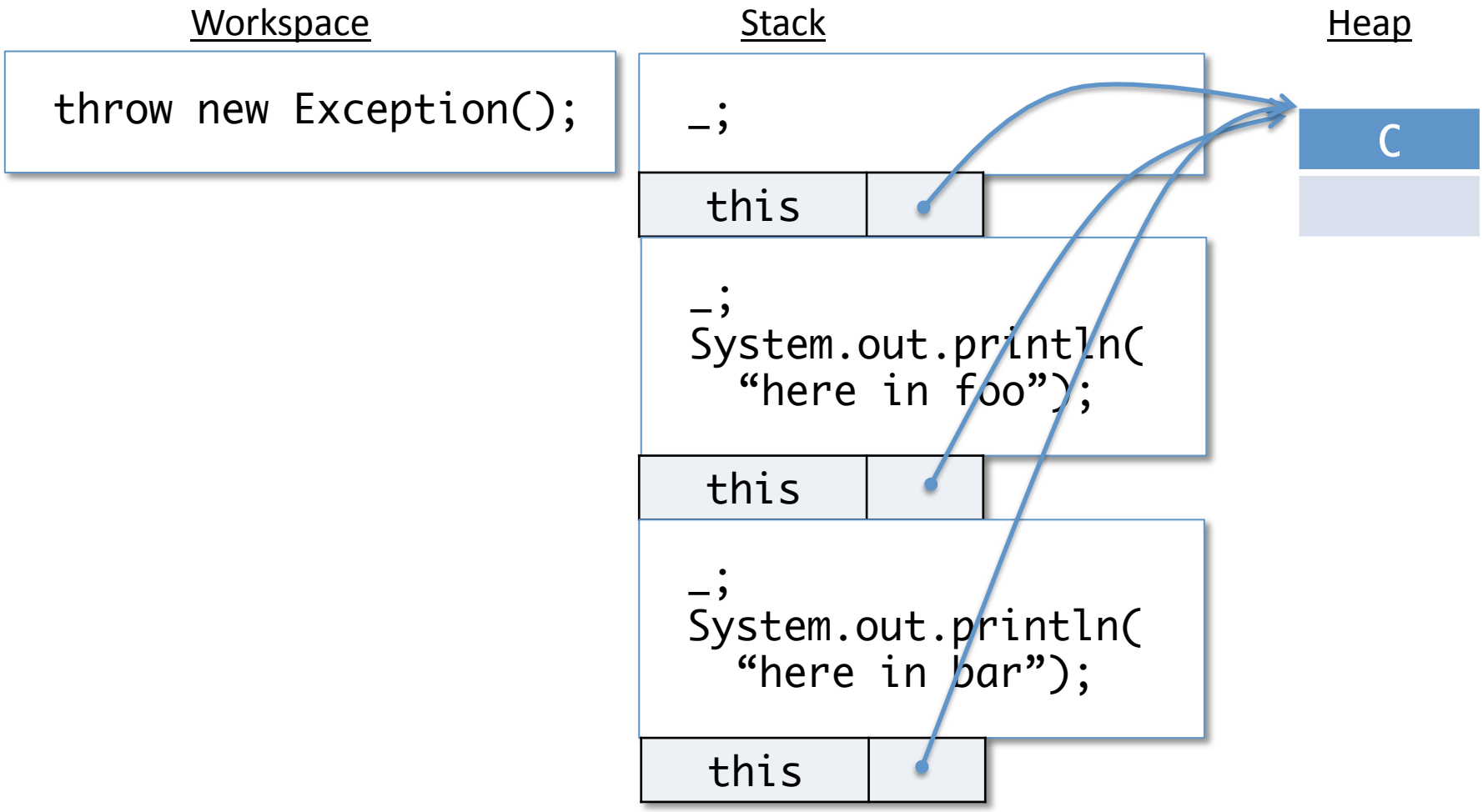
Abstract Stack Machine



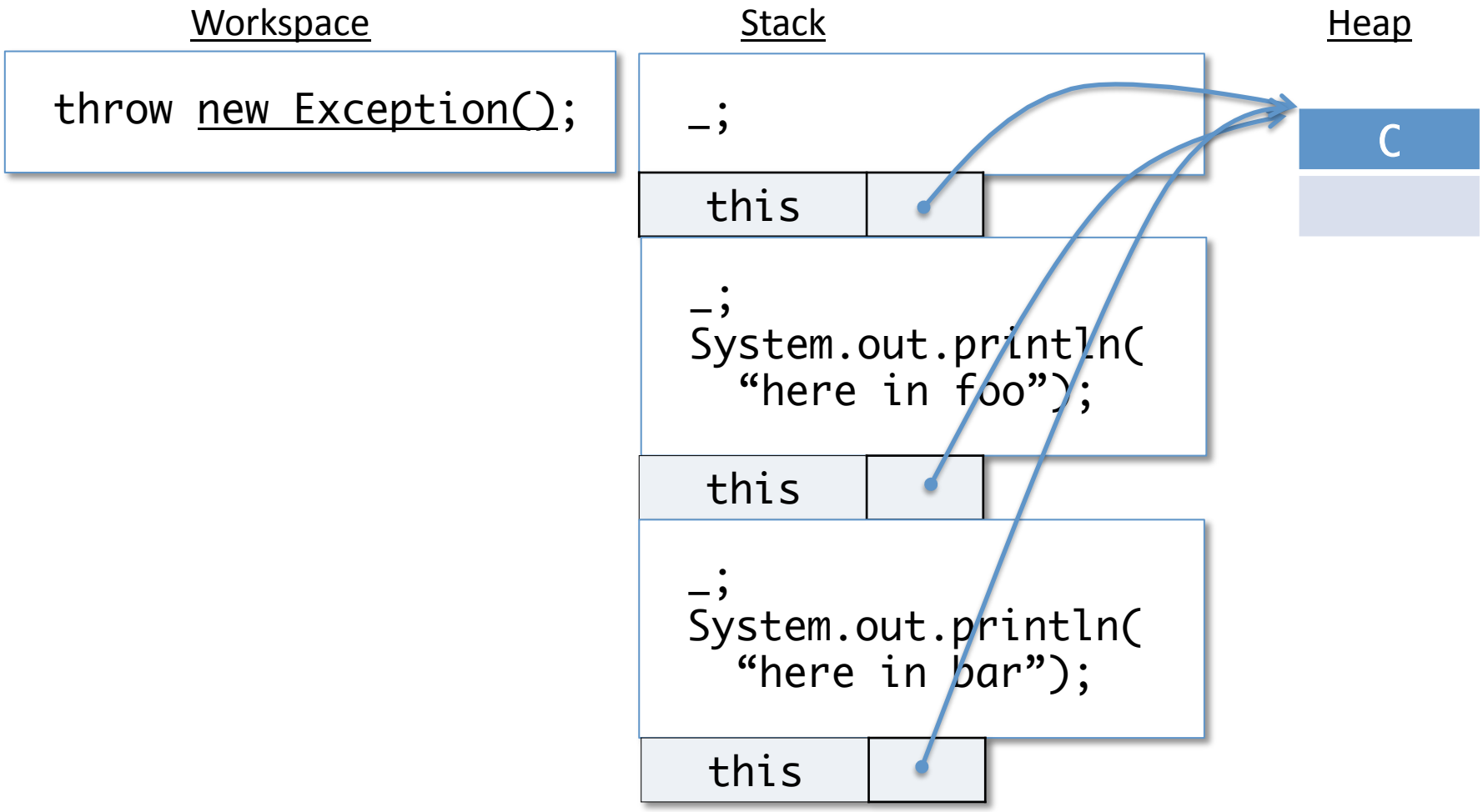
Abstract Stack Machine



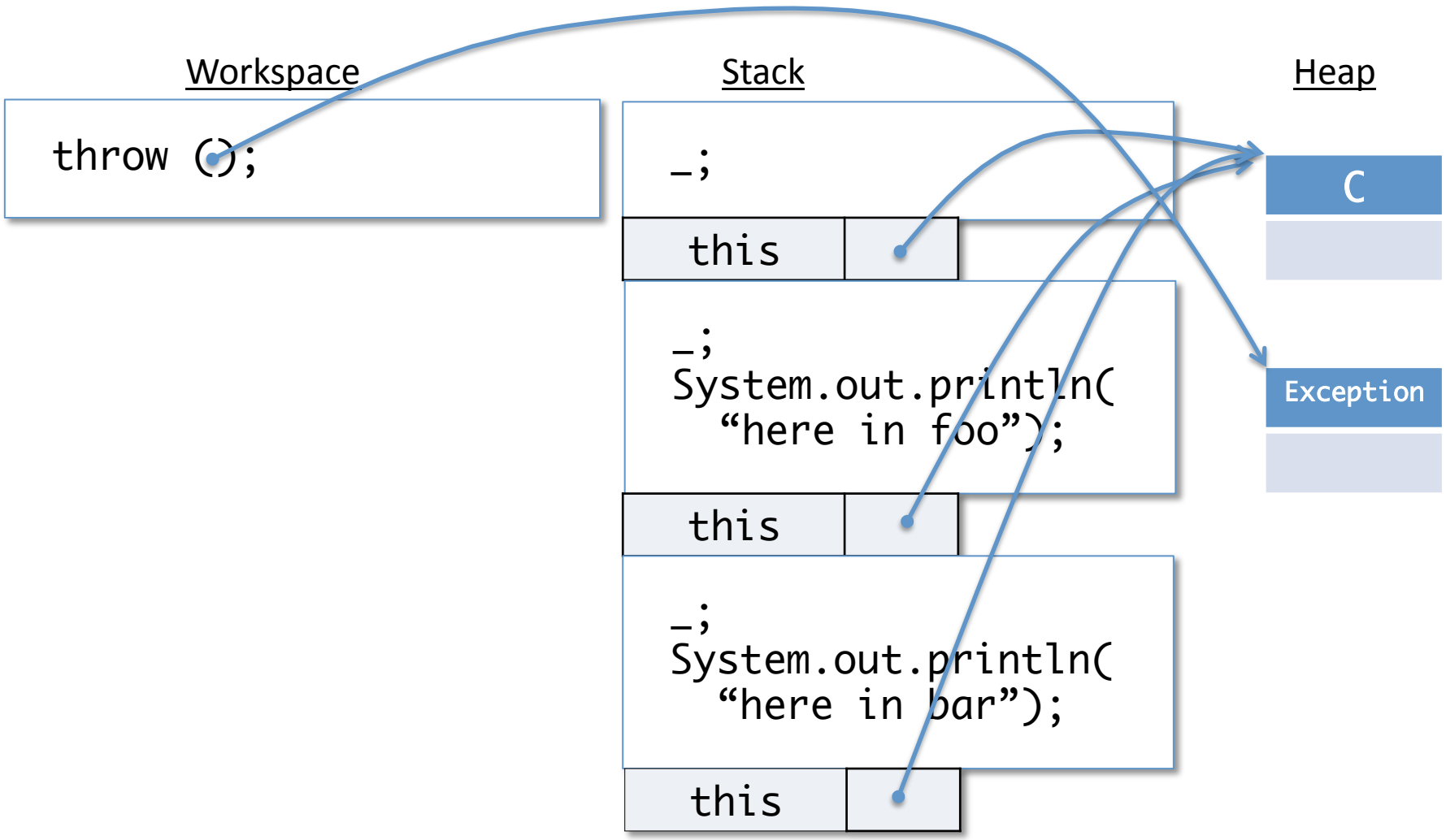
Abstract Stack Machine



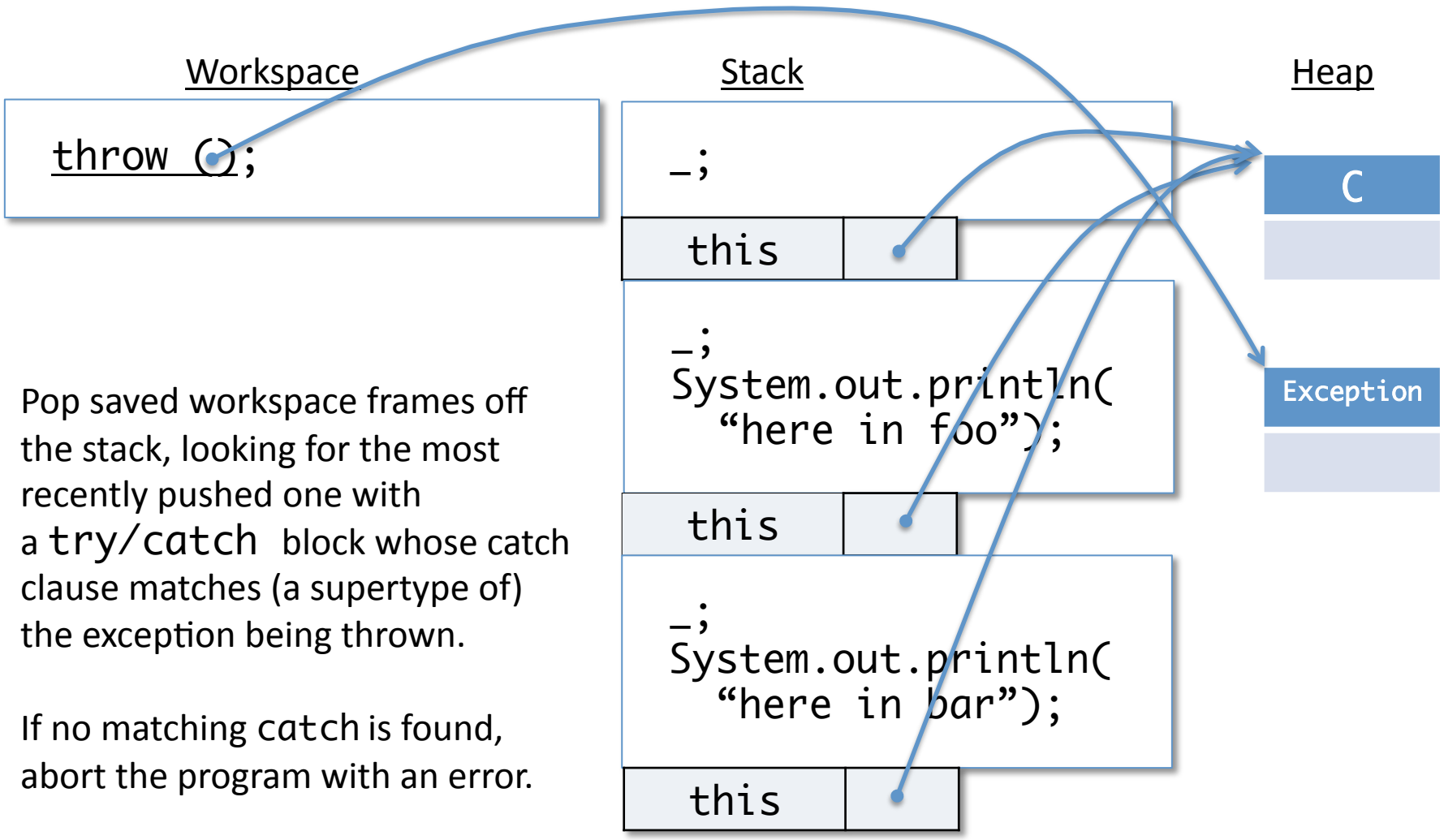
Abstract Stack Machine



Abstract Stack Machine



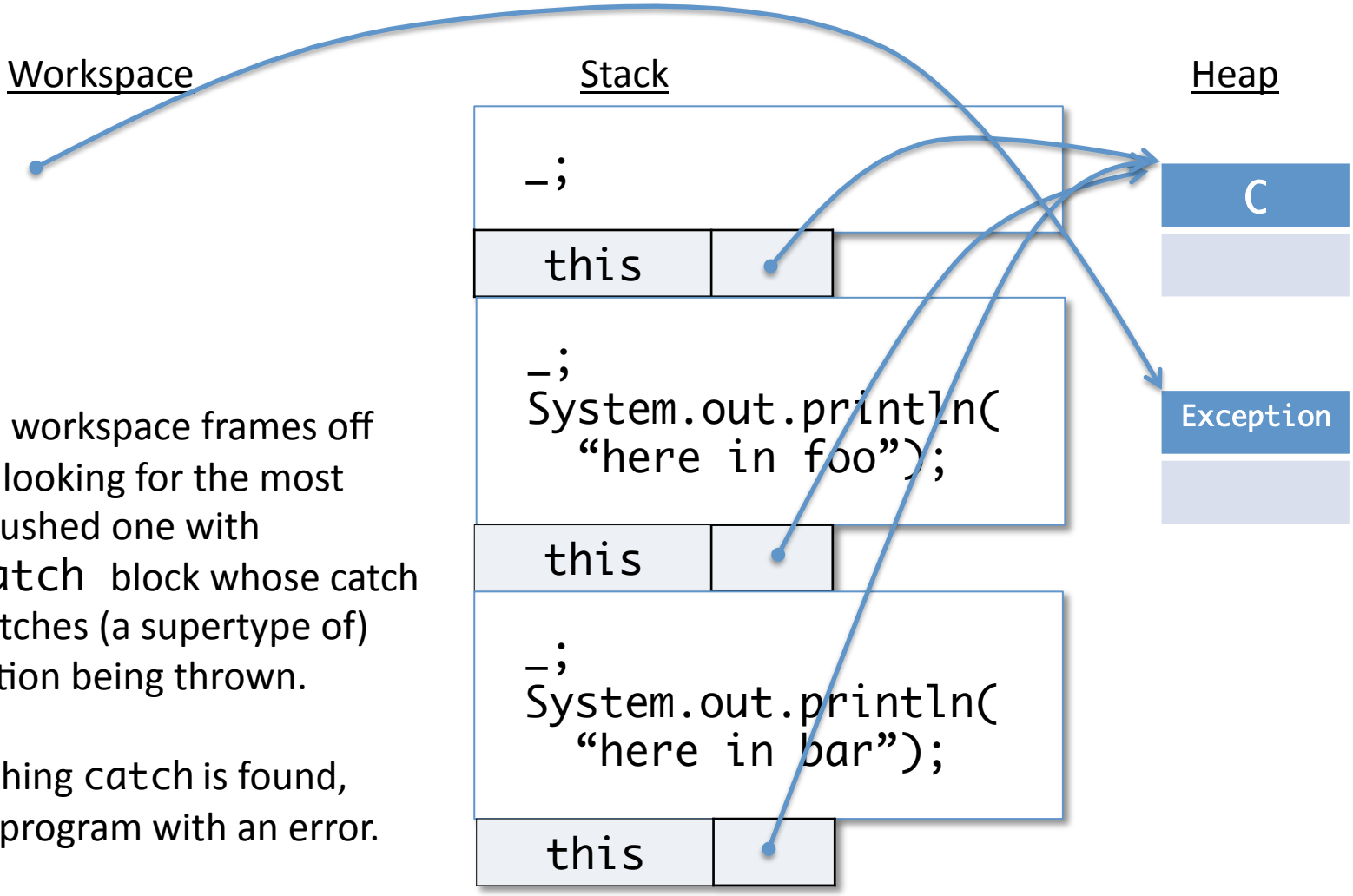
Abstract Stack Machine



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



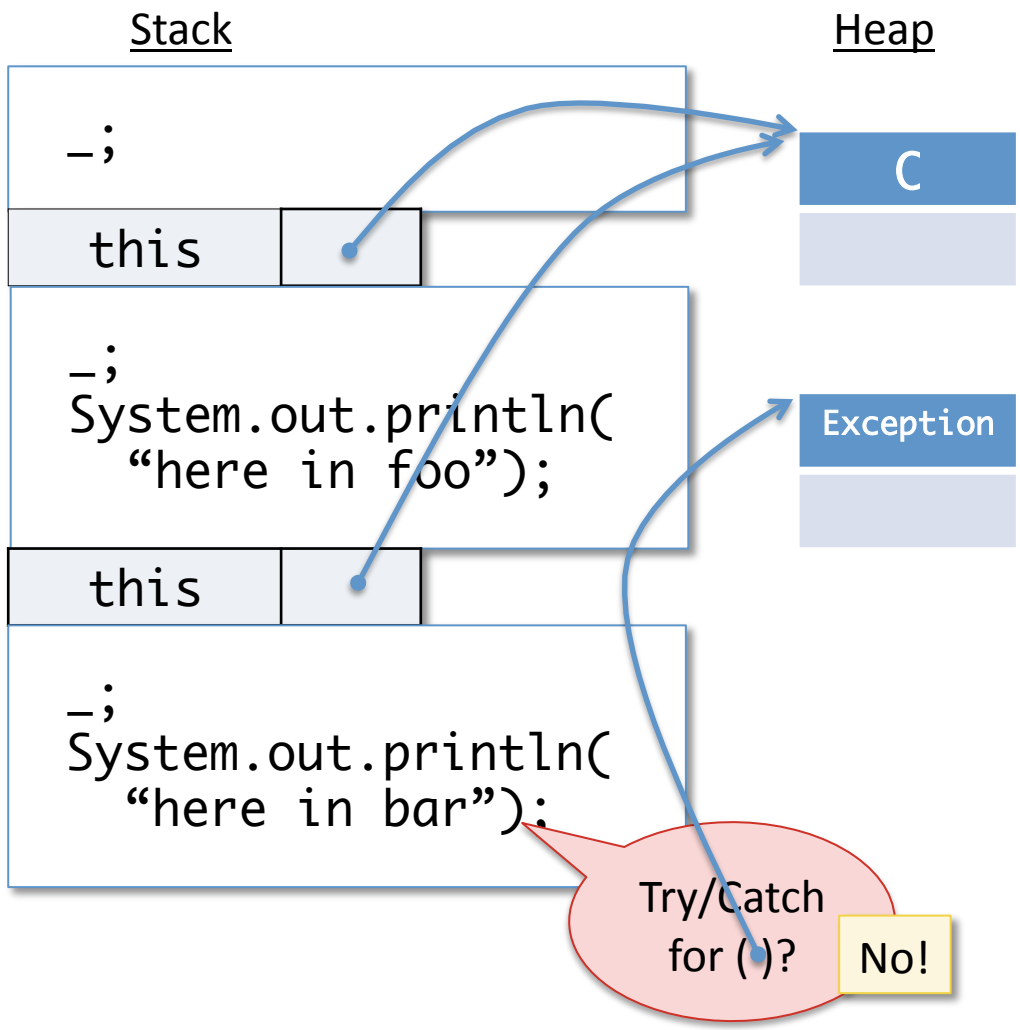
