Programming Languages and Techniques (CIS120)

Lecture 27

November 6, 2017

Java Generics

Collections and Equality

Chapter 26

Announcements

- HW7: Chat Server
  - Available on Codio / Instructions on the web site
  - Due Tuesday, November 14th at 11:59pm

- *Midterm 2 is this Friday, in class*
  - Last names A – M Leidy Labs 10 (here)
  - Last names N – Z Meyerson B1
- *Review Session: Wednesday November 8th at 6:00pm Towne 100*
- Coverage:
  - Mutable state (in OCaml and Java)
  - Objects (in OCaml and Java)
  - ASM (in OCaml and Java)
  - Reactive programming (in Ocaml)
  - Arrays (in Java)
  - Subtyping, Simple Extension, Dynamic Dispatch (in Java)
- Sample exams from recent years posted on course web page
- Makeup exam request form: on the course web pages

Java Generics

Subtype Polymorphism

vs.

Parametric Polymorphism

Review: Subtype Polymorphism*

- Main idea:

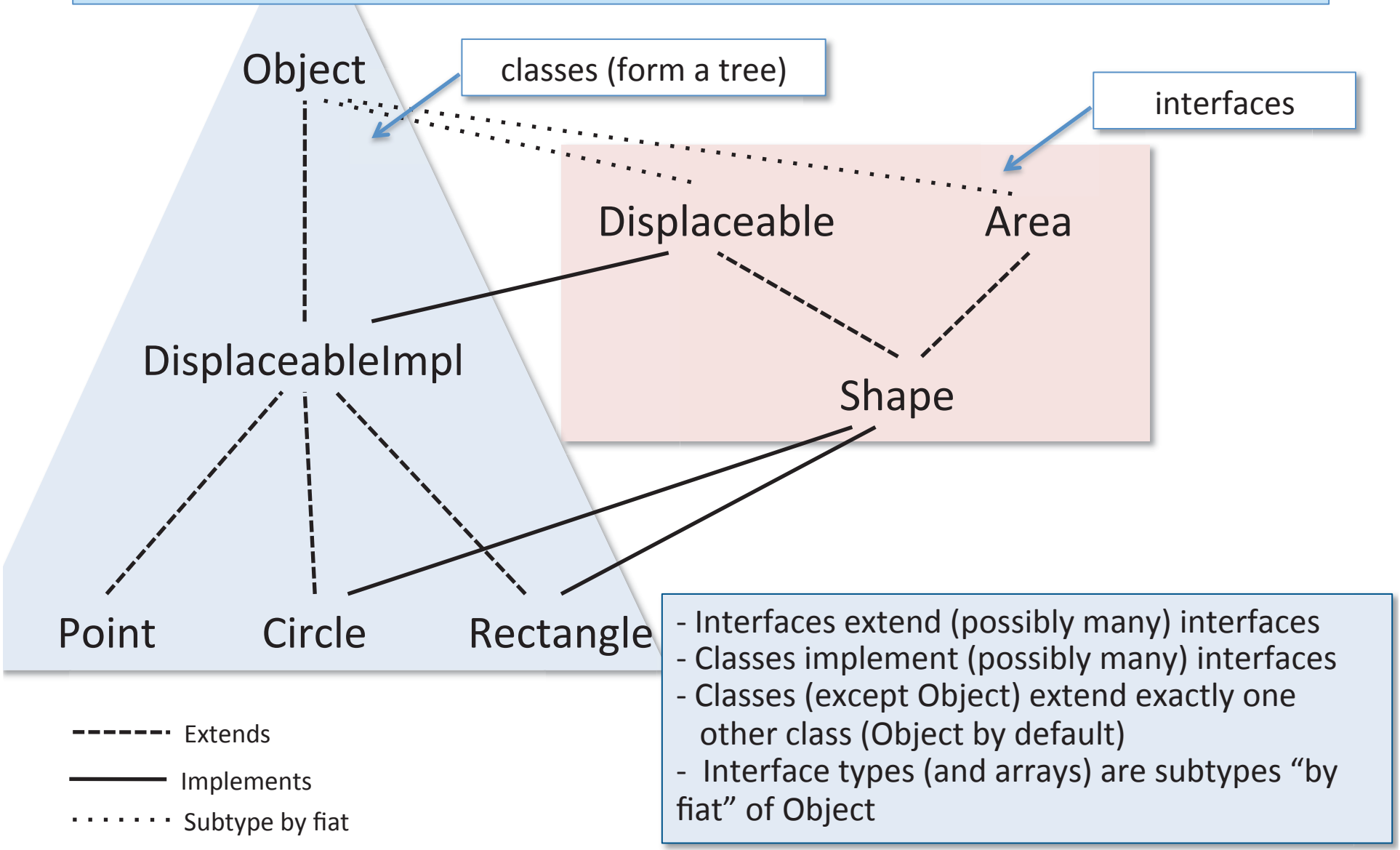
> Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

```
// in class C
public static void times2(Counter c) {
    c.incBy(c.get());
 }
// somewhere else, Decr extends Counter
C.times2(new Decr(3));
```

- If B is a subtype of A, it provides all of A's (public) methods.

*polymorphism = many shapes

Recap: Subtyping



- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object by default)
- Interface types (and arrays) are subtypes "by fiat" of Object

Is subtyping enough?

Mutable Queue ML Interface

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Add a value to the end of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the front value and return it (if any) *)
  val deq : 'a queue -> ‘a

  (* Determine if the queue is empty *)
  val is_empty : ‘a queue -> bool
end
```

How can we translate this interface to Java?

Java Interface

```
module type QUEUE =
sig

  type 'a queue

  val create : unit -> 'a queue

  val enq : 'a -> 'a queue -> unit

  val deq : 'a queue -> 'a

  val is_empty : 'a queue -> bool

end
```

```
interface ObjQueue {



   // no constructors
   // in an interface

     public void enq(Object elt);

     public Object deq();

     public boolean isEmpty();

}
```

Subtype Polymorphism

```
interface ObjQueue {
    public void enq(Object elt);
    public Object deq();
    public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
__A__  x = q.deq();
```

What type for A?

1. String
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object

Subtype Polymorphism

```
interface ObjQueue {
    public void enq(Object elt);
    public Object deq();
    public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
System.out.println(x.trim());
```

← Does this line type check

1. Yes
2. No
3. It depends

ANSWER: No

Subtype Polymorphism

```
interface ObjQueue {
    public void enq(Object elt);
    public Object deq();
    public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
//System.out.println(x.trim());
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();
```

What type for B?

1. Point
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object

Parametric Polymorphism (a.k.a. Generics)

- Big idea:

  Parameterize a type (i.e. interface or class) by another type.

```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();
}
```

- The implementation of a parametric polymorphic interface cannot depend on the implementation details of the parameter.
  - the implementation of enq cannot invoke any methods on 'o' except those inherited from Object
  - i.e. the only thing we know about E is that it is a subtype of Object

Generics (Parametric Polymorphism)

```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();
  …
}
```

```
Queue<String> q = …;

q.enq(" CIS 120 ");
String x = q.deq();                  // What type of x?   String
System.out.println(x.trim());        // Is this valid?    Yes!
q.enq(new Point(0.0,0.0));           // Is this valid?    No!
```

Subtyping and Generics

Subtyping and Generics*

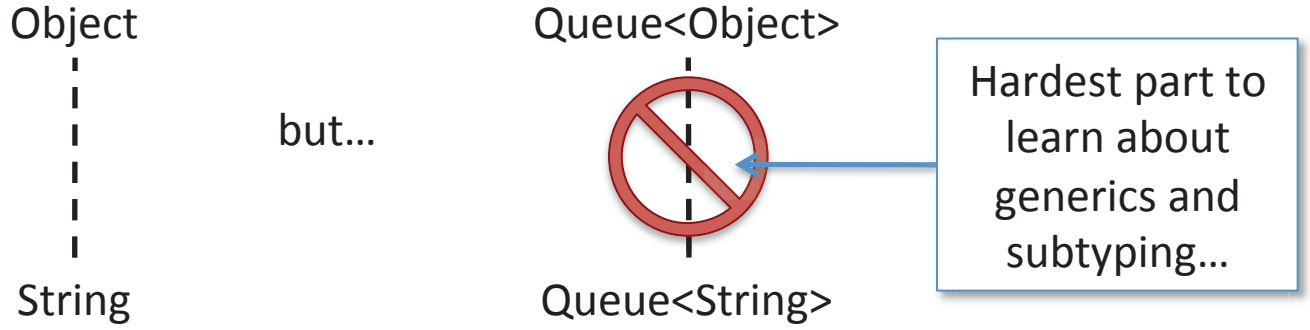
```
Queue<String> qs = new QueueImpl<String>();
Queue<Object> qo = qs;

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:



* Subtyping and generics interact in other ways too. Java supports *bounded polymorphism* and *wildcard types*, but those are beyond the scope of CIS 120.

Subtyping and Generics

Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither

Answer: 1

One Subtlety

- Unlike OCaml, Java classes and methods can be generic only with respect to *reference* types.
  - Not possible to do: `Queue<int>`
  - Must instead do: `Queue<Integer>`

- Java Arrays cannot be generic:
  - Not possible to do:

```
class C<E> {
    E[] genericArray;
    public C() {
        genericArray = new E[];
    }
}
```

The Java Collections Library

A case study in subtyping and generics

(Also very useful!)

Java Packages

- Java code can be organized into *packages* that provide namespace management.
  - Somewhat like OCaml's modules
  - Packages contain groups of related classes and interfaces.
  - Packages are organized hierarchically in a way that mimics the file system's directory structure.
- A .java file can *import* (parts of) packages that it needs access to:

```
import org.junit.Test;     // just the JUnit Test class
import java.util.*;        // everything in java.util
```

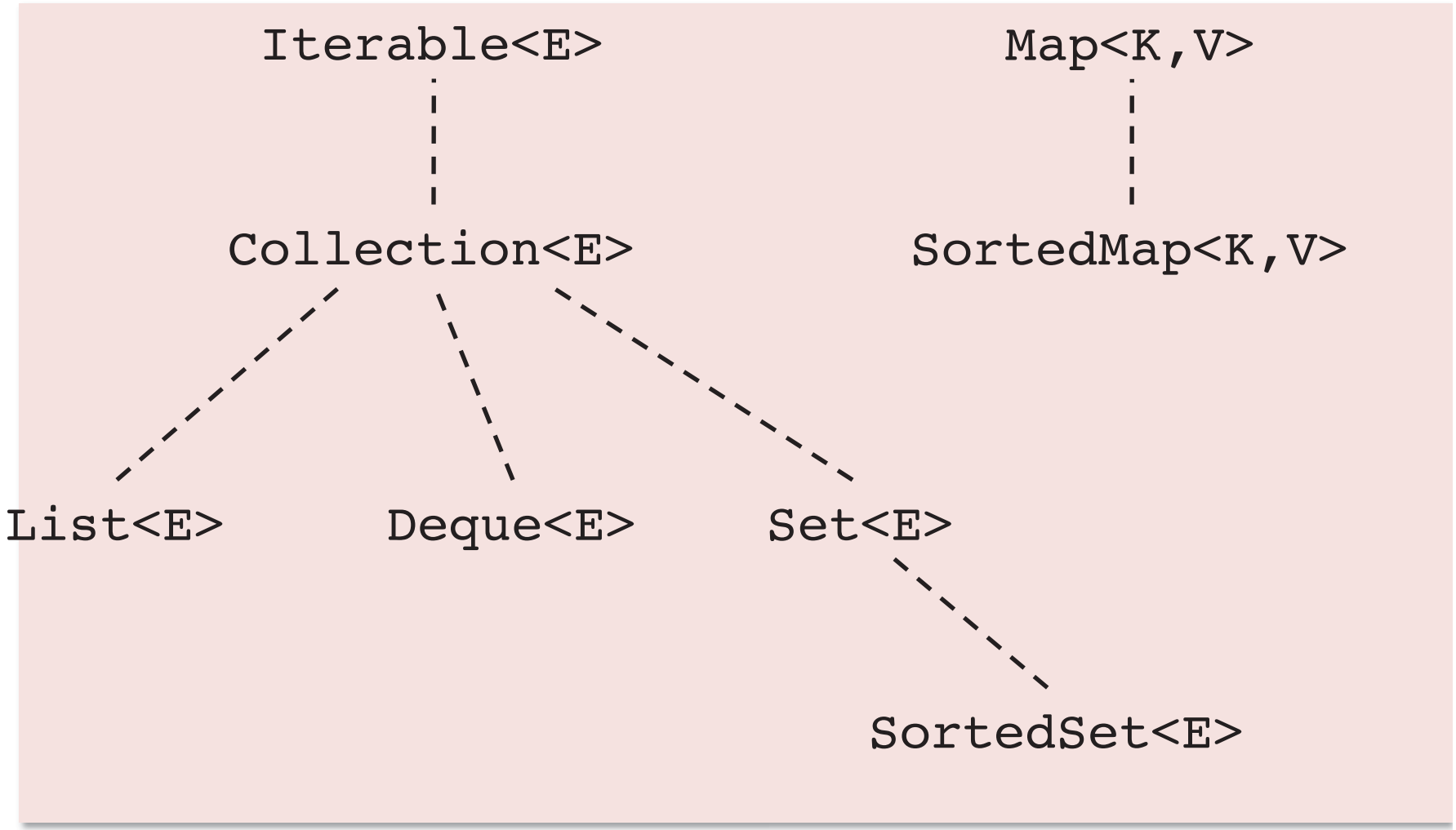
- Important packages:
  - java.lang, java.io, java.util, java.math, org.junit
- See documentation at:
  http://docs.oracle.com/javase/8/docs/api/

Reading Java Docs

java.util

https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html

Interfaces* of the Collections Library



*not all of them!

Collection<E> Interface (Excerpt)

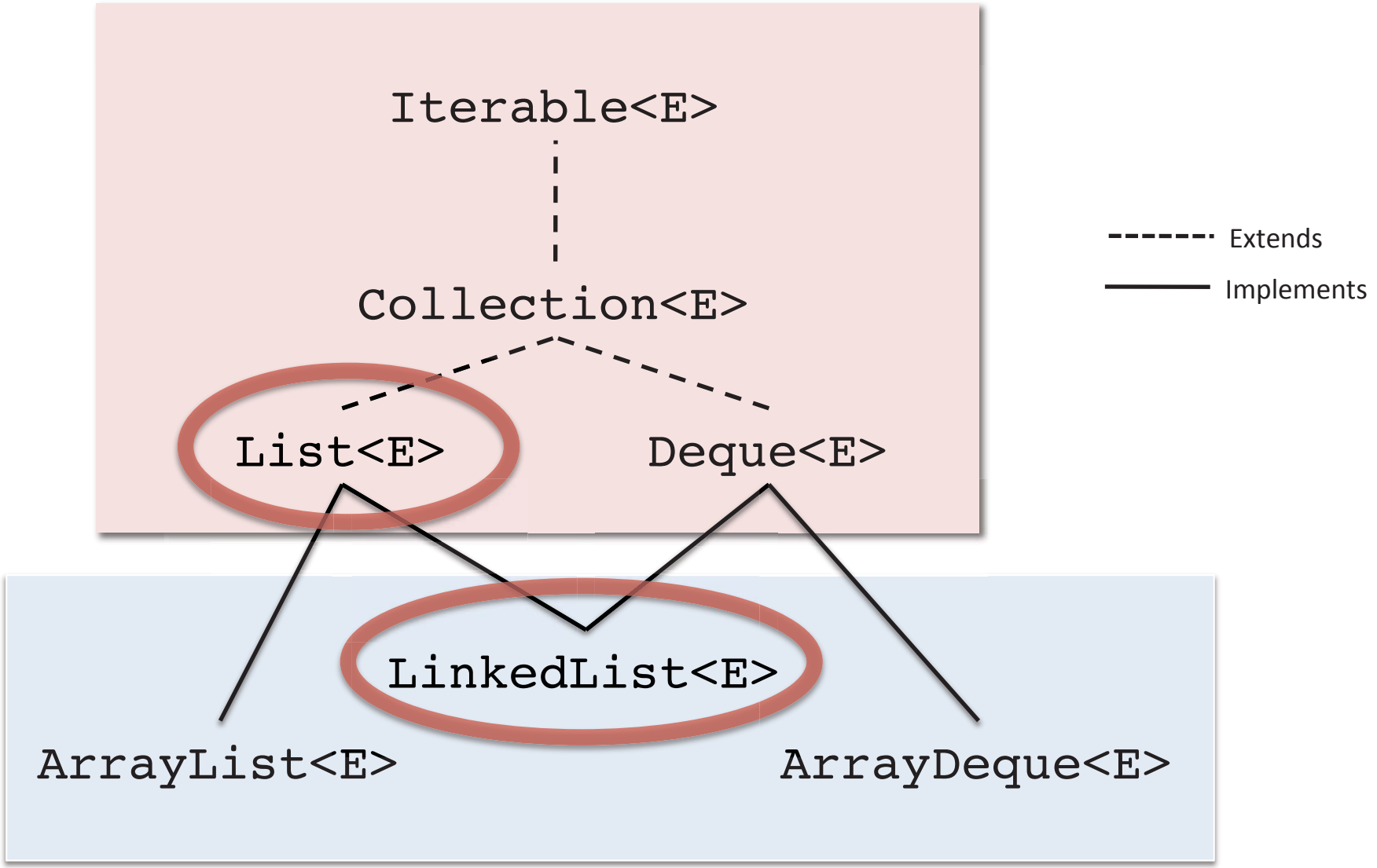
```
interface Collection<E> extends Iterable<E> {
  // basic operations
  int size();
  boolean isEmpty();
  boolean add(E o);
  boolean remove(Object o);     // why not E?*
  boolean contains(Object o);

  // bulk operations
  …
}
```

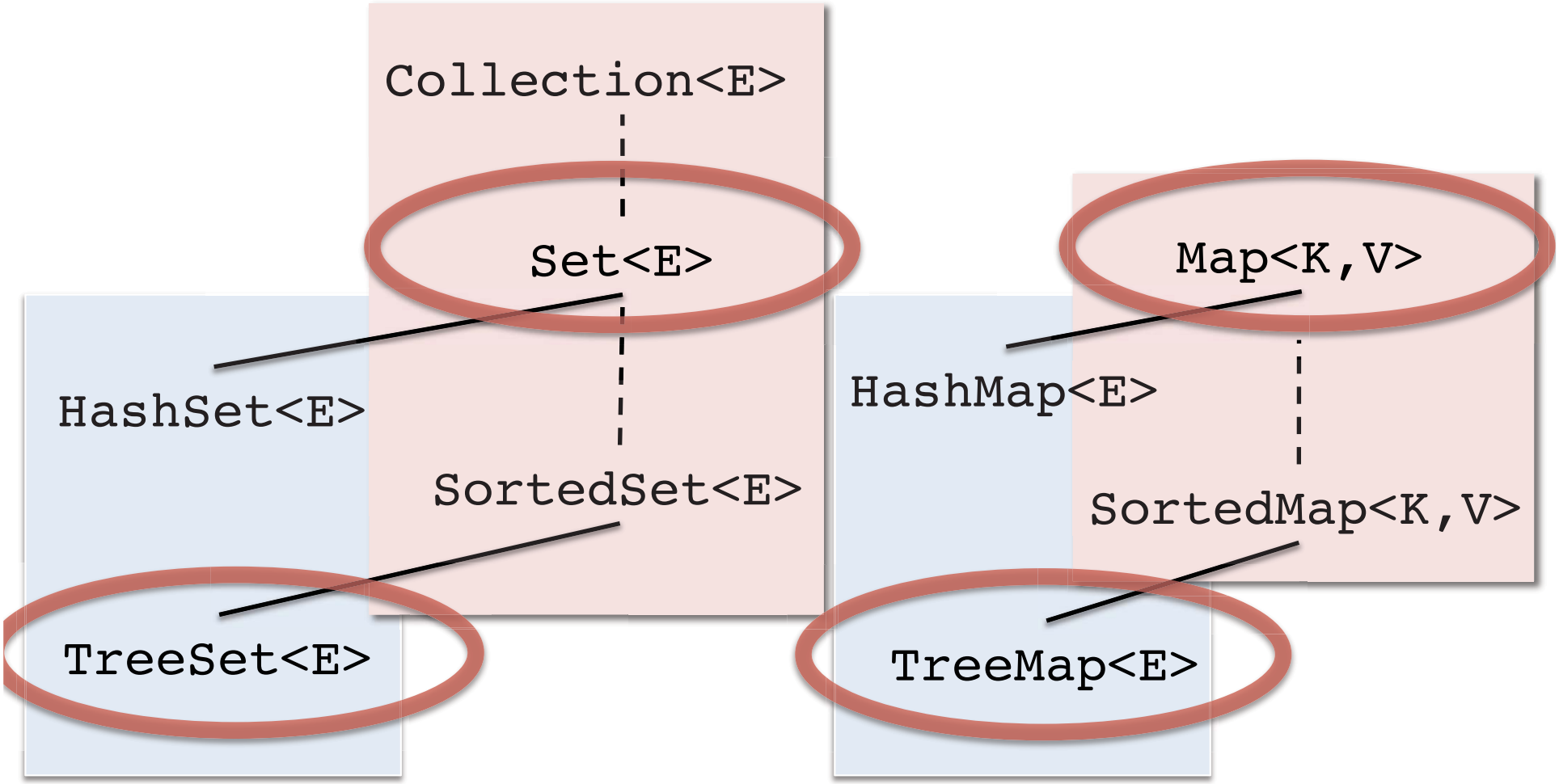
- We've already seen this interface in the OCaml part of the course.
- Most collections are designed to be *mutable* (like queues)

* Why not E? Internally, collections use the `equals` method to check for equality – membership is determined by o.equals, which does not have to be false for objects of different types. Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.

Sequences


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

Sets and Maps*



*Read javadocs before instantiating these classes! There are some important details to be aware of to use them correctly.

Buggy Use of TreeSet implementation

```
import java.util.*;

class Point {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX(){ return x; }
    public int getY(){ return y; }
}


public class TreeSetDemo {
    public static void main(String[] args) {
        Set<Point> s = new TreeSet<Point>();
        s.add(new Point(1,1));
        s.add(new Point(2,3));
        s.add(new Point(4,4));

    }
}
```

A Crucial Detail of TreeSet

Constructor Detail

TreeSet

public TreeSet()

Constructs a new, empty tree set, sorted according to the natural ordering of its elements. All elements inserted into the set must implement the `Comparable` interface. Furthermore, all such elements must be mutually comparable: `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the set. …

The Interface `Comparable`

public interface Comparable<T>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's `compareTo` method is referred to as its natural comparison method. …

Methods of Comparable

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order.

Method Detail

compareTo

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 &&

Adding Comparable to Point

```
import java.util.*;

class Point implements Comparable<Point> {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX(){ return x; }
    public int getY(){ return y; }
    @Override
    public int compareTo(Point o) {
        if (this.x < o.x) {
            return -1;
        } else if (this.x > o.x) {
            return 1;
        } else if (this.y < o.y) {
            return -1;
        } else if (this.y > o.y) {
            return 1;
        }
        return 0;
    }
}
```

Programming Languages and Techniques (CIS120)

Lecture 28

November 8, 2017

Overriding Methods, Equality, Enums

Chapter 26

Announcements

- HW7: Chat Server
  - Available on Codio / Instructions on the web site
  - Due Tuesday, November 14th at 11:59pm

- *Midterm 2 is this Friday, in class*
  - Last names A – M Leidy Labs 10 (here)
  - Last names N – Z Meyerson B1
- *Review Session: Wednesday November 8th at 6:00pm Towne 100*
- Coverage:
  - Mutable state (in OCaml and Java)
  - Objects (in OCaml and Java)
  - ASM (in OCaml and Java)
  - Reactive programming (in Ocaml)
  - Arrays (in Java)
  - Subtyping, Simple Extension, Dynamic Dispatch (in Java)
- Sample exams from recent years posted on course web page
- Makeup exam request form: on the course web pages

Method Overriding

A Subclass can *Override* its Parent

```
class C {
      public void printName() { System.out.println("I'm a C"); }
}

class D extends C {
      public void printName() { System.out.println("I'm a D"); }
}

// somewhere in main
C c = new D();
c.printName();
```

What gets printed to the console?

1. I'm a C
2. I'm a D
3. NullPointerException
4. NoSuchMethodException

A Subclass can *Override* its Parent

```
class C {
      public void printName() { System.out.println("I'm a C"); }
}

class D extends C {
      public void printName() { System.out.println("I'm a D"); }
}

// somewhere in main
C c = new D();
c.printName();
```

- Our ASM model for dynamic dispatch already explains what will happen when we run this code.
- Useful for changing the default behavior of classes.
- But… can be confusing and difficult to reason about if not used carefully.

Overriding Example

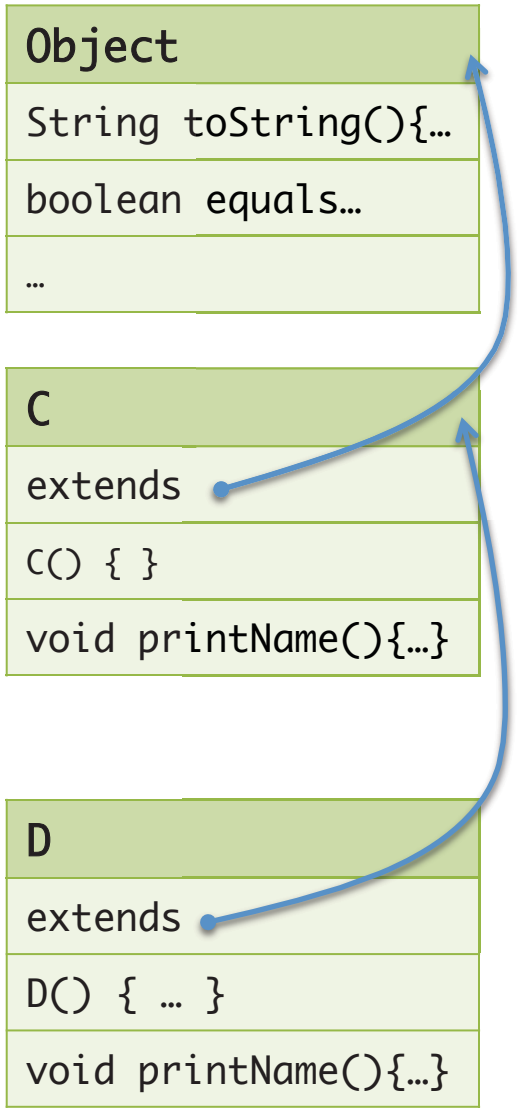
Workspace

```
C c = new D();
c.printName();›
```

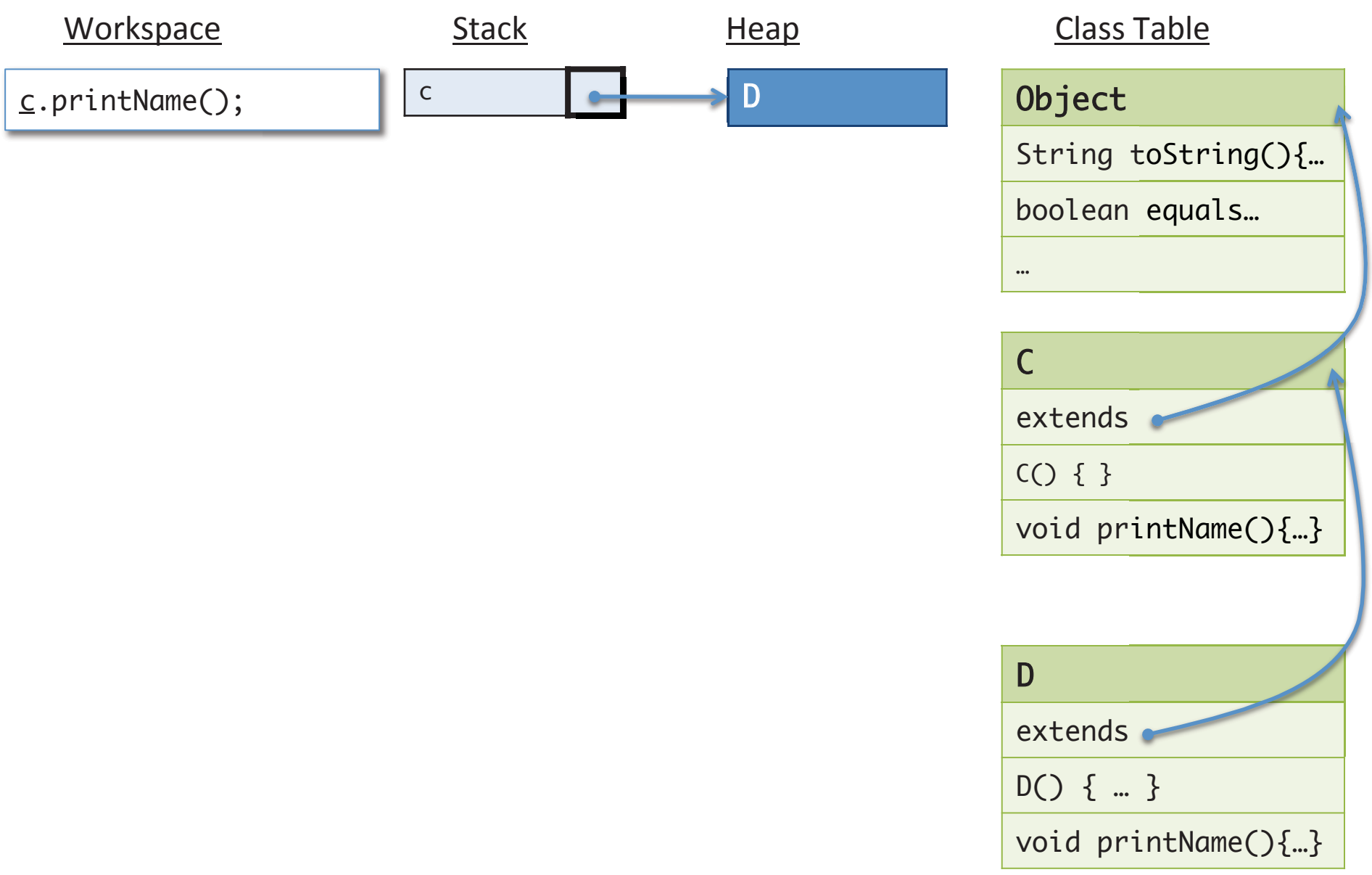
Stack

Heap

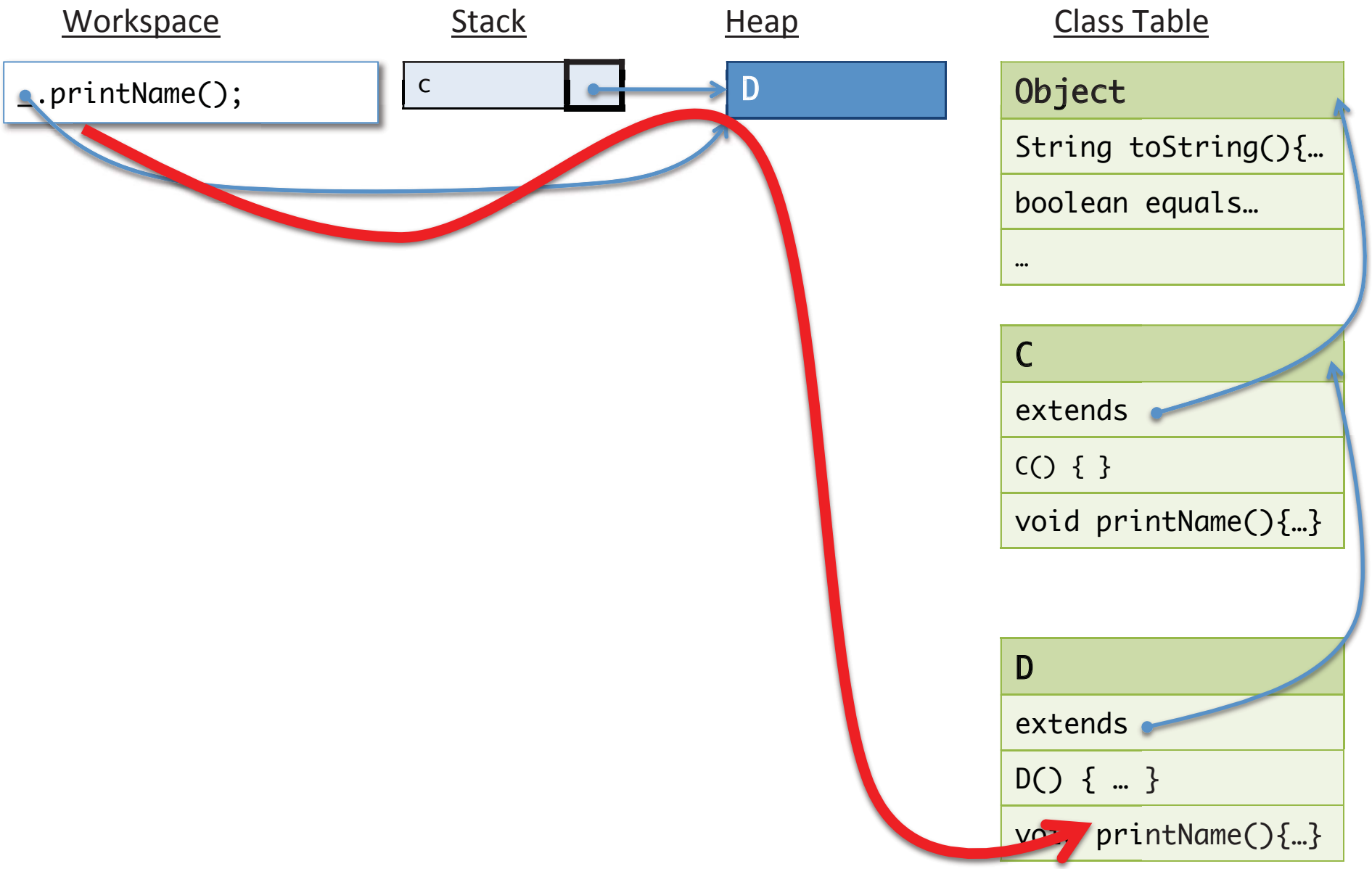
Class Table



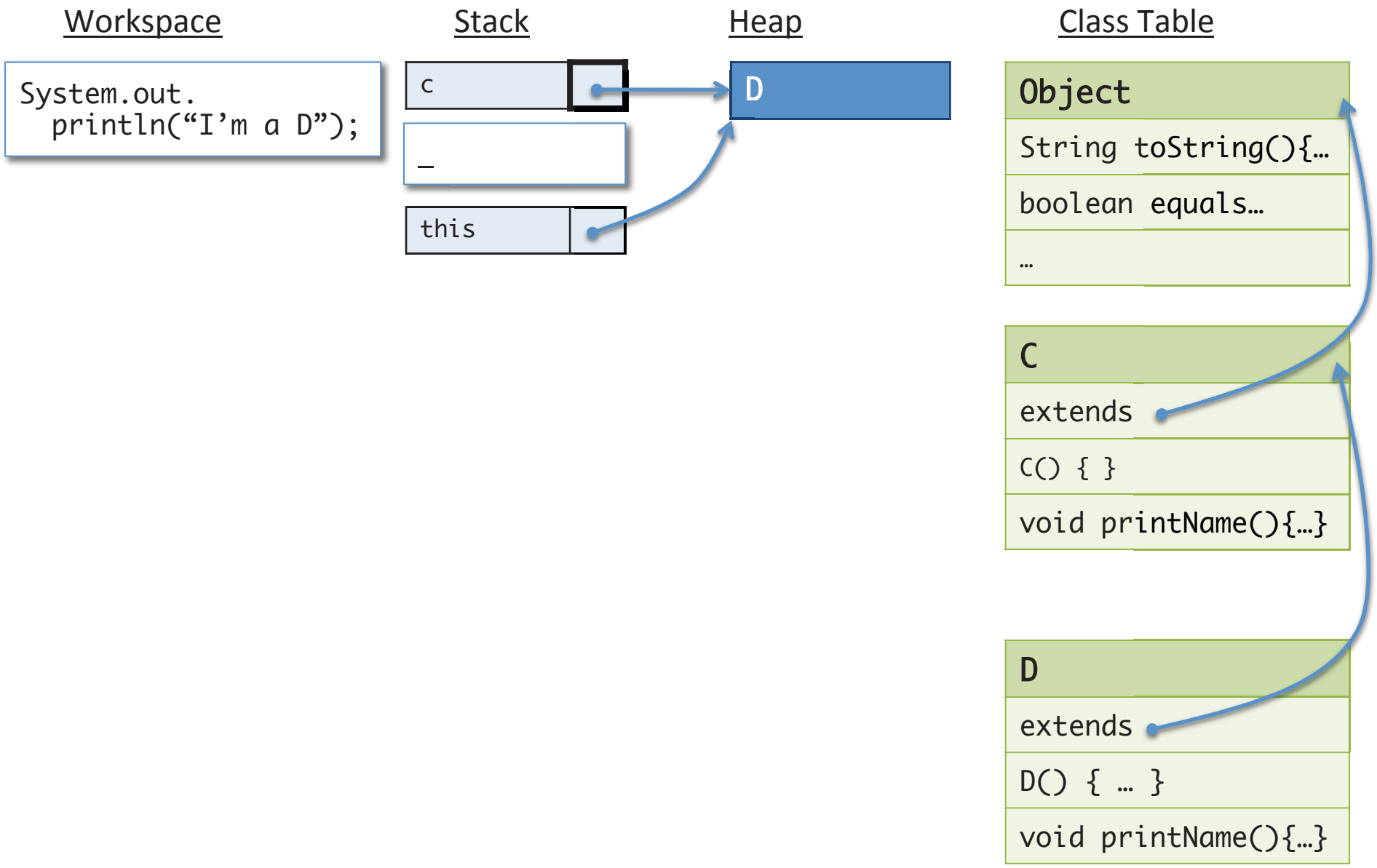
Overriding Example



Overriding Example



Overriding Example



Difficulty with Overriding

```
class C {

  public void printName() {
    System.out.println("I'm a " + getName());
  }

  public String getName() {
    return "C";
  }
}

class E extends C {

  public String getName() {
    return "E";
  }
}

// in main
C c = new E();
c.printName();
```

What gets printed to the console?

