Exceptions

Copyright © 2005-2007 William C. Benton

No method can deal with every possible contingency.

Copyright © 2005-2007 William C. Benton

Examples

- `compareTo(Object o)`
  - How can you compare instances of different classes? (Recall the contract!)
- `Integer.parseInt(“skål”);`
  - what `int` could this sensibly return?
- `new FileInputStream(“bogus.txt”);`
  - what if `bogus.txt` doesn’t exist?

Distinguished values

- One idea: use *distinguished return values* to identify error conditions
- For example, consider the getClicks() method in Clicker
  - could return -1 to indicate Clicker malfunction

In many cases, the return value is an inadequate way to signal what went wrong.

Copyright © 2005-2007 William C. Benton

In many cases, the return value is an inadequate way to signal what went wrong.

(Why?)

Copyright © 2005-2007 William C. Benton

How do you distinguish between valid and error (“out-of-band”) results?

Copyright © 2005-2007 William C. Benton

How do you distinguish between valid and error (“out-of-band”) results?

(You’ll need a return type that can represent more values than are in the range of your function!)

Copyright © 2005-2007 William C. Benton

How do you tell the user to check for errors after each method call?

Copyright © 2005-2007 William C. Benton

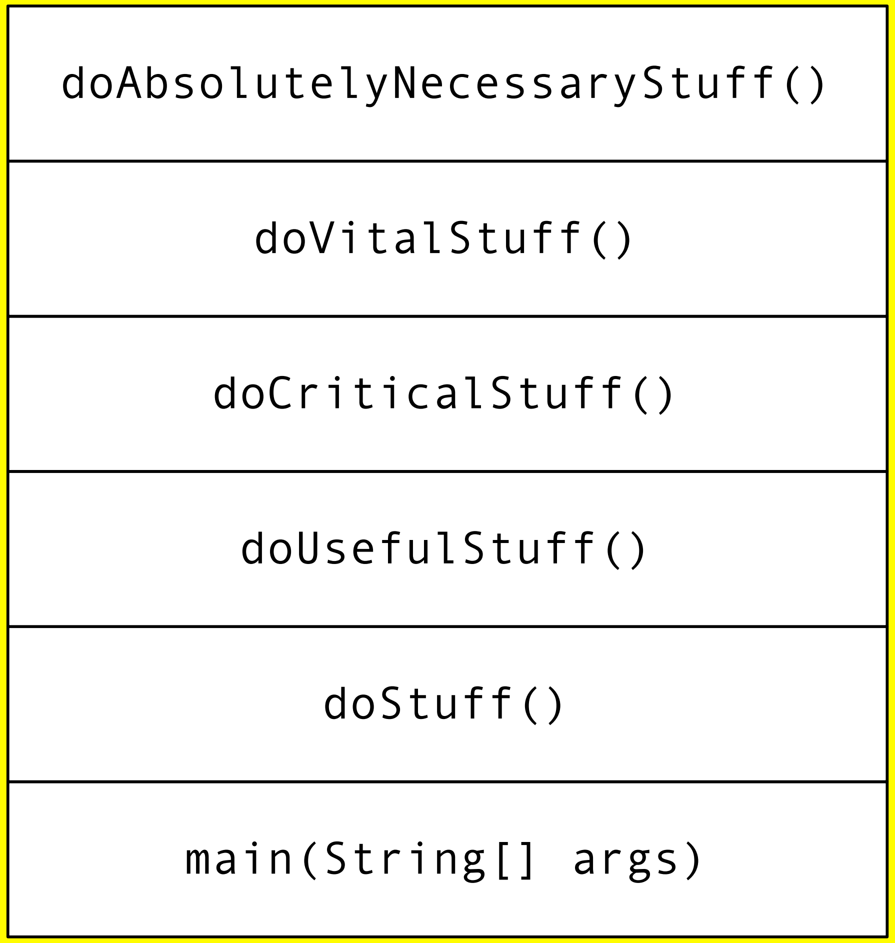
How do you tell the user to check for errors after each method call?

Not with a straight face, I hope.