Pop saved workspace frames off the stack, looking for the most recently pushed one with a `try/catch` block whose catch clause matches (a supertype of) the exception being thrown.

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

Abstract Stack Machine

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.

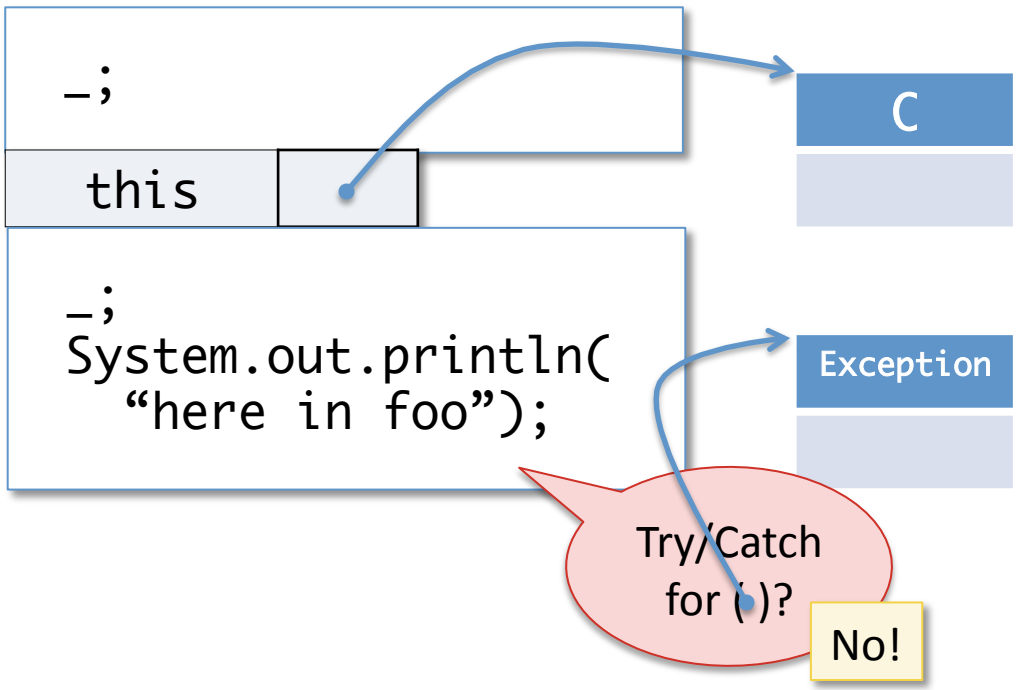


Abstract Stack Machine

Workspace

Stack

Heap

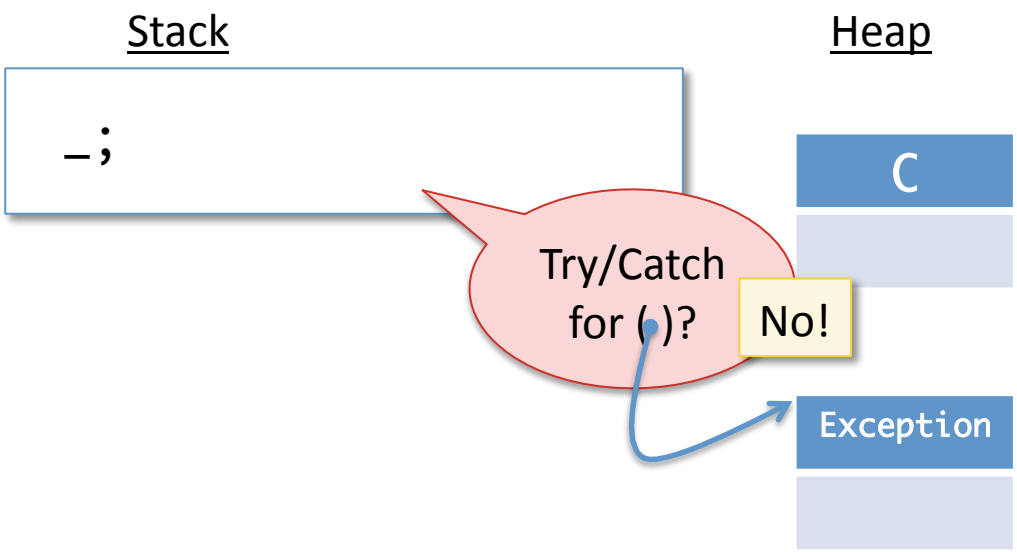


Discard the current workspace.