1. I'm a C
2. I'm a E
3. NullPointerException

Difficulty with Overriding

```
class C {

  public void printName() {
    System.out.println("I'm a " + getName());
  }

  public String getName() {
    return "C";
  }
}

class E extends C {

  public String getName() {
    return "E";
  }
}

// in main
C c = new E();
c.printName();
```

The C class might be in another package, or a library...

Whoever wrote E might not be aware of the implications of changing `getName`.

Overriding the method causes the behavior of `printName` to change!

- Overriding can break invariants/ abstractions relied upon by the superclass.

Case study: Equality

Consider this example

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y; }

    public int getX() { return x; }
    public int getY() { return y; }
}

// somewhere in main…
List<Point> l = new LinkedList<Point>();
l.add(new Point(1,2));
System.out.println(l.contains(new Point(1,2)));
```

What gets printed to the console?

1. true
2. false

Why?

Answer: False

From Java API:

public interface **Collection<E>** extends Iterable<E>

…

Many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the contains(Object o) method says: "returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e)). …

When to override equals

- In classes that represent immutable *values*
  - String already overrides equals
  - Our Point class is a good candidate

- When there is a "logical" notion of equality
  - The collections library overrides equality for Sets (e.g. two sets are equal if and only if they contain equal elements)

- Whenever instances of a class might need to serve as *elements of a set* or as *keys in a map*
  - The collections library uses `equals` internally to define set membership and key lookup
  - (This is the problem with the example code)

When *not* to override equals

- When each instance of a class is inherently unique
  - *Often* the case for mutable objects (since its state might change, the only sensible notion of equality is identity)
  - Classes that represent "active" entities rather than data (e.g. threads, gui components, etc.)

- When a superclass already overrides equals and provides the correct functionality.
  - Usually the case when a subclass is implemented by adding only new methods, but not fields

How to override equals

The contract for equals

- The equals method implements an *equivalence relation* on non-null objects.
- It is *reflexive*:
  - for any non-null reference value x, x.equals(x) should return true
- It is *symmetric*:
  - for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*:
  - for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent:
  - for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified
- For any non-null reference x, x.equals(null) should return false.

Directly from: http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)

First attempt

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {this.x = x; this.y = y;}
  public int getX() { return x; }
  public int getY() { return y; }
  public boolean equals(Point that) {
    return (this.getX() == that.getX() &&
            this.getY() == that.getY());
  }
}
```

Gocha: *overloading*, vs. *overriding*

```
public class Point {
  …
  // overloaded, not overridden
  public boolean equals(Point that) {
    return (this.getX() == that.getX() &&
            this.getY() == that.getY());
  }
}
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
Object o = p2;
System.out.println(p1.equals(o));
// prints false!
System.out.println(p1.equals(p2));
// prints true!
```

The type of equals as declared in Object is:

```
public boolean equals(Object o)
```

The implementation above takes a Point *not* an Object!

instanceof

- The `instanceof` operator tests the *dynamic* type of any object

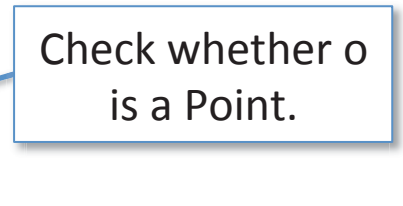
```
Point p = new Point(1,2);
Object o1 = p;
Object o2 = "hello";
System.out.println(p instanceof Point);
       // prints true
System.out.println(o1 instanceof Point);
       // prints true
System.out.println(o2 instanceof Point);
       // prints false
System.out.println(p instanceof Object);
       // prints true
```

- But… use `instanceof` judiciously – usually dynamic dispatch is better.

Type Casts

- We can test whether o is a Point using instanceof

```
@Override
public boolean equals(Object o) {
  boolean result = false;
     if (o instanceof Point) {
        // o is a point - how do we treat it as such?
     }
   return result;
}
```



- Use a type *cast*: `(Point) o`
  - At compile time: the expression `(Point) o` has type `Point`.
  - At runtime: check whether the dynamic type of o is a subtype of `Point`, if so evaluate to o, otherwise raise a `ClassCastException`
  - As with instanceof, use casts judiciously – i.e. almost never. Instead use generics

Refining the `equals` implementation

```
@Override
public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof Point) {
        Point that = (Point) o;
        result = (this.getX() == that.getX() &&
                  this.getY() == that.getY());
    }
    return result;
}
```

This cast is guaranteed to succeed.

What about subtypes?

Suppose we extend Point like this

```
public class ColoredPoint extends Point {
  private final int color;
  public ColoredPoint(int x, int y, int color) {
    super(x,y);
    this.color = color;
  }

  @Override
  public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof ColoredPoint) {
      ColoredPoint that = (ColoredPoint) o;
      result = (this.color == that.color &&
                super.equals(that));
    }
    return result;
  }
}
```

This version of equals is suitably modified to check the color field too.

Keyword `super` is used to invoke overridden methods.

Broken Symmetry

```
Point p = new Point(1,2);
ColoredPoint cp = new ColoredPoint(1,2,17);
System.out.println(p.equals(cp));
    // prints true
System.out.println(cp.equals(p));
    // prints false
```

What gets printed? (1=true, 2=false)

- The problem arises because we mixed Points and ColoredPoints, but ColoredPoints have more data that allows for finer distinctions.
- Should a Point *ever* be equal to a ColoredPoint?

Suppose Points *can* equal ColoredPoints

```
public class ColoredPoint extends Point {
  …
  public boolean equals(Object o) {
       boolean result = false;
       if (o instanceof ColoredPoint) {
          ColoredPoint that = (ColoredPoint) o;
          result = (this.color == that.color &&
                    super.equals(that));
       } else if (o instanceof Point) {
          result = super.equals(o);
       }
       return result;
  }
}
```

I.e., we repair the symmetry violation by checking for Point explicitly

Does this really work? (1=yes, 2=no)

Broken Transitivity

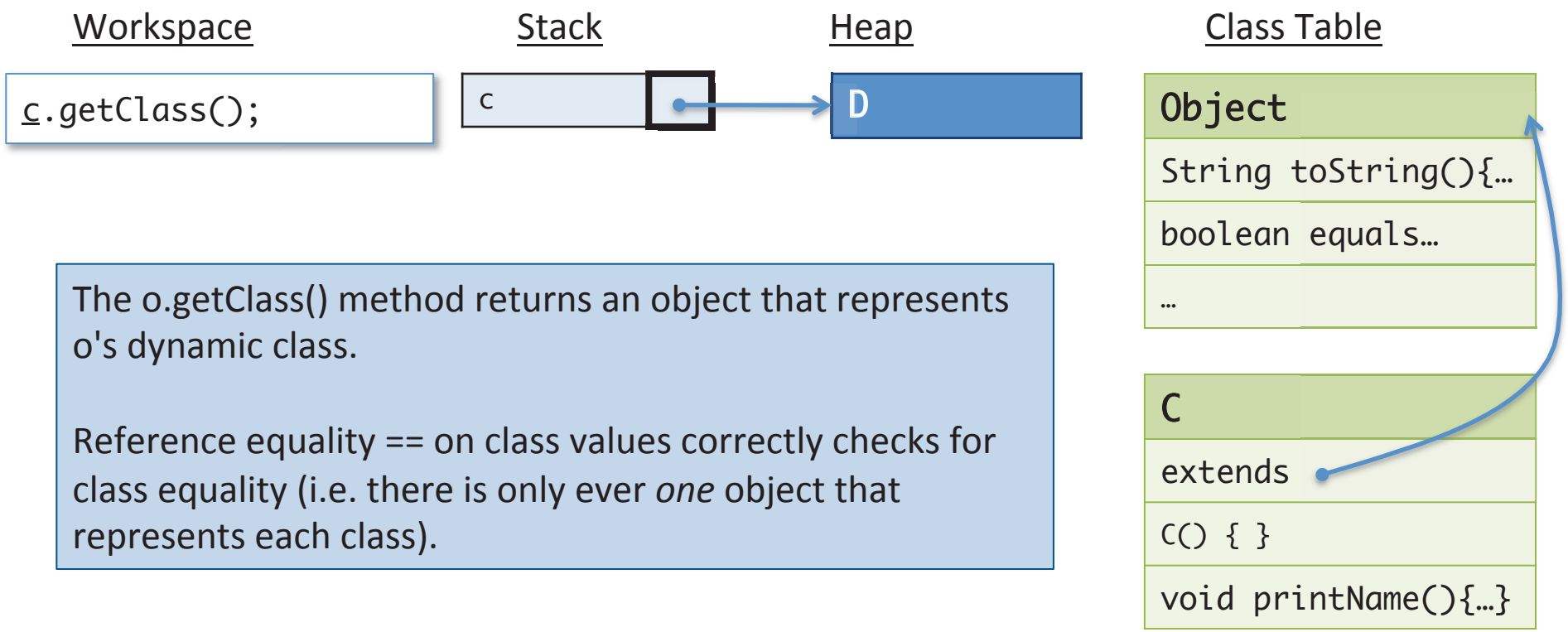
```
Point p = new Point(1,2);
ColoredPoint cp1 = new ColoredPoint(1,2,17);
ColoredPoint cp2 = new ColoredPoint(1,2,42);
System.out.println(p.equals(cp1));
    // prints true
System.out.println(cp1.equals(p));
    // prints true(!)
System.out.println(p.equals(cp2));
    // prints true
System.out.println(cp1.equals(cp2));
    // prints false(!!)
```

- We fixed symmetry, but broke transitivity!
- Should a Point *ever* be equal to a ColoredPoint?

No!

Should equality use `instanceof`?

- To correctly account for subtyping, we need the classes of the two objects to match *exactly*.
- instanceof only lets us ask about the subtype relation
- How do we access the dynamic class?



The o.getClass() method returns an object that represents o's dynamic class.

Reference equality == on class values correctly checks for class equality (i.e. there is only ever *one* object that represents each class).

Overriding equals, take two

Properly overridden equals

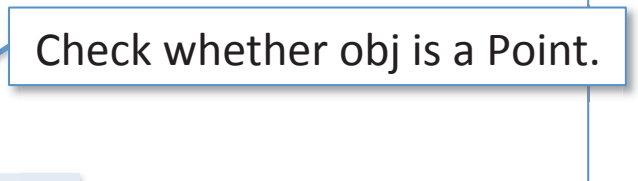
```
public class Point {
    …
    @Override
    public boolean equals(Object o) {
        // what do we do here???
    }
}
```

- Use the `@Override` annotation when you *intend* to override a method so that the compiler can warn you about accidental overloading.

- Now what? How do we know whether the o is even a Point?
  - We need a way to check the *dynamic* type of an object.

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;
}
```



Equality and Hashing

- Whenever you override equals you ***must*** *also* override `hashCode` in a compatible way
  - If `o1.equals(o2)` then `o1.hashCode() == o2.hashCode()`
  - hashCode is used by the HashSet and HashMap collections

- Forgetting to do this can lead to extremely puzzling bugs!

Overriding Equality in Practice

- Some tools (e.g. Eclipse) can autogenerate equality methods of the kind we developed.
  - But you need to specify which fields should be taken into account.
  - and you should know why some comparisons use == and some use .equals

Enumerations

Enumerations (a.k.a. Enum Types)

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

```
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

```
// Use of 'enum' in CommandParser.java (PennPals 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.*, plus Enum types and 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?

```
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

Answer: 5 something else!

break

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

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

- Use an explicit **break** to avoid fallthrough:

```
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");
}
```

Enumerations

- Enum types are just a convenient way of defining a class along with some standard methods.
  - 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();

Programming Languages and Techniques (CIS120)

Lecture 29

November 13, 2017

Enums, Iterators, Exceptions

Chapter 27

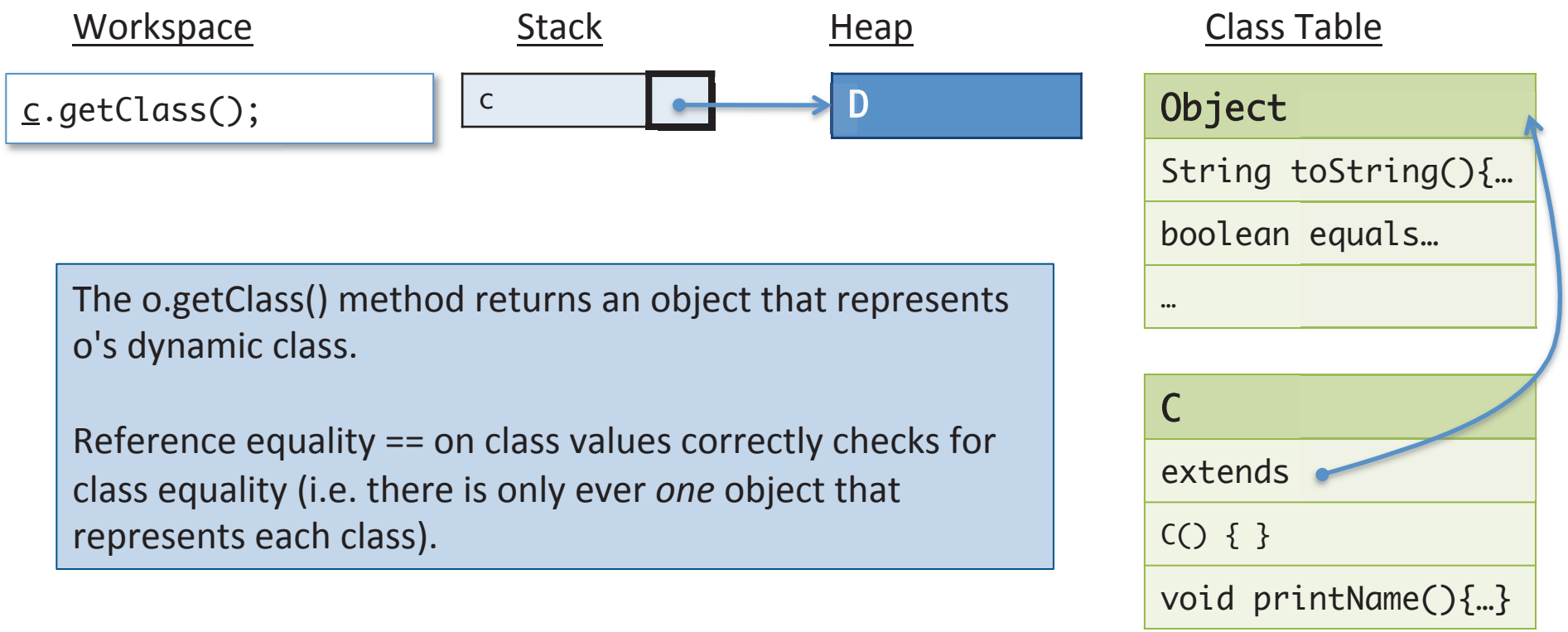
Announcements

- HW7: Chat Server
  - Due *TOMORROW* November 14th
- HW8: SpellChecker
  - Available Weds.
  - Due next Tuesday, November 21st
- Midterm Results / Solutions, etc. will be available after the make-up exams have been graded.
- Next Week is *Thanksgiving Break*:
  - No lab sections
  - Wednesday, November 22nd: Bonus Lecture offered ONLY at the 11:00-noon timeslot (everyone welcome to attend)
  - Topic: Code *is* Data (fun, but not relevant to HW or exam materials)

Overriding equals, take two

Review: `instanceof` vs. `getClass()`

- To correctly account for subtyping, we need the classes of the two objects to match *exactly*.
- `instanceof` only lets us ask about the *subtype* relation
- How do we access the dynamic class?

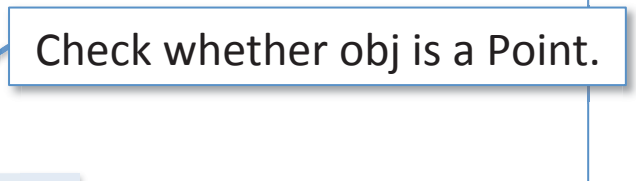


The o.getClass() method returns an object that represents o's dynamic class.

Reference equality == on class values correctly checks for class equality (i.e. there is only ever *one* object that represents each class).

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;
    }
```



Equality and Hashing

- Whenever you override equals you ***must*** *also* override `hashCode` in a compatible way
  - If `o1.equals(o2)` then `o1.hashCode() == o2.hashCode()`
  - hashCode is used by the HashSet and HashMap collections
- Forgetting to do this can lead to extremely puzzling bugs!
  - For HW07 *we do not allow* the use of HashSet, etc., for this reason.

Overriding Equality in Practice

- Some tools (i.e. Eclipse) can autogenerate equality methods of the kind we developed.
  - But you need to specify which fields should be taken into account.
  - and you should know why some comparisons use == and some use .equals
  - You need to understand why the generated code is the way it is.

Enumerations

Enumerations (a.k.a. Enum Types)

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

```
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

```
// Use of 'enum' in CommandParser.java (PennPals 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.*, plus Enum types and 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?

```
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

Answer: 5 something else!

break

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

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

- Use an explicit **break** to avoid fallthrough:

```
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");
}
```

Enumerations

- Enum types are just a convenient way of defining a class along with some standard methods.
  - 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();

Iterating over collections

iterators, while, for, for-each loops

Iterator and Iterable

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

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

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
}
```

boolean *guard* expression

statement

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).

Iterator example

```
public static void iteratorExample() {
    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;
    }

    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

For-each version

```
public static void forEachExample() {
    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;
    }

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

Another Iterator example

```
public static void nextNextExample() {
    List<Integer> nums = new LinkedList<Integer>();
    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