Copyright © 2005-2007 William C. Benton

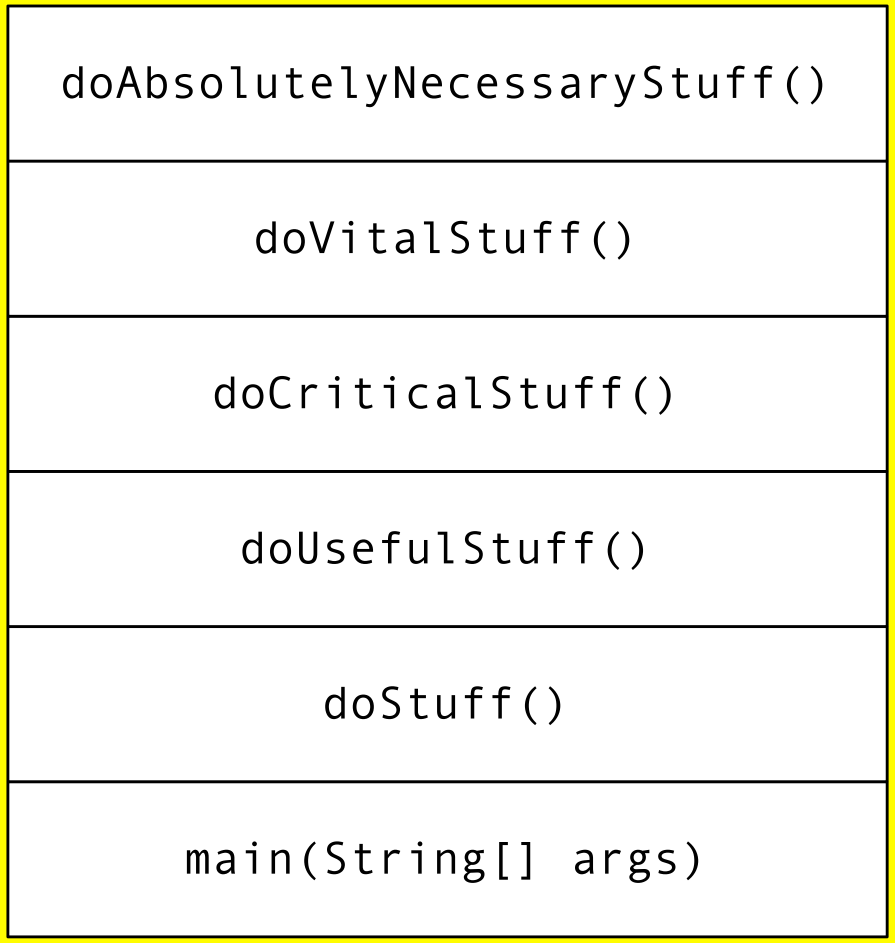
(Even worse: handling an error may require many methods to return.)

Copyright © 2005-2007 William C. Benton



Copyright © 2005-2007 William C. Benton

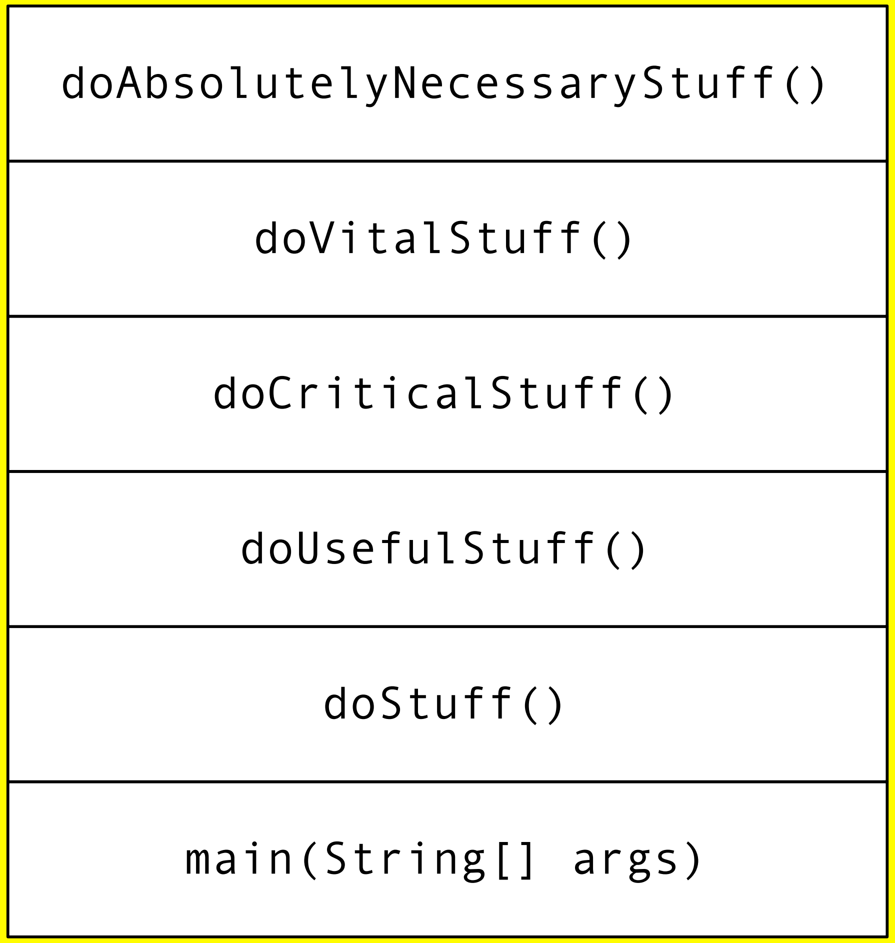
Consider this call stack.