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 Exception();
    }
}
```

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

Abstract Stack Machine

Workspace

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

Stack

Heap

Abstract Stack Machine

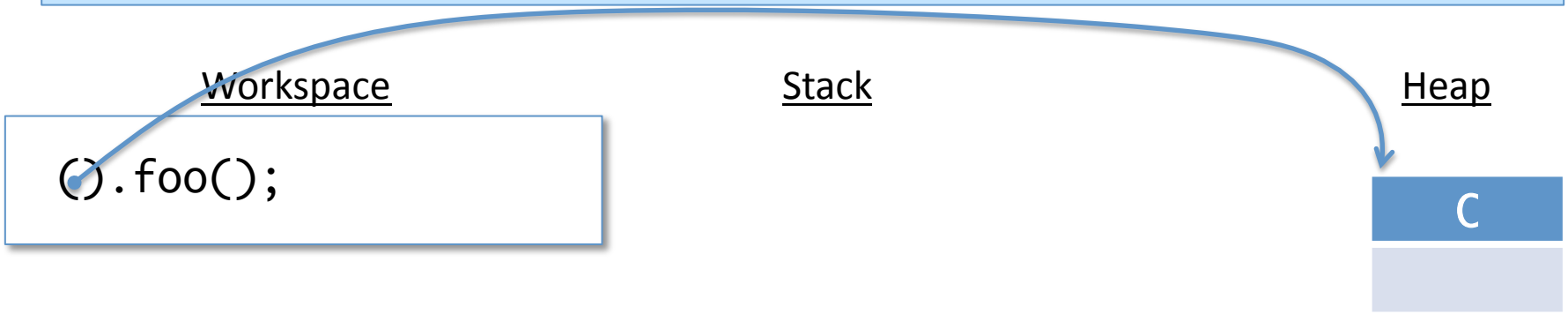
Workspace

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

Stack

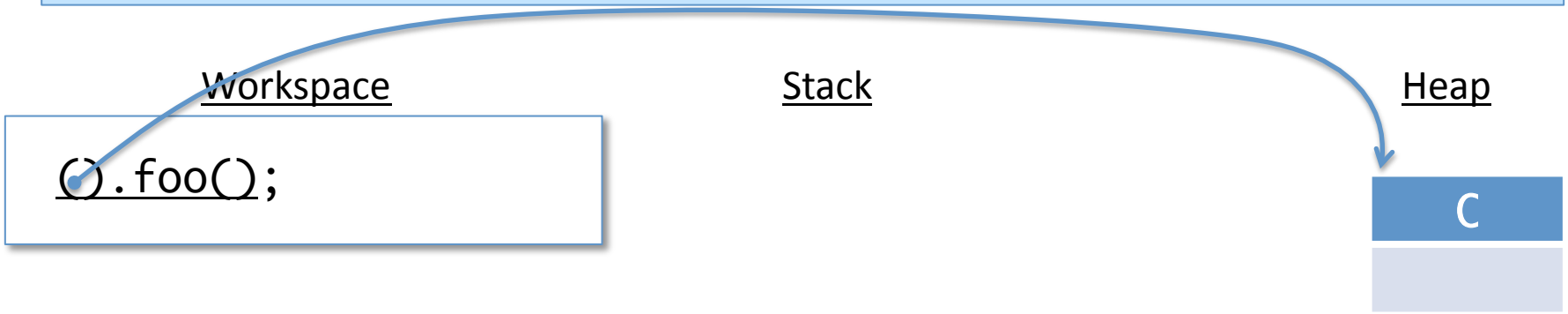
Heap

Abstract Stack Machine

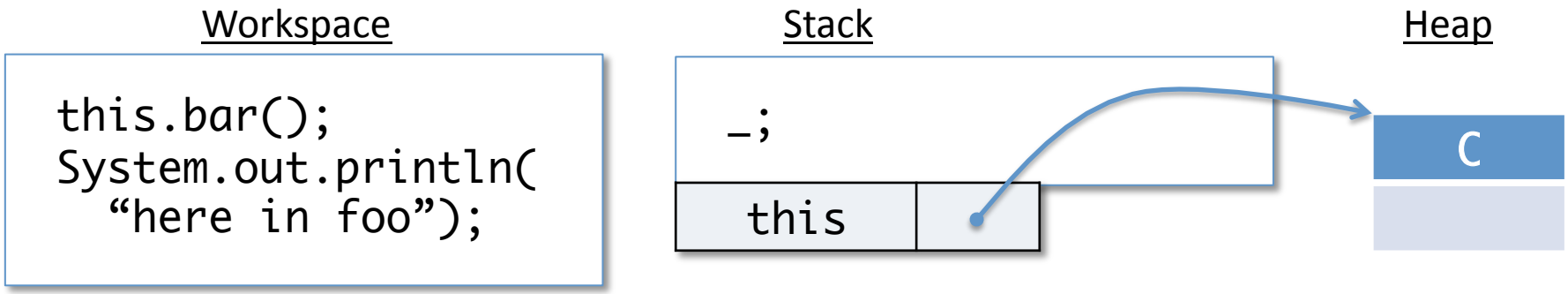


Allocate a new instance of C in the heap.

Abstract Stack Machine



Abstract Stack Machine

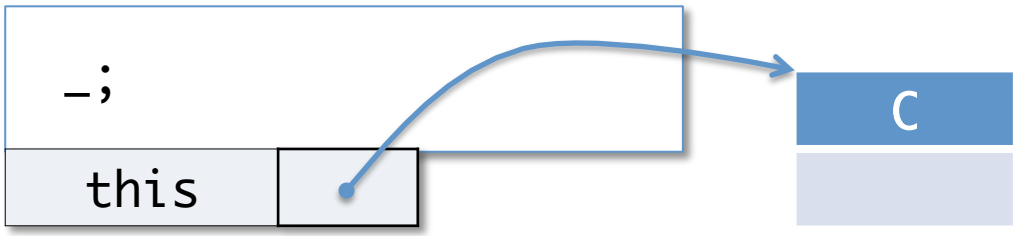


Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack.

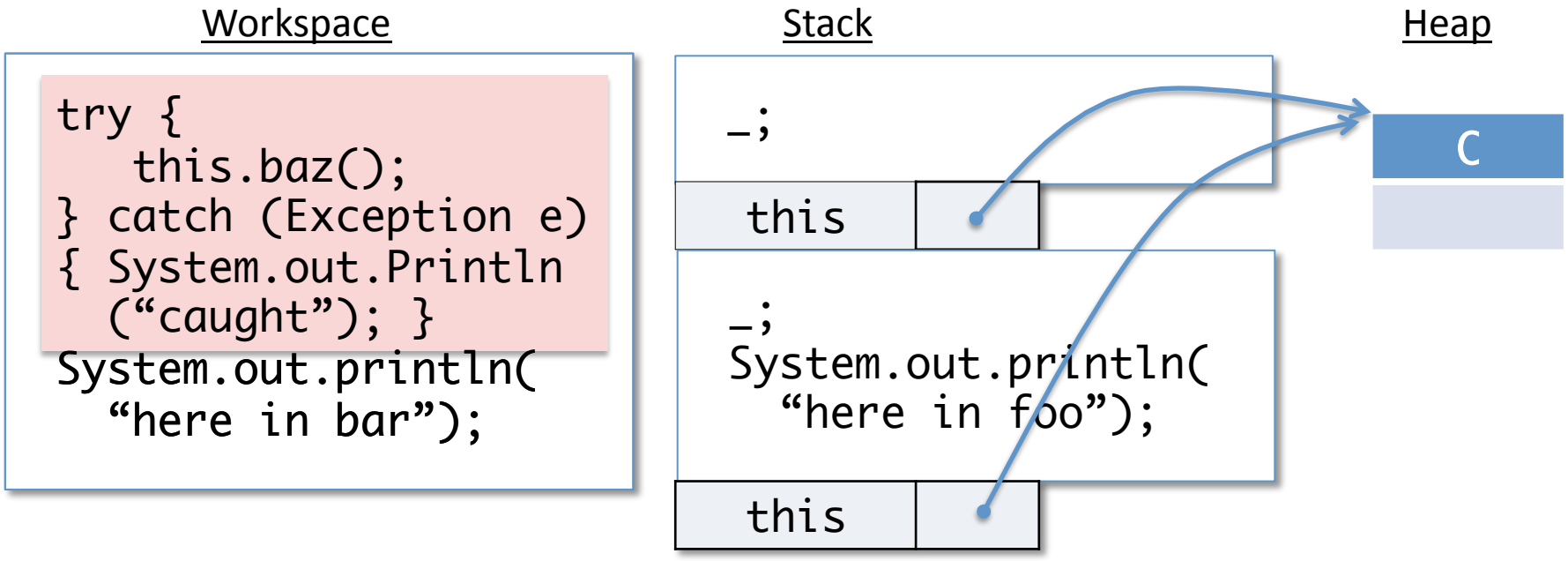
Abstract Stack Machine

Workspace