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 that doesn't exist
- Resources might be exhausted
  - Program uses all of the computer's memory or 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 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
  - e.g. in Java, we can create a special class like option
  - ☹ *Passes responsibility to caller, who is 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

```
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

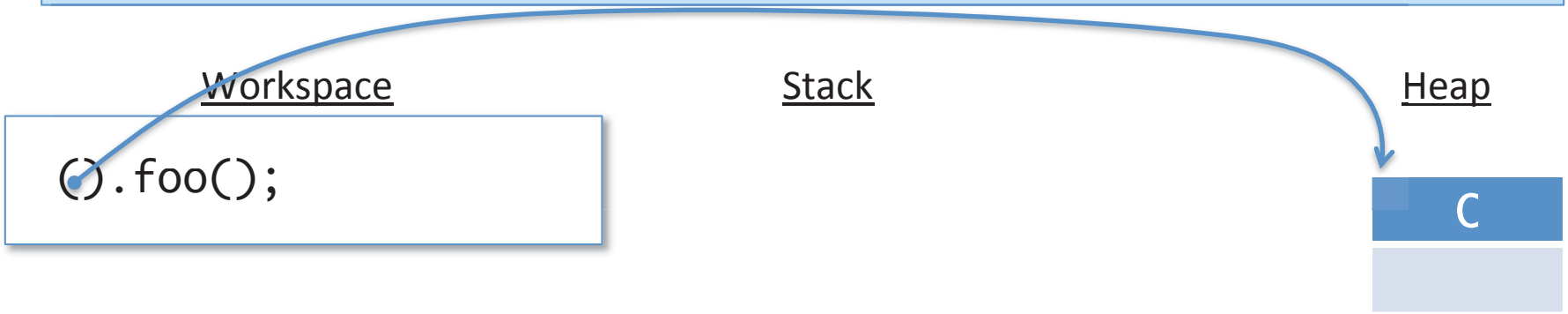
Workspace

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

Stack

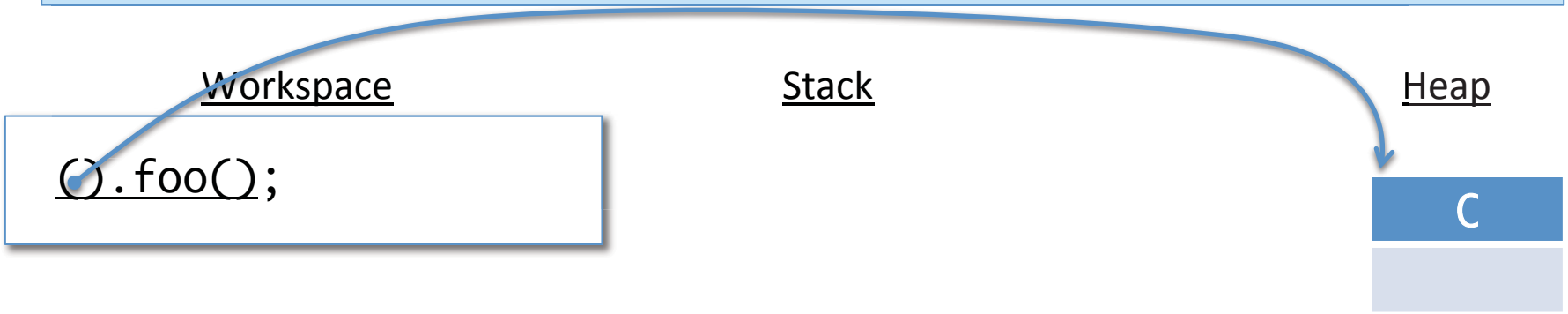
Heap

Abstract Stack Machine

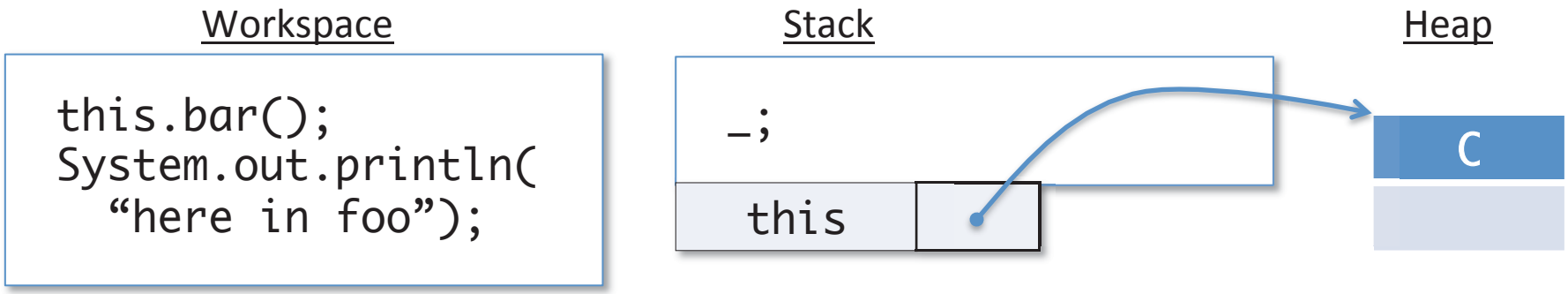


Allocate a new instance of C in the heap. (Skipping details of trivial constructor for C.)

Abstract Stack Machine

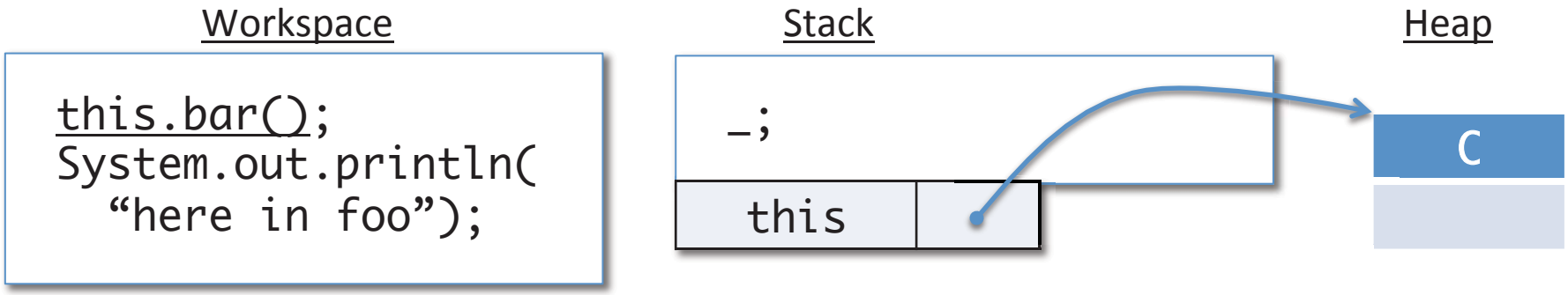


Abstract Stack Machine

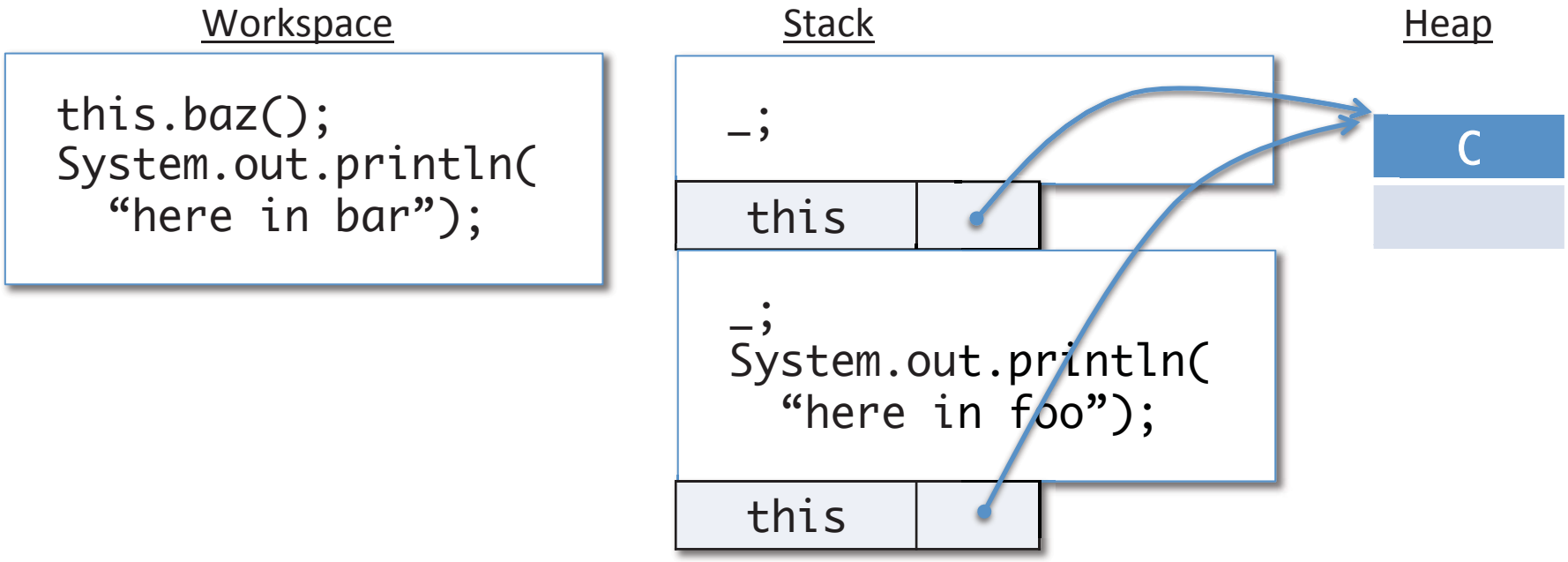


Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack. Use the dynamic class to lookup the method body from the class table.

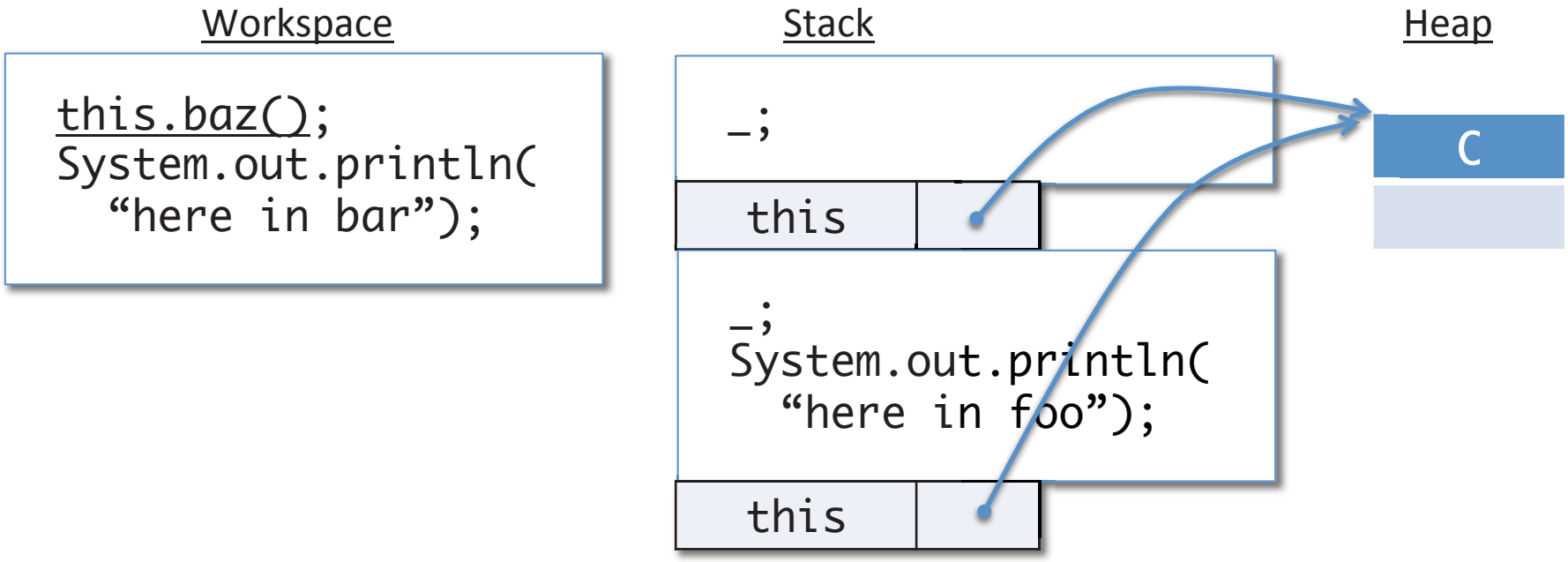
Abstract Stack Machine



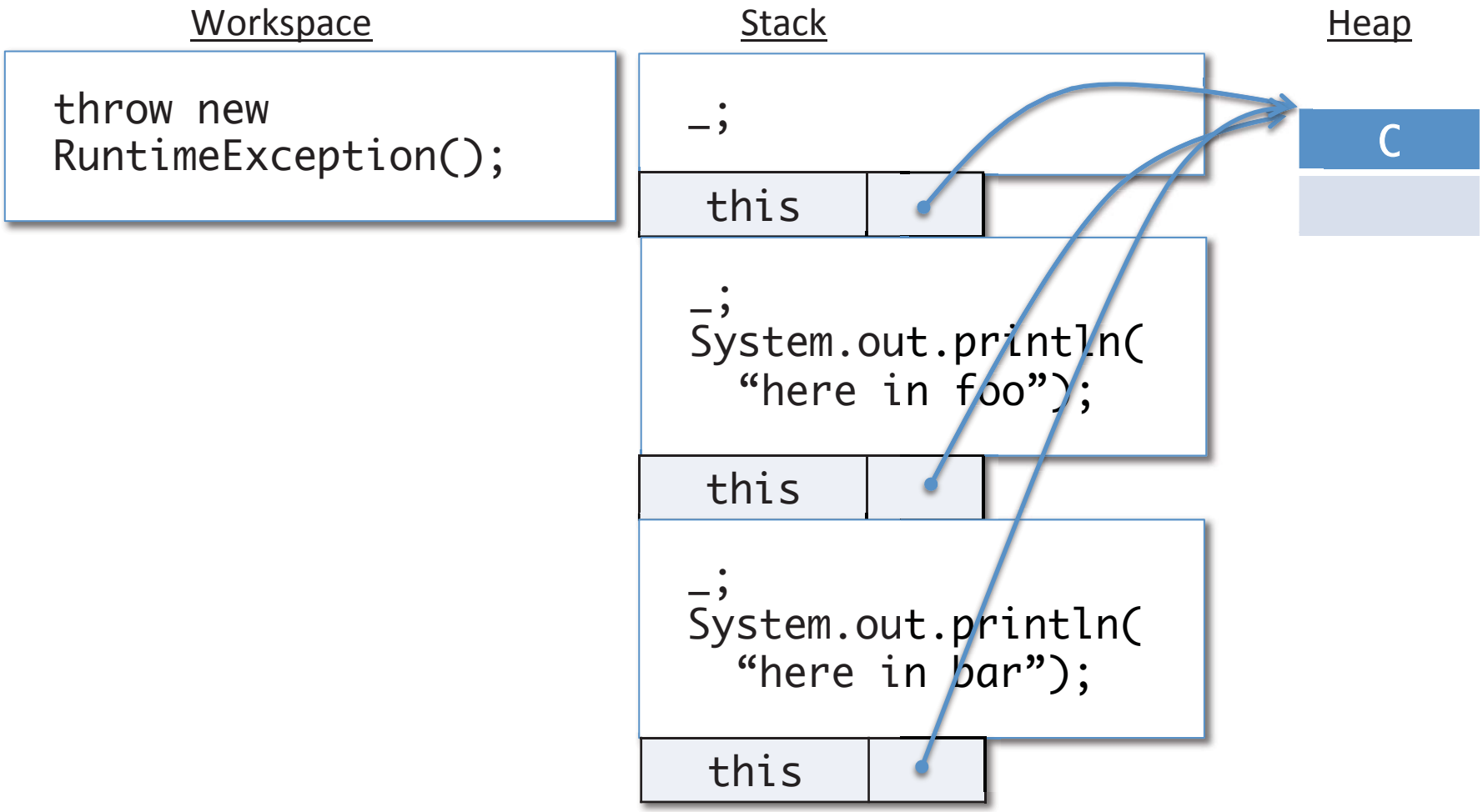
Abstract Stack Machine



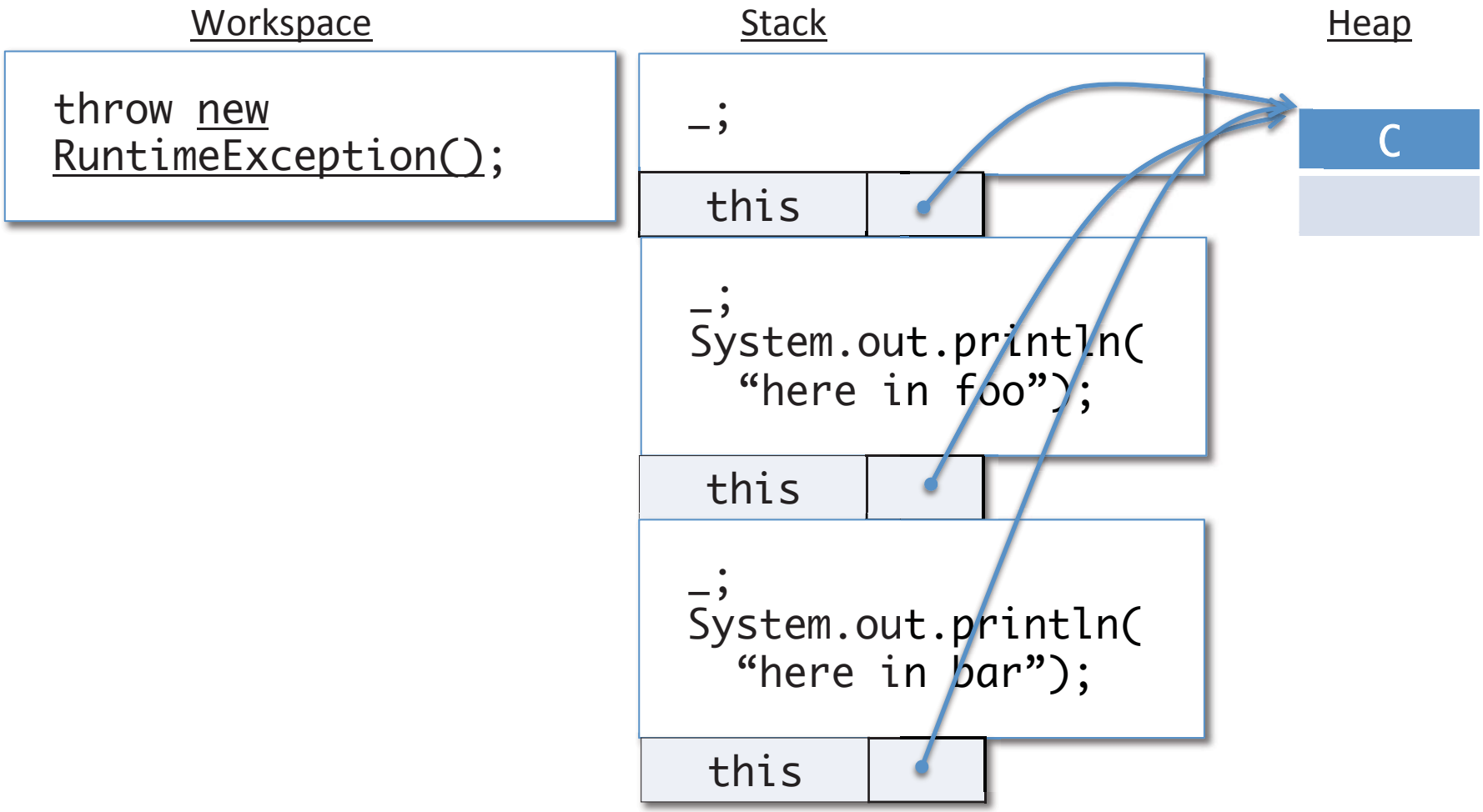
Abstract Stack Machine



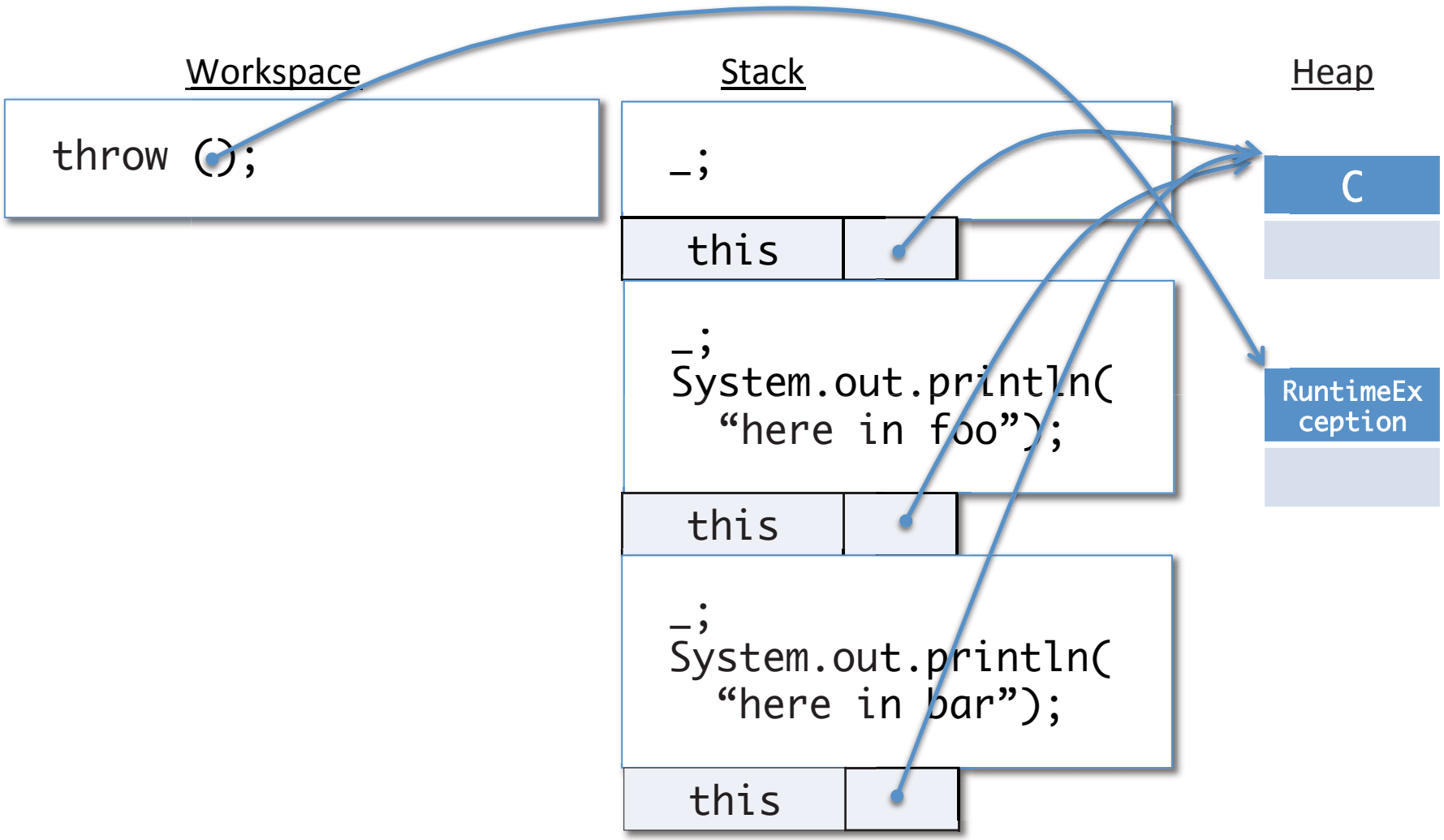
Abstract Stack Machine



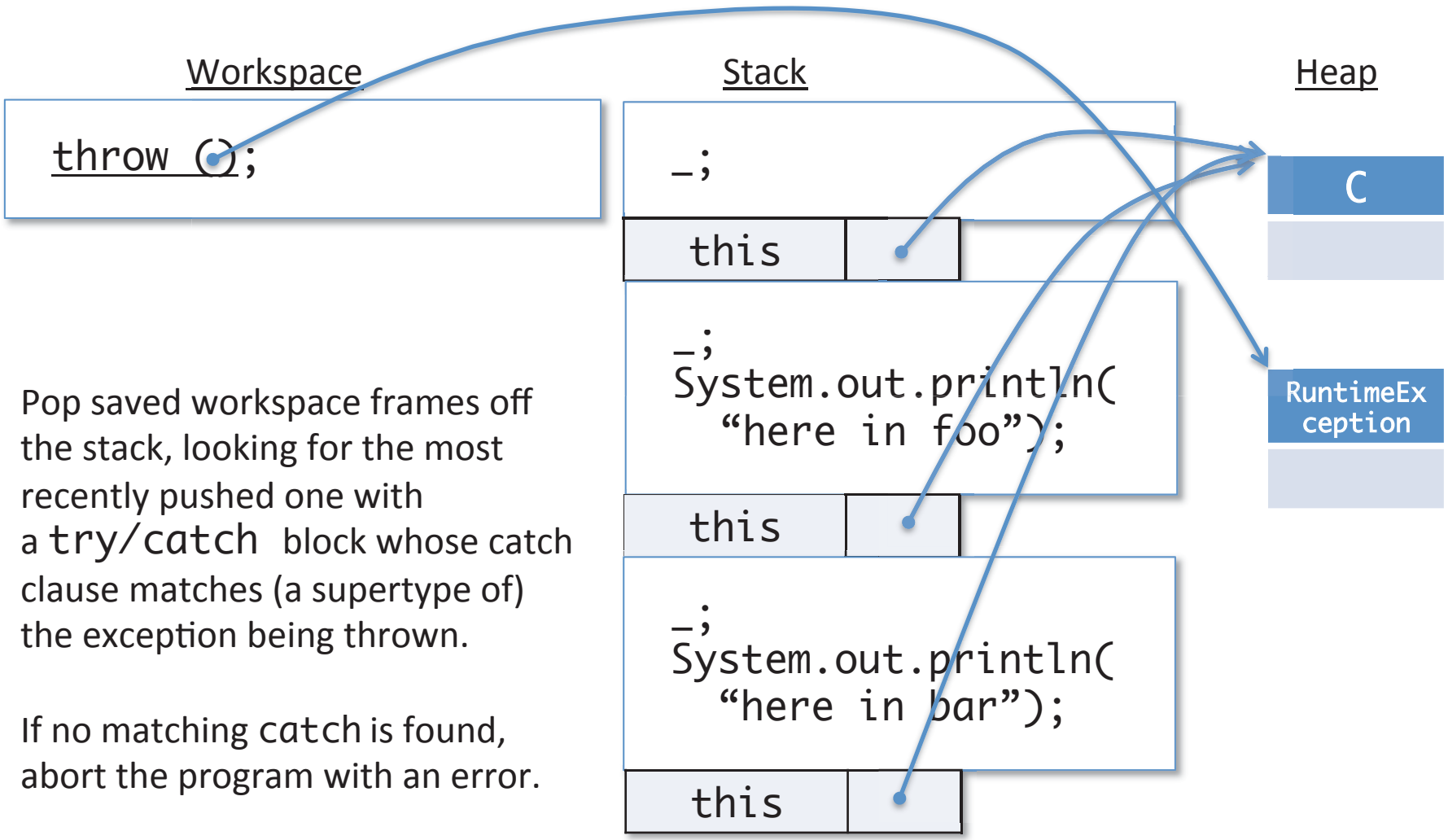
Abstract Stack Machine



Abstract Stack Machine



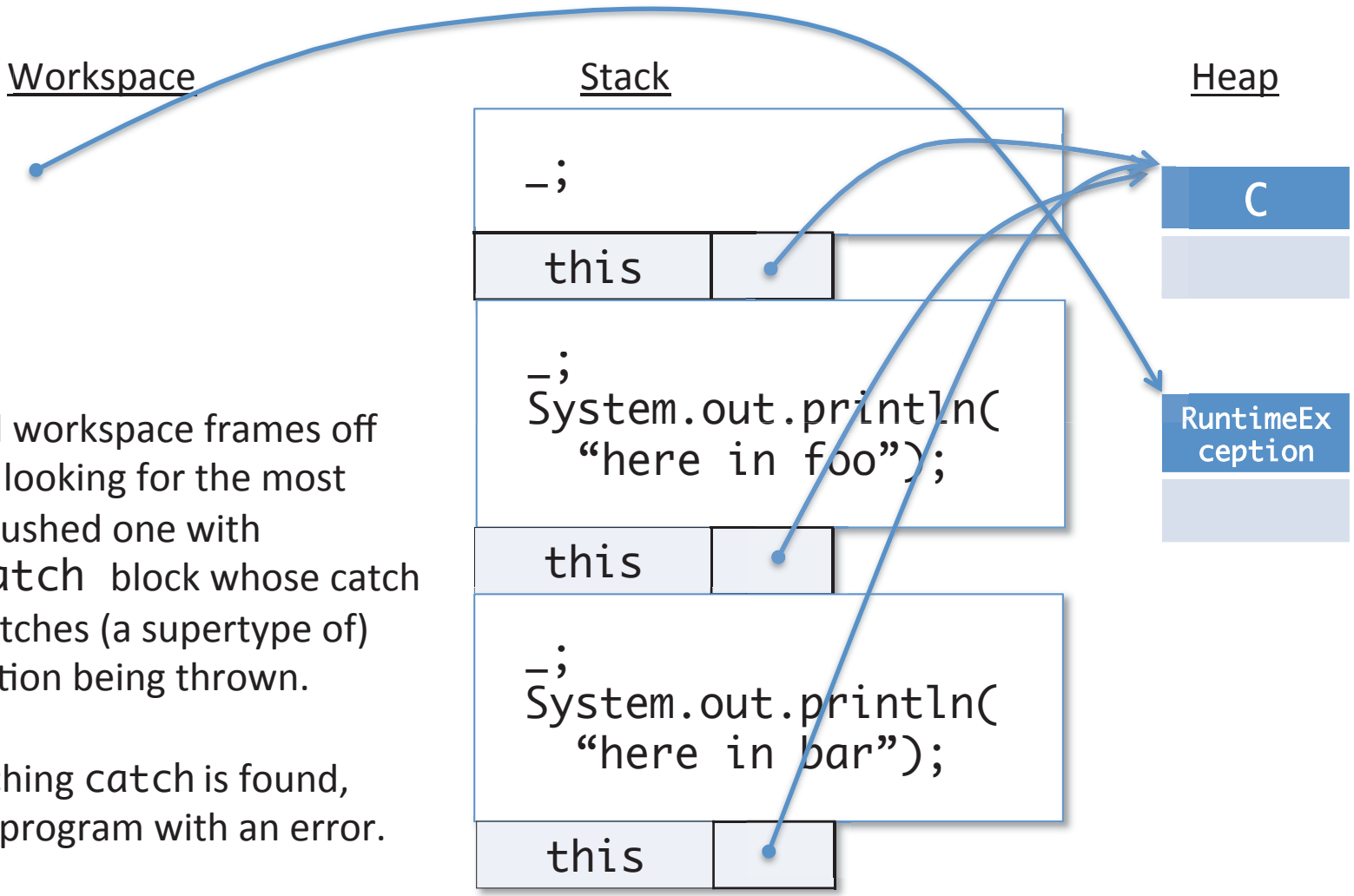
Abstract Stack Machine



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.

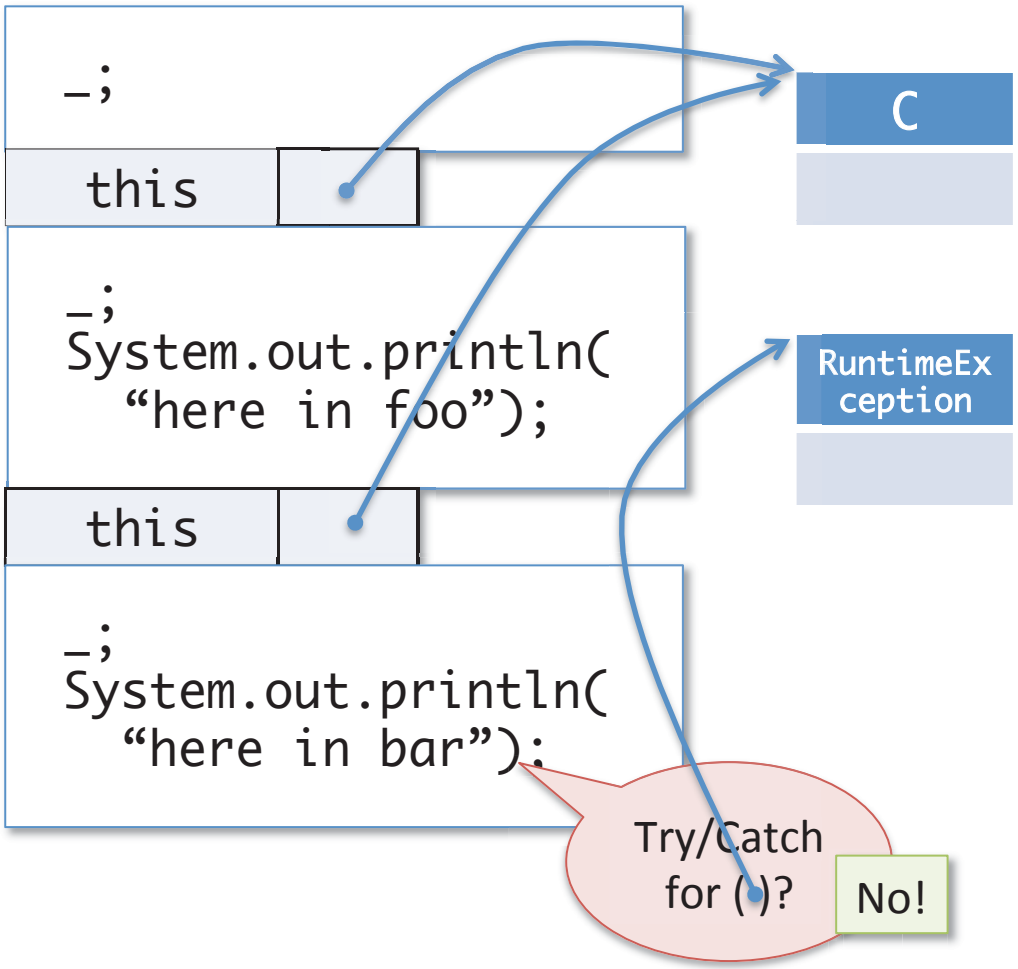
If no matching `catch` is found, abort the program with an error.

Abstract Stack Machine

Workspace

Stack

Heap



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

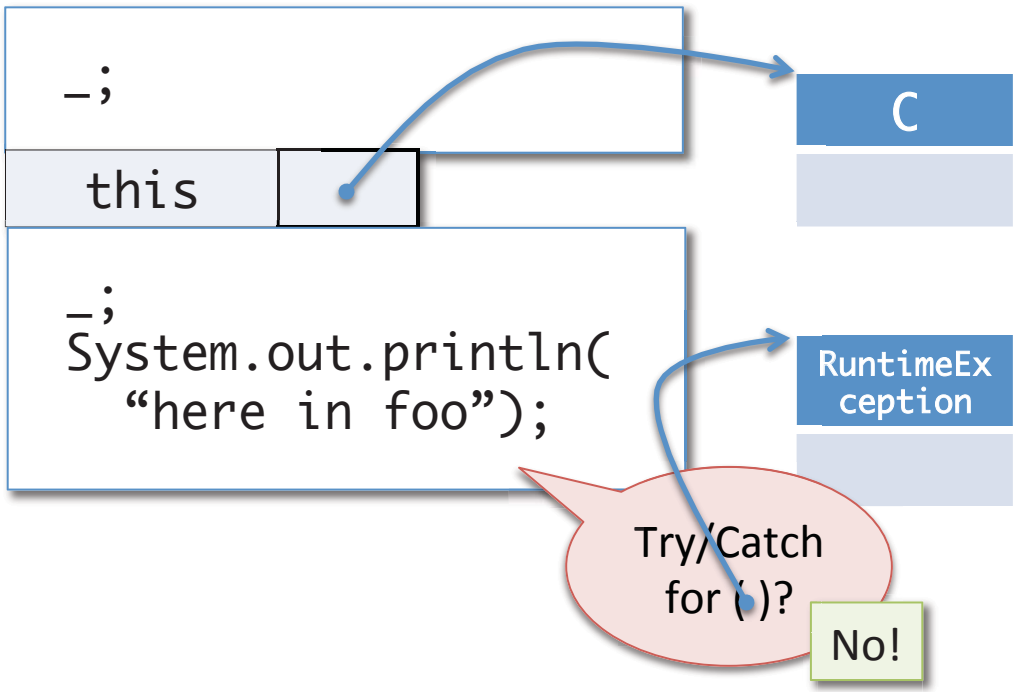
If no matching `catch` is found, abort the program with an error.

Abstract Stack Machine

Workspace

Stack

Heap

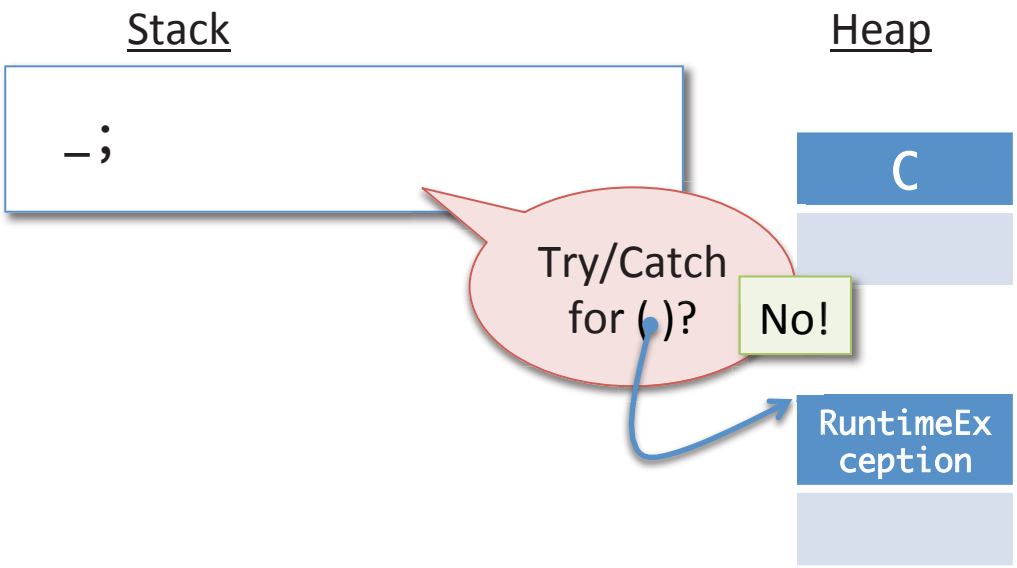


Discard the current workspace.

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

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

Abstract Stack Machine



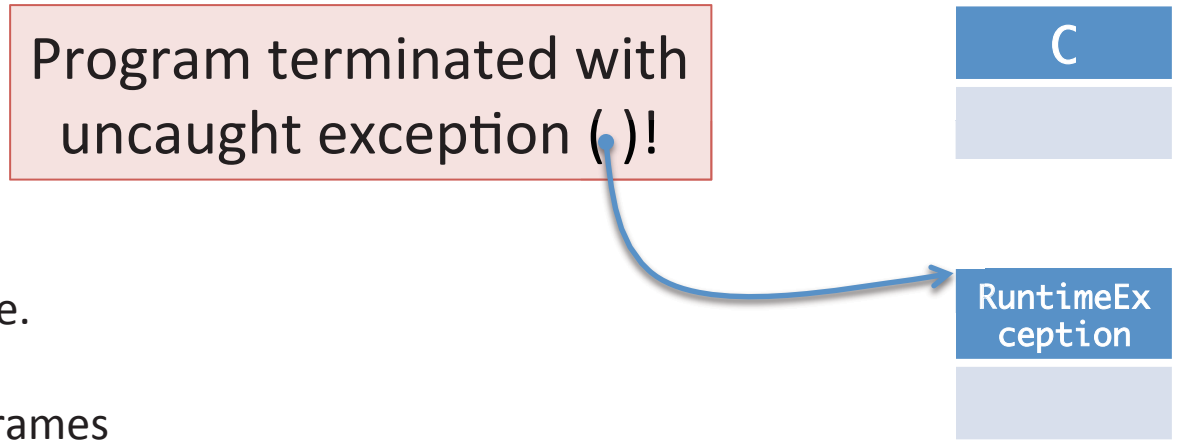
Discard the current workspace.

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

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

Abstract Stack Machine

Workspace | Stack | Heap



Discard the current workspace.

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

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

Catching the Exception

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

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

Abstract Stack Machine

Workspace

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

Stack

Heap

Abstract Stack Machine

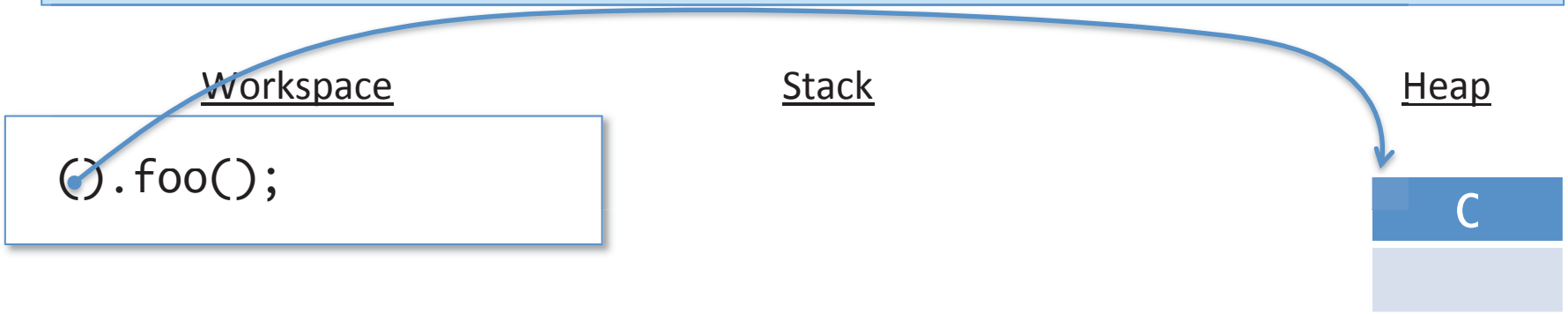
Workspace

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

Stack

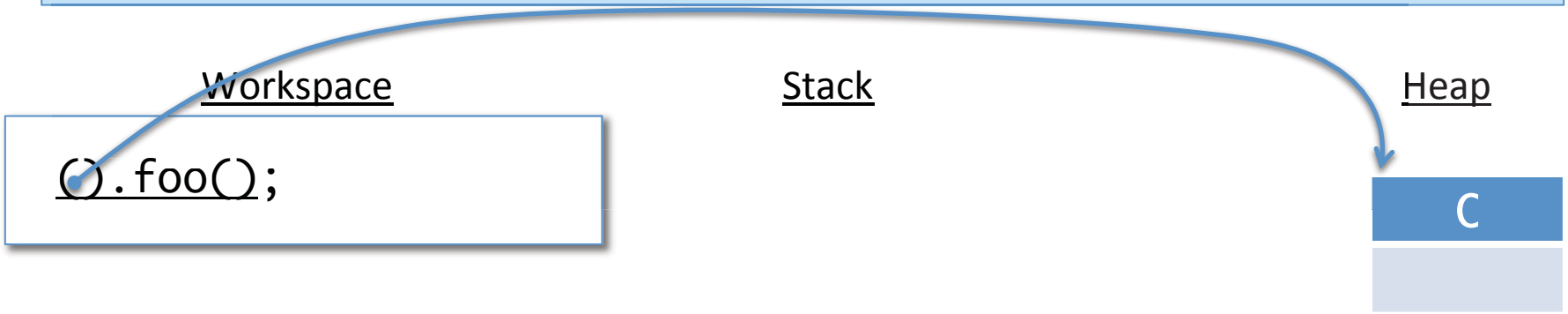
Heap

Abstract Stack Machine

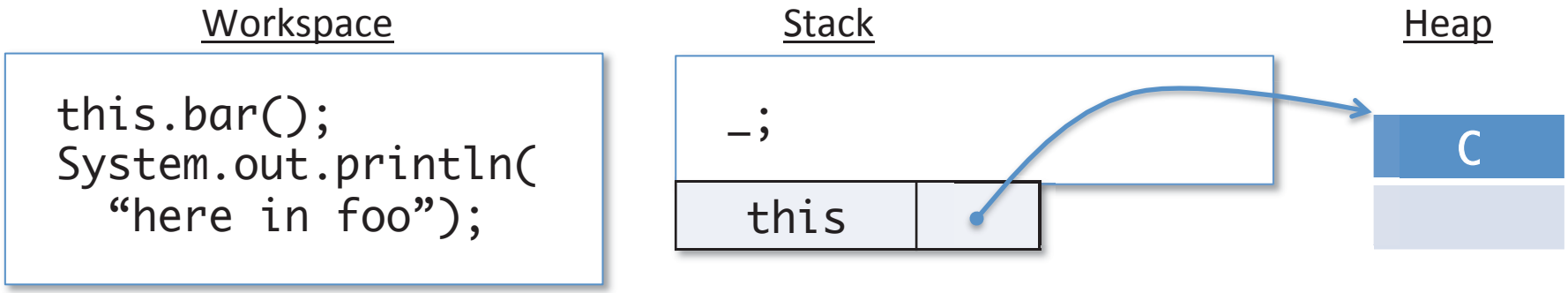


Allocate a new instance of C in the heap.

Abstract Stack Machine

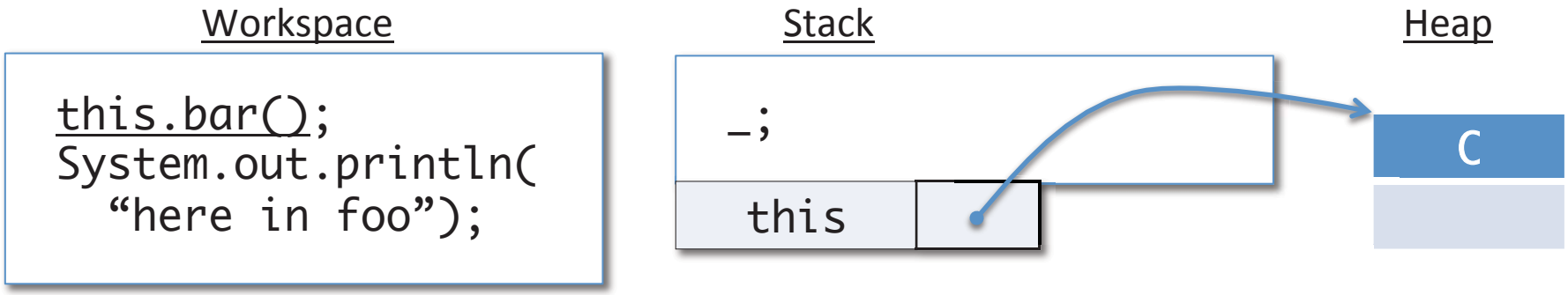


Abstract Stack Machine

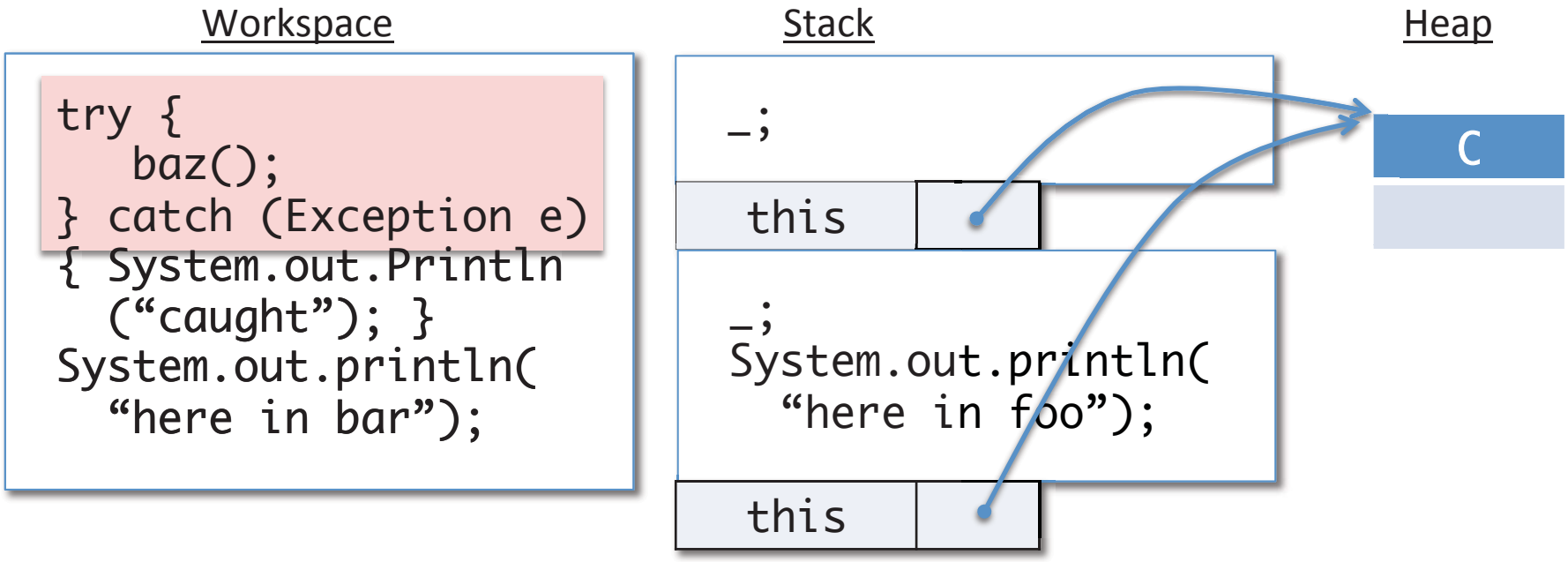


Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack.

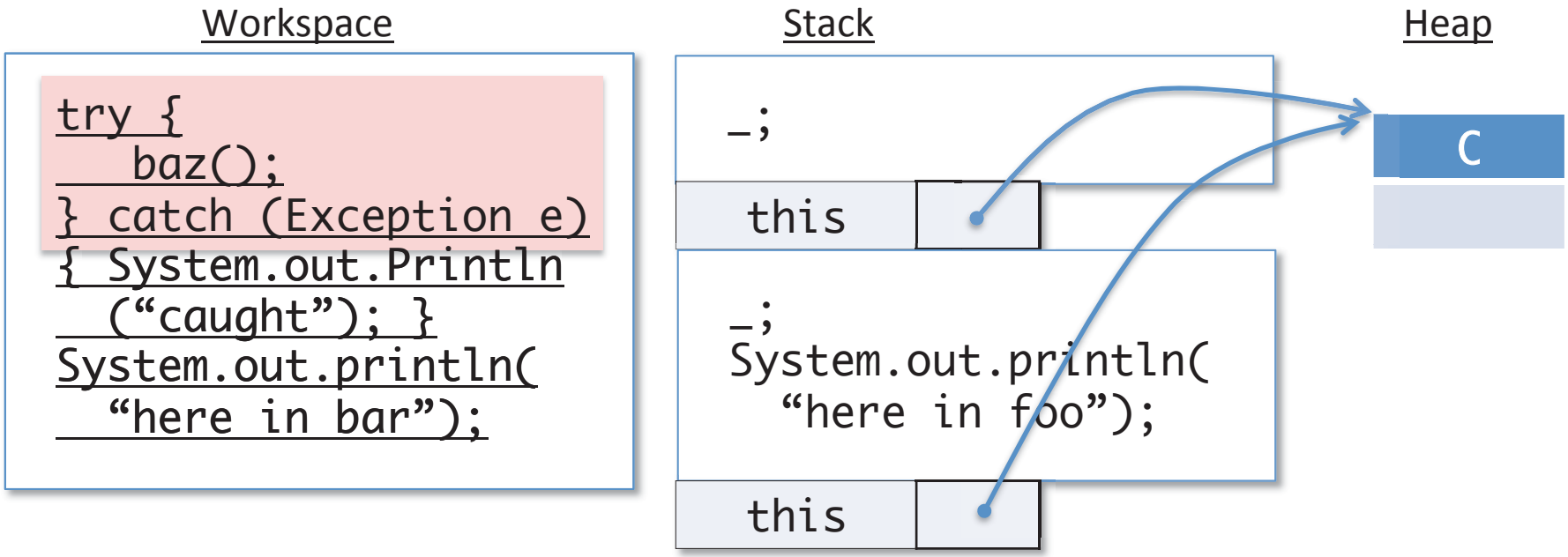
Abstract Stack Machine



Abstract Stack Machine



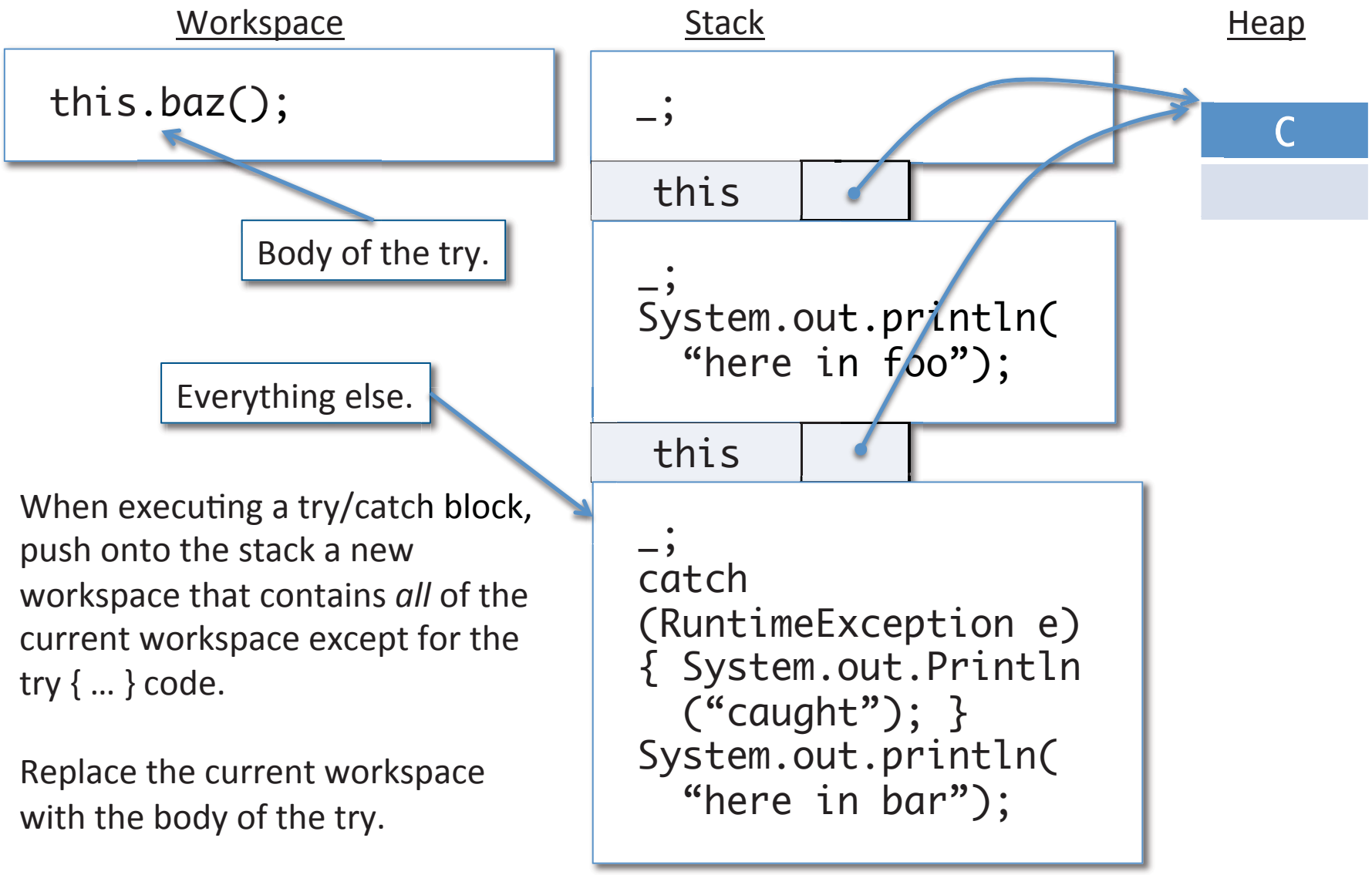
Abstract Stack Machine



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.

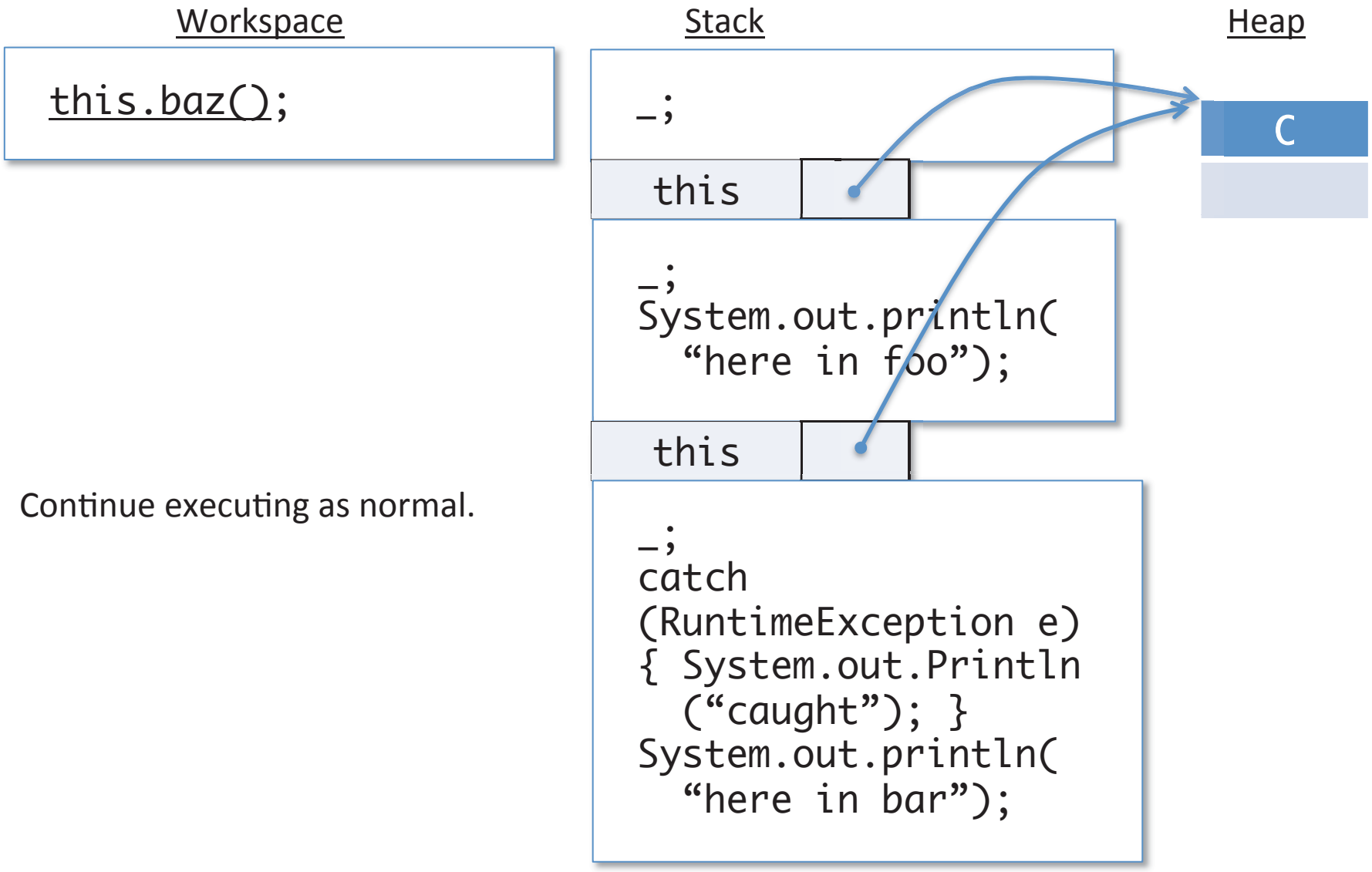
Abstract Stack Machine



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.

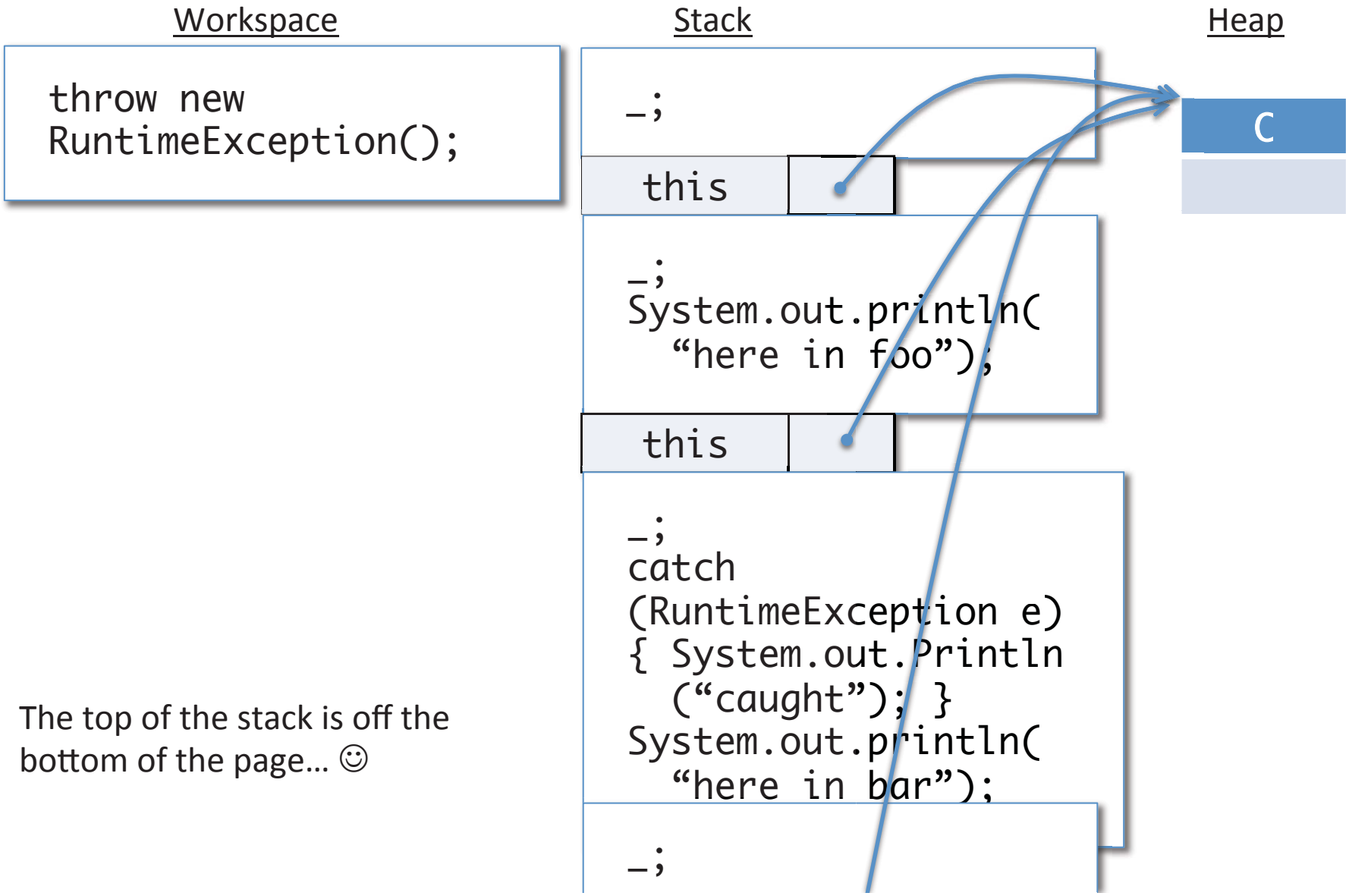
Replace the current workspace with the body of the try.

Abstract Stack Machine



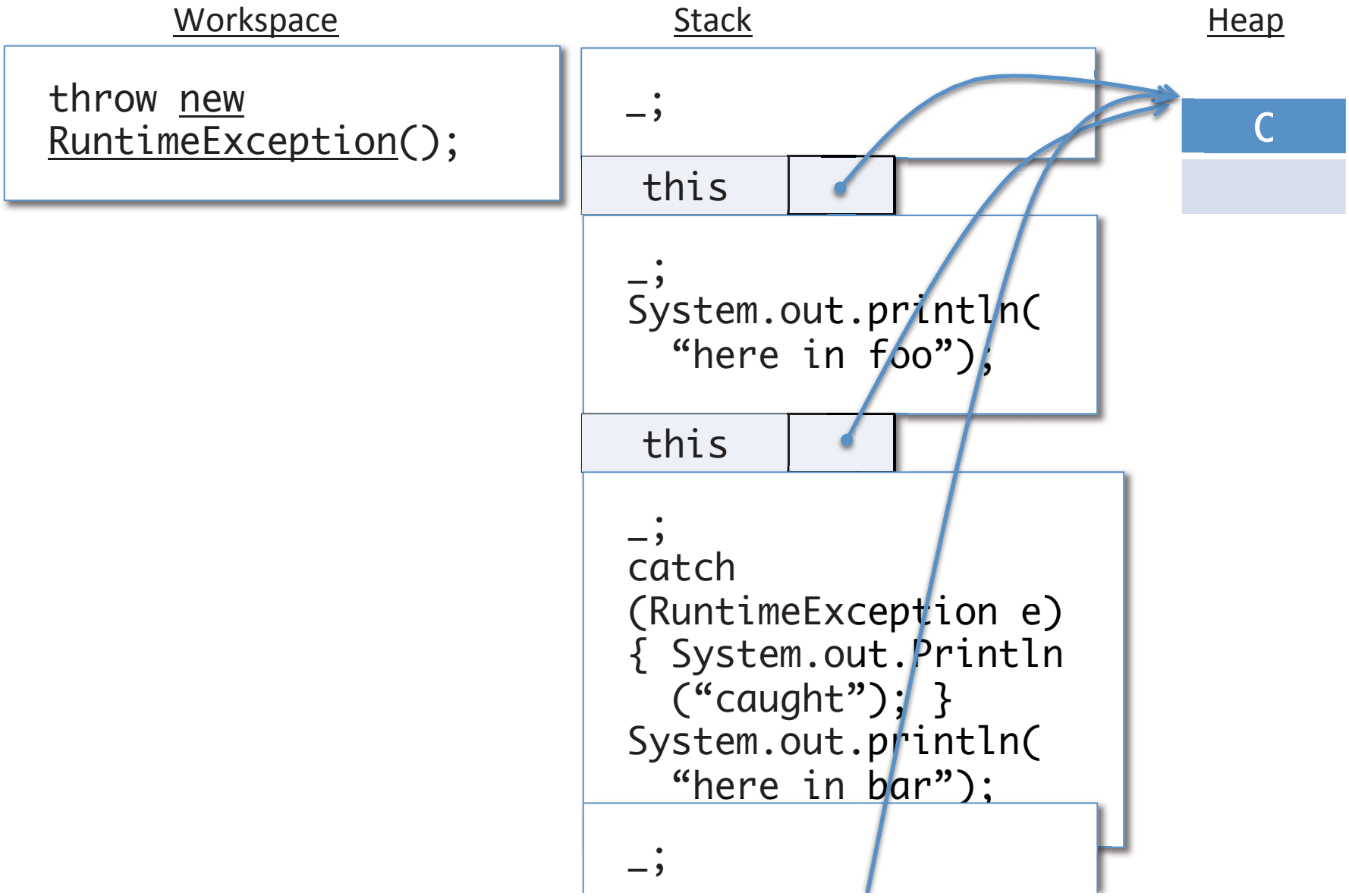
Continue executing as normal.

Abstract Stack Machine

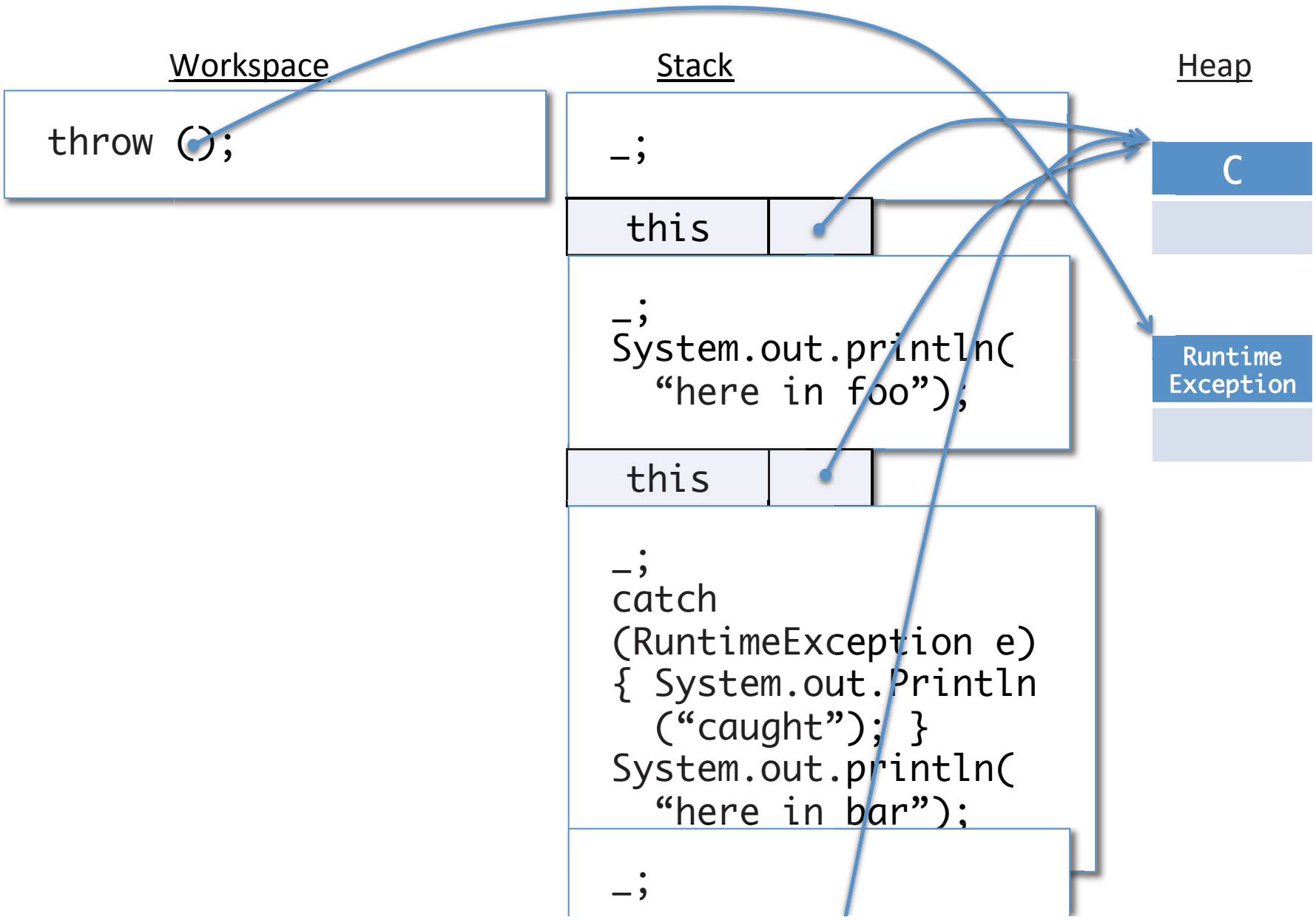


The top of the stack is off the bottom of the page… ☺

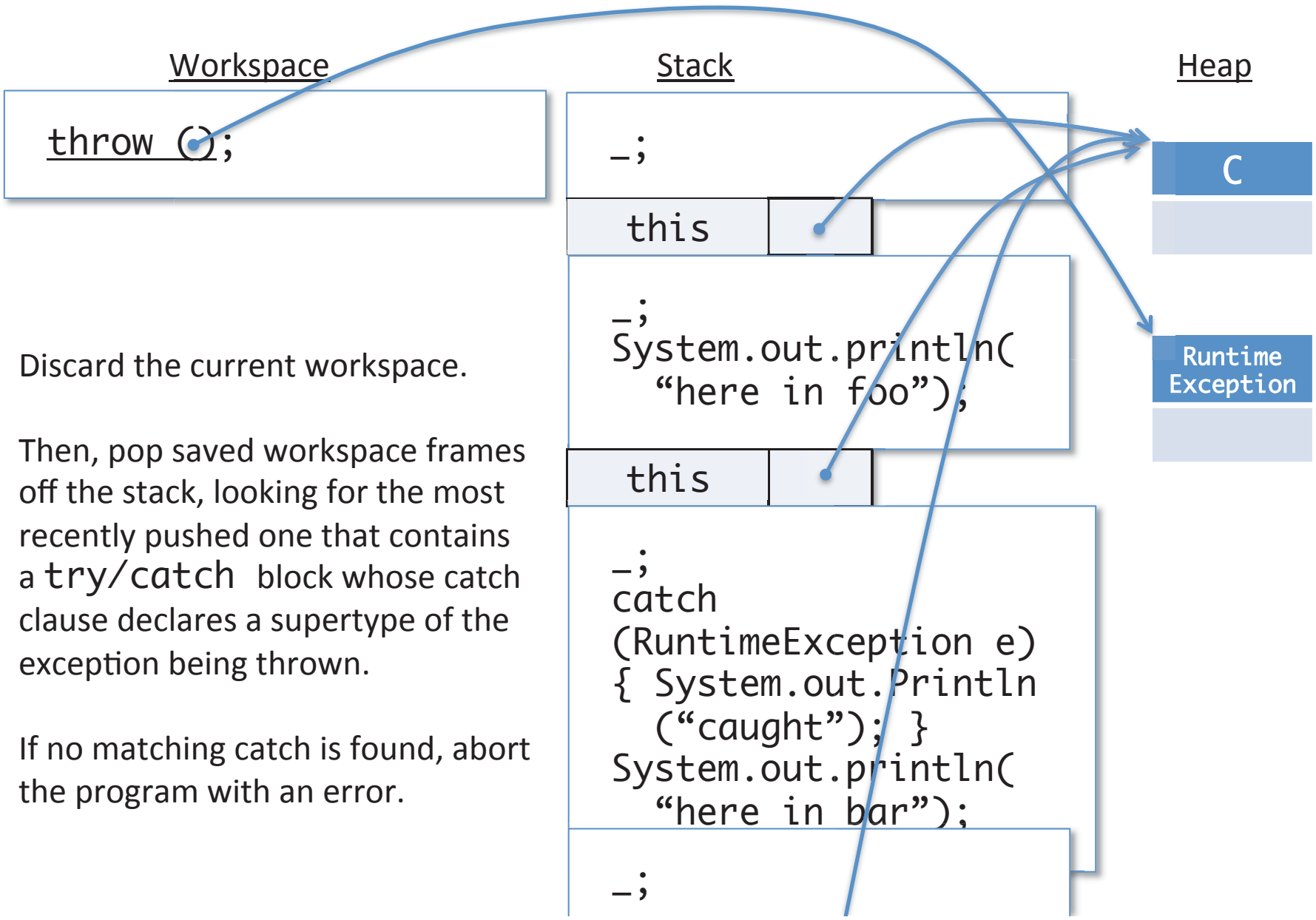
Abstract Stack Machine



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.

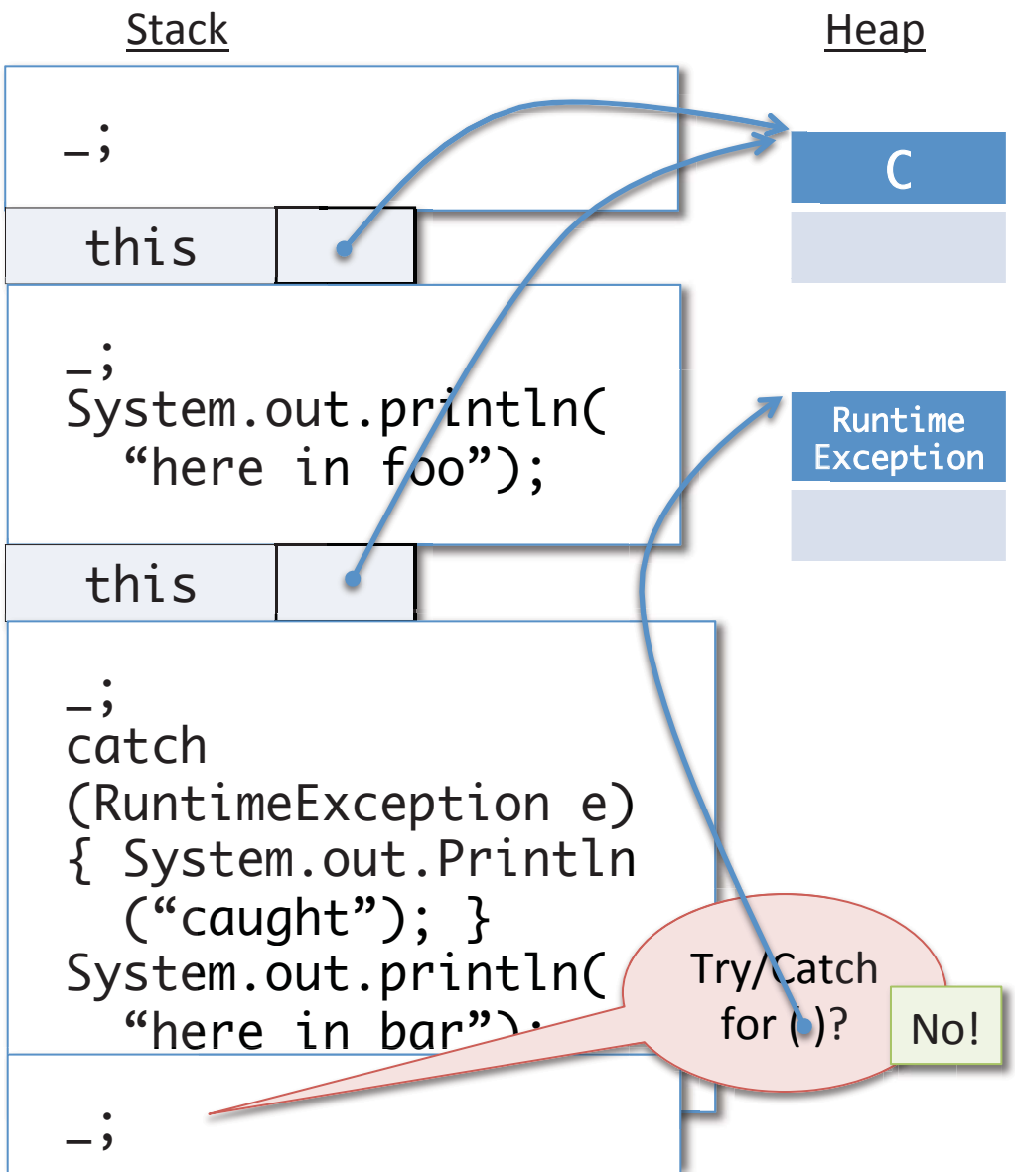
If no matching catch is found, abort the program with an error.

Abstract Stack Machine

Discard the current workspace.

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

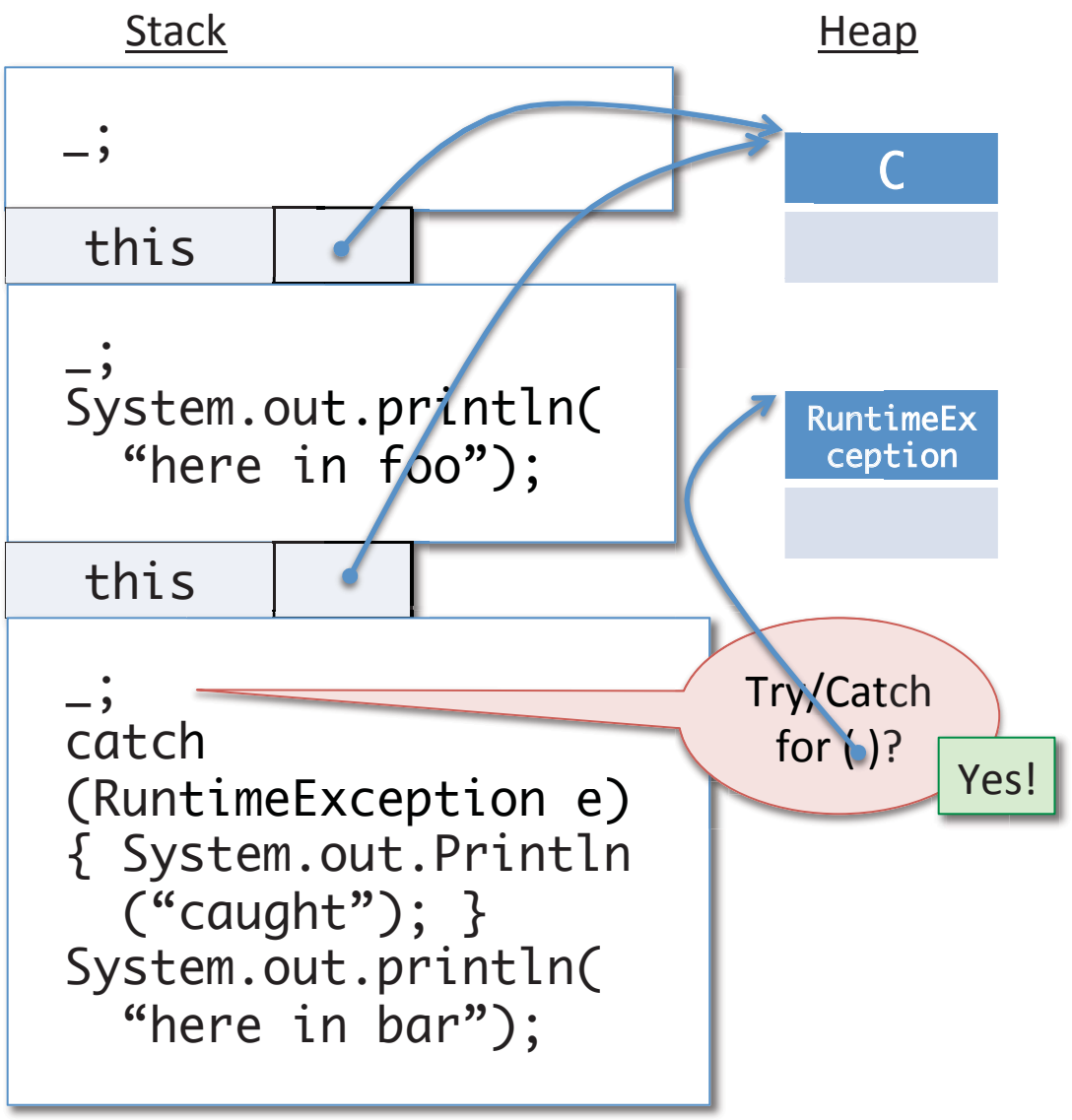
If no matching catch is found, abort the program with an error.



Abstract Stack Machine

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

Continue executing as usual.



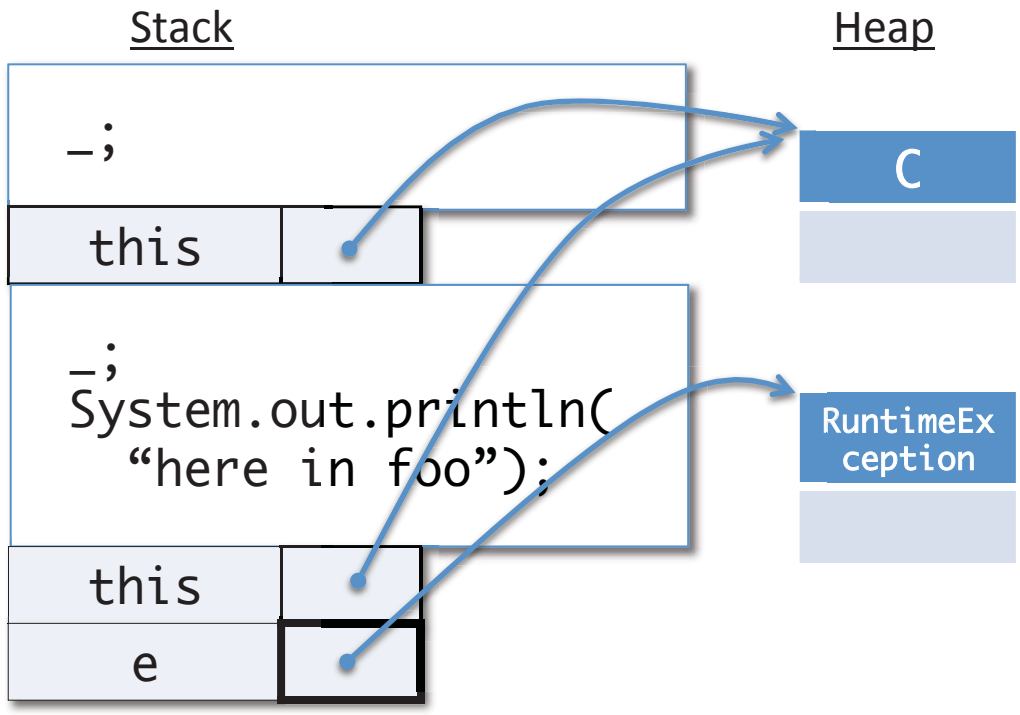
Abstract Stack Machine

Workspace

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

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

Continue executing as usual.

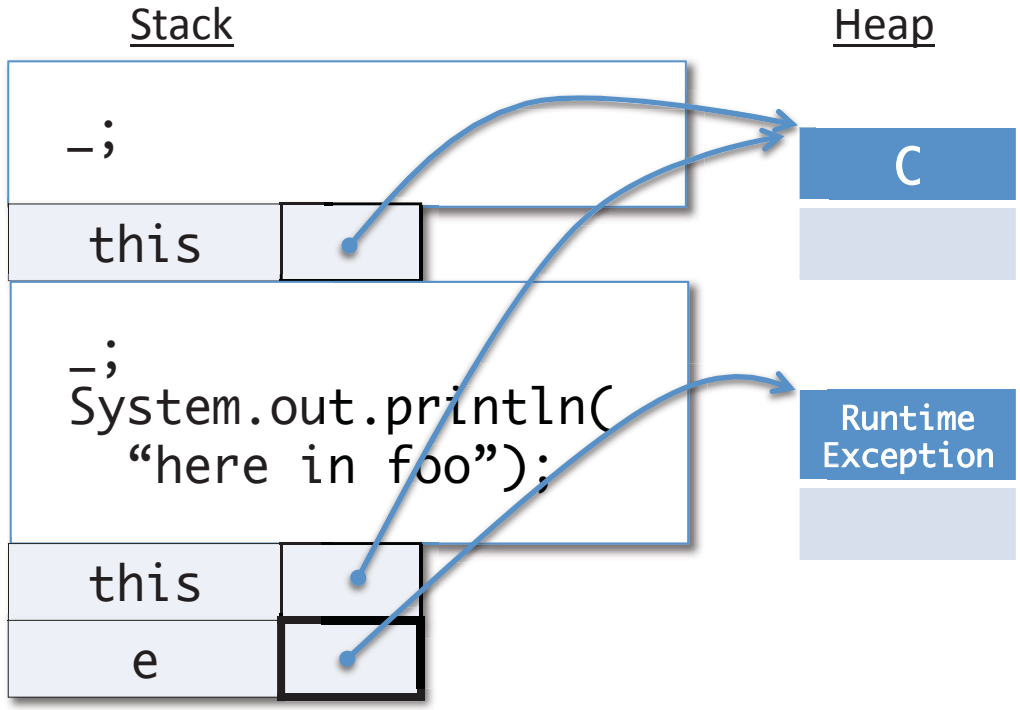


Abstract Stack Machine

Workspace

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

Continue executing as usual.

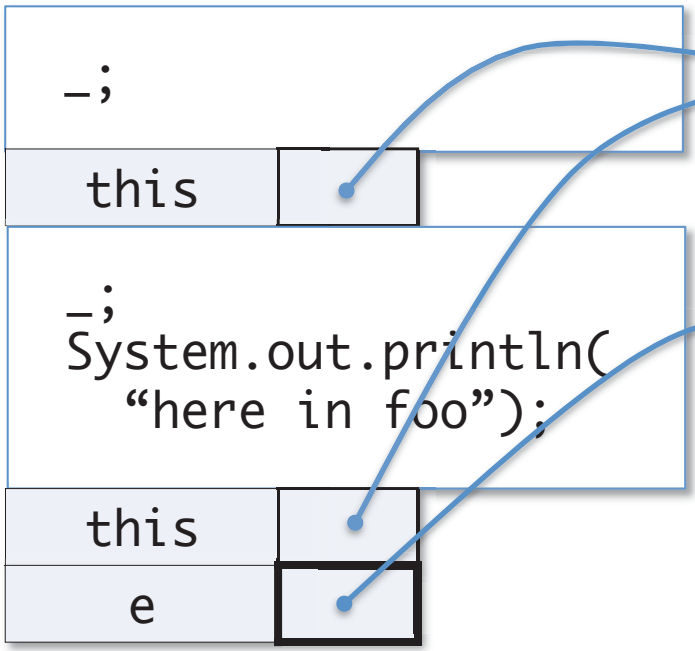


Abstract Stack Machine

Workspace

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

Stack



Heap

Continue executing as usual.

Console

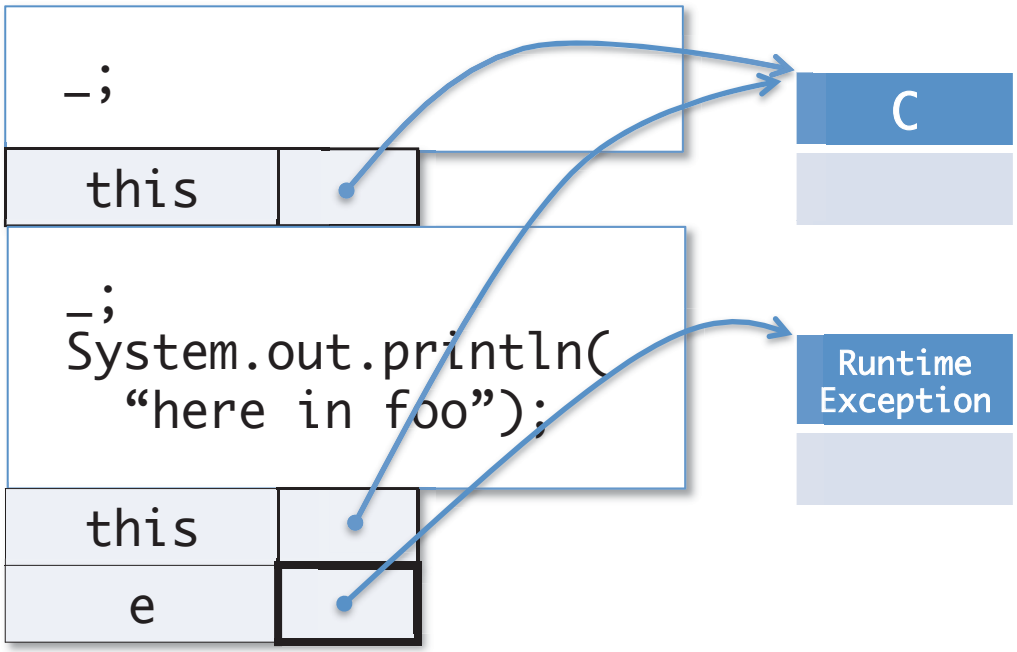
```
caught
```

Abstract Stack Machine

Workspace

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

Stack

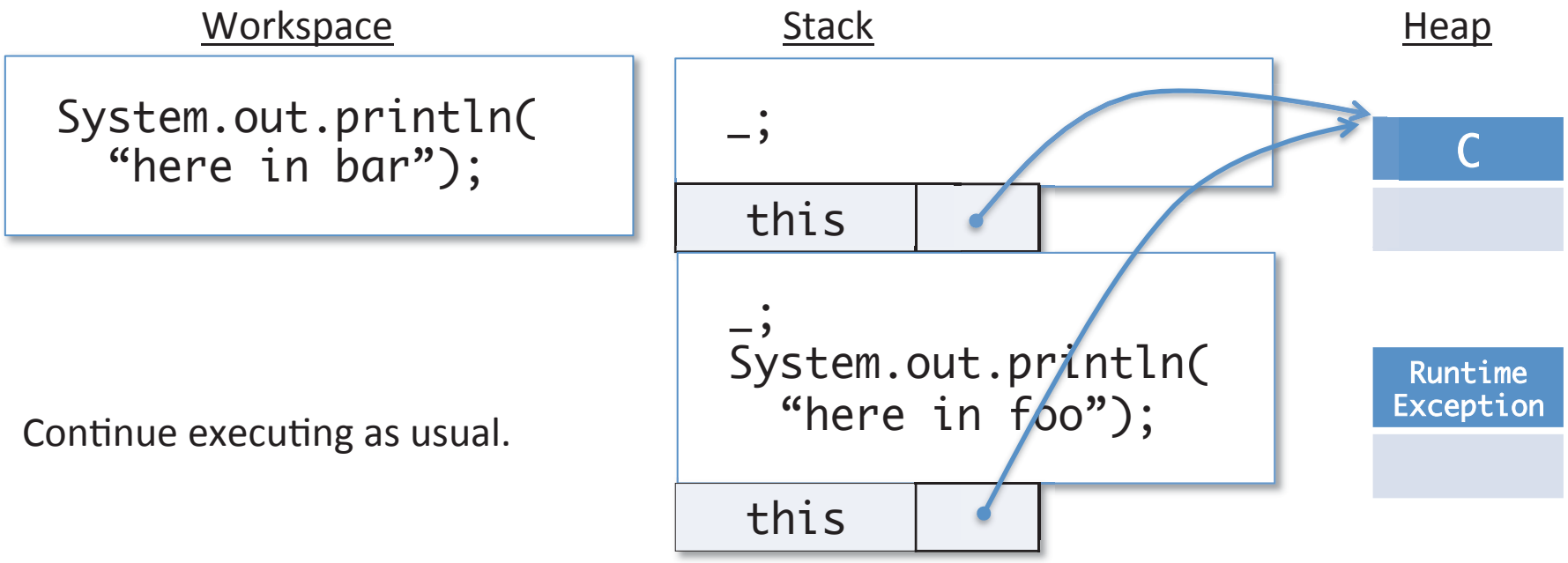


We’re sweeping a few details about lexical scoping of variables under the rug – the scope of e is just the body of the catch, so when that is done, e must be popped from the stack.

Console