Copyright © 2005-2007 William C. Benton

Consider this call stack.

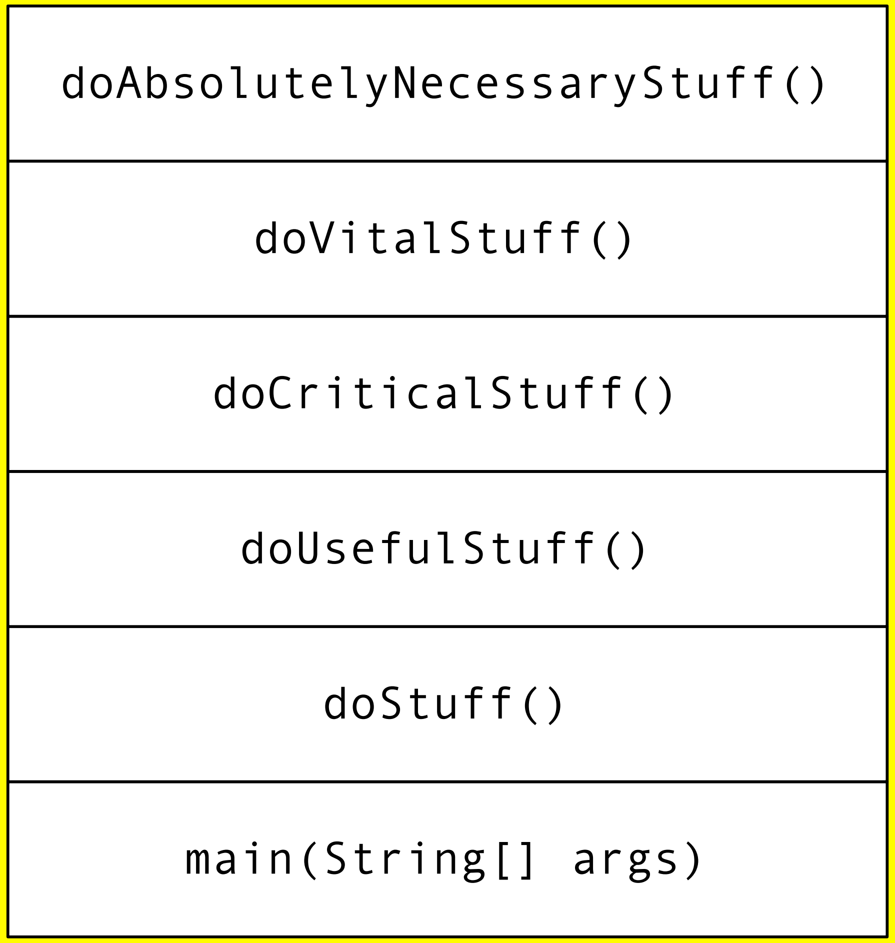
Assume `doStuff` cannot succeed unless every method it calls succeeds, and so on.



Consider this call stack.

Assume `doStuff` cannot succeed unless every method it calls succeeds, and so on.

What happens if there is an error in `doAbsolutelyNecessaryStuff`?