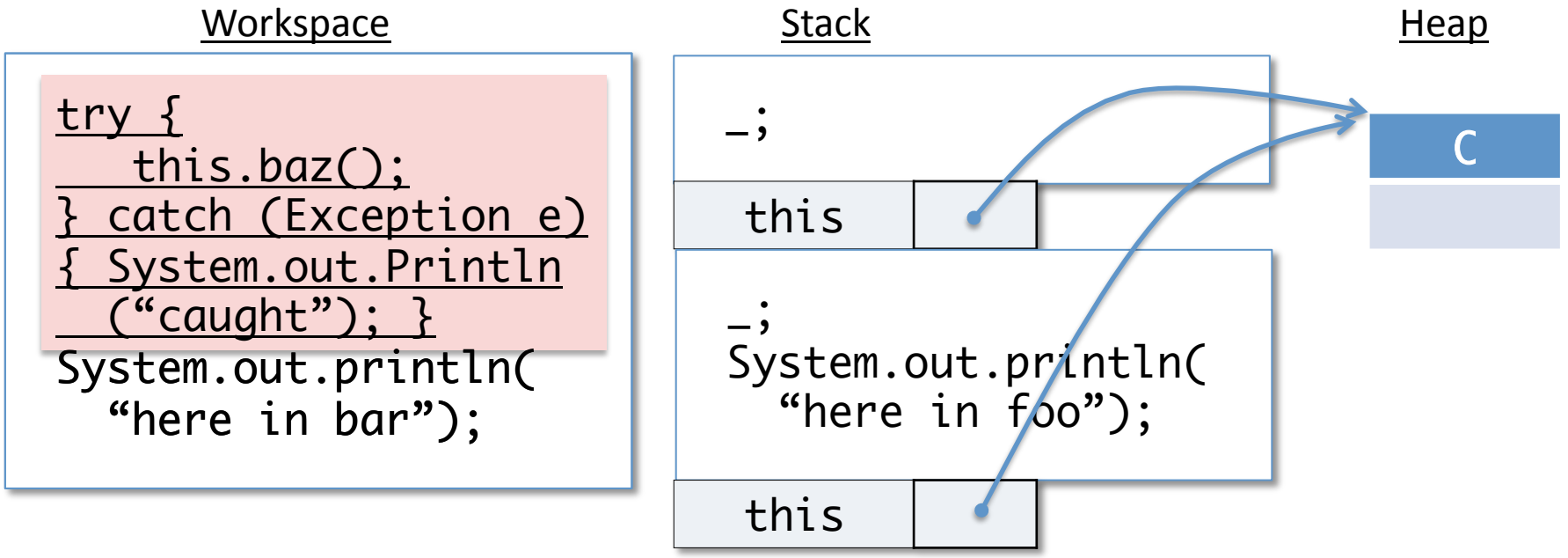
```
this.bar();
System.out.println(
  "here in foo");
```



Abstract Stack Machine



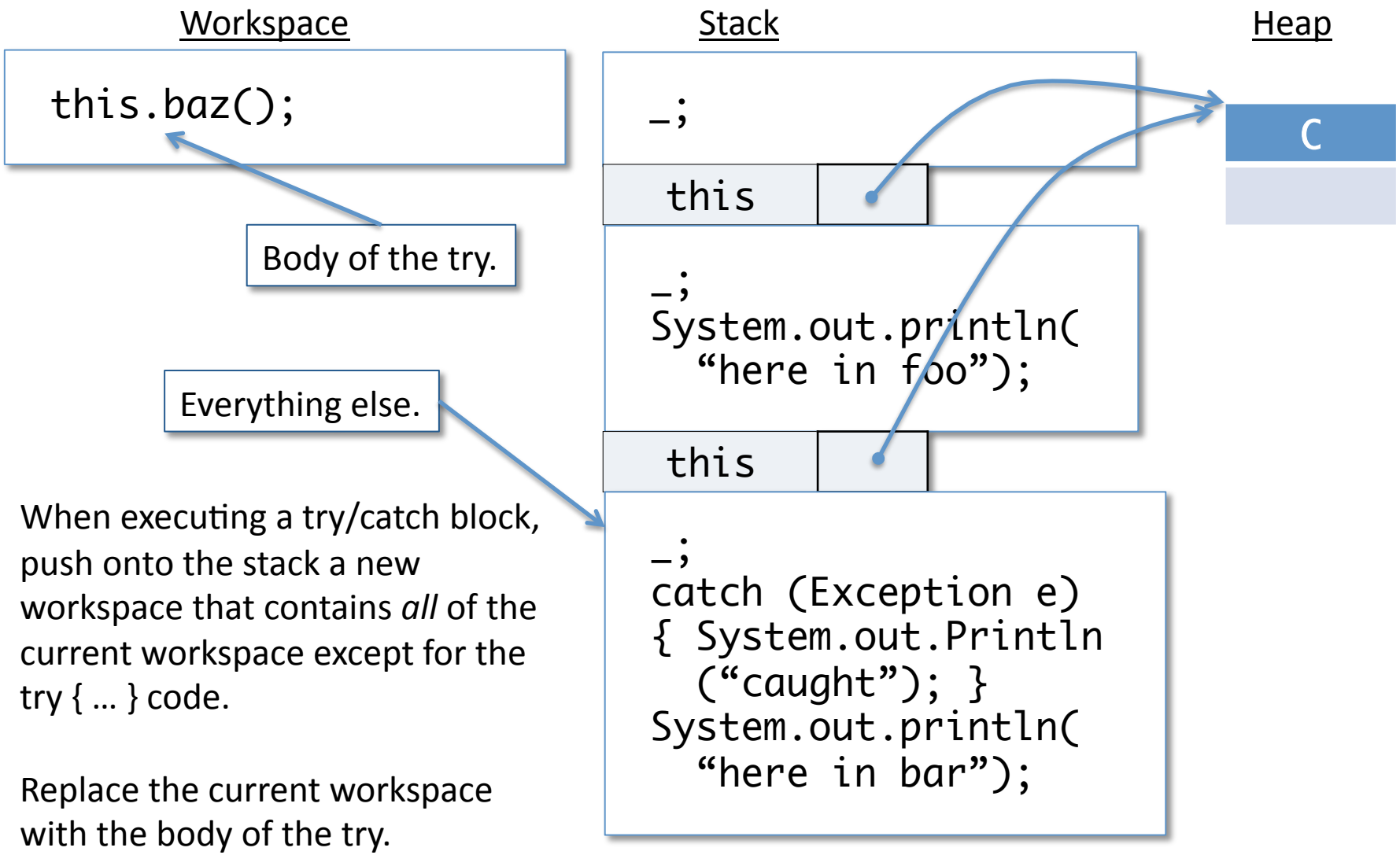
Abstract Stack Machine



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.

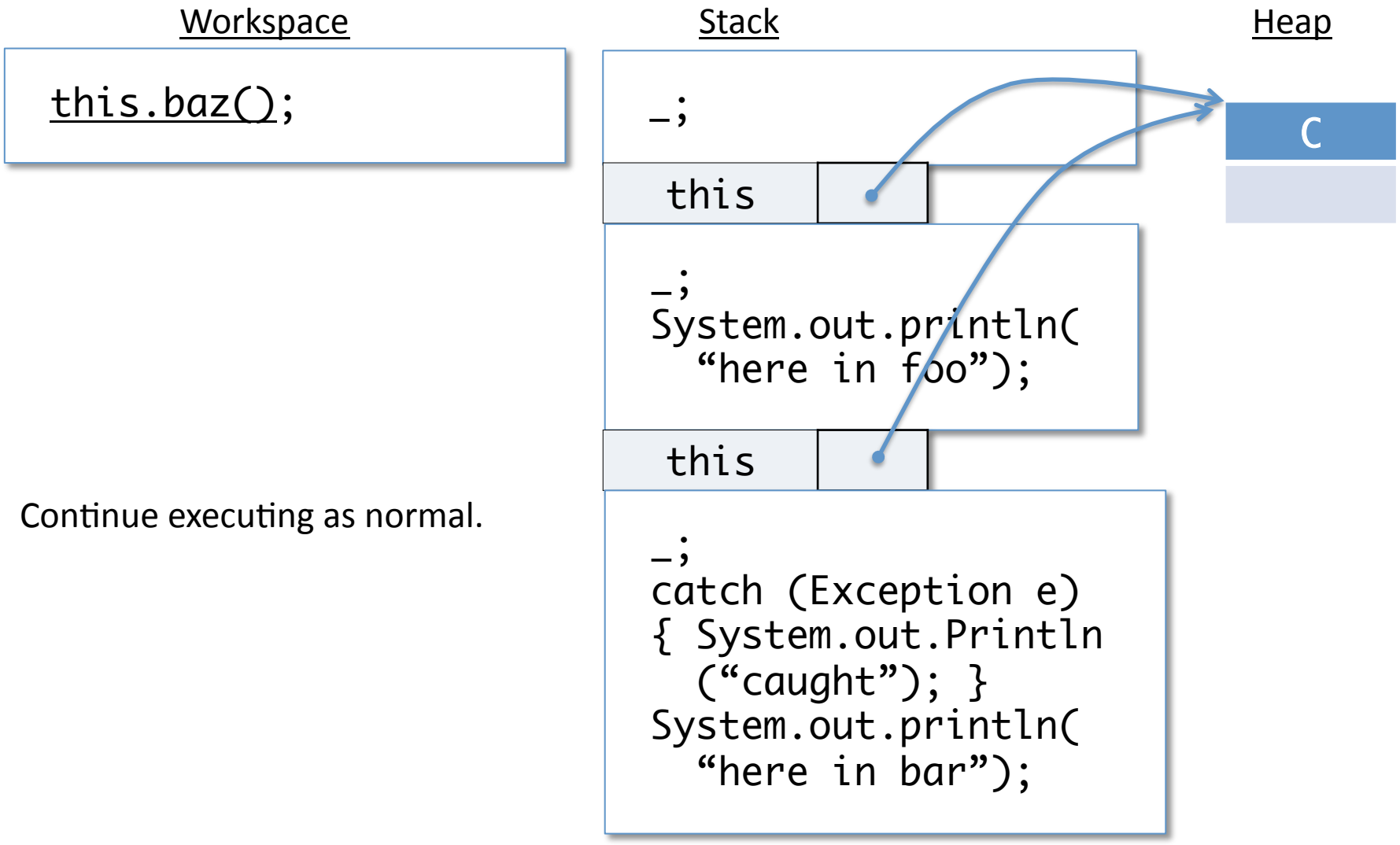
Abstract Stack Machine



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.

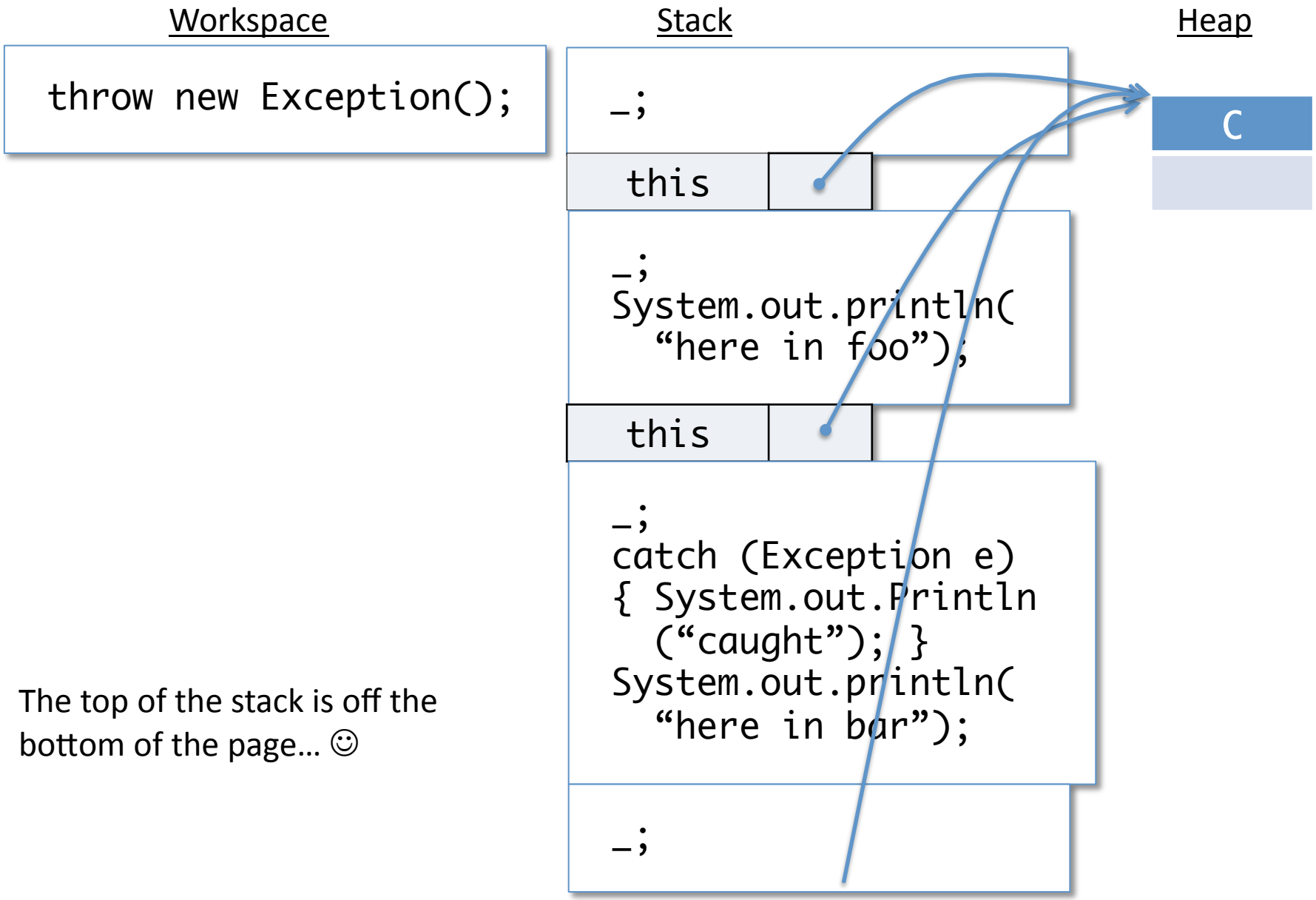
Replace the current workspace with the body of the try.

Abstract Stack Machine



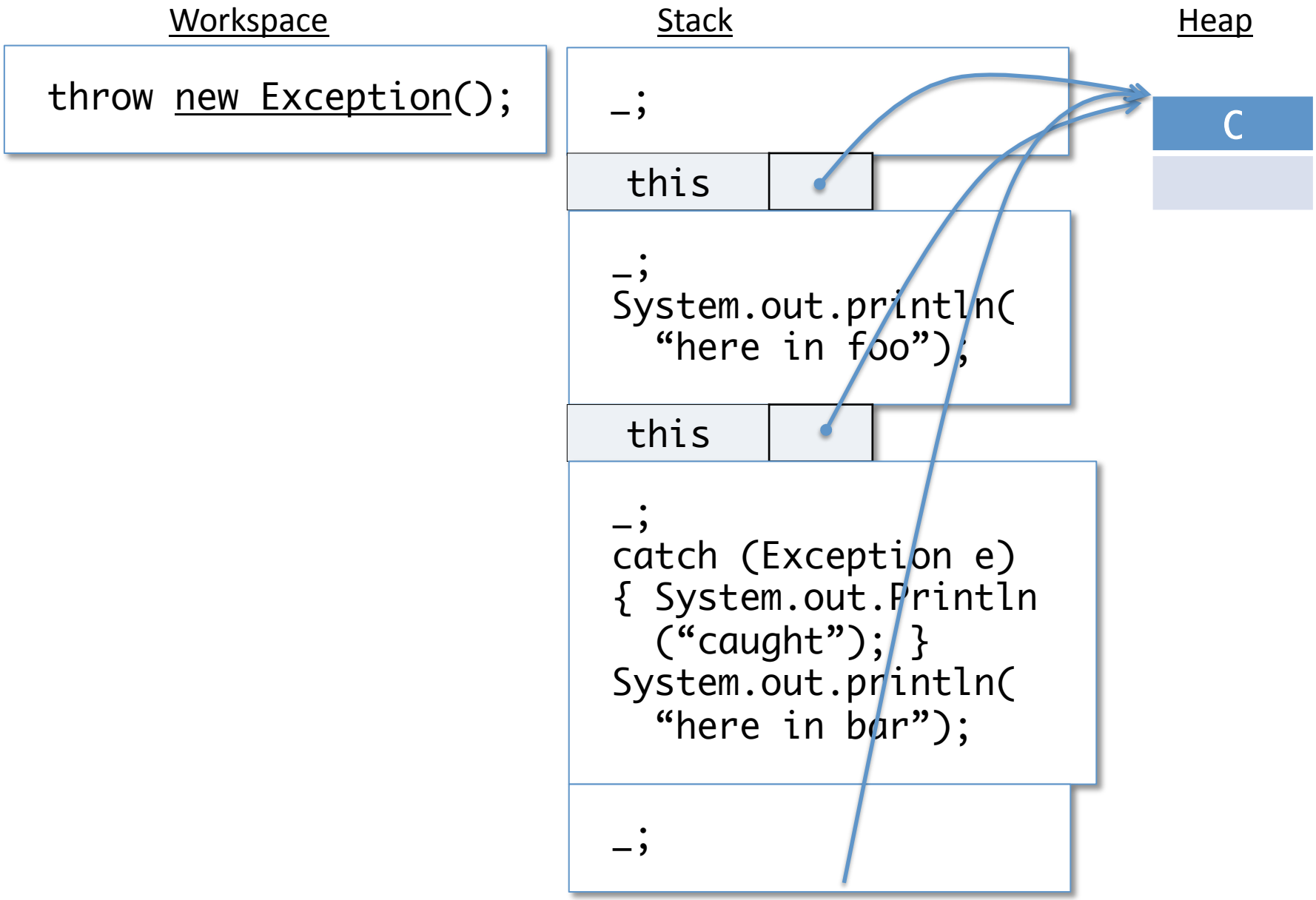
Continue executing as normal.

Abstract Stack Machine

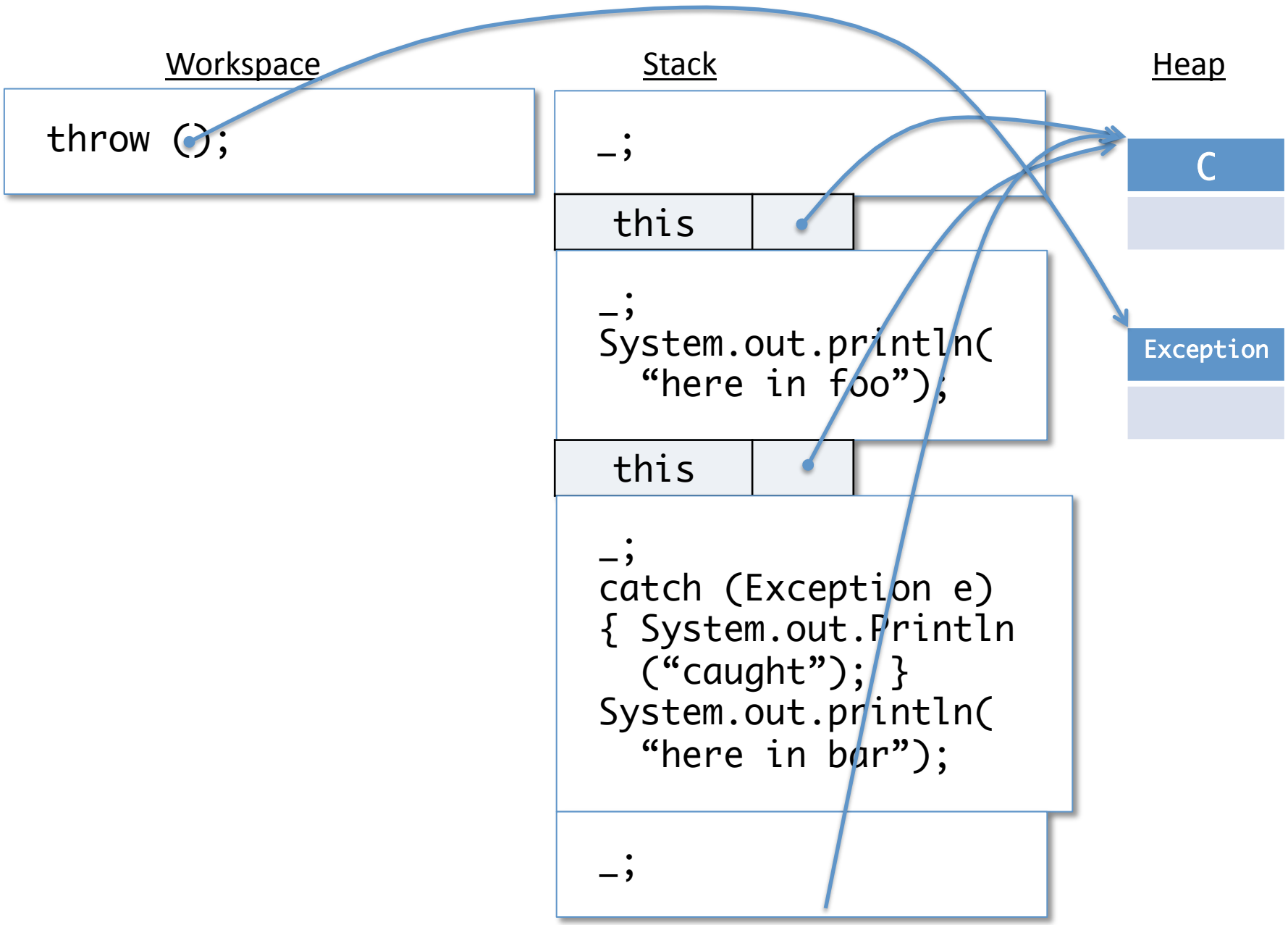


The top of the stack is off the bottom of the page… ☺

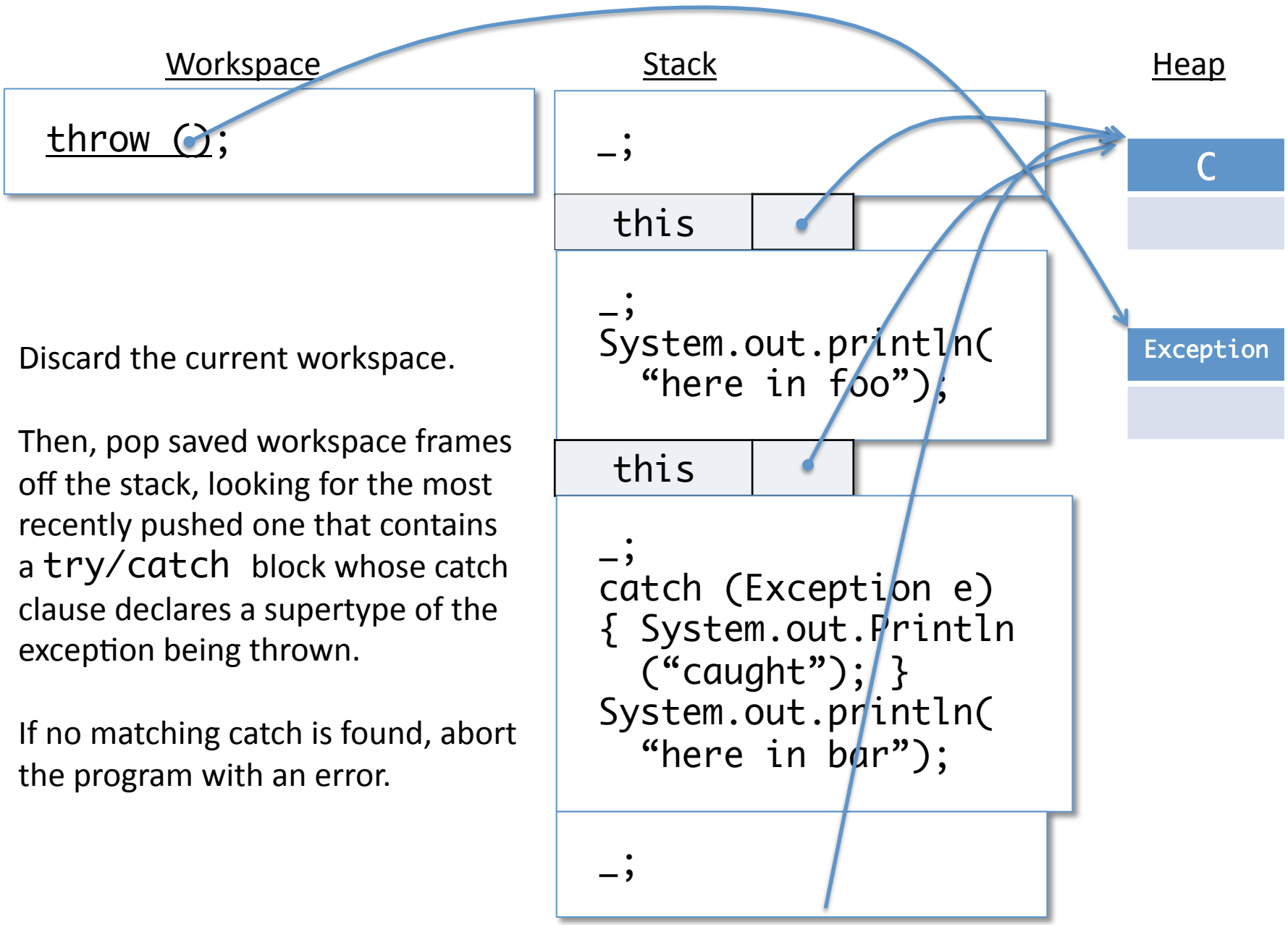
Abstract Stack Machine