```
caught
```

Abstract Stack Machine



Continue executing as usual.

Console

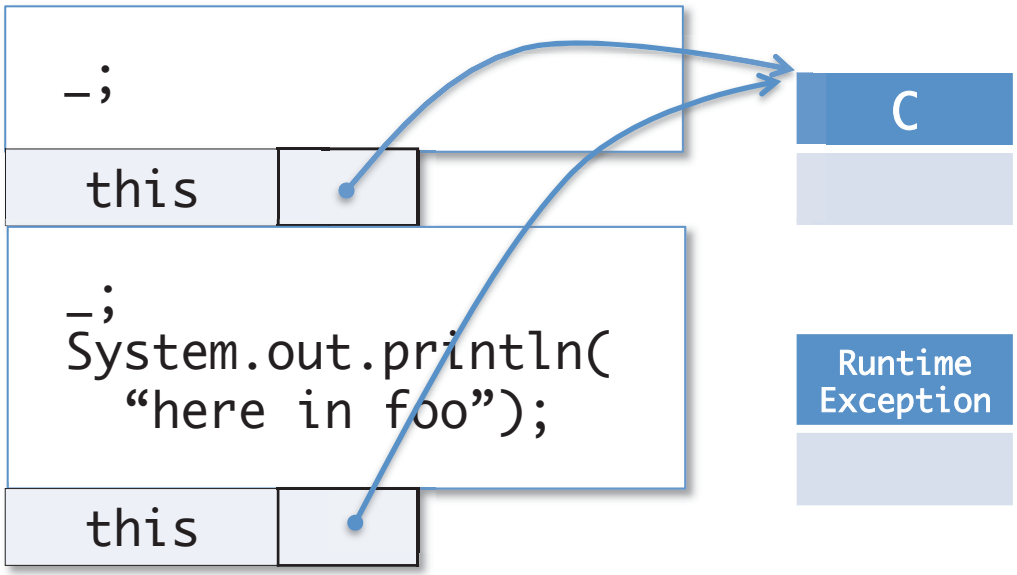
caught

Abstract Stack Machine

Workspace

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

Stack



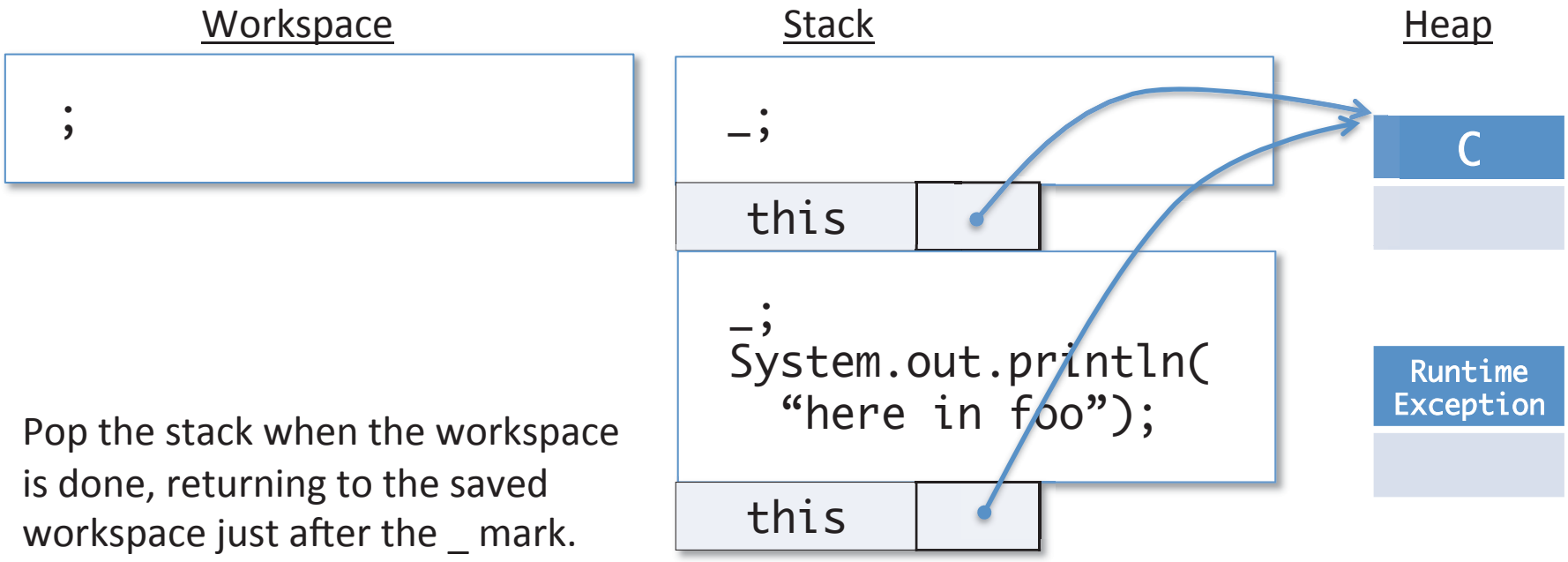
Heap

Continue executing as usual.

Console