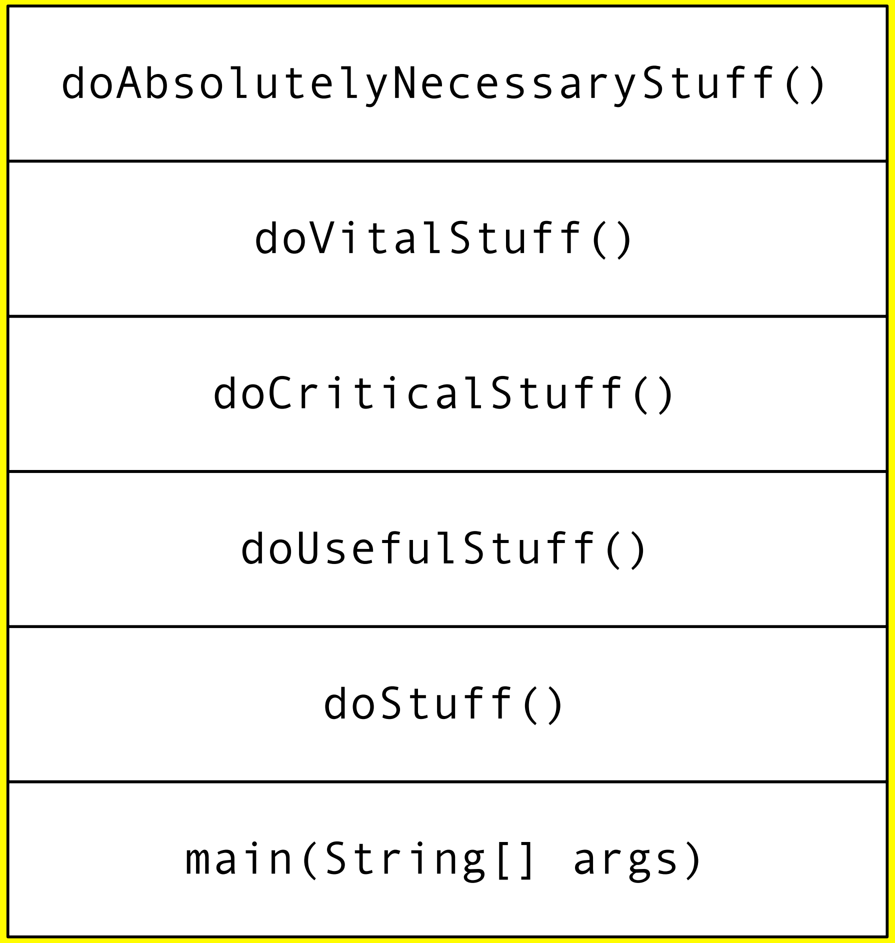
Copyright © 2005-2007 William C. Benton

Consider this call stack.

Assume `doStuff` cannot succeed unless every method it calls succeeds, and so on.

What happens if there is an error in `doAbsolutelyNecessaryStuff`?

A big pain for the programmer, that’s what happens.



Copyright © 2005-2007 William C. Benton

How do you ensure that the user *actually checks for errors*?

Copyright © 2005-2007 William C. Benton

How do you ensure that the user *actually checks for errors*?

Stern warnings? Strongly-worded documentation? Shame? Bribes?

Copyright © 2005-2007 William C. Benton

…and let’s not get started on constructors!

Copyright © 2005-2007 William C. Benton

…and let’s not get started on constructors!

(OK, twist my arm. Why are constructors even harder to deal with than methods?)

Copyright © 2005-2007 William C. Benton

It should be apparent that error-handling is a hard problem.

Copyright © 2005-2007 William C. Benton

A solution should...

- Provide some way to distinguish between legitimate return values and error notifications,
- Not excessively burden method callers with error code checking,
  - allow returning "up" multiple methods on the call stack, and
- Force the user to deal with (at least some kinds of) errors.

Copyright © 2005-2007 William C. Benton

One solution: pass the buck.

Copyright © 2005-2007 William C. Benton

"I am unhappy with the service that your company is providing me because of *blah blah blah*...."

Copyright © 2005-2007 William C. Benton

“I am unhappy with the service that your company is providing me because of *blah* *blah* *blah*....”

“I can’t address your specific complaint, but please remember that we value your business!”

Copyright © 2005-2007 William C. Benton

"I am unhappy with the service that your company is providing me because of *blah blah blah*...."

"I can't address your specific complaint, but please remember that we value your business!"

"Indeed, I plan to cancel my service and contract with your fiercest competitor!"

Copyright © 2005-2007 William C. Benton

"I am unhappy with the service that your company is providing me because of *blah blah blah*...."

"I can't address your specific complaint, but please remember that we value your business!"

"Indeed, I plan to cancel my service and contract with your fiercest competitor!"

"Sir/madam, let me transfer you to my supervisor. She may be able to help you."

Copyright © 2005-2007 William C. Benton

(Passing the buck has rich and lengthy precedent.)

Copyright © 2005-2007 William C. Benton

Exceptions

- The “may I speak to your supervisor?” of contemporary programming languages
- *Thrown* to signal all sorts of error conditions and
- *Caught* by code that can handle the error.

Idea: `try` to execute code that may contain an error and `catch` any problems that may occur.

