Programming Languages and Techniques (CIS120)

Lecture 30
April 4, 2016

Exceptions
Chapter 27
Announcements

• HW7: PennPals Chat
  – Due: Tuesday
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 (runtime error)
2. Program prints “here in bar”, then stops
3. Program prints “here in bar”, then “here in foo”, then stops
4. Something else
Exceptions

Dealing with the unexpected
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
Abstract Stack Machine

Workspace

(new C()).foo();

Stack

Heap
Allocate a new instance of C in the heap. (Skipping details of trivial constructor for C.)
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. Use the dynamic class to lookup the method body from the class table.
this.bar();
System.out.println("here in foo");
Abstract Stack Machine

Workspace

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

Stack

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

Heap

```
C
```
Abstract Stack Machine

**Workspace**

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

**Stack**

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

**Heap**

```java
C
```

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

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

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

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

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

Try/Catch for ()? No!
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.

Program terminated with uncaught exception ( )!
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();
    }
}

What happens if we do (new C()).foo()?
1. Program stops without printing anything (other than runtime error message)
2. Program prints "here in bar", then "here in foo"
3. Program prints "here in bar", then "here in foo", then "caught"
4. Program prints "caught", then "here in bar", then "here in 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

C

foo();

Stack

Heap
Save a copy of the current workspace in the stack, leaving a “hole”, written _, where we return to. Push the this pointer, followed by arguments (in this case none) onto the stack.
```java
this.bar();
System.out.println("here in foo");
```
try {
    baz();
} catch (Exception e) {
    System.out.println("caught");
}
System.out.println("here in bar");

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

Continue executing as normal.
Abstract Stack Machine

Workspace

throw new
RuntimeException();

--- Stack ---

_;

this

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

this

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

_;

--- Heap ---

C

The top of the stack is off the bottom of the page... 😊
throw new RuntimeException();

private void bar()
{
    System.out.println("here in bar");
}

private void foo()
{
    System.out.println("here in foo");

    try
    {
        System.out.println("try");
    }
    catch (RuntimeException e)
    {
        System.out.println("catch");
    }
}

class AbstractStackMachine

throw C;

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

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

throw ();

catch (RuntimeException e) {
    System.out.println("caught");
}
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.

Try/Catch for ()? No!
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

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

Continue executing as usual.

Console caught

Abstract Stack Machine

Workspace

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

Stack

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

Heap

```
C
Runtime Exception
```
Abstract Stack Machine

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

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

Continue executing as usual.

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

Console caught

Runtime Exception
Abstract Stack Machine

Workspace

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

Console caught

Stack

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

Heap

C

Runtime Exception

Continue executing as usual.
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

```
_;
this
```

Heap

```
C
```

Console

Caught here in bar

Continue executing as usual.
Abstract Stack Machine

Workspace

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

Stack

-;

this

Heap

C

Console

catched

here in bar

Continue executing as usual.
Abstract Stack Machine

Workspace

;  

Stack

-;

this

Heap

C

Console
caught here in bar here in foo

Continue executing as usual.
Abstract Stack Machine

Program terminated normally.

Console
cought
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 (Runtime Exception e) {
    System.out.println("caught");
}
System.out.println("here in bar");

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

- There can be more than one “catch” clause associated with each “try”
  - Matched in order, according to the *dynamic* class of the exception thrown
  - The *first* clause that declares a supertype of the exception is triggered
  - Helps refine 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.
}
```
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.
• **Finally** is 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) {...}
        }
    }
}
```
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"
Informative Exception Handling
**Exception Class Hierarchy**

- **Object**
  - **Throwable**
    - **Exception**
      - **Error**
        - **IOException**
        - **RuntimeException**
          - **IllegalArgumentException**
        - **FileNotFoundException**

Subtypes of Exception *must be declared.*

Subtypes of RuntimeException *do not have to be declared.*

Fatal Errors: should never be caught.

Type of all throwable objects.
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 by 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!
        ...
    }

    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...
Which methods need a "throws" clause?

Note:
IllegalArgumentExcepti
on 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
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 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
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) {...}