```
caught
```

Abstract Stack Machine



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

Console

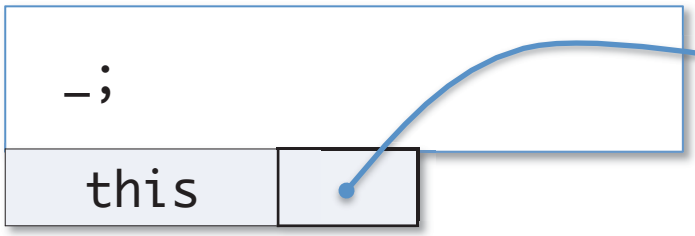
```
caught
here in bar
```

Abstract Stack Machine

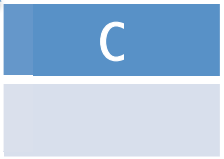
Workspace

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

Stack



Heap

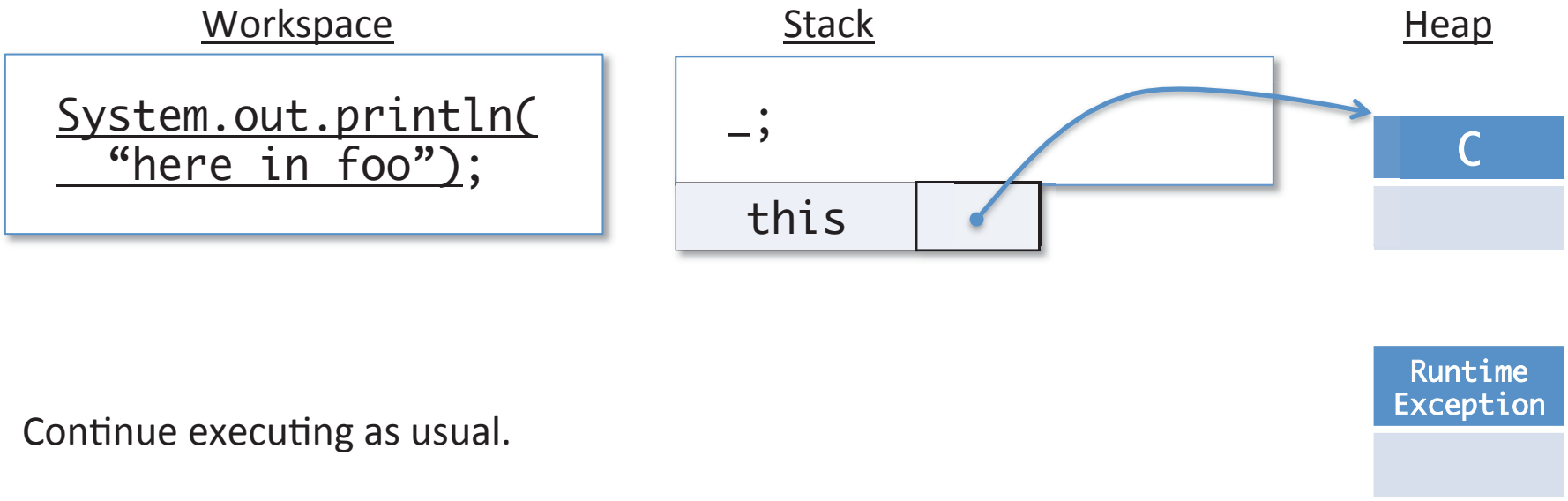


Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine

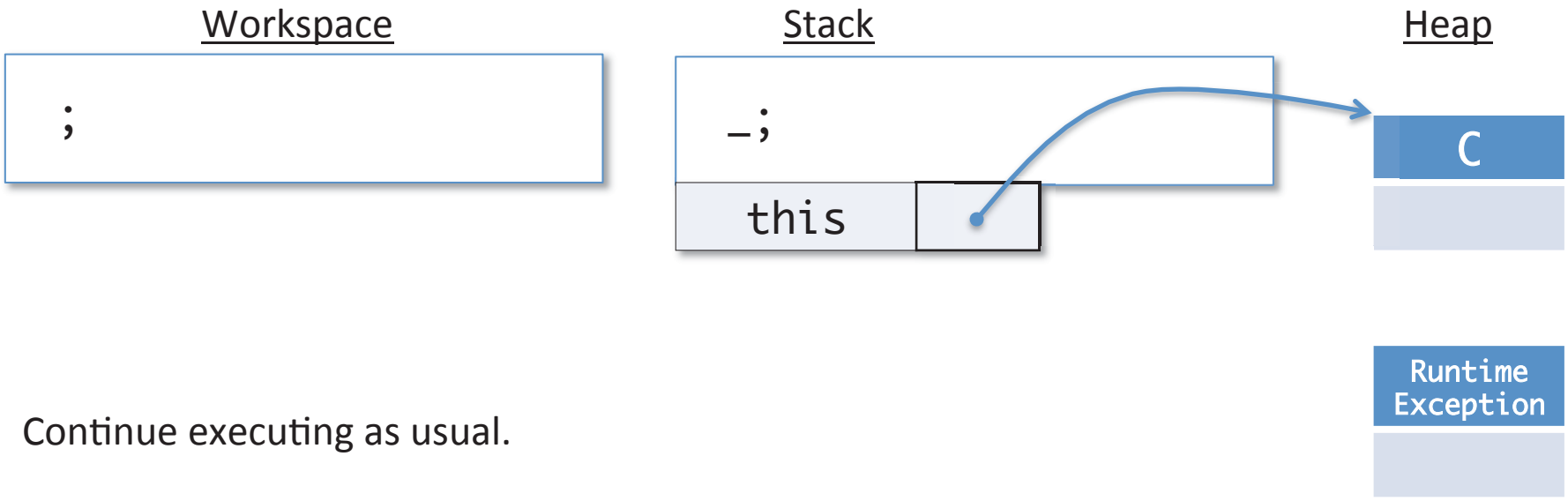


Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine



Continue executing as usual.

Console

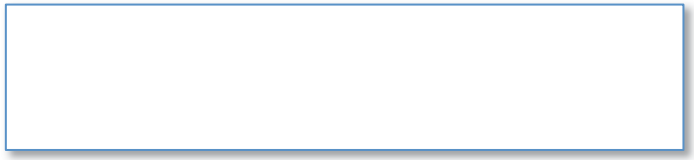
```
caught
here in bar
here in foo
```

Abstract Stack Machine

Workspace

Stack

Heap



Program terminated normally.

Console

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

When No Exception is Thrown

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

Workspace

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

Workspace

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

Catching Exceptions

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

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

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

Programming Languages and Techniques (CIS120)

Lecture 30

November 14, 2017

Exceptions

Chapter 27

Announcements

- HW8: SpellChecker
  - Available on the website and Codio
  - Due next Tuesday, November 21st

- Next Week is *Thanksgiving Break*:
  - No lab sections
  - Wednesday, November 22nd: Bonus Lecture offered ONLY at the 11:00-noon timeslot (everyone welcome to attend)
  - Topic: Code *is* Data (fun, but not relevant to HW or exam materials)

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 that doesn't exist
- Resources might be exhausted
  - Program uses all of the computer's memory or 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 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
  - e.g. in Java, we can create a special class like option
  - ☹ *Passes responsibility to caller, who is 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

```
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

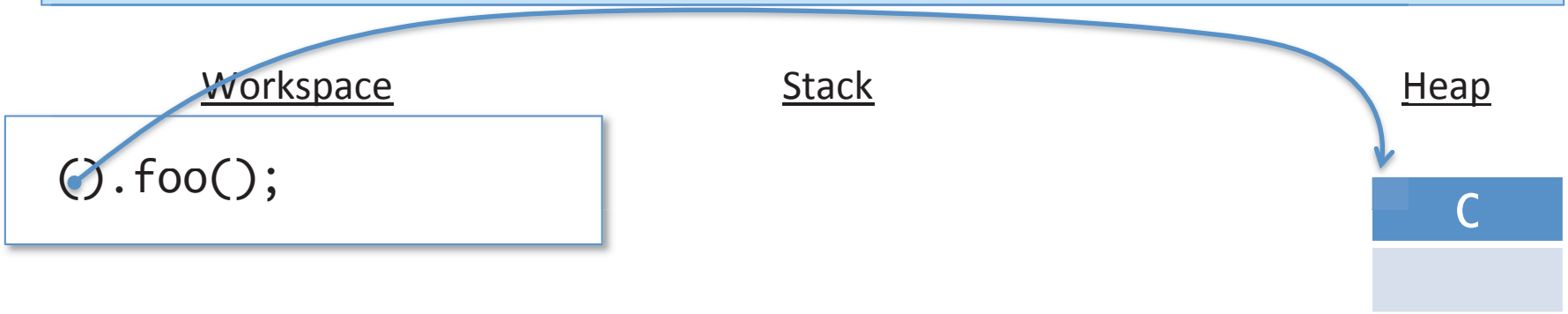
Workspace

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

Stack

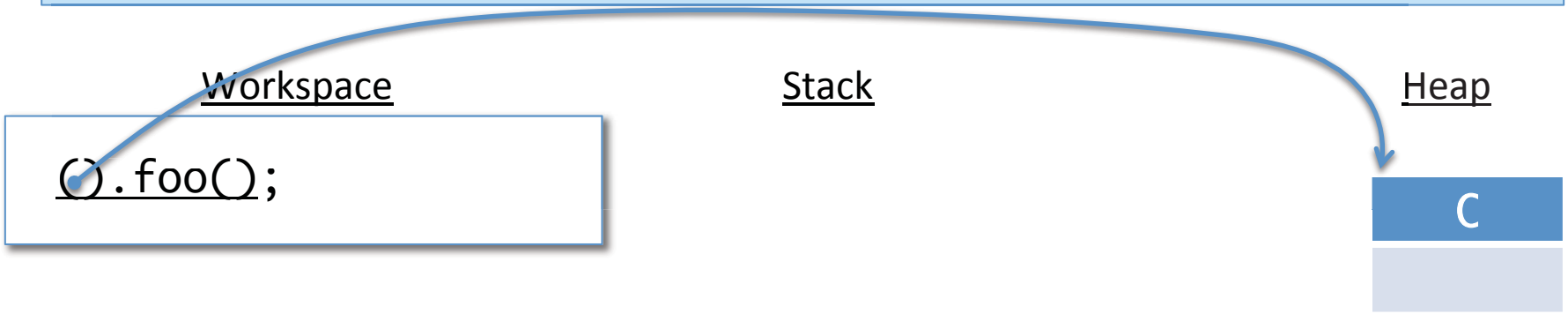
Heap

Abstract Stack Machine

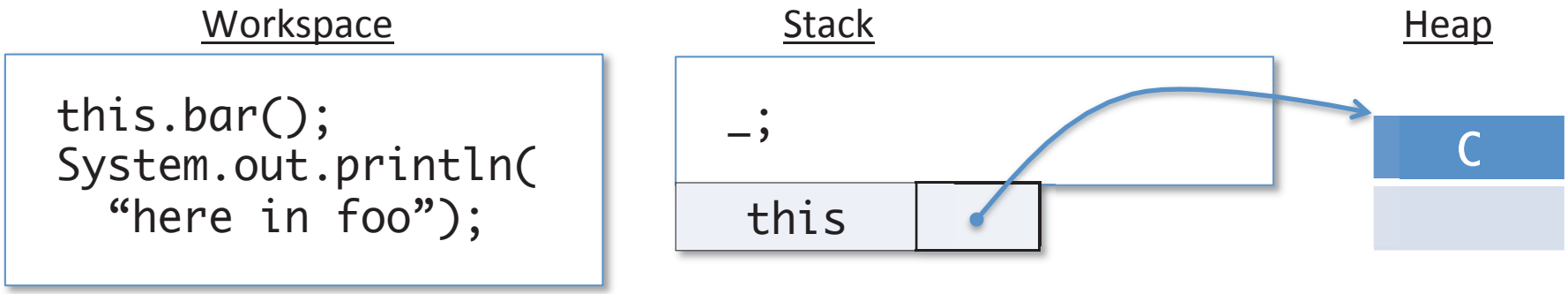


Allocate a new instance of C in the heap. (Skipping details of trivial constructor for C.)

Abstract Stack Machine

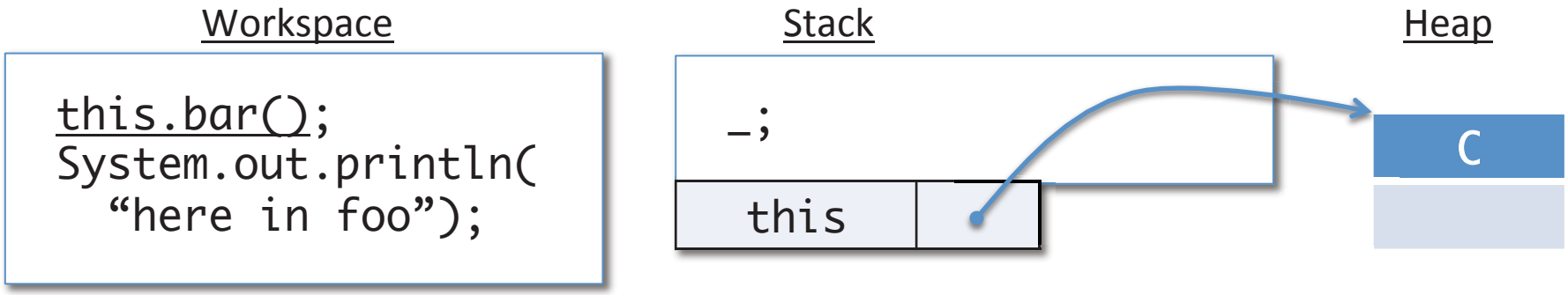


Abstract Stack Machine

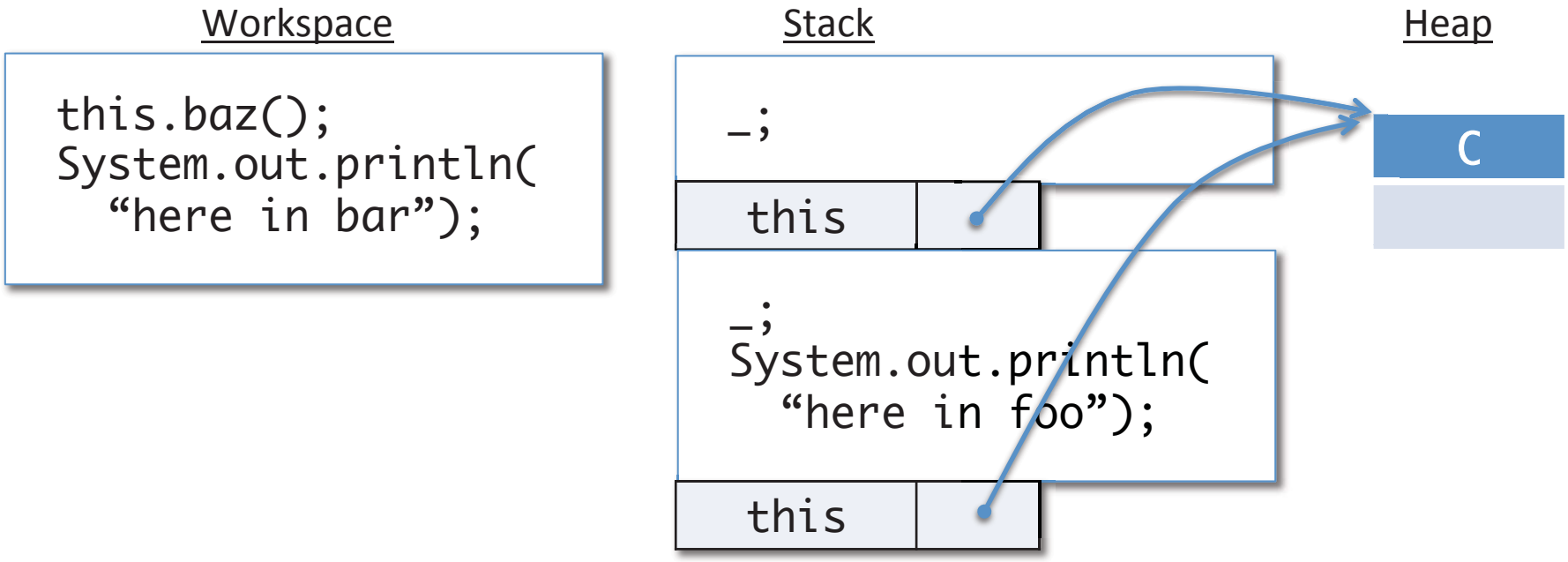


Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack. Use the dynamic class to lookup the method body from the class table.