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.

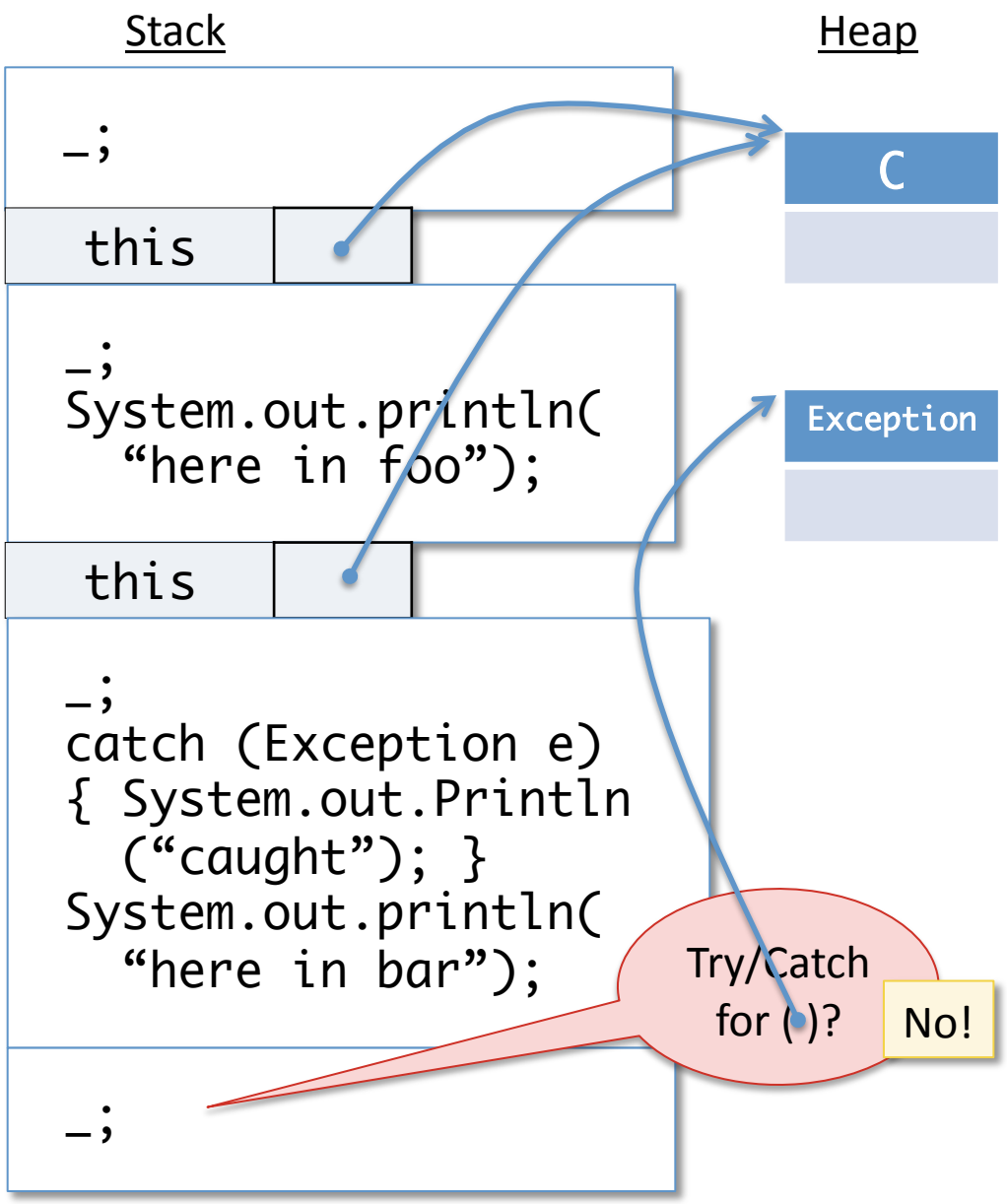
Abstract Stack Machine

Workspace

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.

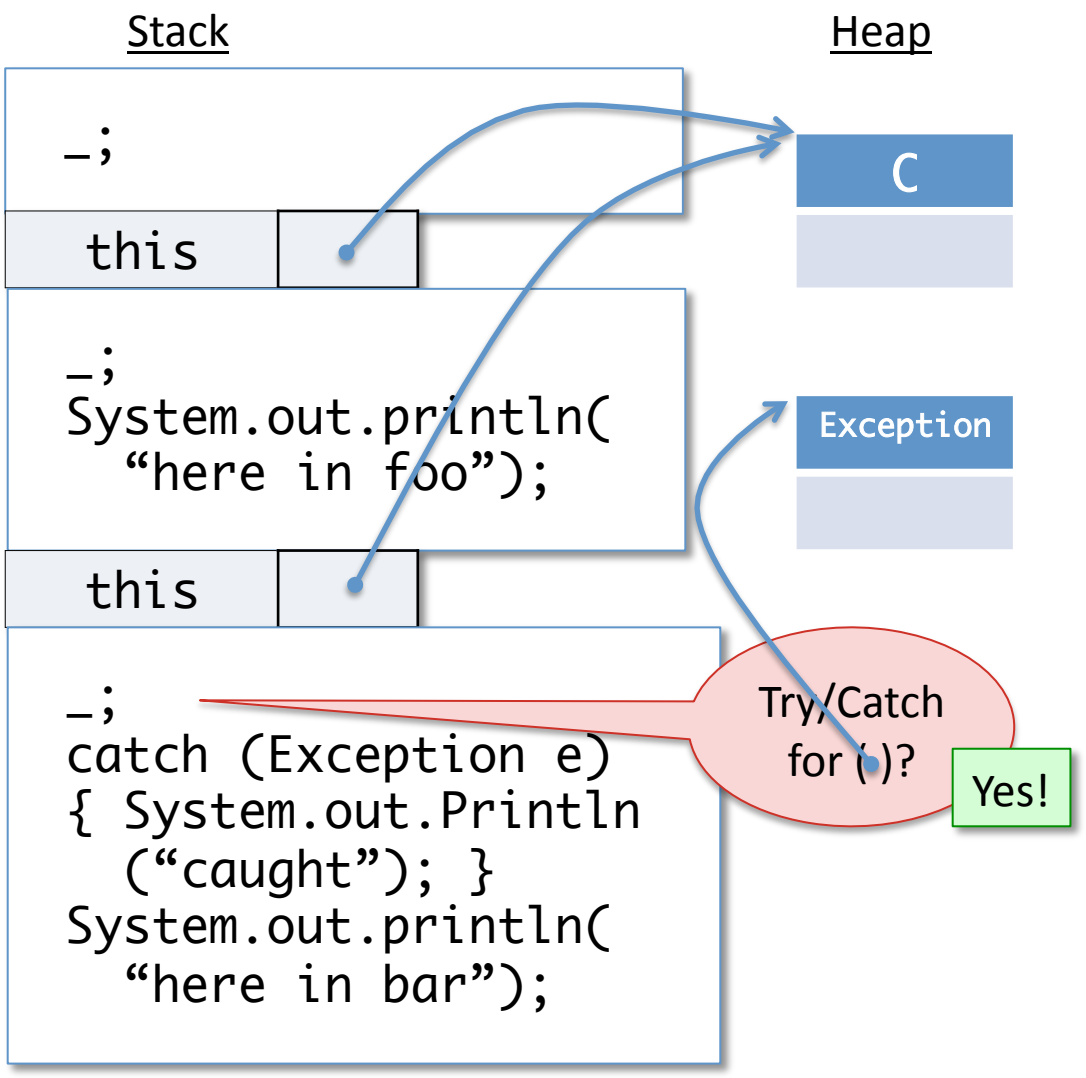
If no matching catch is found, abort the program with an error.



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.

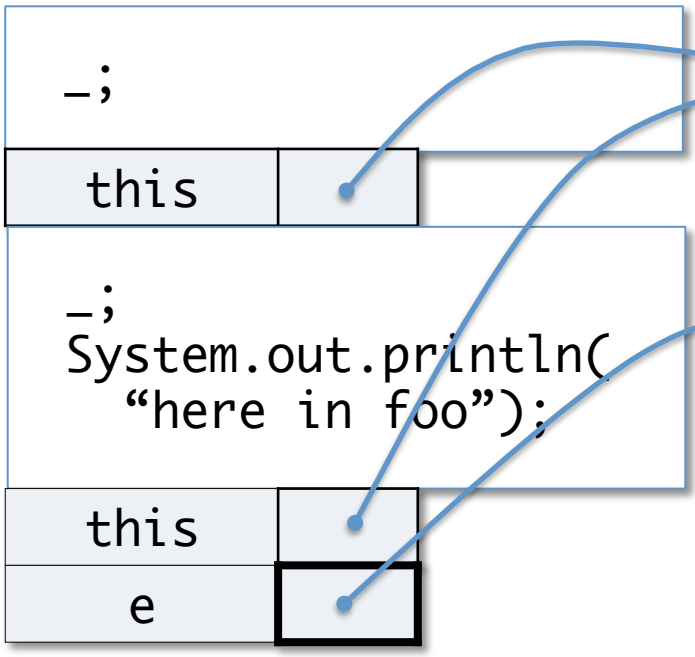


Abstract Stack Machine

Workspace

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

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.

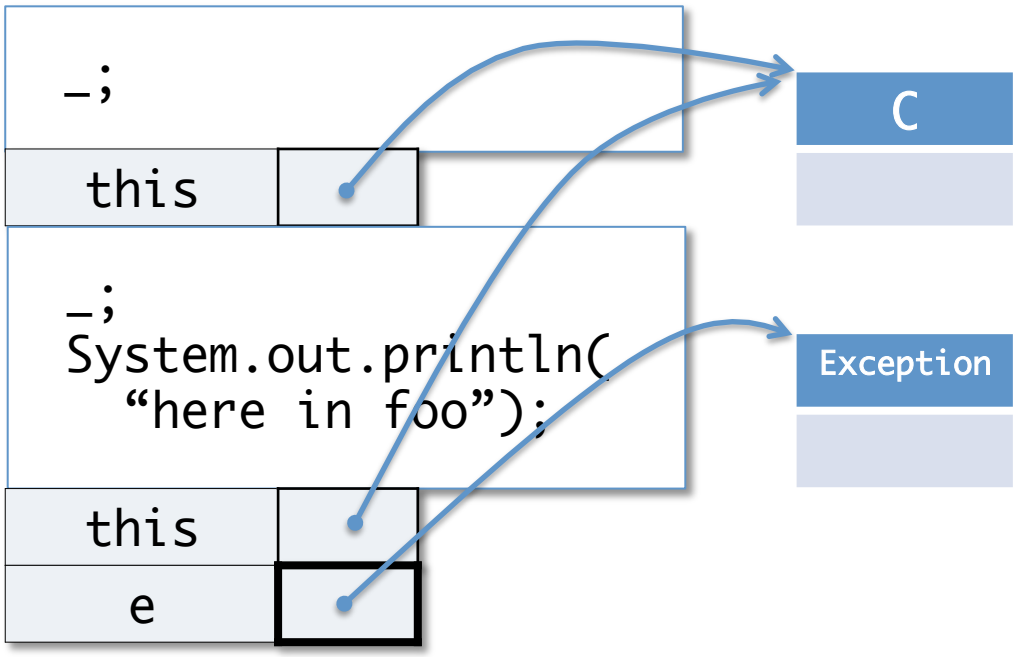
Continue executing as usual.

Abstract Stack Machine

Workspace

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

Continue executing as usual.

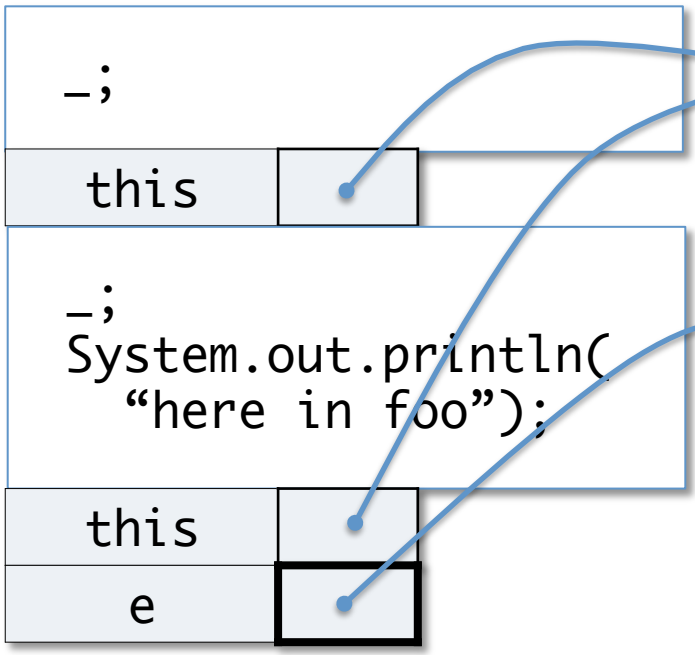


