Programming Languages and Techniques (CIS120e)

Lecture 25

Nov. 8, 2010

Exceptions and the Java Abstract Stack Machine
Announcements

• Homework 8 (SpellChecker) is due Nov 15th.

• Midterm 2 is this Friday, November 12th
Exceptions

Dealing with failure.
Exceptions

• An exception is an object representing abnormal conditions.
  – Its internal state describes what went wrong.
  – Examples: NullPointerException, IllegalArgumentException, IOException

• **Throwing** an exception is an *emergency exit* from the current method.
  – The exception propagates up the invocation stack until it either
    reaches the top and the stack, in which case the program aborts with
    the error, or the exception is caught.

• **Catching** an exception lets callers take appropriate actions to
  handle the abnormal circumstances.
Example

class C {
    public void foo() {
        bar();
        System.out.println("here in foo");
    }
    public void bar() {
        baz();
        System.out.println("here in bar");
    }
    public void baz() {
        throw new Exception();
        System.out.println("here in baz");
    }
}

• What happens if we do (new C()).foo();?
Abstract Stack Machine

Workspace

Stack

Heap

(new C()).foo();
Abstract Stack Machine

Workspace

Stack

Heap

(new C()).foo();
Allocate a new instance of C in the heap.
Abstract Stack Machine

Workspace

foo();

Stack

Heap

C
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.

*We’ll explain the “this” pointer in more detail later. Suffice it to say that it’s how we can find which code to run when an instance method like bar is invoked.
Abstract Stack Machine

Workspace

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

Stack

_;
this

Heap

C
Abstract Stack Machine

Workspace

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

Stack

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

Heap

C
Abstract Stack Machine

Workspace

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

Stack

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

Heap

```
c
```
Abstract Stack Machine

Workspace

throw new Exception();
System.out.println("here in baz");

Stack

_heap

this

_heap

this

_heap

this

Heap

C

this

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

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

throw new Exception();
System.out.println("here in baz");
throw ();
System.out.println("here in baz");
Abstract Stack Machine

Workspace

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

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.
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.
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.
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.
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.
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

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

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

Workspace  Stack  Heap

(new C()).foo();
Abstract Stack Machine

Workspace

Stack

Heap

(new C()).foo();
Allocate a new instance of C in the heap.
Abstract Stack Machine

Workspace

Stack

Heap

(()).foo();
Abstract Stack Machine

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

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

*We’ll explain the “this” pointer in more detail in the next lecture. Suffice it to say that it’s how we can find which code to run when an object-local method like bar is invoked.
Abstract Stack Machine

Workspace

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

Stack

_heap
this

Heap

C
try {
    baz();
} catch (Exception e) {
    System.out.println("caught");
} System.out.println("here in bar");
When executing a try/catch block, push onto the stack a new workspace that contains all of the current workspace except for the try { ... } code.

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

Replace the current workspace with the body of the try.
Abstract Stack Machine

Workspace

baz();

Stack

_;

this

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

this

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

Heap

c

Continue executing as normal.
Abstract Stack Machine

Workspace

throw new Exception();
System.out.println("here in baz");

Stack

_;
this

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

_;  

Heap

C

The top of the stack is off the bottom of the page... 😊
Abstract Stack Machine

Workspace

```
throw new Exception();
System.out.println("here in baz");
```

Stack

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

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

Heap

```
C
```
Abstract Stack Machine

throw ();
System.out.println(“here in baz”);

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

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

Workspace
Stack
Heap
C
Exception

System.out.Println
System.out.Println
throw ();
System.out.println(“here in baz”);
Abstract Stack Machine

Workspace

```
throw ()
System.out.println( "here in baz");
```

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.
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.
Workspace

```java
{ 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.
Continue executing as usual.

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

_,
this
System.out.println("here in foo");
this
e

C
Exception
```
Abstract Stack Machine

Workspace

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

Continue executing as usual.

Console
caught

Heap

```
_c;
this

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

Exception

executed as usual.
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.
Abstract Stack Machine

Workspace

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

Continue executing as usual.

Stack

_;
this

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

Heap

C

Exception

Console caught
Abstract Stack Machine

Workspace

System.out.println(“here in bar”);

Continue executing as usual.

Heap

C

Exception

System.out.println(“here in foo”);

this

Console caught
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

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

Continue executing as usual.

Console
caught
here in bar
Abstract Stack Machine

Workspace

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

Stack

this

Heap

C

Exception

Continue executing as usual.

Console
caught
here in bar
Abstract Stack Machine

Workspace

Stack

Heap

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:

```java
Workspace

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

Workspace

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

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

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

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

Object

Throwable

Exception

Error

IOException

RuntimeException

IllegalArgumentException

Type of all throwable objects.

Subtypes of Exception must be declared.

Fatal Errors, should never be caught.

Subtypes of RuntimeException do not have to be declared.
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

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

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

```java
public void doSomeIO (String file) throws IOException {
    Reader r = new FileReader(file);  // might throw
    ...
}
```
Unchecked (Undeclared) Exceptions

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

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

- The original intent was that such exceptions represent disastrous conditions from which it was impossible to sensibly recover...
Declared vs. Undeclared?

• Tradeoffs in the software design process:
  • Declared = better documentation
    – forces callers to acknowledge that the exception exists
  • Undeclared = fewer static guarantees
    – but, much easier to refactor code

• In practice: test-driven development encourages “fail early/fail often” model of code design and lots of code refactoring, so “undeclared” exceptions are prevalent.

• A good compromise?
  – Declared exceptions for libraries, where the documentation and usage enforcement are critical
  – Undeclared for client-exceptions to facilitate more flexible code
Good Style for Exceptions

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

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

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

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

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

• Catch exceptions with as much precision as you can
  – i.e. Don’t do: try {...} catch (Exception e) {...}
    instead do: try {...} catch (IOException e) {...}
Finally

• A “finally” clause of a try/catch/finally statement always gets run, regardless of whether there is no exception, a propagated exception, a caught exception, or even if the method returns from inside the try.

• “Finally” is most often used for releasing resources that might have been held/created by the “try” block:

```java
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) {…}
        }
    }
}
```
The Java Abstract Stack Machine

1. Class tables
2. Method invocations and “this”
3. Static members
Consider This Example

```java
public class Ctr {
    private int x;
    public Ctr () { x = 0; }
    public void incBy(int d) { x = x + d; }
    public int get() { return x; }
}

public class Decr extends Ctr {
    private int y;
    public Decr (int initY) { y = initY; }
    public void dec() { incBy(-y); }
}

// ... somewhere in main:
Decr d = new Decr(2);
d.dec().get();
```