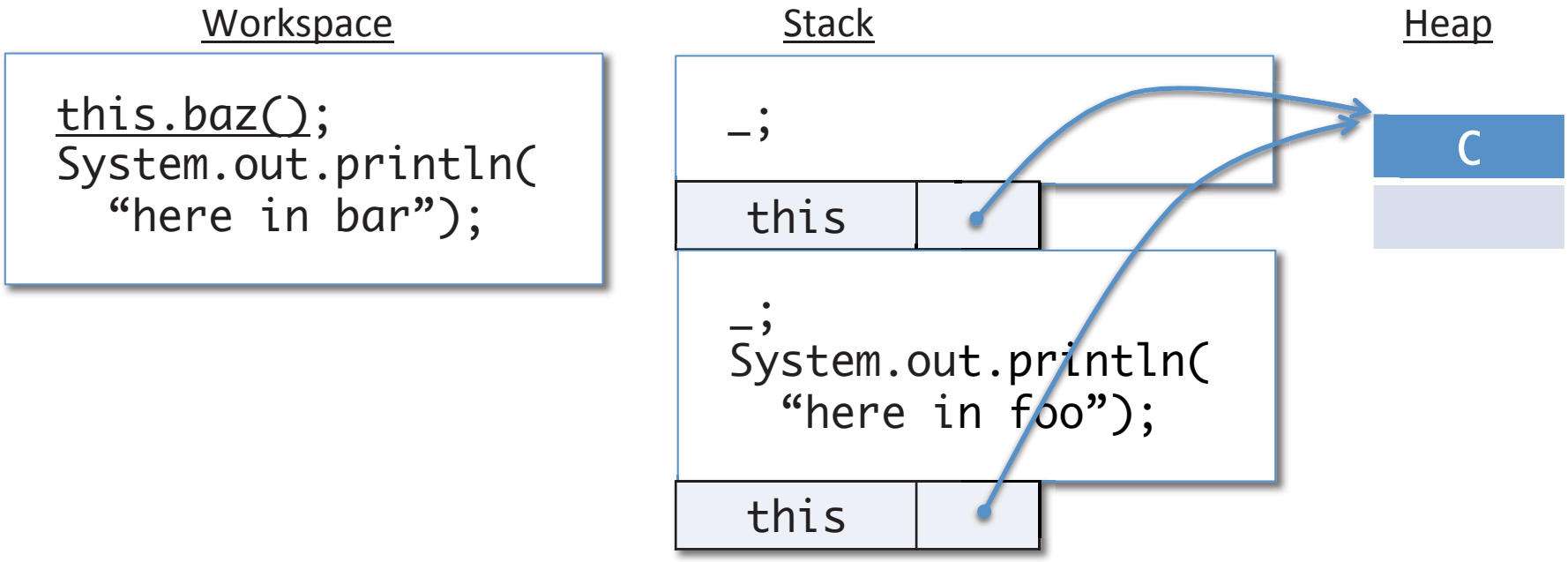
Abstract Stack Machine



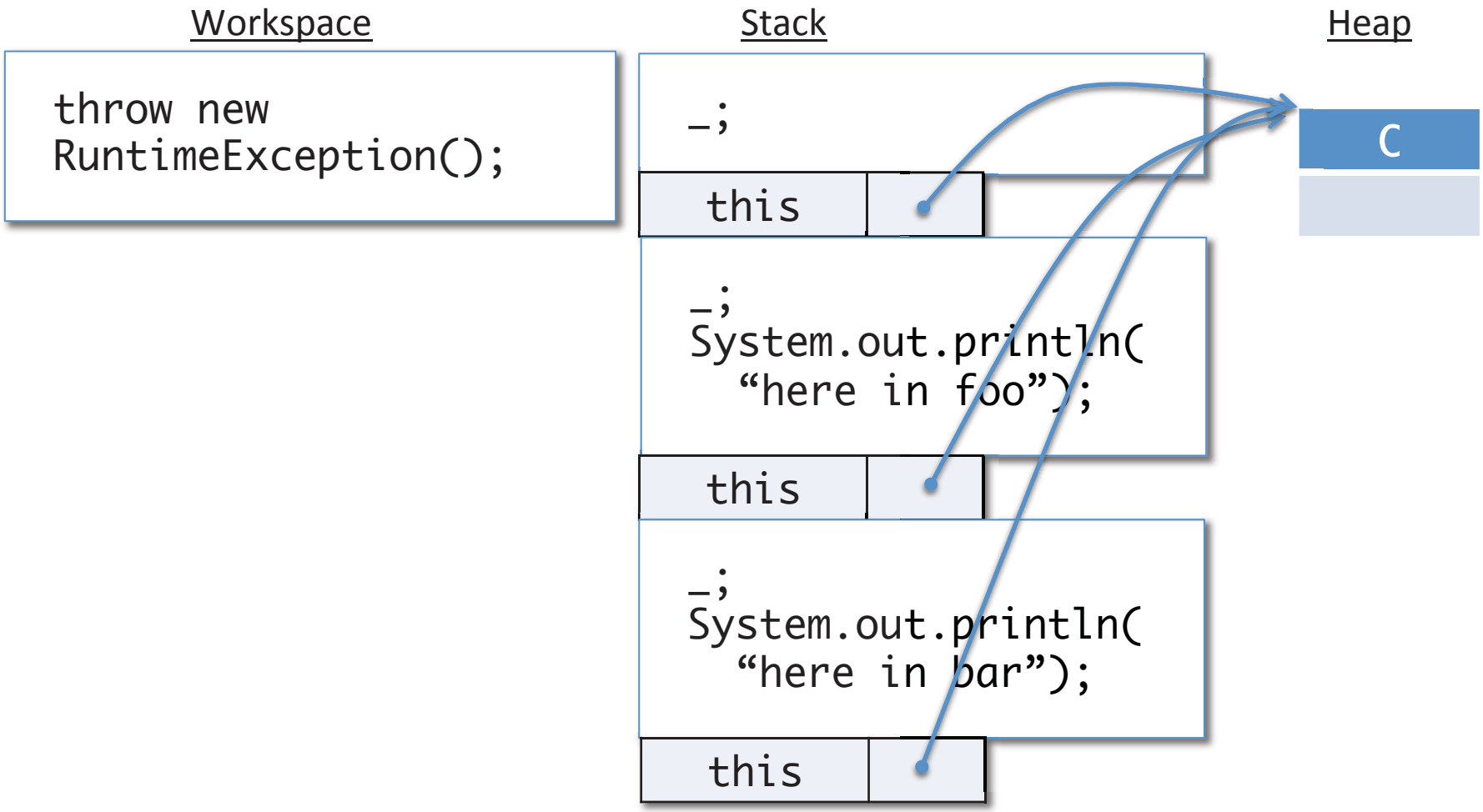
Abstract Stack Machine



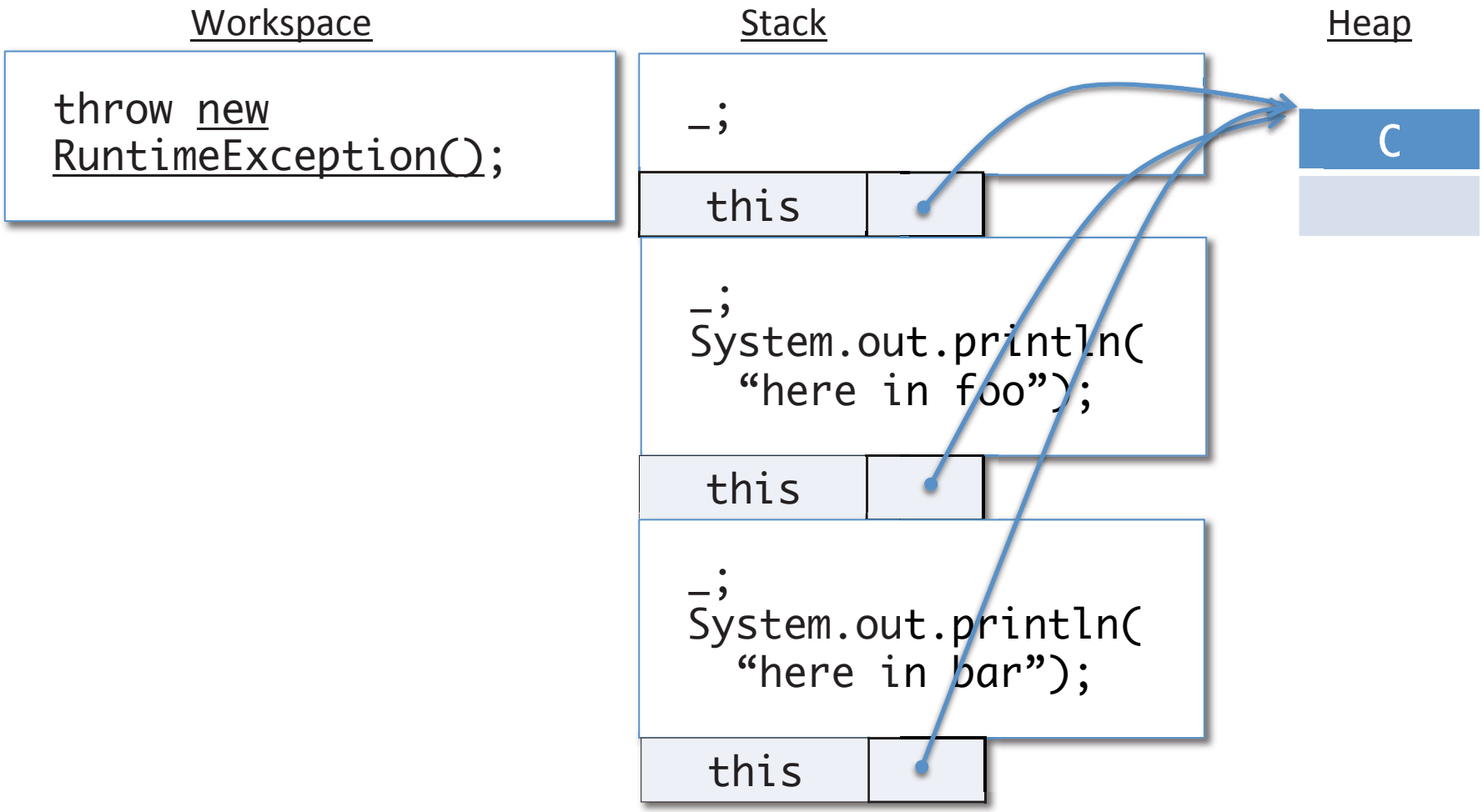
Abstract Stack Machine



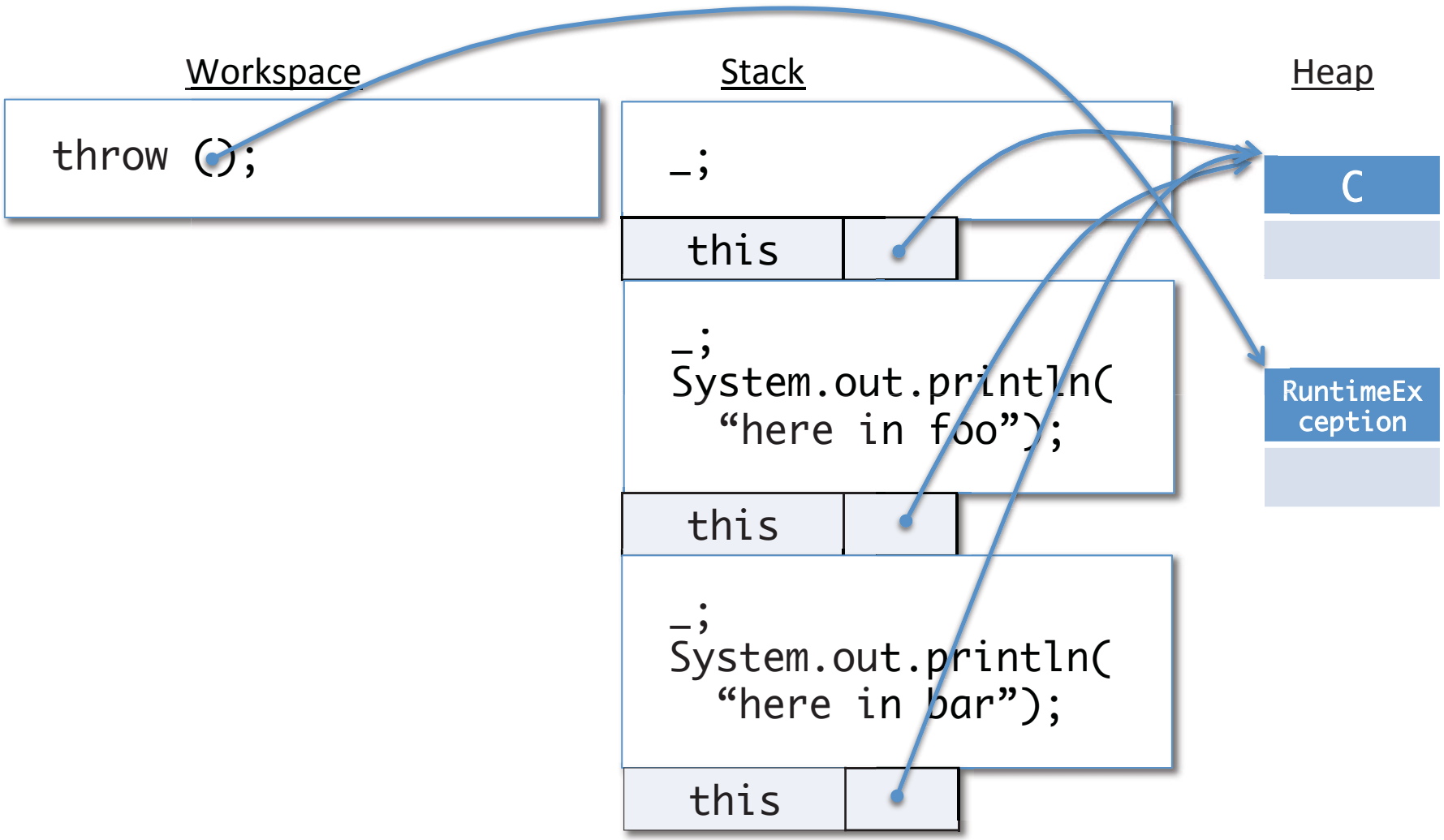
Abstract Stack Machine



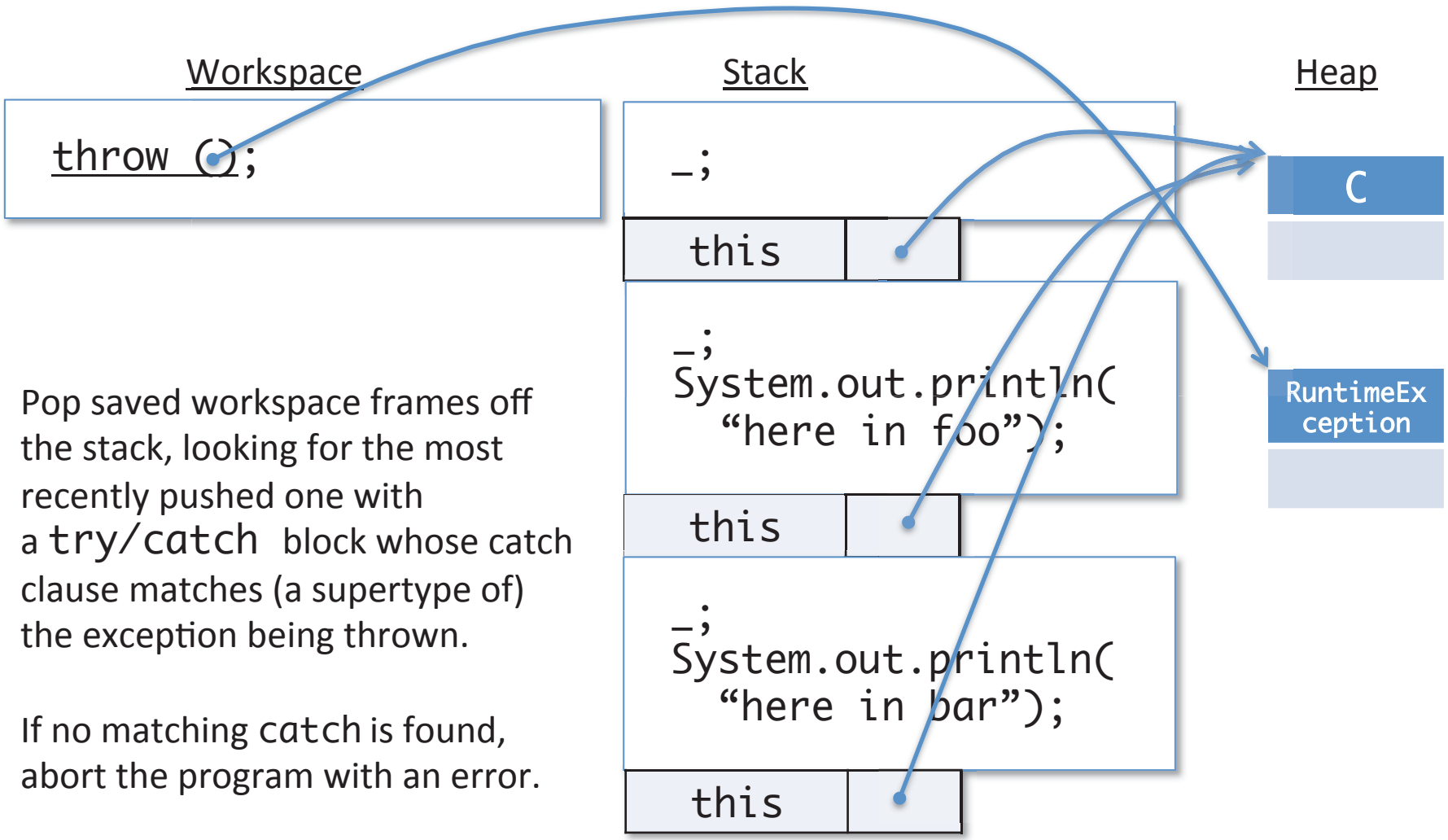
Abstract Stack Machine



Abstract Stack Machine



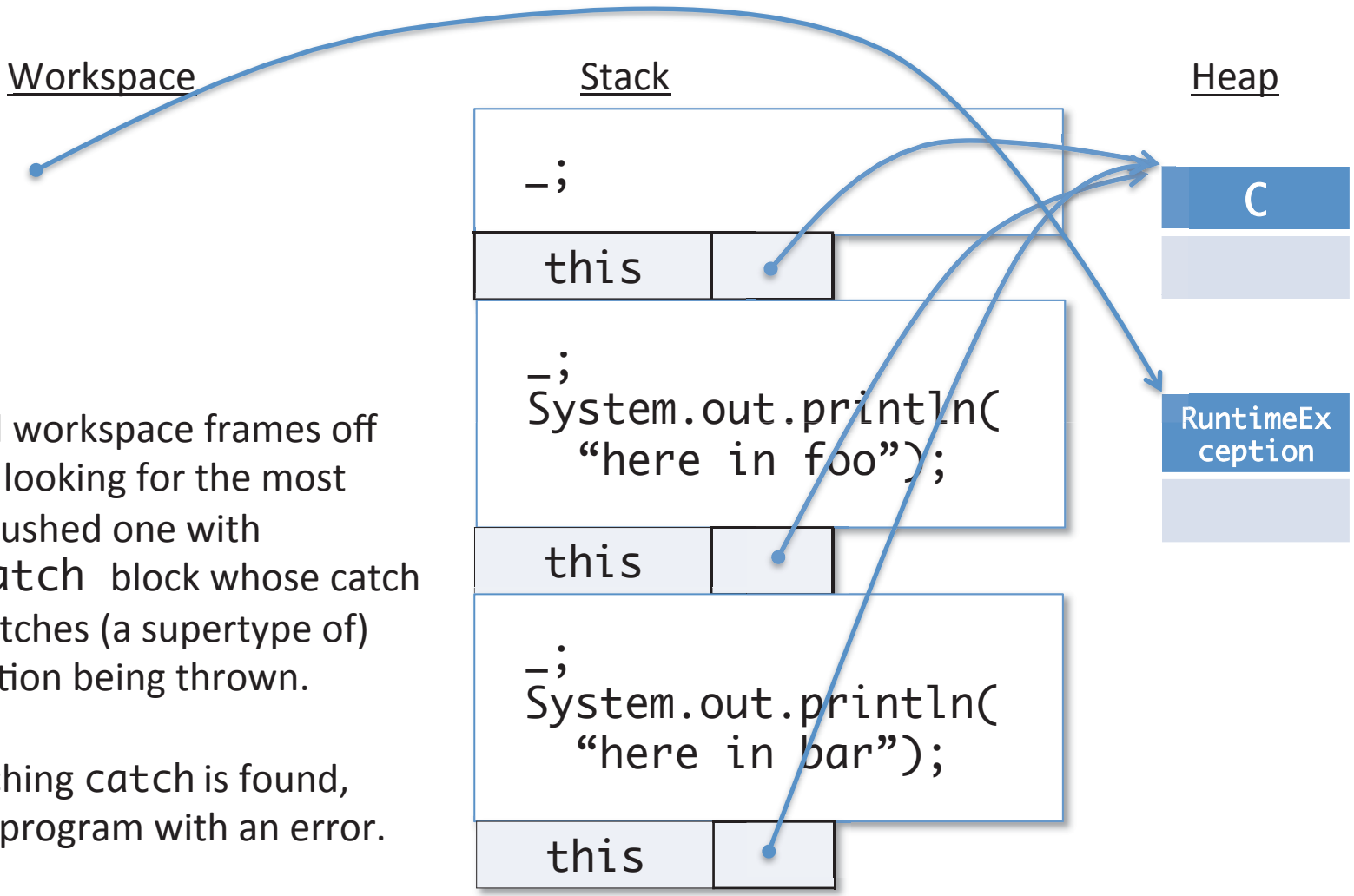
Abstract Stack Machine



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.

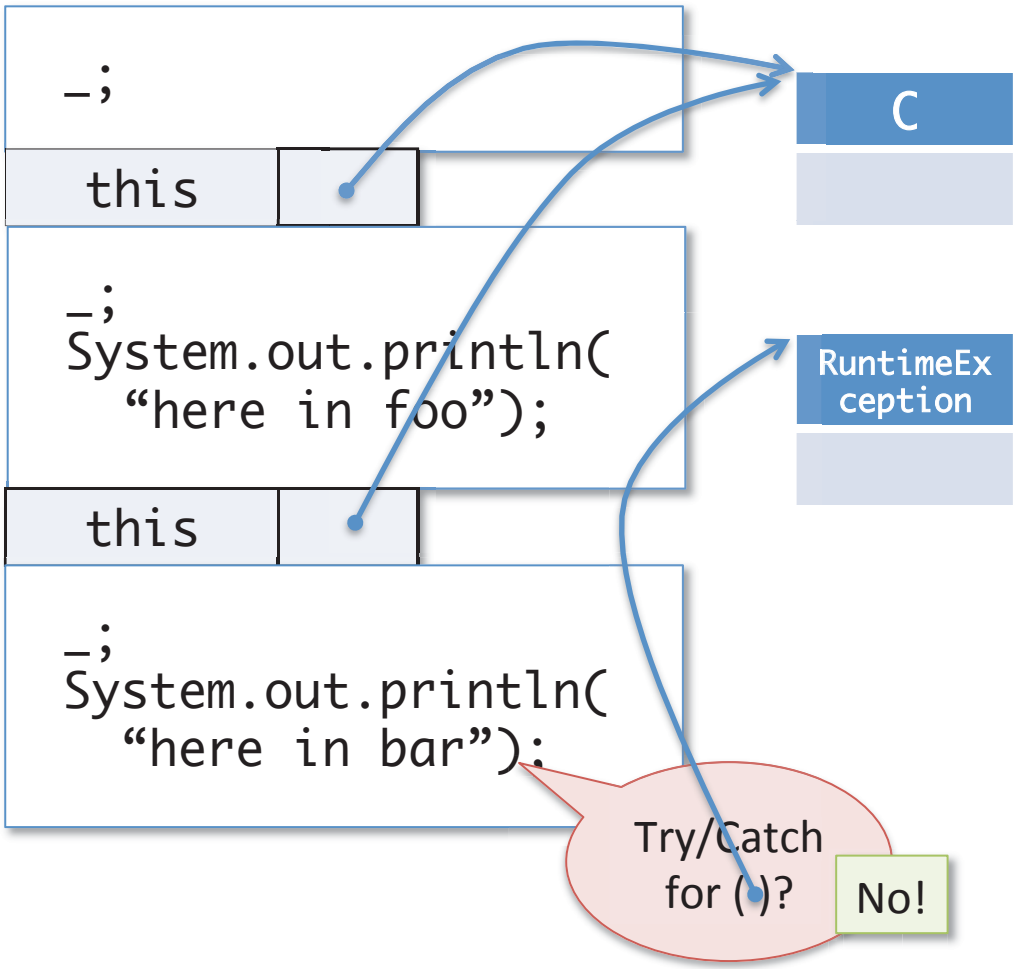
If no matching `catch` is found, abort the program with an error.

Abstract Stack Machine

Workspace

Stack

Heap



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

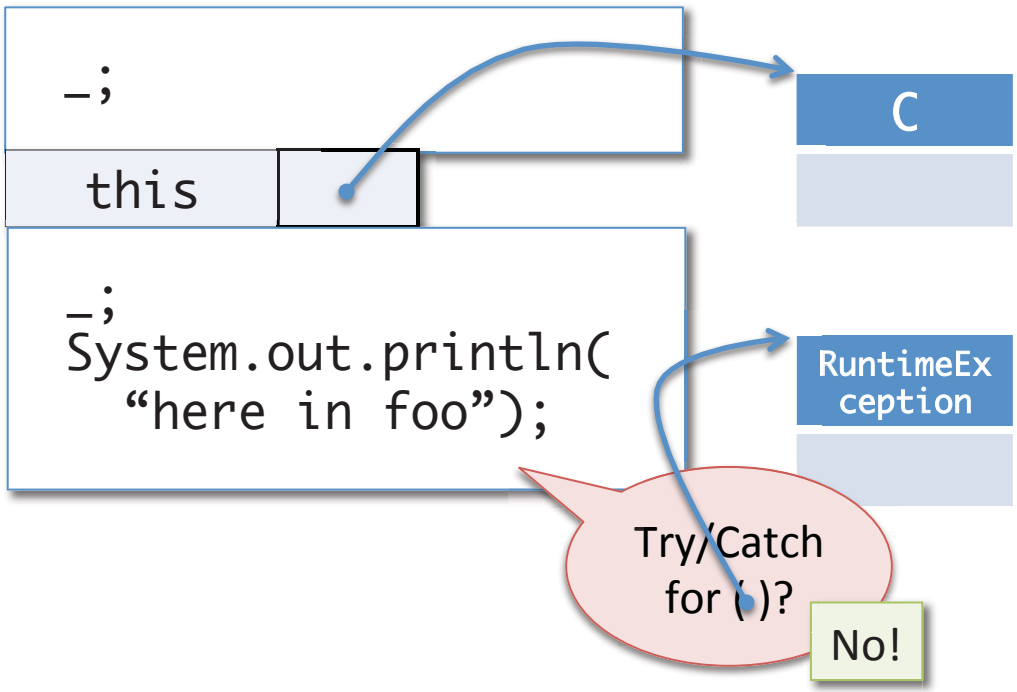
If no matching `catch` is found, abort the program with an error.

Abstract Stack Machine

Workspace

Stack

Heap

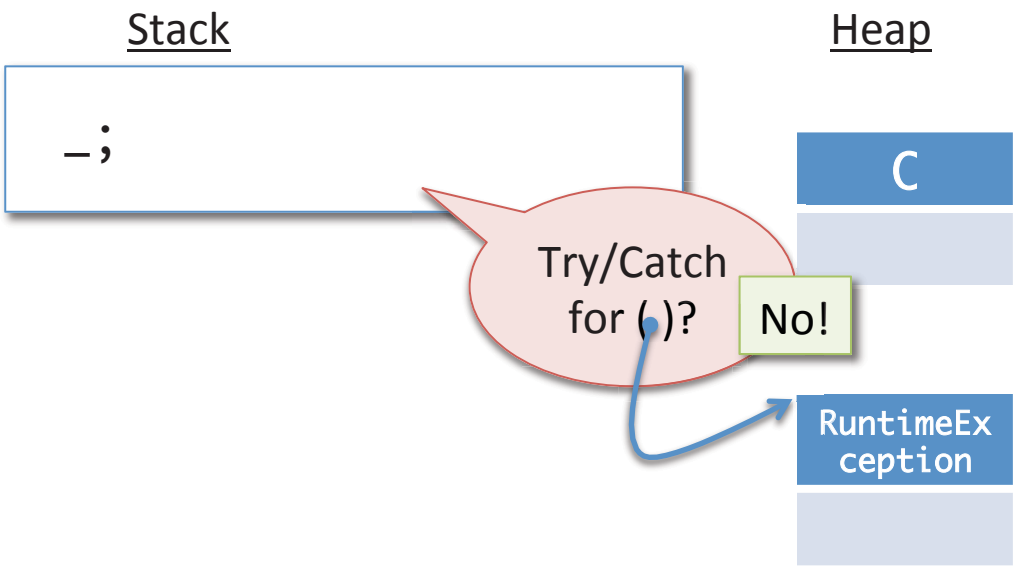


Discard the current workspace.

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

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

Abstract Stack Machine



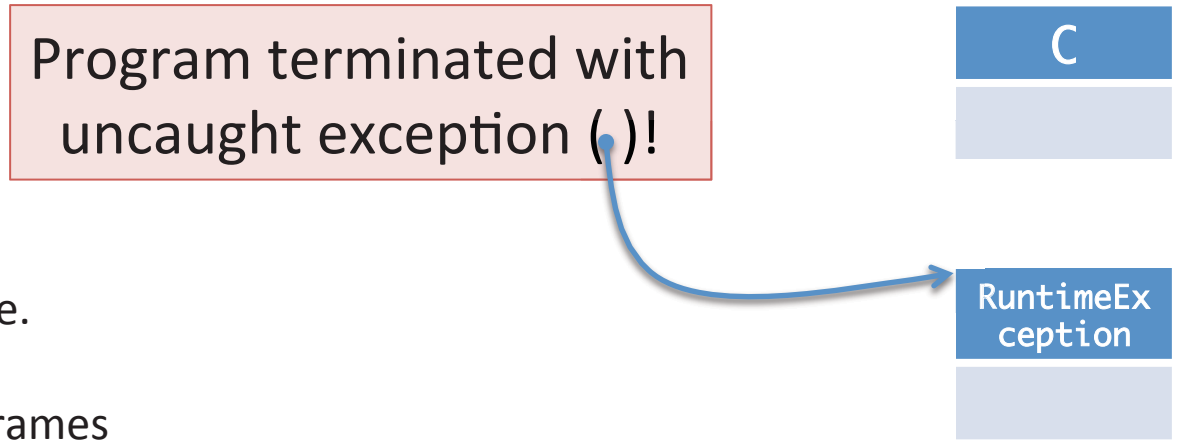
Discard the current workspace.

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

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

Abstract Stack Machine

Workspace | Stack | Heap



Discard the current workspace.

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

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

Catching the Exception

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

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

Abstract Stack Machine

Workspace

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

Stack

Heap

Abstract Stack Machine

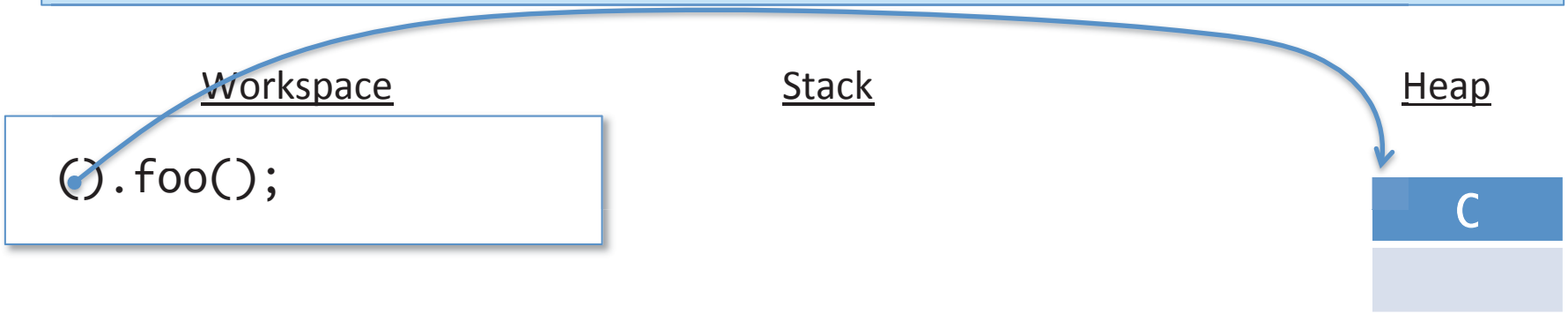
Workspace

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

Stack

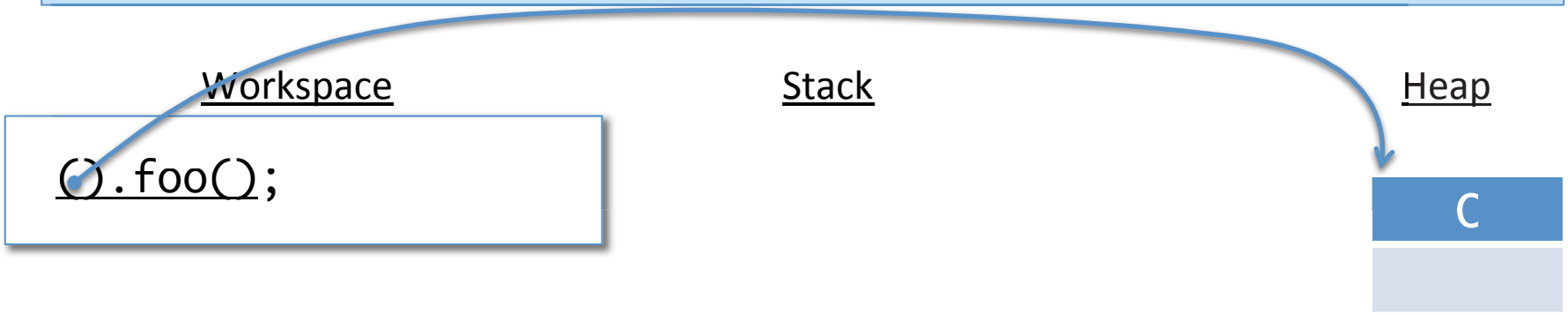
Heap

Abstract Stack Machine

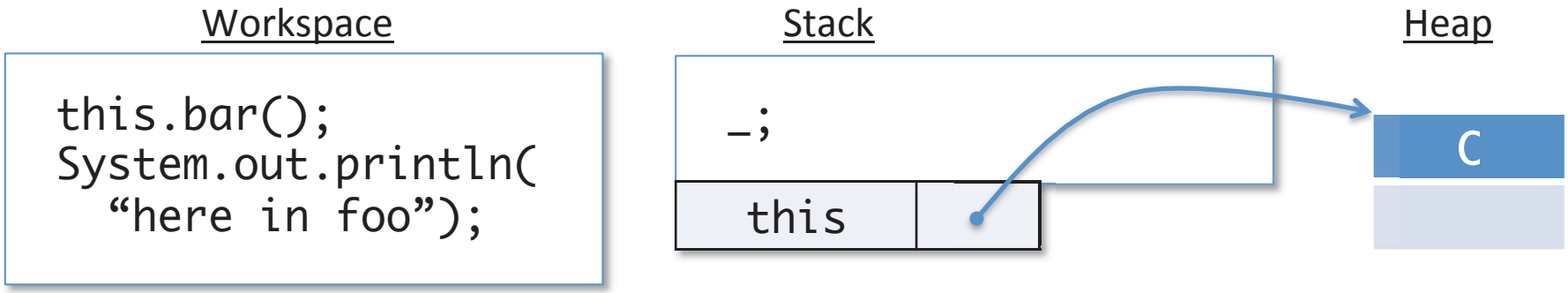


Allocate a new instance of C in the heap.

Abstract Stack Machine

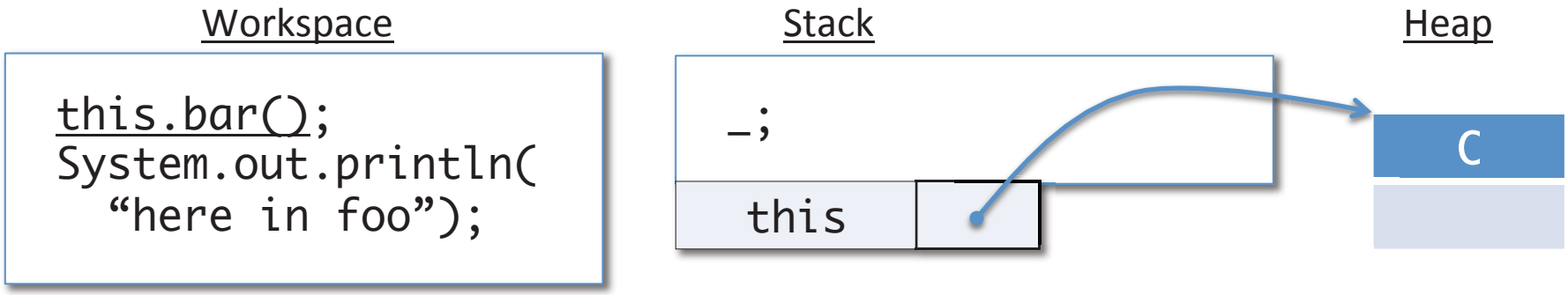


Abstract Stack Machine

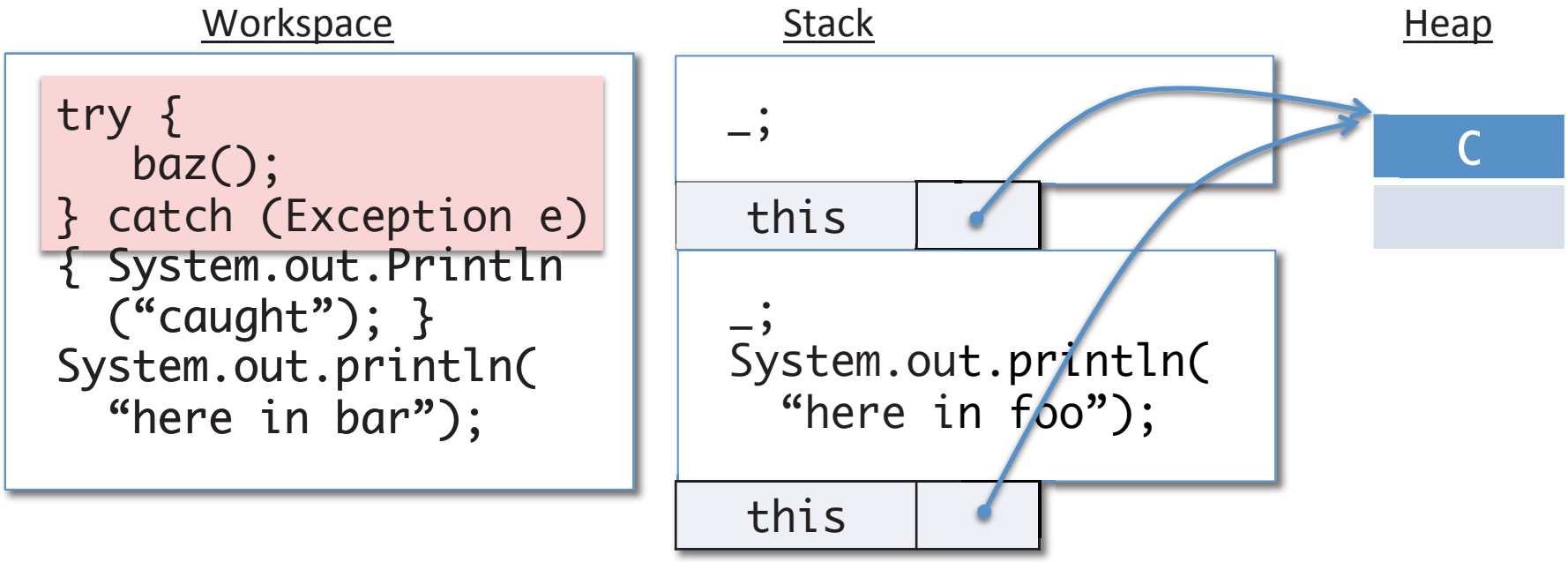


Save a copy of the current workspace in the stack, leaving a "hole", written _, where we return to. Push the `this` pointer, followed by arguments (in this case none) onto the stack.

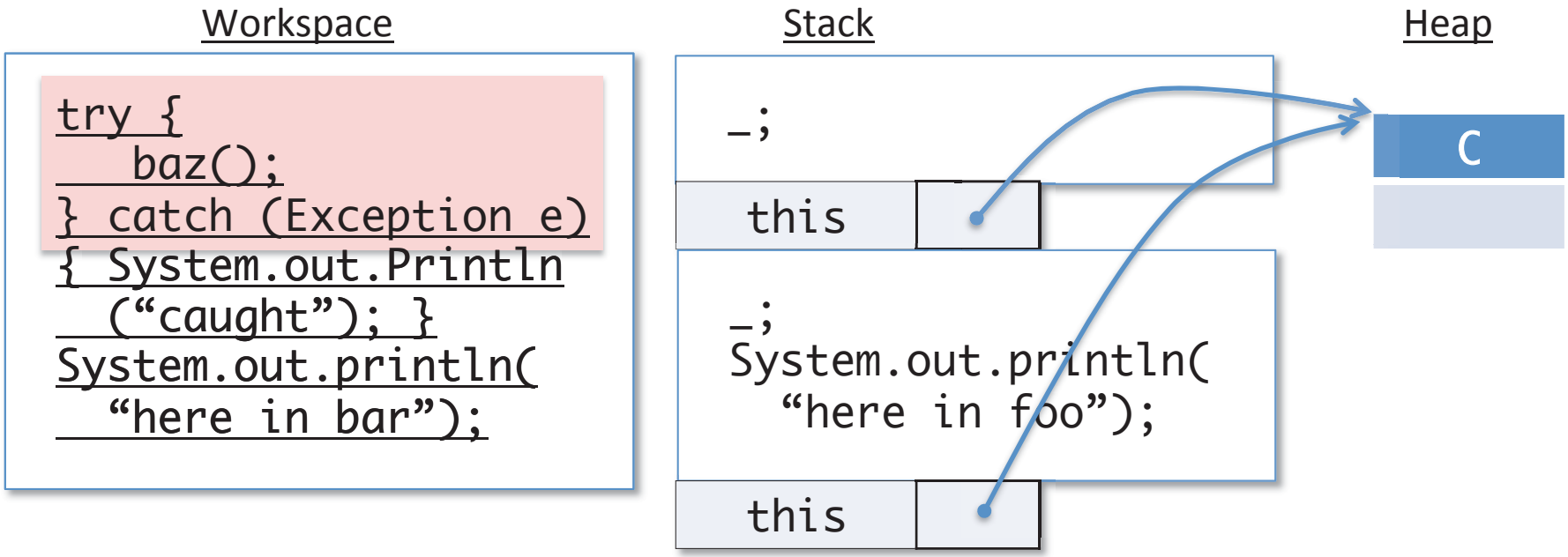
Abstract Stack Machine



Abstract Stack Machine



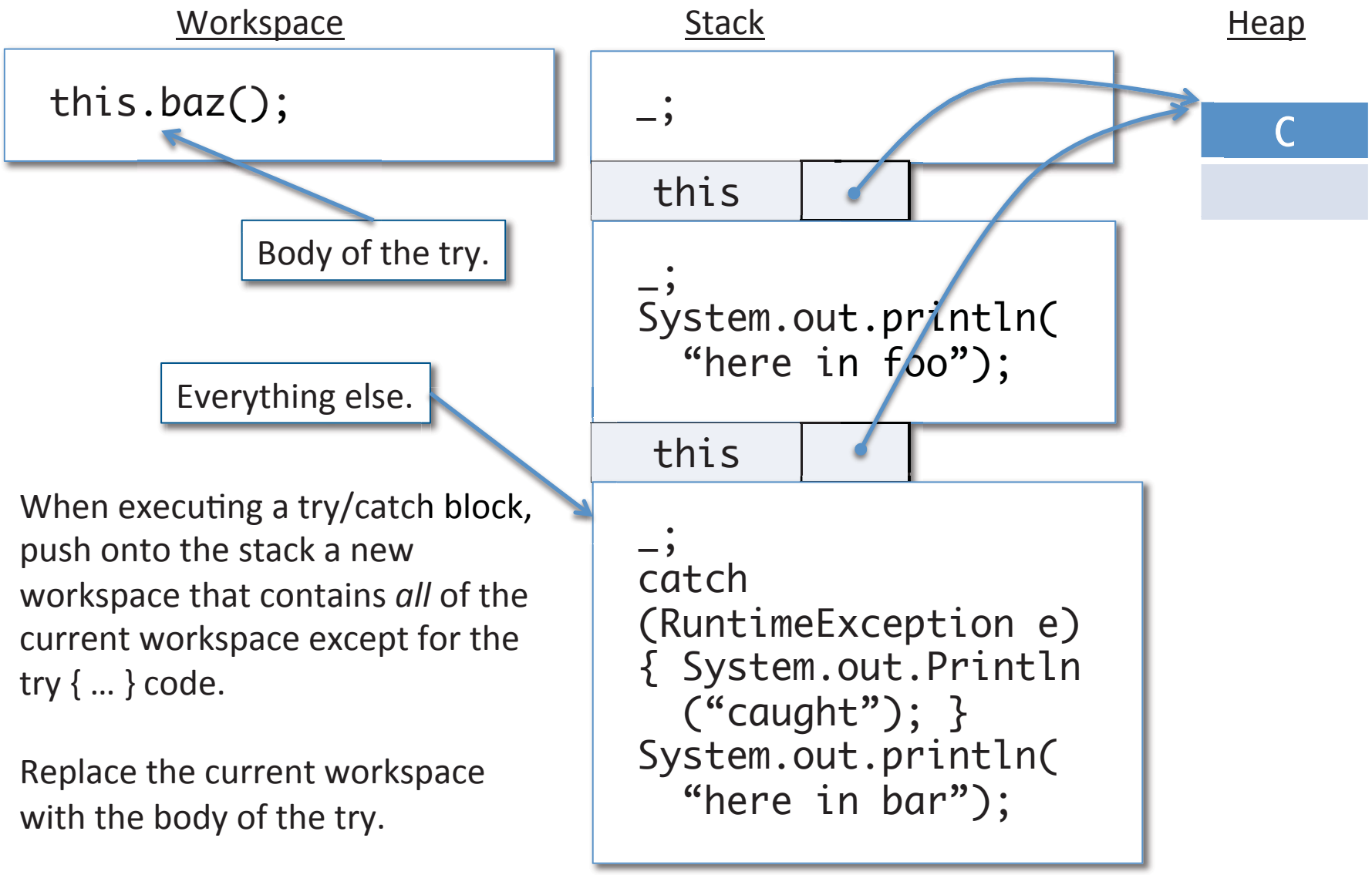
Abstract Stack Machine



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.

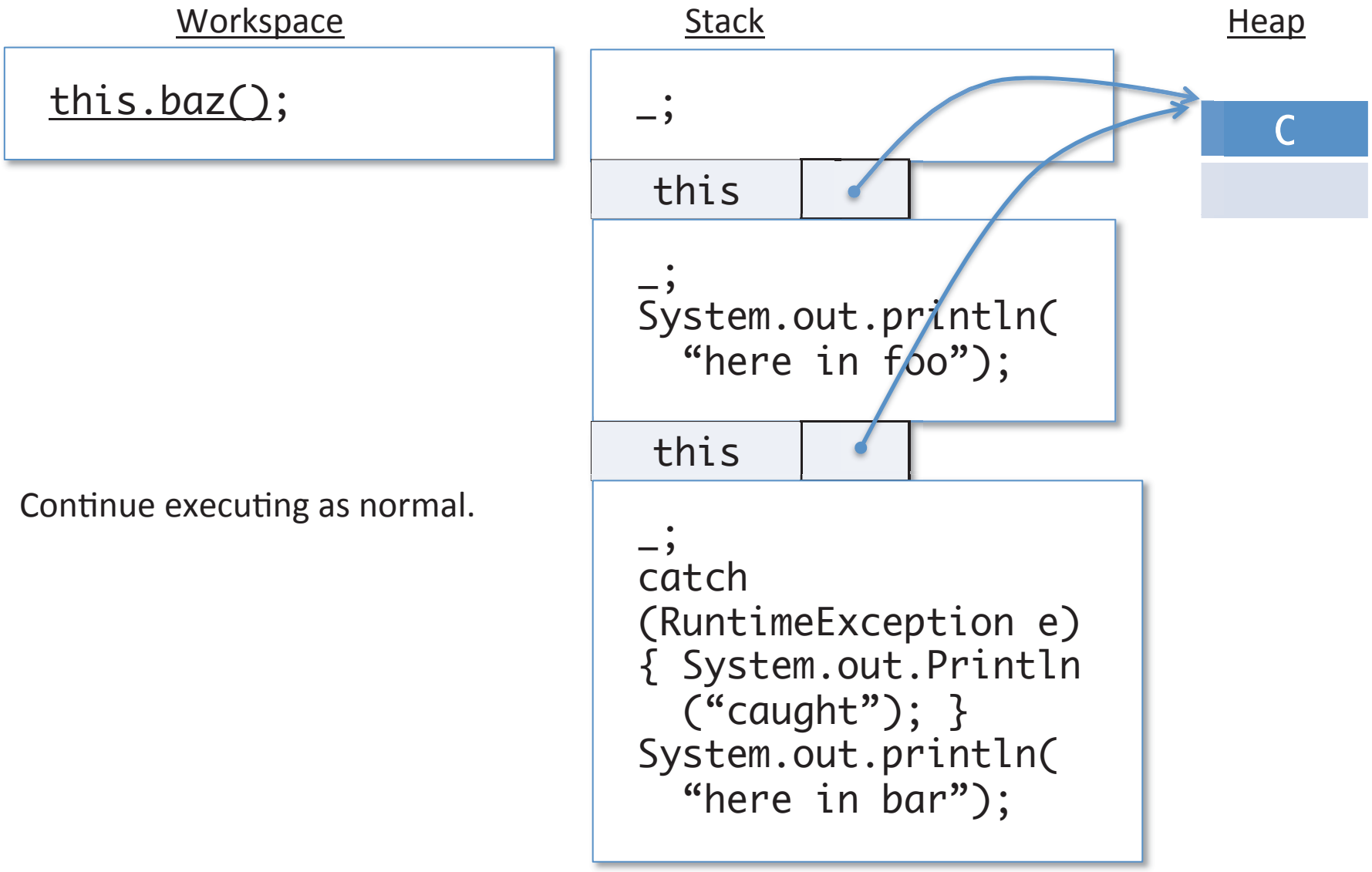
Abstract Stack Machine



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.