Abstract Stack Machine

Workspace

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

Stack



Heap

Continue executing as usual.

Console

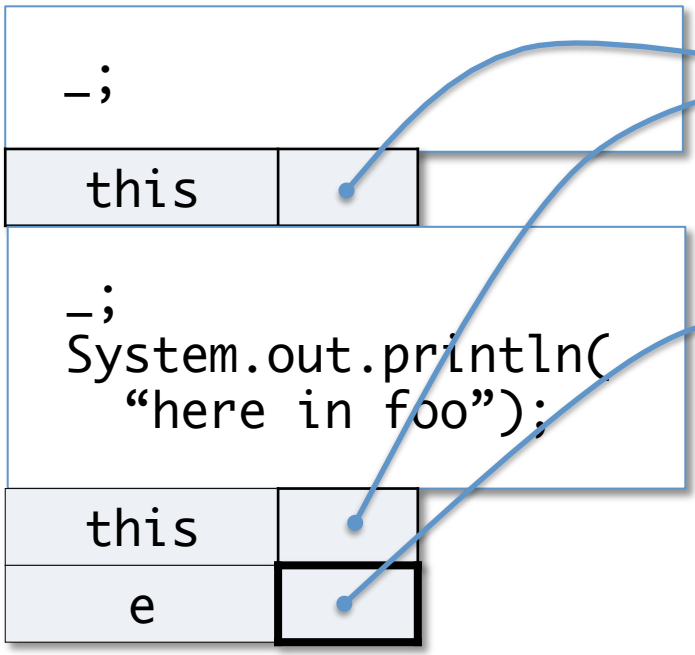
```
caught
```

Abstract Stack Machine

Workspace

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

Stack



Heap

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

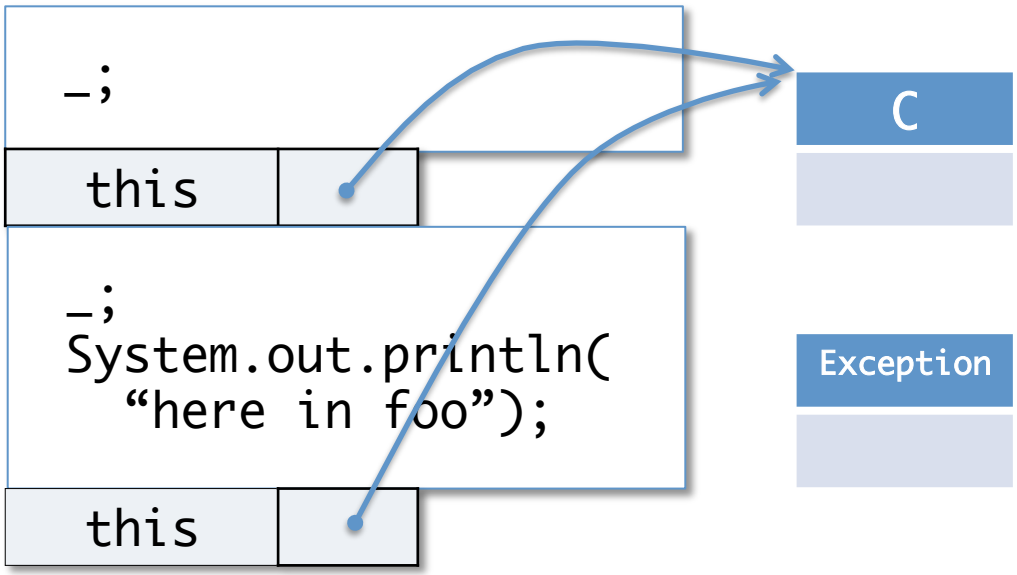
```
caught
```

Abstract Stack Machine

Workspace

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

Stack



Heap

Continue executing as usual.

Console

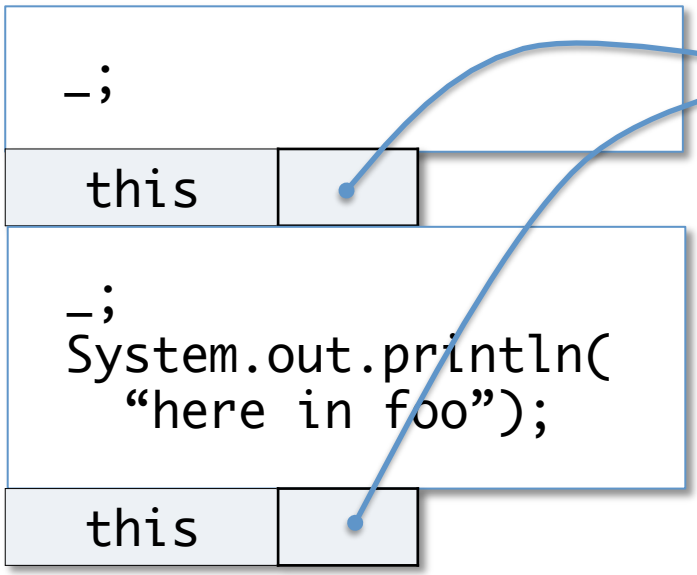
```
caught
```

Abstract Stack Machine

Workspace

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

Stack



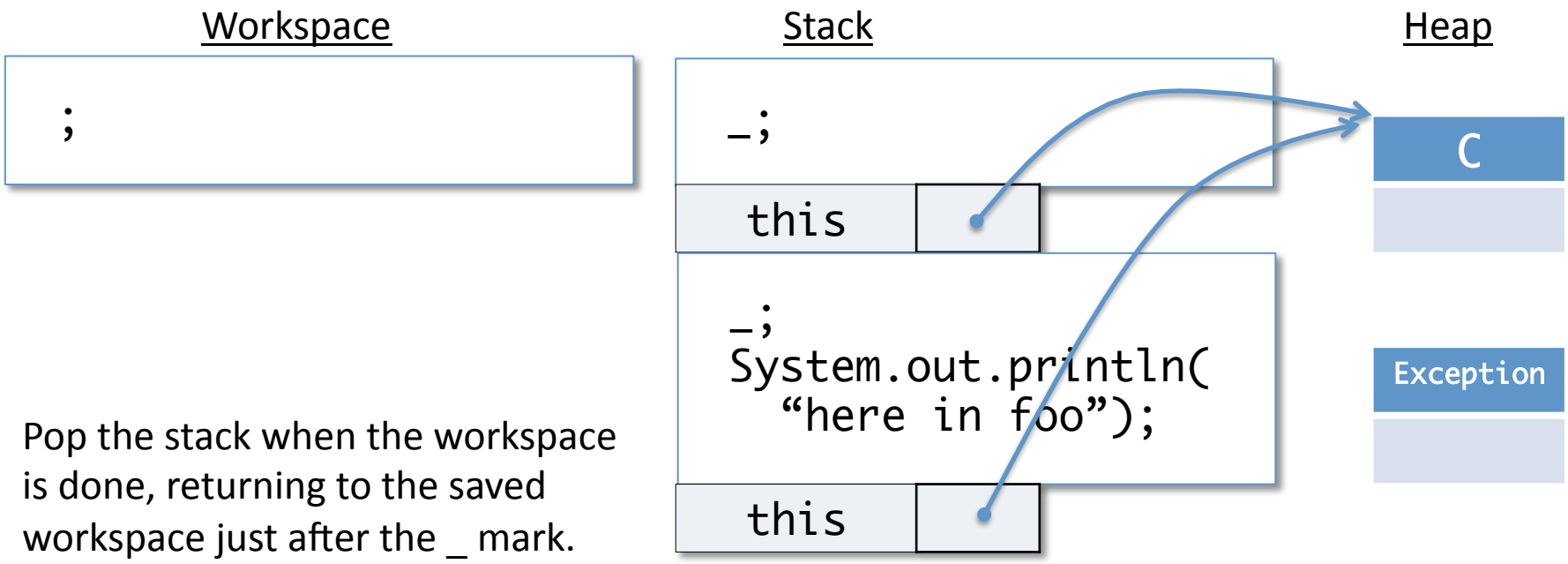