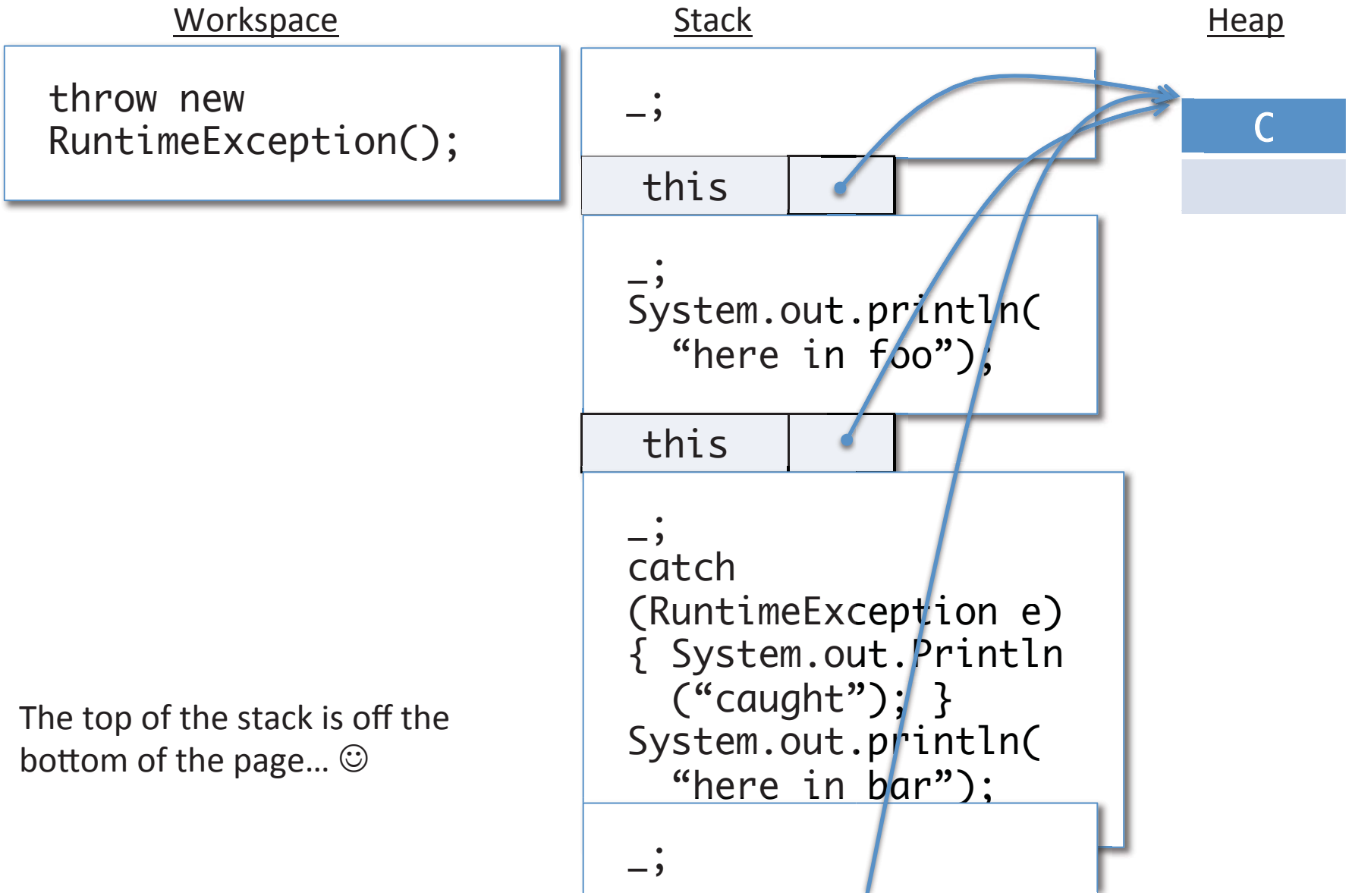
Replace the current workspace with the body of the try.

Abstract Stack Machine



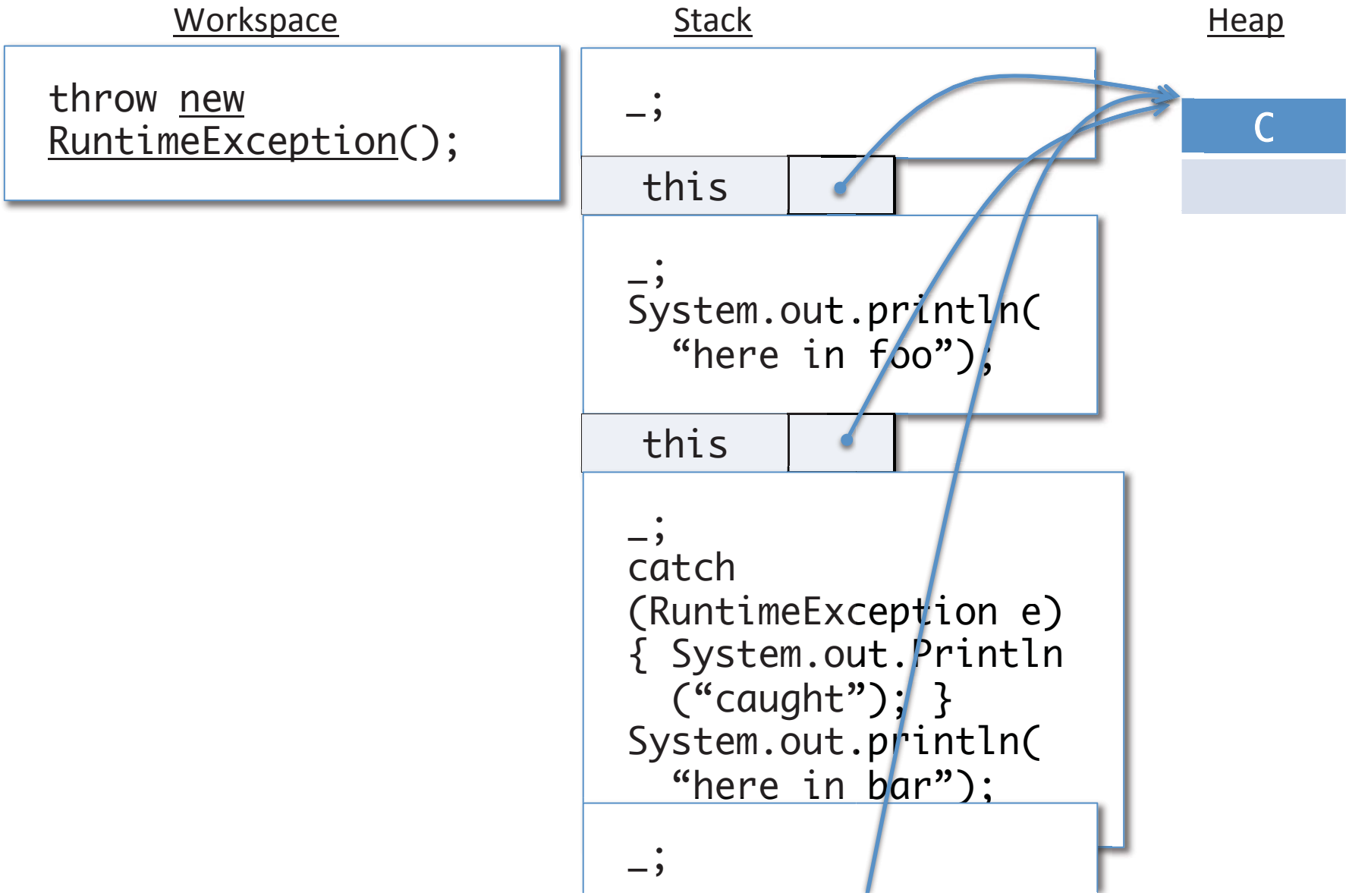
Continue executing as normal.

Abstract Stack Machine

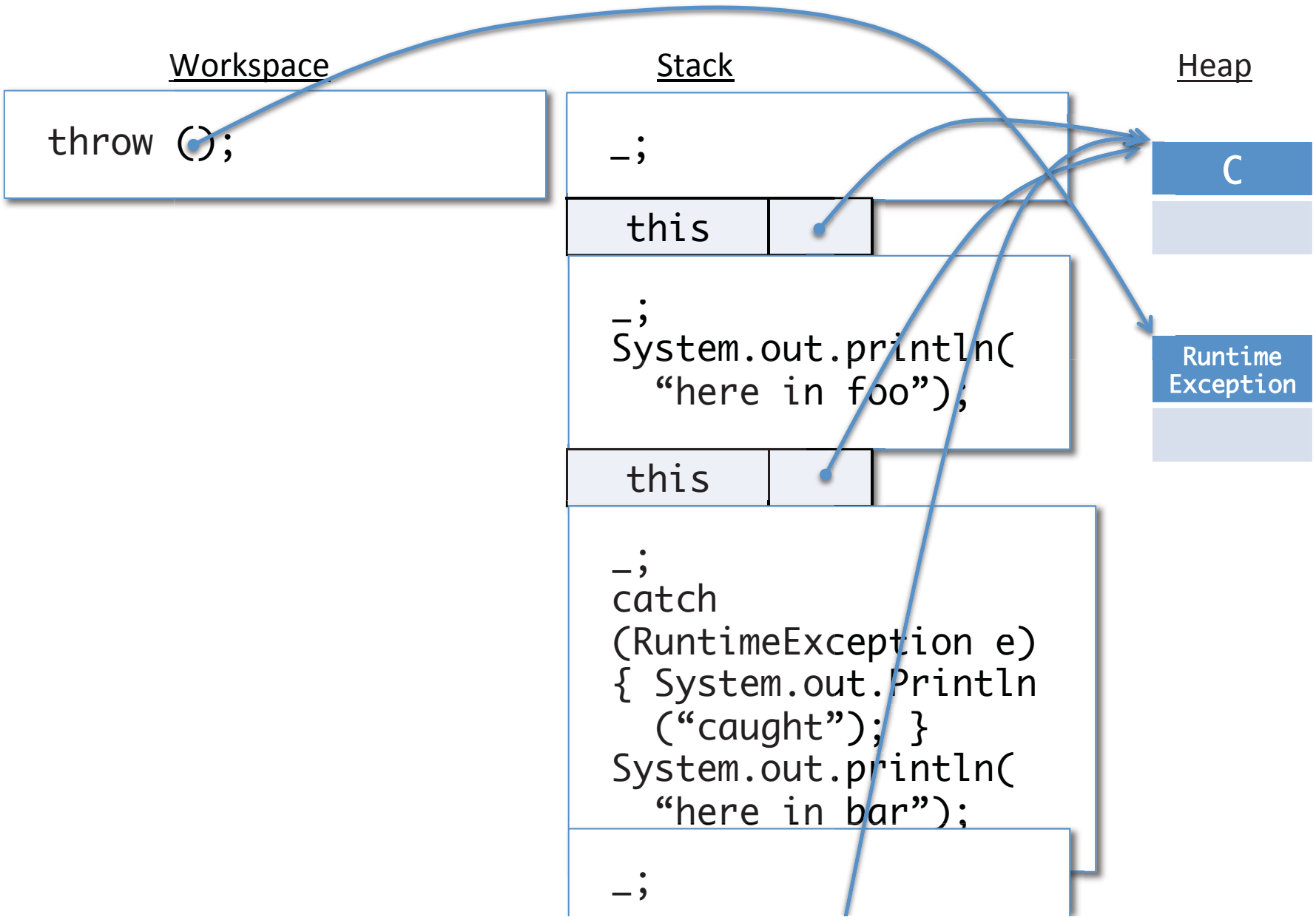


The top of the stack is off the bottom of the page… ☺

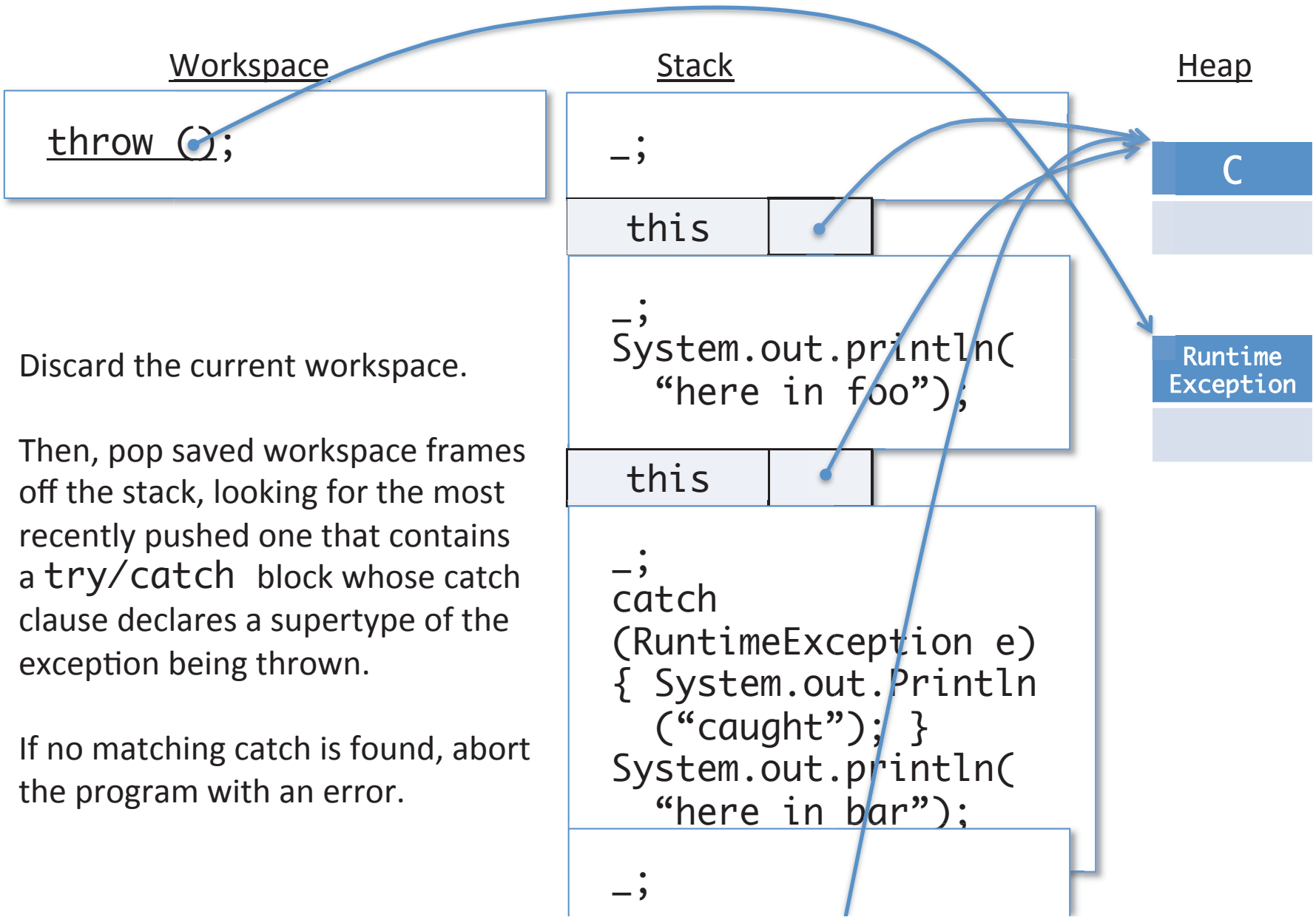
Abstract Stack Machine



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.

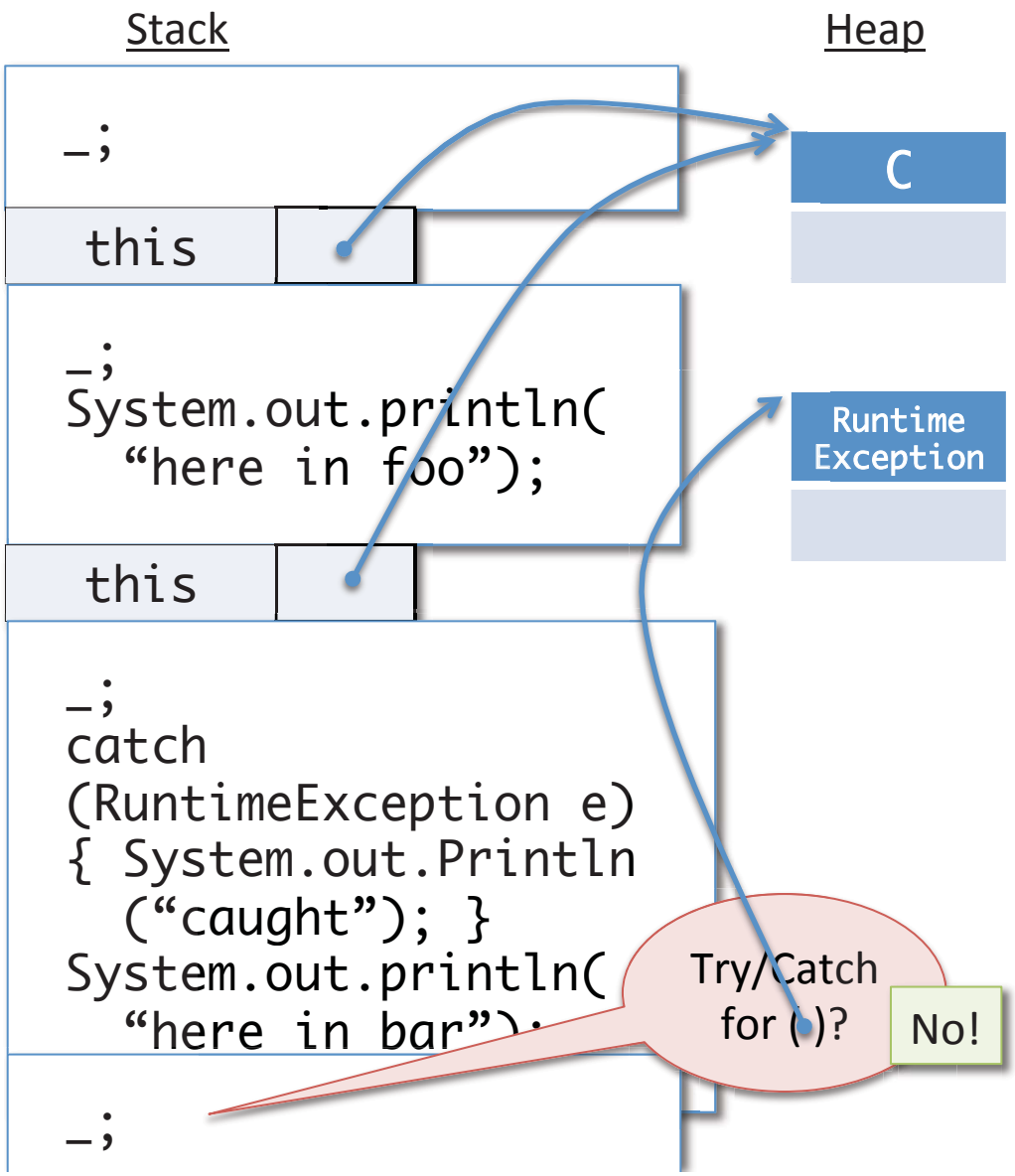
If no matching catch is found, abort the program with an error.

Abstract Stack Machine

Discard the current workspace.

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

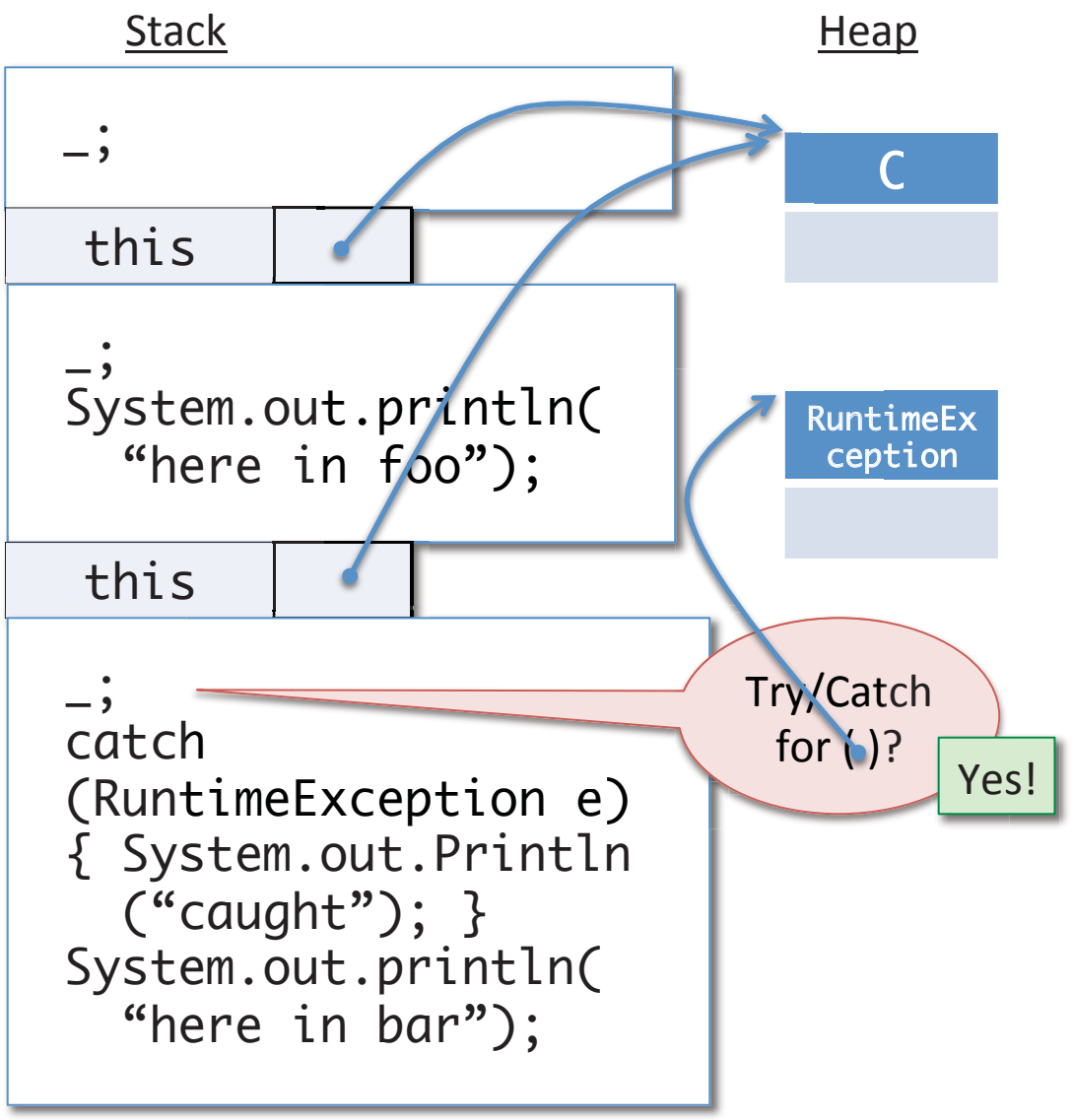
If no matching catch is found, abort the program with an error.



Abstract Stack Machine

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

Continue executing as usual.



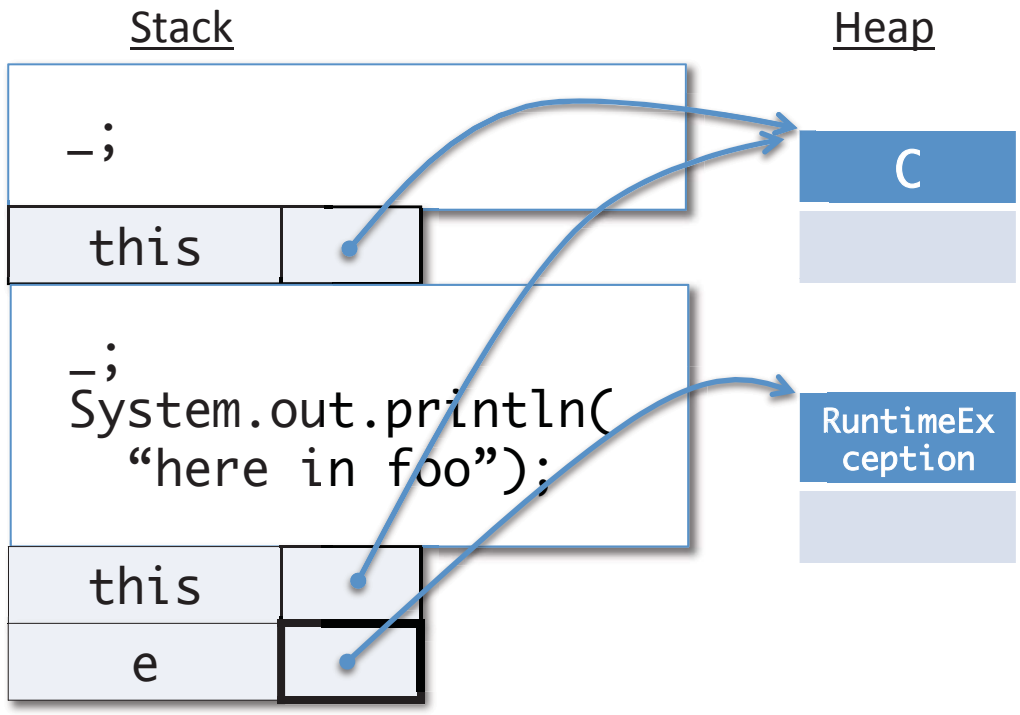
Abstract Stack Machine

Workspace

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

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

Continue executing as usual.

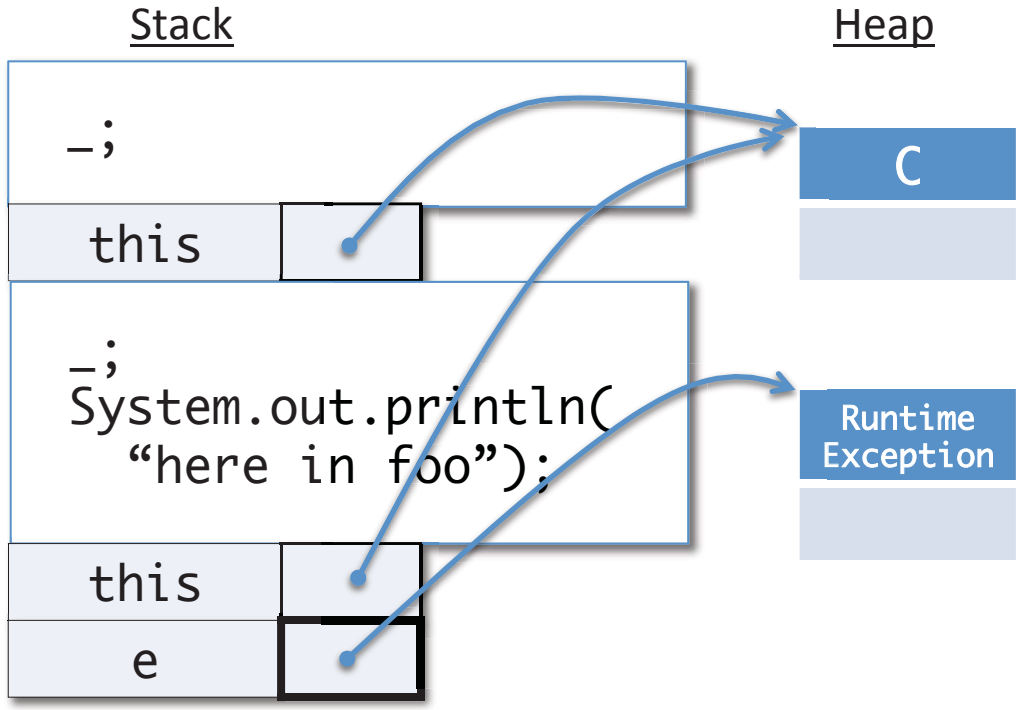


Abstract Stack Machine

Workspace

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

Continue executing as usual.

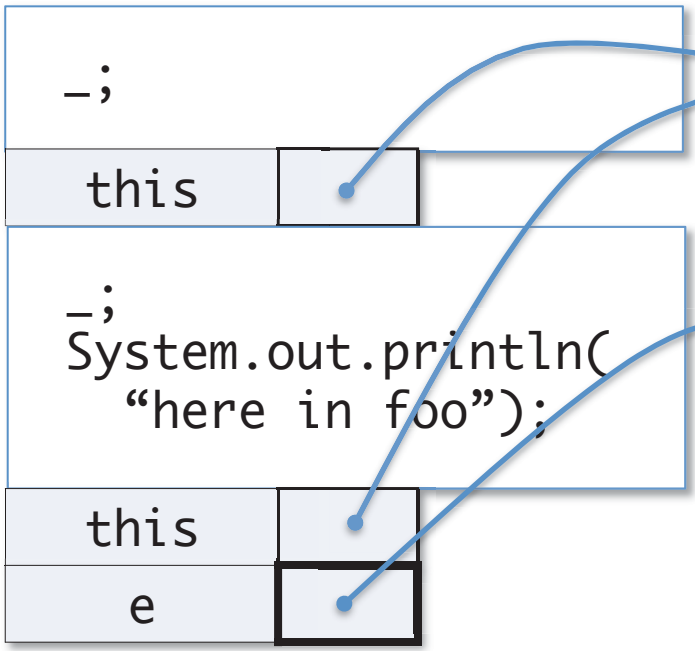


Abstract Stack Machine

Workspace

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

Stack



Heap

Continue executing as usual.

Console

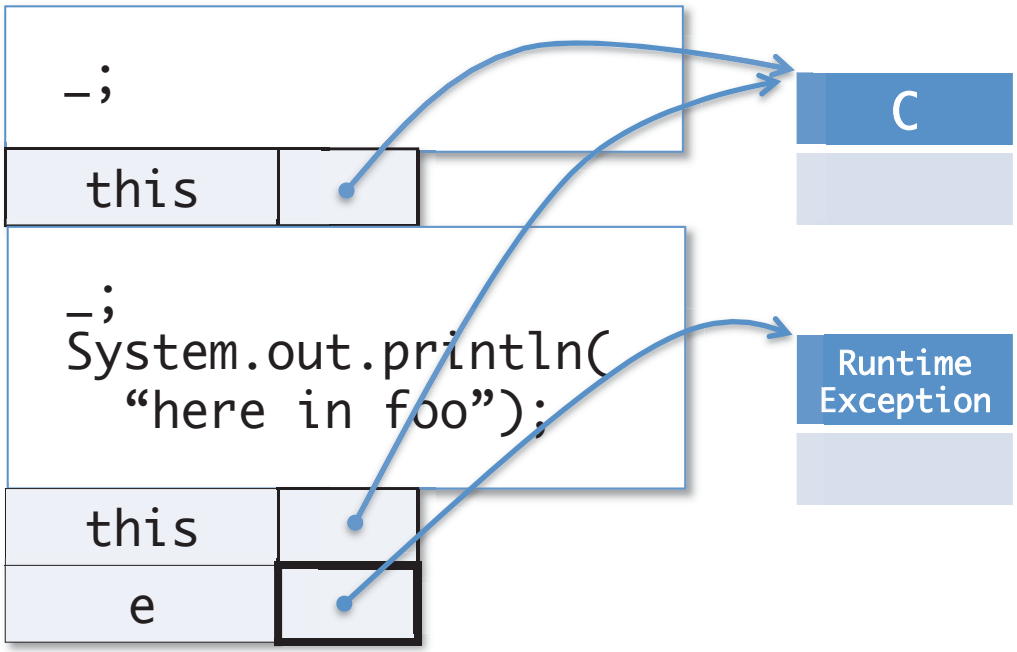
```
caught
```

Abstract Stack Machine

Workspace

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

Stack

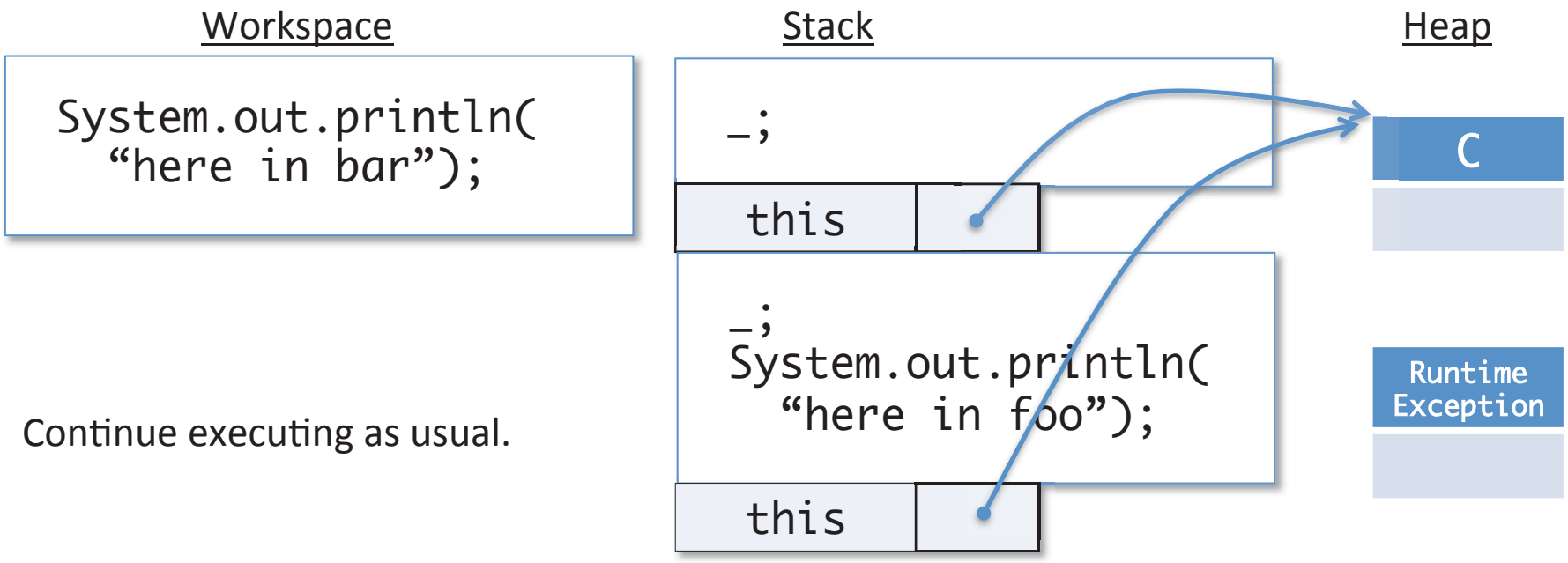


We’re sweeping a few details about lexical scoping of variables under the rug – the scope of e is just the body of the catch, so when that is done, e must be popped from the stack.

Console

```
caught
```

Abstract Stack Machine



Continue executing as usual.

Console

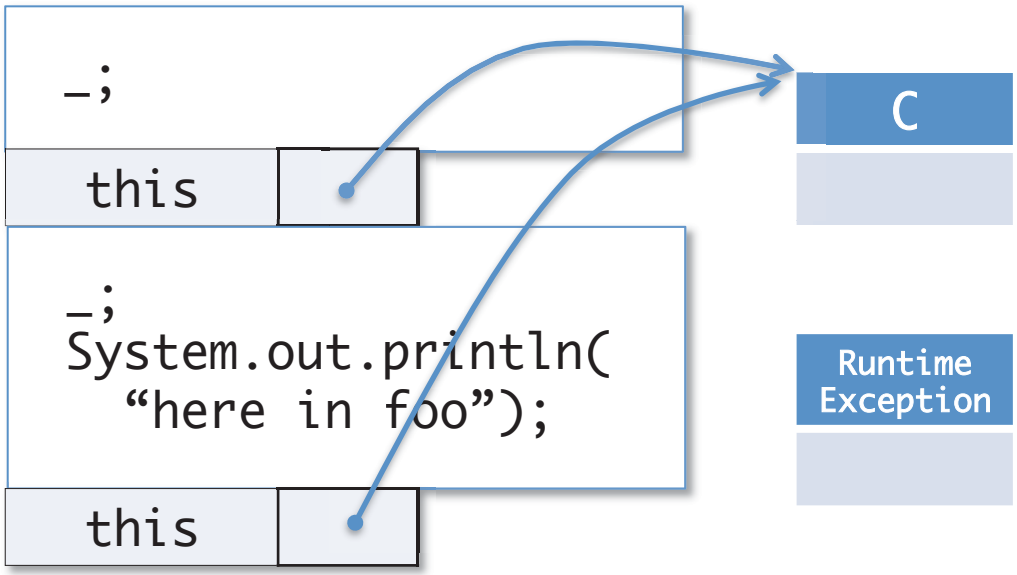
caught

Abstract Stack Machine

Workspace

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

Stack



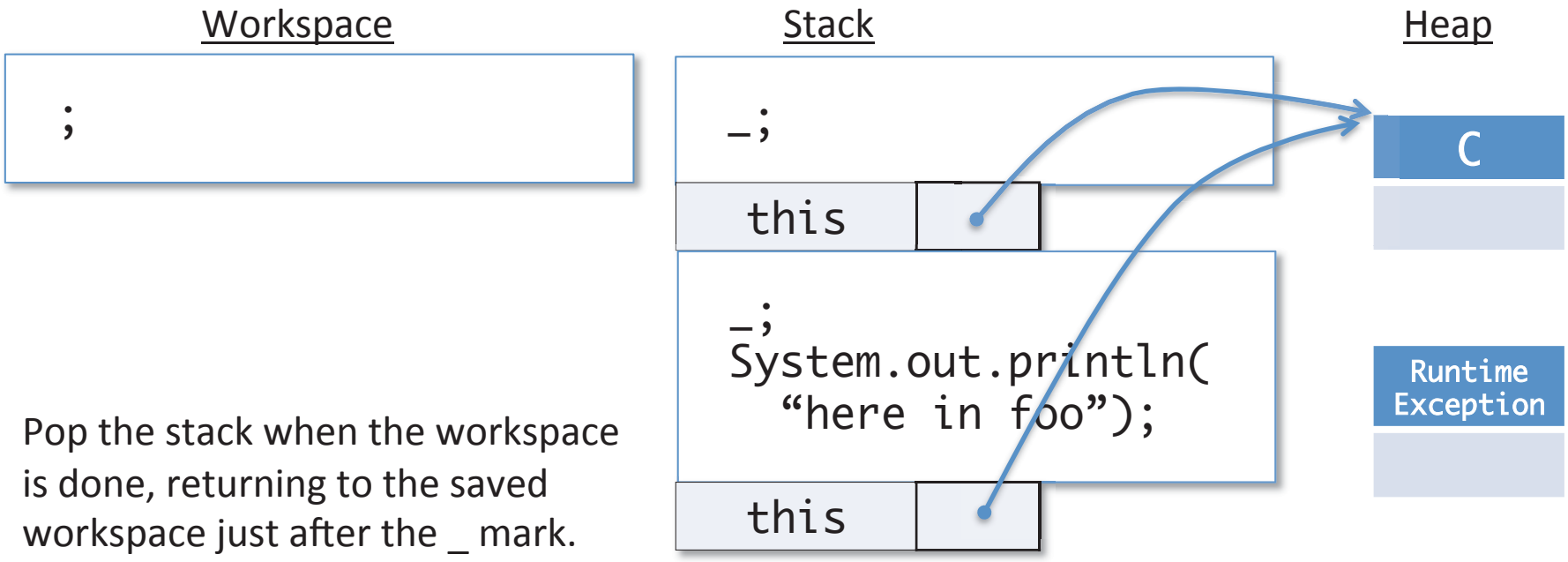
Heap

Continue executing as usual.

Console

```
caught
```

Abstract Stack Machine



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

Console

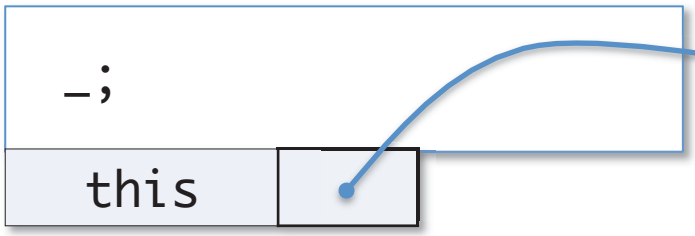
```
caught
here in bar
```

Abstract Stack Machine

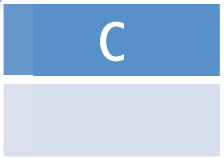
Workspace

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

Stack



Heap

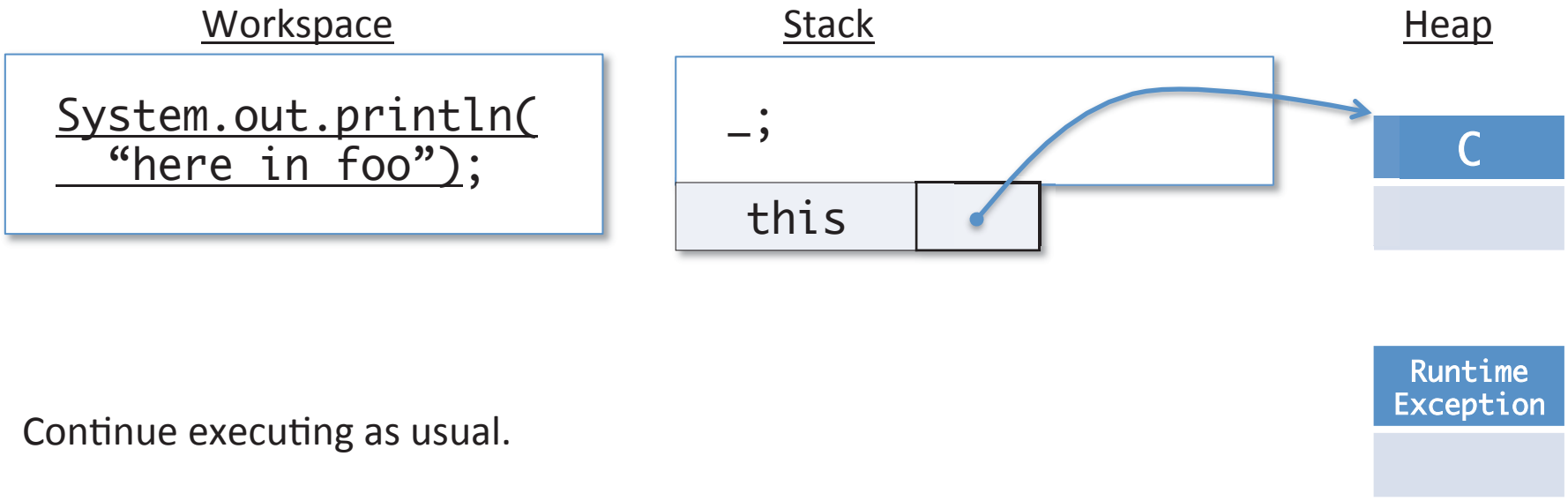


Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine

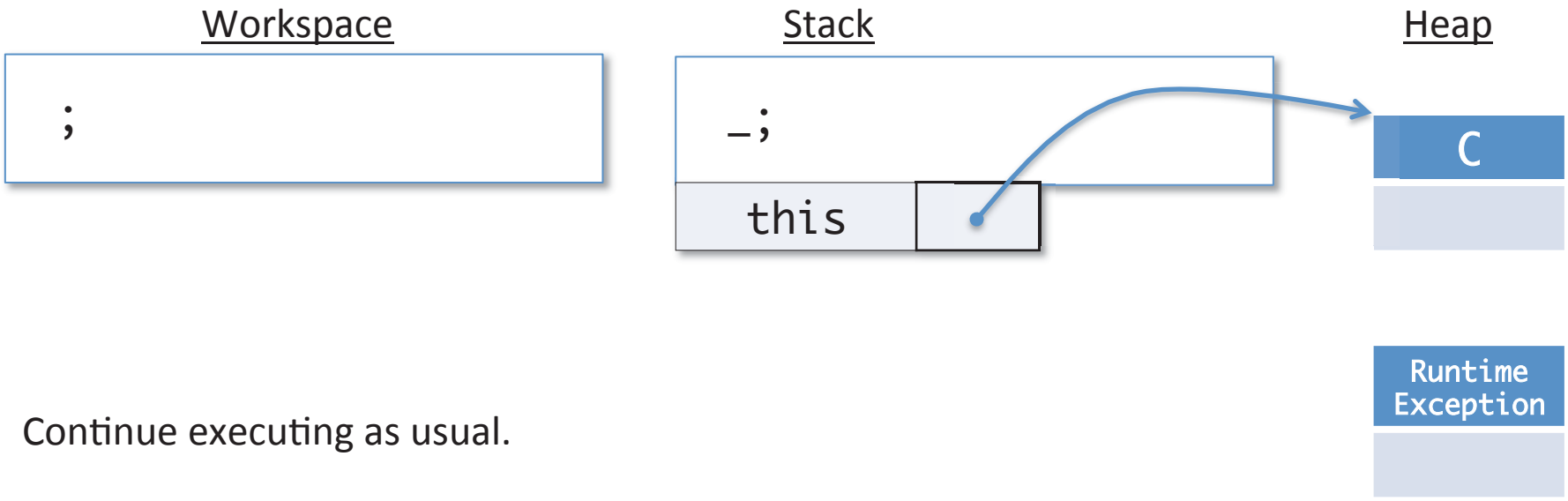


Continue executing as usual.

Console

```
caught
here in bar
```

Abstract Stack Machine



Continue executing as usual.

Console

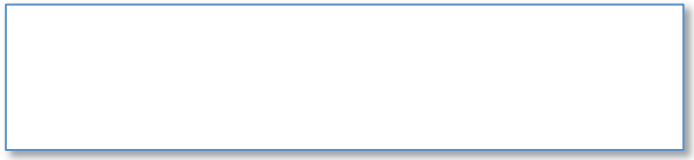
```
caught
here in bar
here in foo
```

Abstract Stack Machine

Workspace

Stack

Heap



Program terminated normally.

Console

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

When No Exception is Thrown

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

Workspace

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

Workspace

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

Catching Exceptions

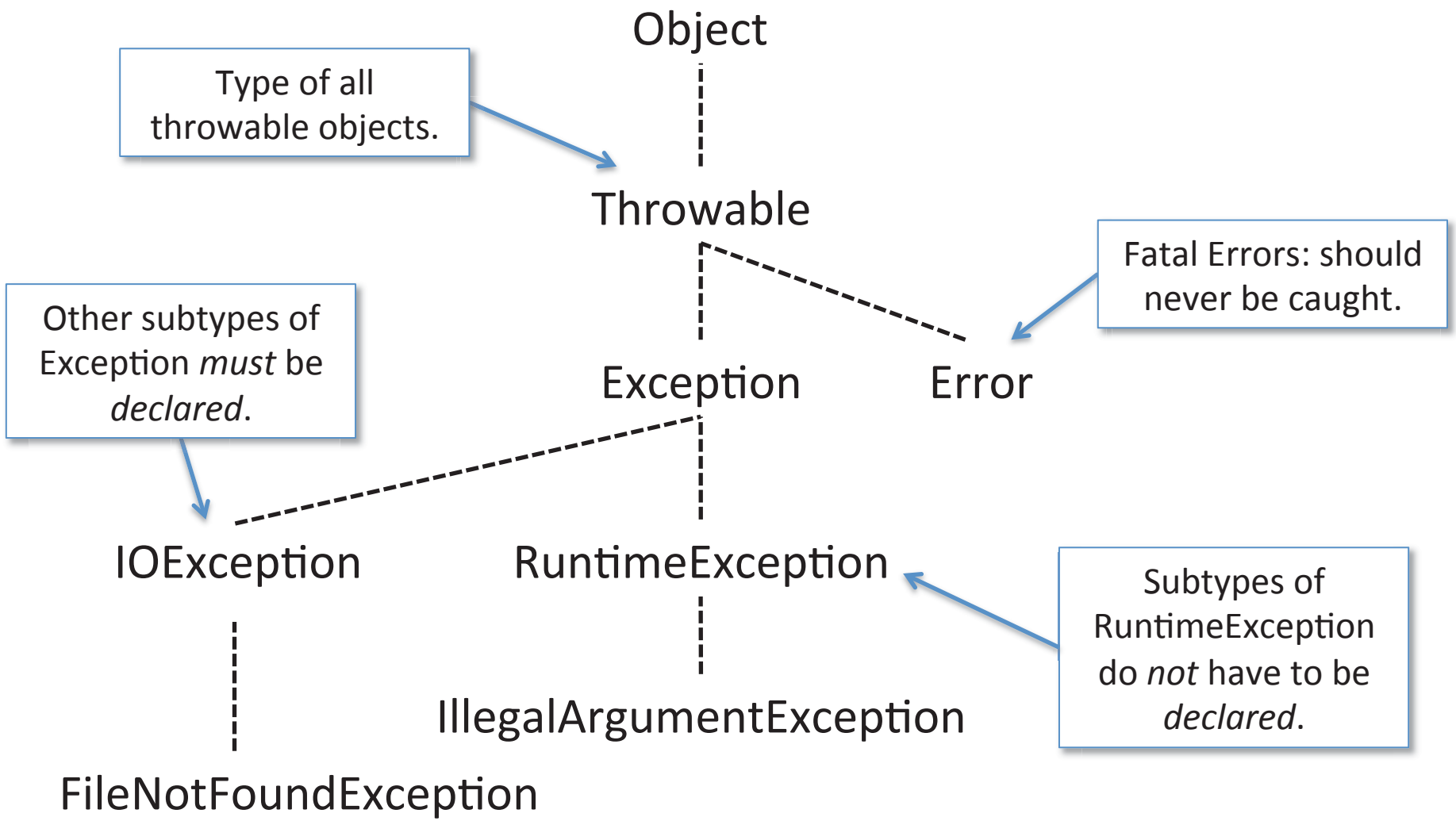
- There can be more than one "catch" clause associated with each "try"
  - Matched in order, according to the *dynamic* class of the exception thrown
  - Helps refine error handling

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

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

Informative Exception Handling

Exception Class Hierarchy



Checked (Declared) Exceptions

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

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

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

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

Unchecked (Undeclared) Exceptions

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

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

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

Checked vs. Unchecked Exceptions

Which methods need a "throws" clause?

Note: IllegalArgumentException is a subtype of RuntimeException.

IOException is not.

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

Answer:
n, q and s should say throws IOException

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

Declared vs. Undeclared?

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

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

Finally

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

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

Using Finally

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

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

Using Finally

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

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

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

Good Style for Exceptions

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

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

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

Good Style for Exceptions

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

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

- Catch exceptions with as much precision as you can

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

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

Programming Languages and Techniques (CIS120)

Lecture 31

November 17, 2017

I/O & Histogram Demo

Chapters 28

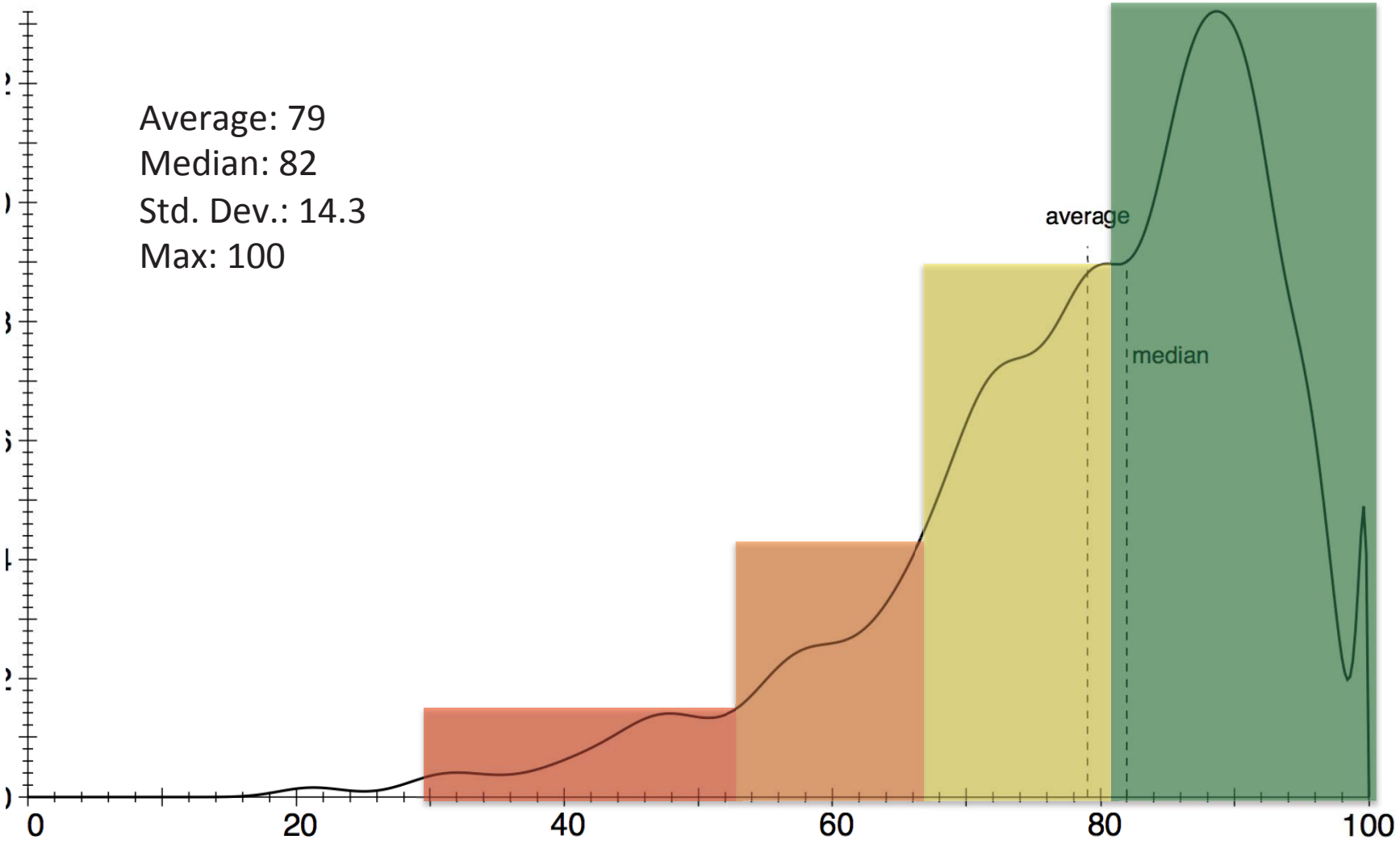
Announcements

- HW8: SpellChecker
  - Available on the website and Codio
  - Due next Tuesday, November 21st

- Next Week is *Thanksgiving Break*:
  - No lab sections
  - Wednesday, November 22nd: Bonus Lecture offered ONLY at the 11:00-noon timeslot (everyone welcome to attend)
  - Topic: Code *is* Data (fun, but not relevant to HW or exam materials)

Midterm 2 Results

Average: 79
Median: 82
Std. Dev.: 14.3
Max: 100



Good Style for Exceptions

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

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

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

Good Style for Exceptions

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

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

- Catch exceptions with as much precision as you can

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

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

java.io

Poll

How many of these these classes have you used before CIS 120 (all part of the Java standard library)?

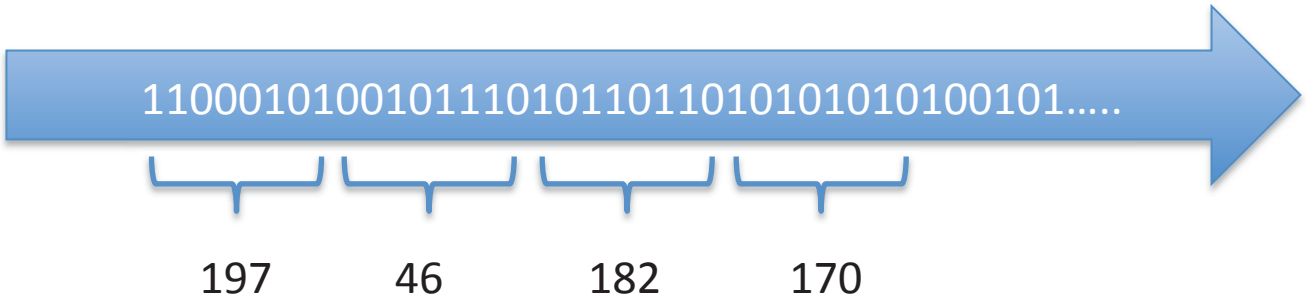
- Scanner
- Reader
- InputStream (e.g. System.in)
- FileReader
- BufferedReader
- Something else from java.io?

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:

```
int read ();          // Reads the next byte of data
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

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();
```

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

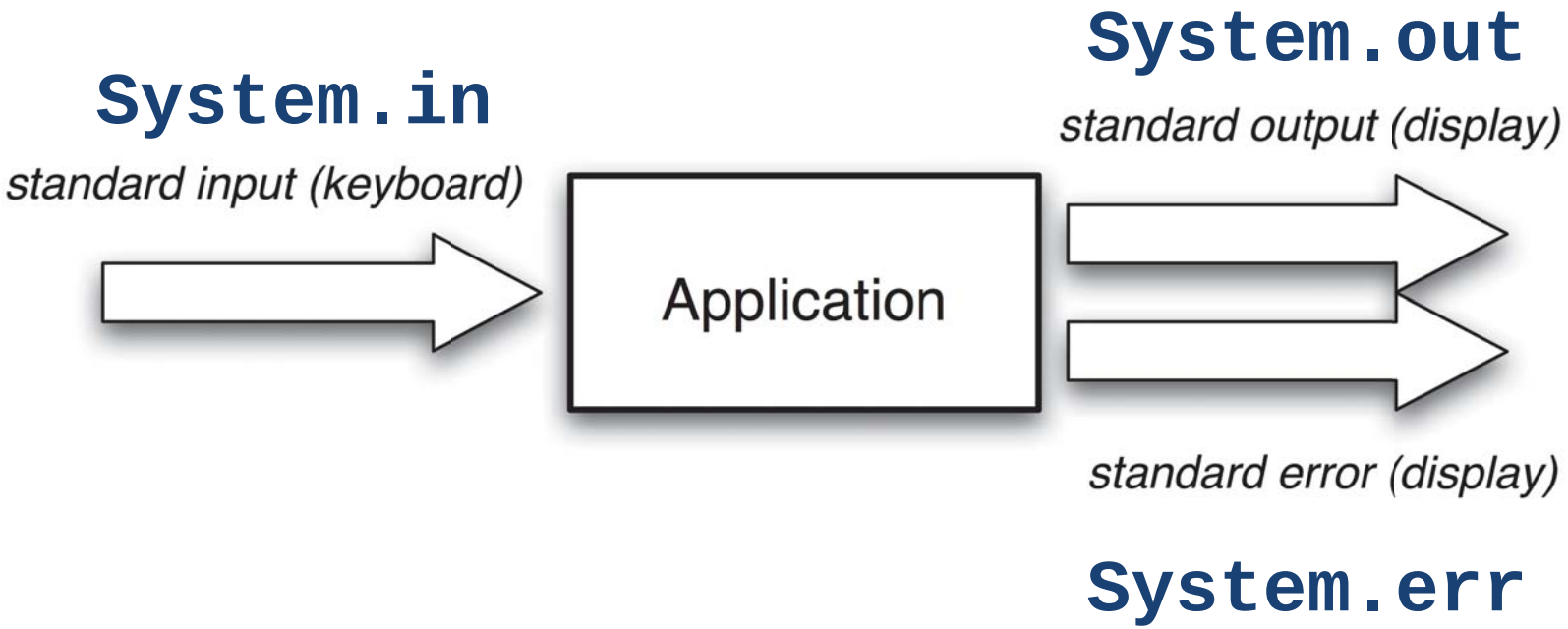
Buffering Example

```
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();
```

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.

PrintStream Methods

PrintStream adds buffering and binary-conversion methods to OutputStream

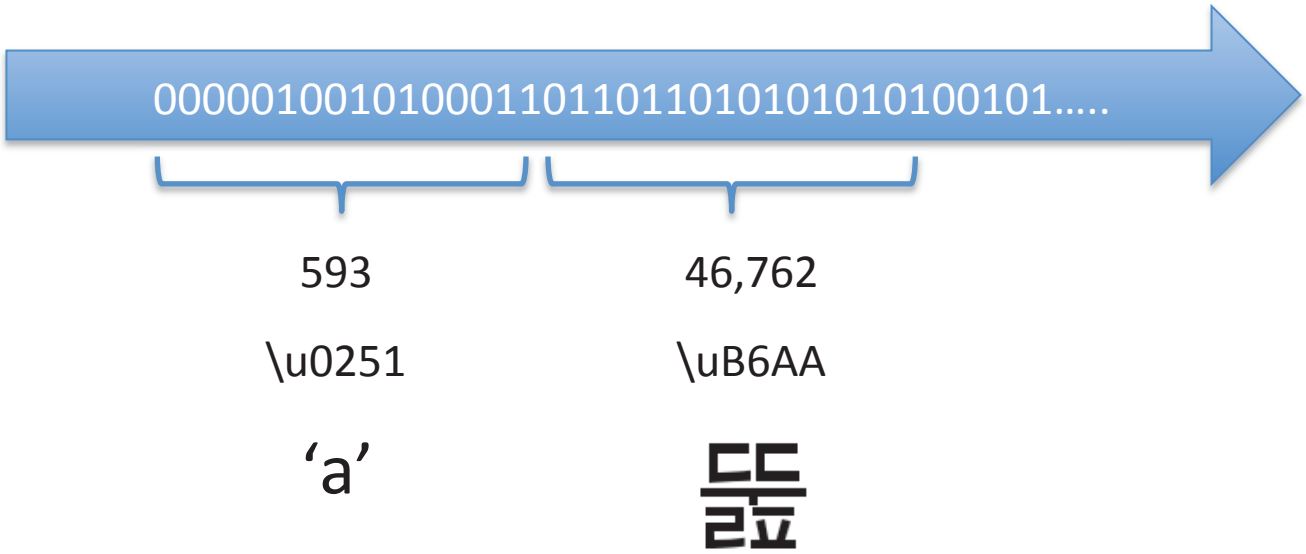
```
void println(boolean b); // write b followed by a new line
void println(String s);  // write s followed by a newline
void println();          // write a newline to the stream

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

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:

```
int read ();          // Reads the next character
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

Design Example: Histogram.java

A design exercise using java.io and the generic collection libraries

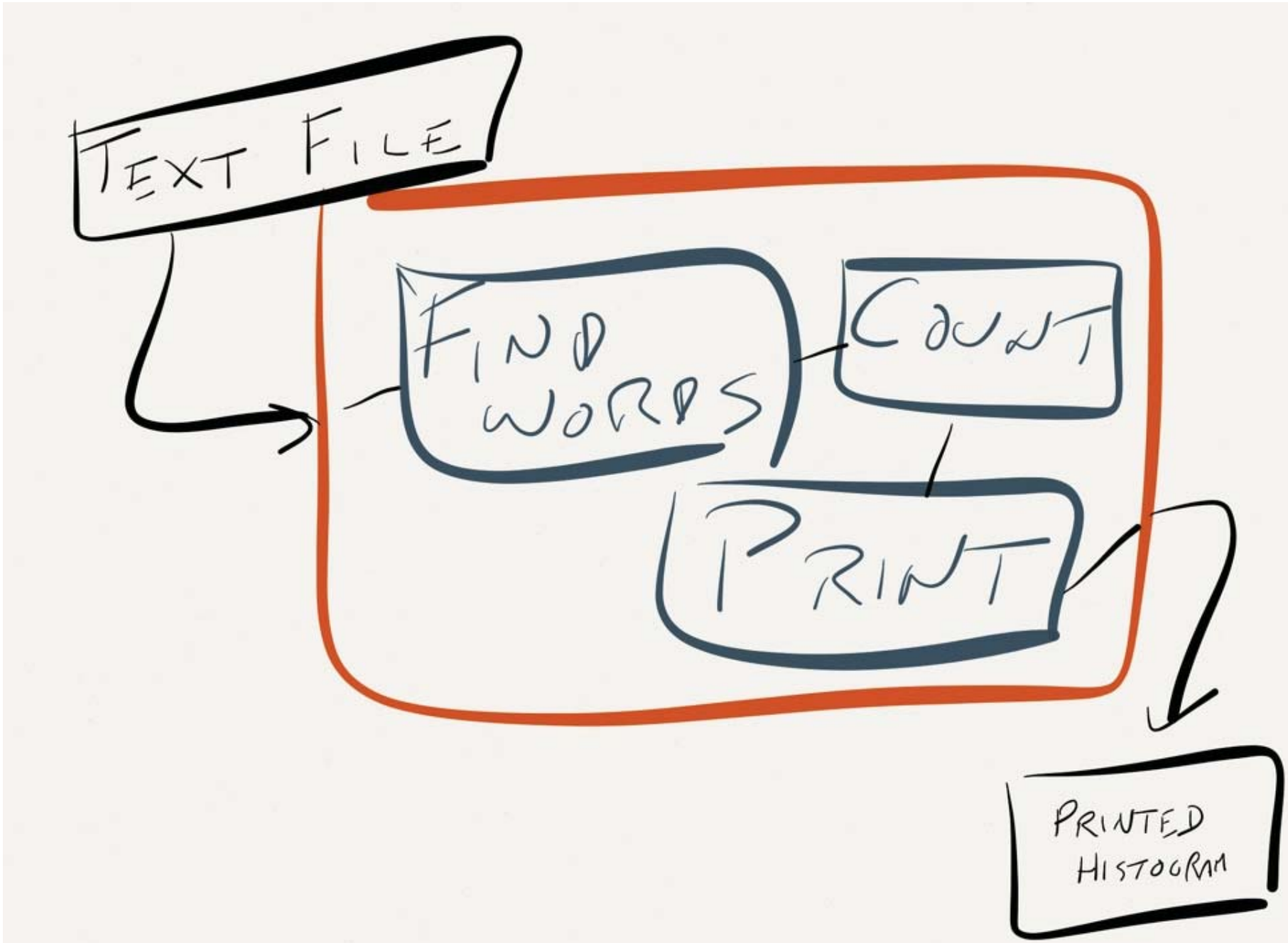
(SEE COURSE NOTES FOR THE FULL STORY)

Problem Statement

Write a program that, given a filename for a text file as input, calculates the frequencies (i.e. number of occurrences) of each distinct word of the file. The program should then print the frequency distribution to the console as a sequence of "word: freq" pairs (one per line).

Histogram result:

The : 1
Write : 1
a : 4
as : 2
calculates : 1
command : 1
console : 1
distinct : 1
distribution : 1
e : 1
each : 1
file : 2
filename : 1
for : 1
freq : 1
frequencies : 1
frequency : 1
given : 1
i : 1
input : 1
line : 2
number : 1
occurrences : 1
of : 4
one : 1
pairs : 1
per : 1
print : 1
program : 2
sequence : 1
should : 1
text : 1
that : 1
the : 4
then : 1
to : 1
word : 2


TEXT FILE
FIND WORDS
COUNT
PRINT
PRINTED HISTOGRAM

Decompose the problem

- Sub-problems:
  1. How do we iterate through the text file, identifying all of the words?
  2. Once we can produce a stream of words, how do we calculate their frequency?
  3. Once we have calculated the frequencies, how do we print out the result?

- What is the interface between these components?
- Can we test them individually?

How to produce a stream of words?

1. How do we iterate through the text file, identifying all of the words?

```
public interface Iterator<T> {
    // returns true if the iteration has more elements
    public boolean hasNext();
    // returns the next element in the iteration
    public T next();
    // Optional: removes last element returned
    public void remove();
}
```

- **Key idea:** Define a class (WordScanner) that implements this interface by reading words from a text file.

Coding: Histogram.java

WordScanner.java

Histogram.java

Programming Languages and Techniques (CIS120)

Lecture 32

November 20, 2017

Histogram Demo

Chapter 28

Announcements

- HW8: SpellChecker
  - Available on the website and Codio
  - Due Tomorrow, November 21st
- *Thanksgiving Break*:
  - No lab sections
  - Wednesday, November 22nd: Bonus Lecture offered ONLY at the 11:00-noon timeslot (everyone welcome to attend)
  - Topic: Code *is* Data (fun, but not relevant to HW or exam materials)
- NOTE: Office Hours schedules are different due to Thanksgiving
  - Check the calendar

- Interested in becoming a Teaching Assistant?
  - Applications are open for CIS 110, 160, 120, 121
  - See post on Piazza

HW09: Create your own Game

- Instructions available soon.

- Checkpoint: Describe your game, what features you will use.
  - DUE: Next Week! (After break)
  - Not a huge time commitment
  - Intended to get you thinking about your project. Be creative!
  - Submission via Gradescope

- Game Project
  - Due: MONDAY, December 11th at midnight
  - No late days!

Design Example: Histogram.java

A design exercise using java.io and the generic collection libraries

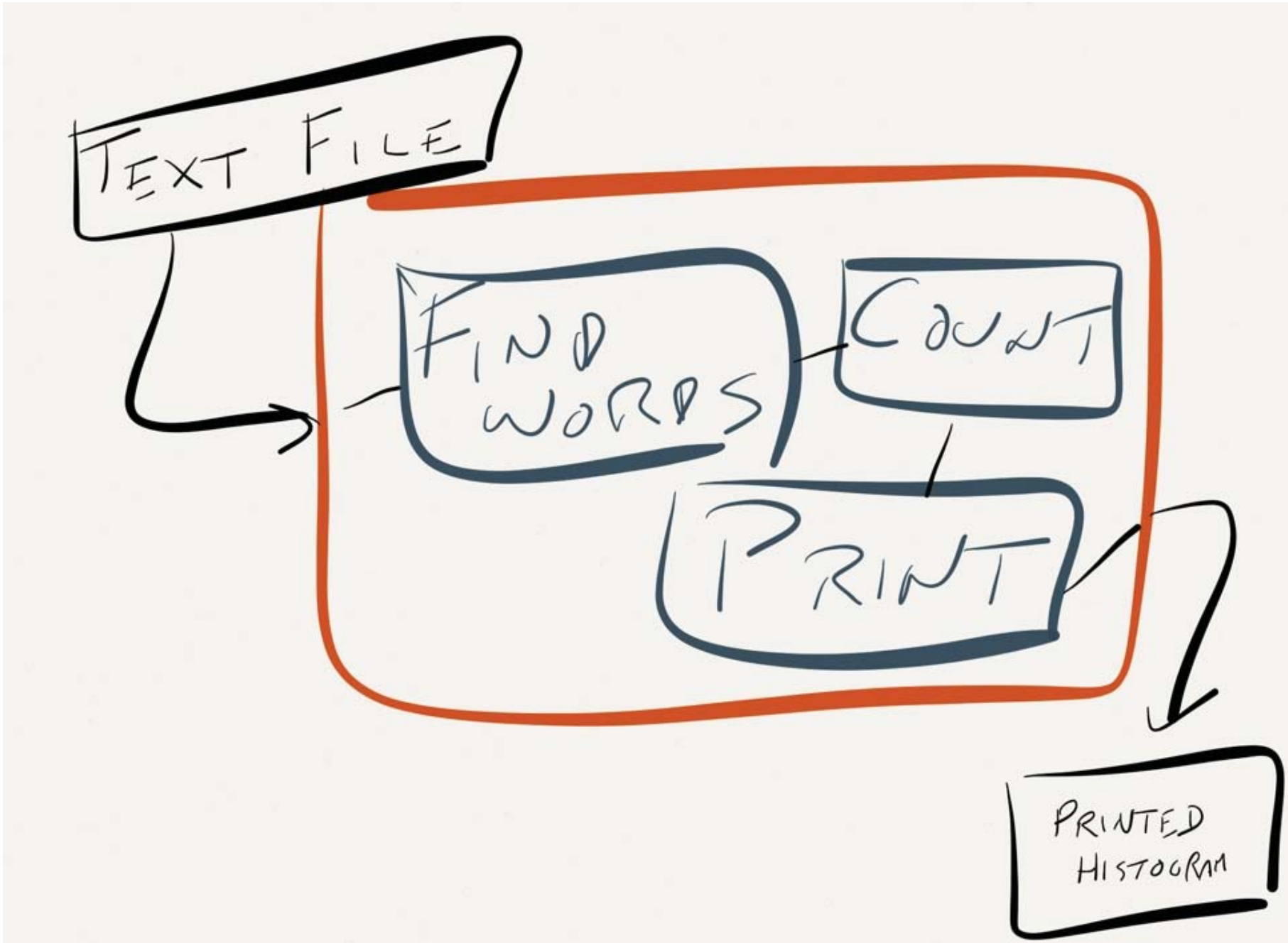
(SEE COURSE NOTES FOR THE FULL STORY)

Problem Statement

Write a program that, given a filename for a text file as input, calculates the frequencies (i.e. number of occurrences) of each distinct word of the file. The program should then print the frequency distribution to the console as a sequence of "word: freq" pairs (one per line).

Histogram result:

The : 1
Write : 1
a : 4
as : 2
calculates : 1
command : 1
console : 1
distinct : 1
distribution : 1
e : 1
each : 1
file : 2
filename : 1
for : 1
freq : 1
frequencies : 1
frequency : 1
given : 1
i : 1
input : 1
line : 2
number : 1
occurrences : 1
of : 4
one : 1
pairs : 1
per : 1
print : 1
program : 2
sequence : 1
should : 1
text : 1
that : 1
the : 4
then : 1
to : 1
word : 2


TEXT FILE
FIND WORDS
COUNT
PRINT
PRINTED HISTOGRAM

Reading Data

Which I/O class should we use to open the text file?

1. InputStream
2. FileInputStream
3. FileReader
4. BufferedReader

Decompose the problem

- Sub-problems:
  1. How do we iterate through the text file, identifying all of the words?
  2. Once we can produce a stream of words, how do we calculate their frequency?
  3. Once we have calculated the frequencies, how do we print out the result?

- What is the interface between these components?
- Can we test them individually?

How to produce a stream of words?

1. How do we iterate through the text file, identifying all of the words?

```
public interface Iterator<T> {
    // returns true if the iteration has more elements
    public boolean hasNext();
    // returns the next element in the iteration
    public T next();
    // Optional: removes last element returned
    public void remove();
}
```

- **Key idea:** Define a class (WordScanner) that implements this interface by reading words from a text file.

Coding: Histogram.java

WordScanner.java

Histogram.java