Heap

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

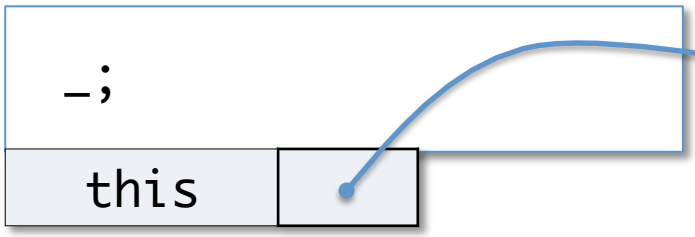
```
caught
here in bar
```

Abstract Stack Machine

Workspace

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

Stack



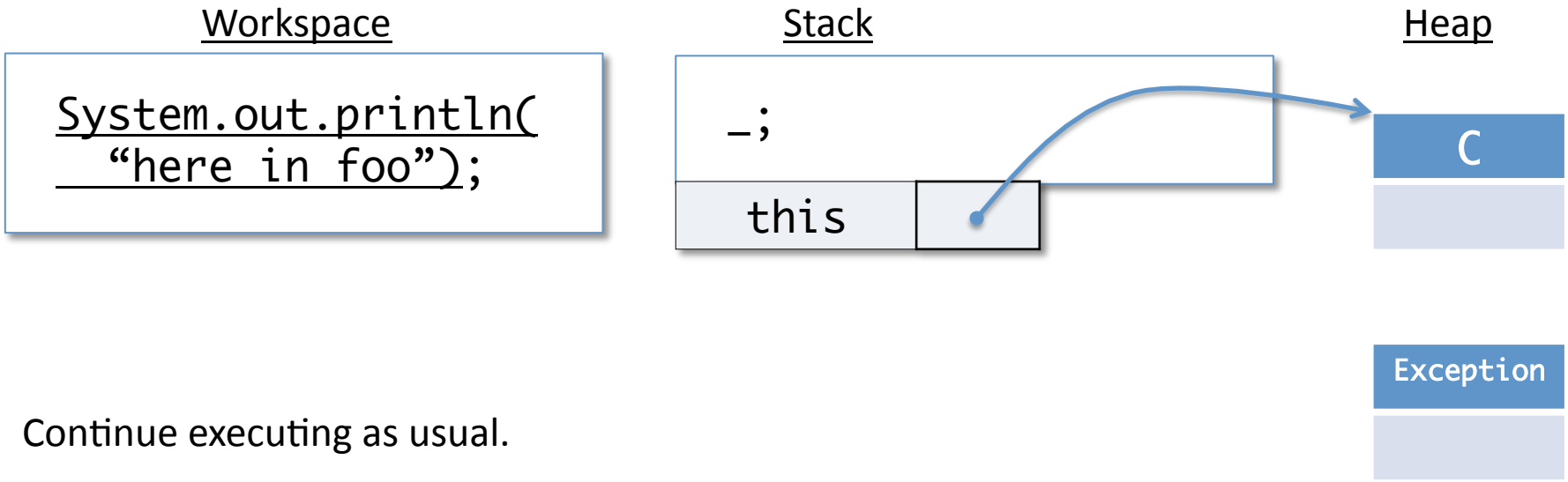
Heap

Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine

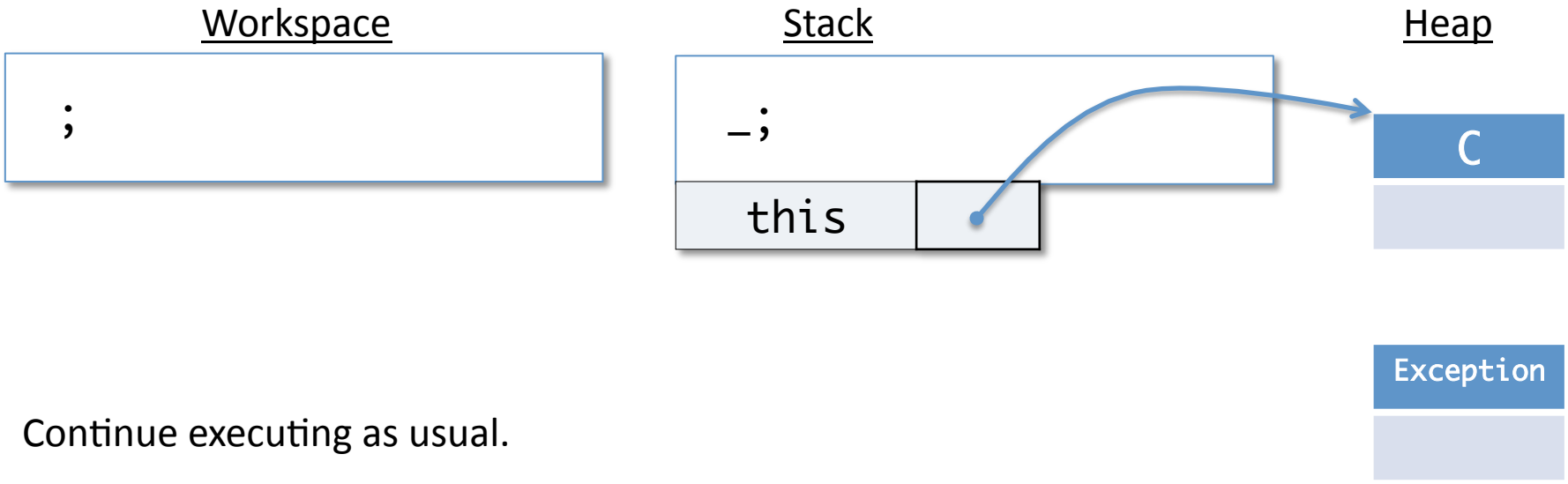


Continue executing as usual.

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

Workspace

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

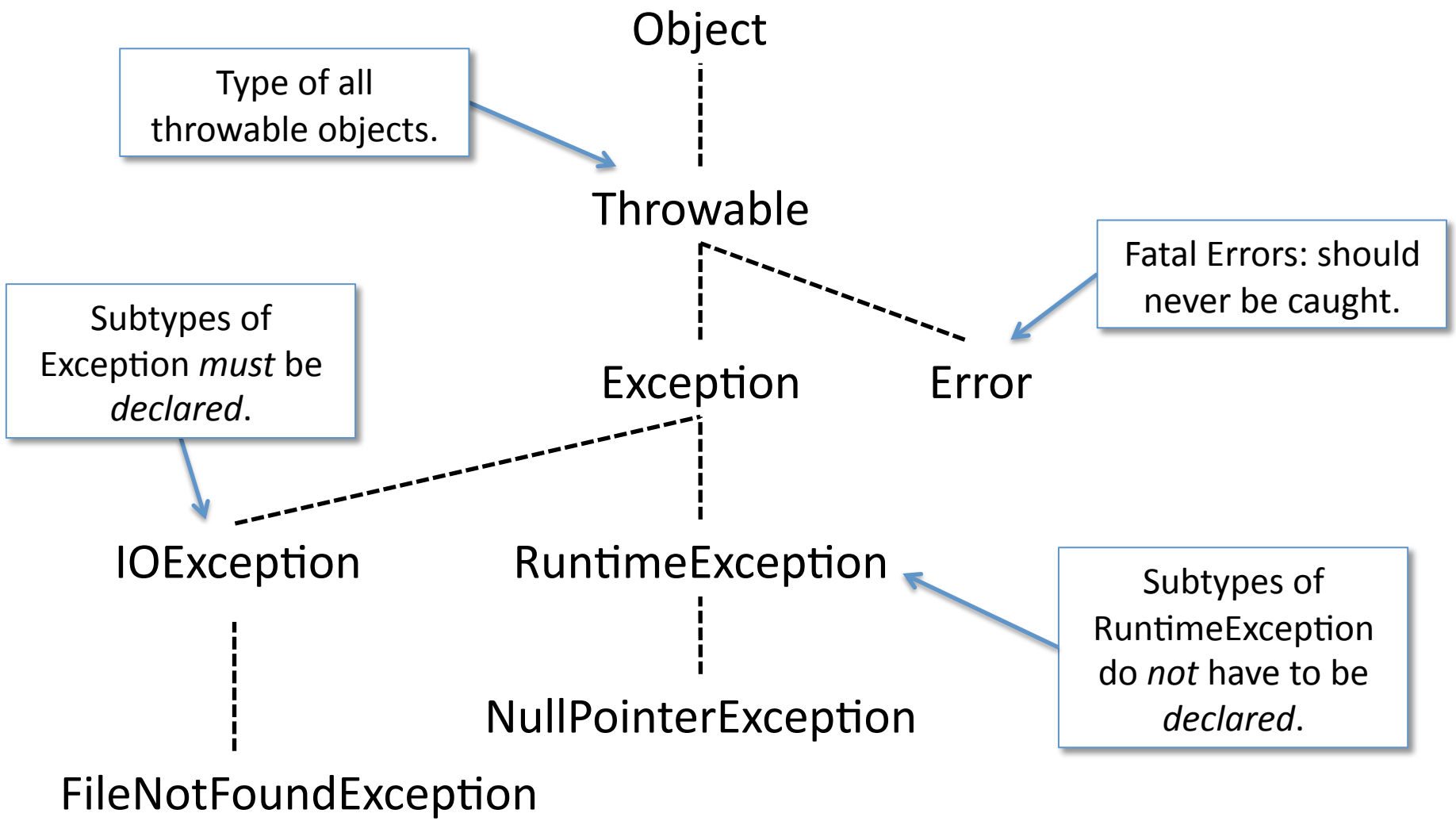
Catching Exceptions

- 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.

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

- Even if it doesn't throw the exception directly

```
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: "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

Style Points

- In Java, exceptions should be used to capture *exceptional* circumstances
  - Try/catch/throw incur performance costs and complicate reasoning about the program, so **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.
- It is often sensible to **catch one exception and re-throw** a different (more meaningful) kind of exception.
  - e.g. when implementing file base iterator (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
  - Don't put your only try {…} block in main
- Catch exceptions **with** as much **precision** as you can
  - i.e. Don't do: try {…} catch (Exception e) {…}
    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, or a caught exception — or even if the method returns from inside the try.
- “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) {…}
    }
  }
}
```

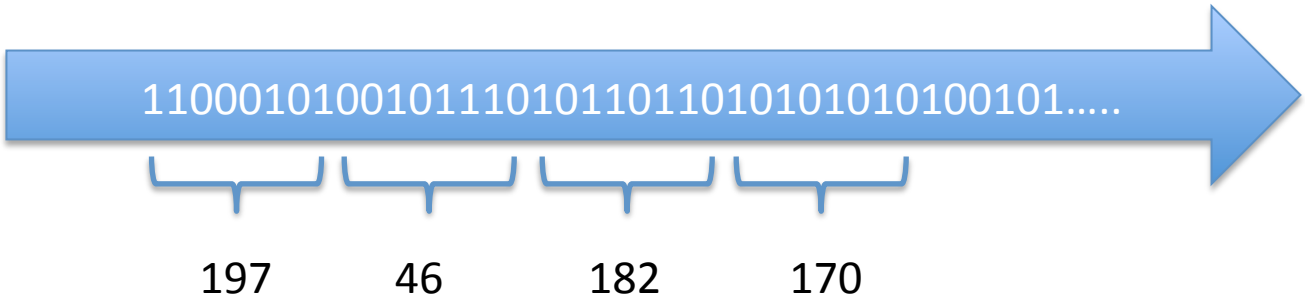
java.io

I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.

Low-level Streams

- At the lowest level, a stream is a sequence of binary numbers



- The simplest IO classes break up the sequence into 8-bit chunks, called *bytes*. Each byte corresponds to an integer in the range 0 – 255.

`InputStream` and `OutputStream`

- Abstract classes* that provide basic operations for the Stream class hierarchy:

```
abstract int read ();          // Reads the next byte of data
abstract void write (int b); // Writes the byte b to the output
```

- These operations read and write `int` values that represent *bytes*
  - range `0-255` represents a byte value
  - `-1` represents "no more data" (when returned from read)
- java.io provides many subclasses for various sources/sinks of data:
  - files, audio devices, strings, byte arrays, serialized objects
- Subclasses also provides rich functionality:
  - encoding, buffering, formatting, filtering

*Abstract classes are classes that cannot be directly instantiated (via `new`). Instead, they provide partial, concrete implementations of some operations. In this way, abstract classes are a bit like interfaces (they provide a partial specification) but also a bit like classes (they provide some implementation). They are most useful in building big libraries, which is why we aren't focusing on them in this course.

Demo

Binary input demo

Binary IO example

```
InputStream fin = new FileInputStream(filename);

int[] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

Binary IO example

```
public Image() throws IOException {
   InputStream fin = new FileInputStream(“mandrill.pgm”);

   data = new int[width][height];
   for (int i=0; i < width; i++) {
      for (int j=0; j < height; j++) {
          int ch = fin.read();
          if (ch == -1) {
              fin.close();
              throw new IOException("File ended too early");
          }
          data[j][i] = ch;
      }
   }
   fin.close();
}
```

BufferedInputStream

- Reading one byte at a time can be slow!
- Each time a stream is read there is a fixed overhead, plus time proportional to the number of bytes read.

  disk -> operating system -> JVM -> program
  disk -> operating system -> JVM -> program
  disk -> operating system -> JVM -> program

- A `BufferedInputstream` presents the same interface to clients, but internally reads many bytes at once into a *buffer* (incurring the fixed overhead only once)

  disk -> operating system ->>>> JVM -> program
  JVM -> program
  JVM -> program
  JVM -> program

Buffering Example

```
FileInputStream fin1 = new FileInputStream(filename);
InputStream fin = new BufferedInputStream(fin1);

int[] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

Buffering example

```
public Image() throws IOException {
    FileInputStream fin1 = new FileInputStream("mandrill.pgm");
    InputStream fin = new BufferedInputStream(fin1);

    data = new int[width][height];
    for (int i=0; i < width; i++) {
        for (int j=0; j < height; j++) {
            int ch = fin.read();
            if (ch == -1) {
                throw new IOException("File ended too early");
            }
            data[j][i] = ch;
        }
    }
    fin.close();
}
```

`PrintStream` Methods

`PrintStream` adds buffering and binary-conversion methods to `OutputStream`

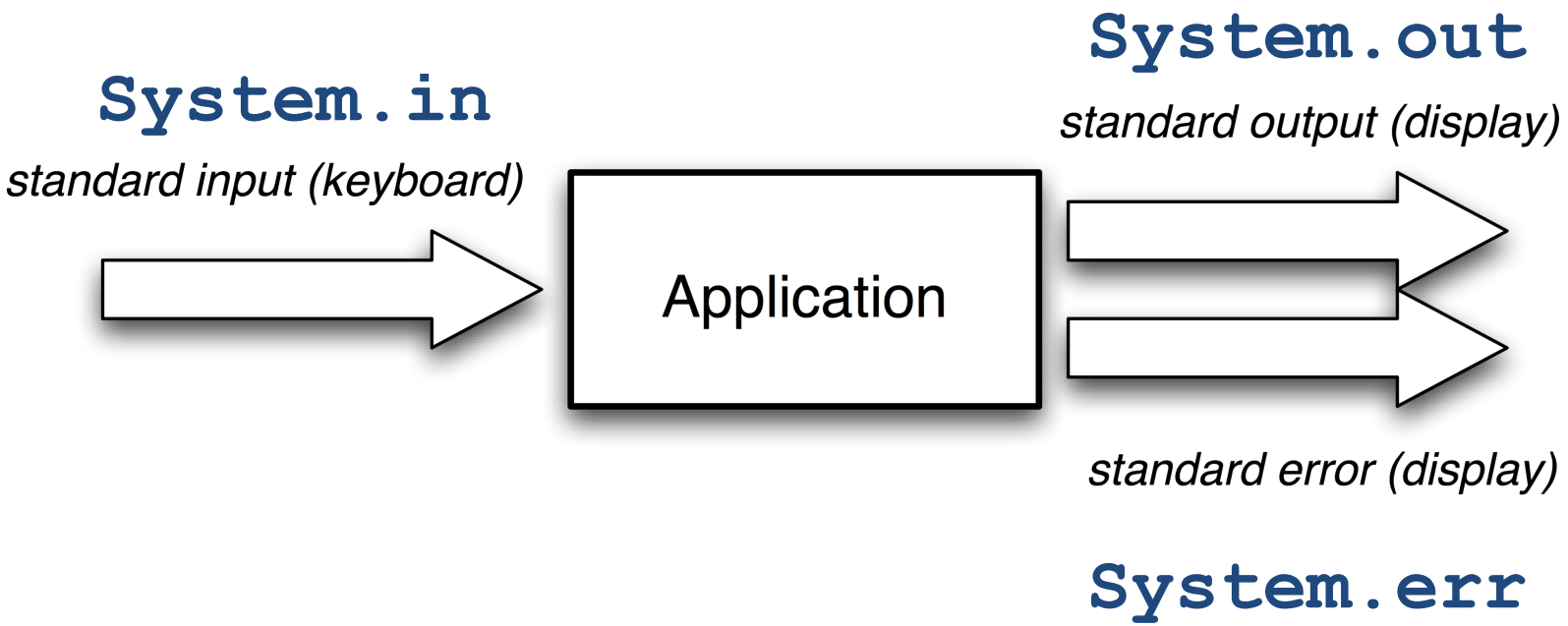
```
void println(boolean b); // write b followed by a new line
void println(String s);  // write s followed by a newline
void println();          // write a newline to the stream

void print(String s);    // write s without terminating the line
                         //  (output may not appear until the stream is flushed)
void flush();            // actually output characters waiting to be sent
```

- Note the use of *overloading*: there are *multiple* methods called `println`
  - The compiler figures out which one you mean based on the number of arguments, and/or the *static* type of the argument you pass in at the method's call site.
  - The java I/O library uses overloading of constructors pervasively to make it easy to "glue together" the right stream processing routines

The Standard Java Streams

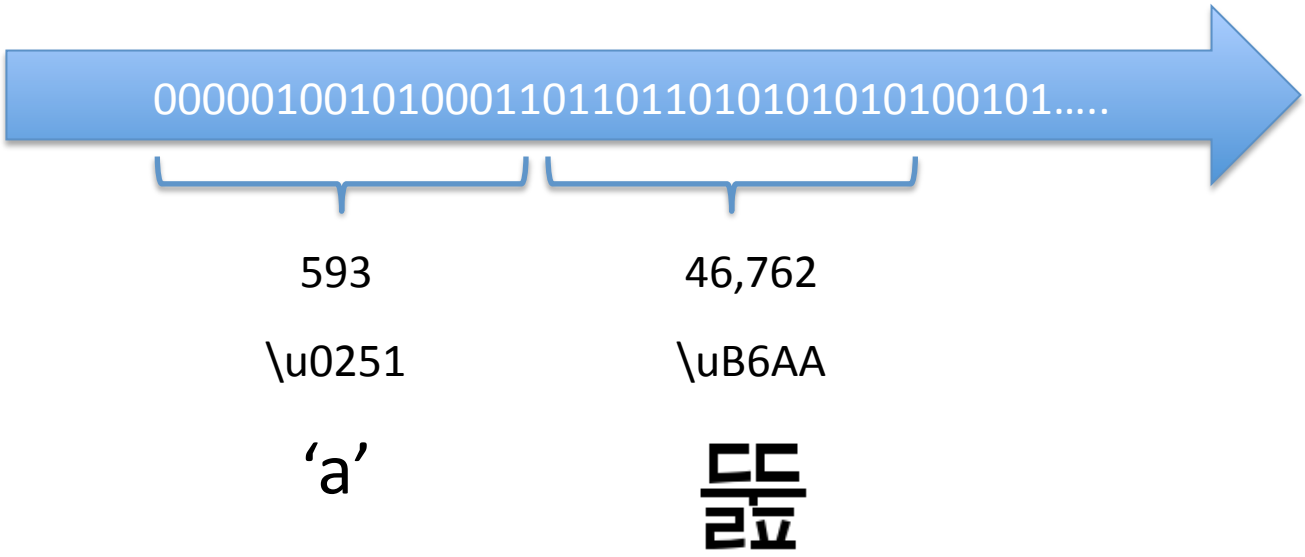
java.lang.System provides an `InputStream` and two standard `PrintStream` objects for doing console I/O.



Note that `System.in`, for example, is a *static member* of the class System – this means that the field "`in`" is associated with the *class*, not an *instance* of the class. Recall that static members in Java act like global variables. Methods can also be static – the most common being "`main`", but see also the `Math` class.

Character based IO

A character stream is a sequence of 16-bit binary numbers



The character-based IO classes break up the sequence into 16-bit chunks, of type `char`. Each character corresponds to a letter (specified by a *character encoding*).

Reader and Writer

- Similar to the `InputStream` and `OutputStream` classes, including:

```
abstract int read ();          // Reads the next character
abstract void write (int b); // Writes the char to the output
```

- These operations read and write `int` values that represent *unicode characters*
  - read returns an integer in the range 0 to 65535 (i.e. 16 bits)
  - value `-1` represents "no more data" (when returned from read)
  - requires an "encoding" (e.g. UTF-8 or UTF-16, set by a `Locale`)
- Like byte streams, the library provides many subclasses of Reader and Writer Subclasses also provides rich functionality.
  - use these for portable text I/O
- Gotcha: `System.in`, `System.out`, `System.err` are *byte* streams
  - So wrap in an InputStreamReader / PrintWriter if you need unicode console I/O

Demo

How do you read from a file into a String?

FileReadingTest.java

Java I/O Design Strategy

1. Understand the concepts and how they relate:
   - What kind of stream data are you working with?
   - Is it byte-oriented or text-oriented?
     - InputStream vs. InputReader
   - What is the source of the data?
     - e.g. file, console, network, internal buffer or array
   - Does the data have any particular format?
     - e.g. comma-separated values, line-oriented, numeric
     - Consider using Scanner or another parser

2. Design the interface:
   - Browse through java.io libraries (to remind yourself what's there!)
   - Determine how to compose the functionality your need from the library
   - Some data formats require more complex *parsing* to convert the data stream into a useable structure in memory