Copyright © 2005-2007 William C. Benton

```
try
    stmt_try
catch (type id)
    stmt_catch
```

Copyright © 2005-2007 William C. Benton

```
try
    stmt_try
catch (type id)
    stmt_catch
```

(*type* must be an *exception type*; i.e. it must be a reference to a class that extends `Exception` or implements `Throwable`)

Copyright © 2005-2007 William C. Benton

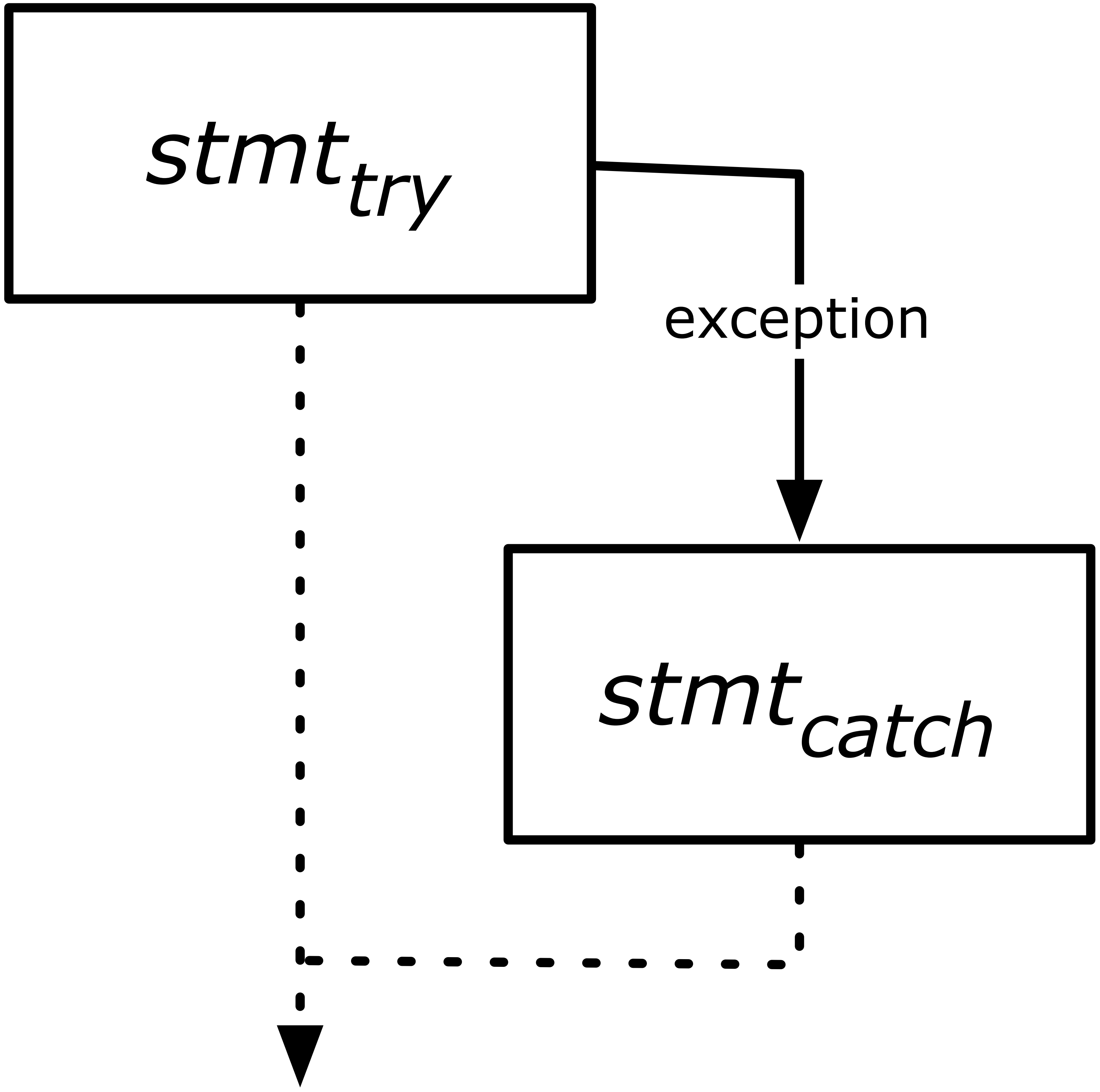
```
Scanner s = new Scanner(System.in);
boolean gotInputYet = false;
int result = 0;

do {
    try {
        System.out.print("Please enter a number:  ");
        String input = s.nextLine();
        result = Integer.parseInt(input);
        // only reached if parseInt succeeds
        gotInputYet = true;
    } catch (NumberFormatException e){
        System.out.println("That's not a number.");
    }
} while(!gotInputYet);
```

Copyright © 2005-2007 William C. Benton

```
try
   $stmt_{try}$
catch ($type_1$ $id$)
     $stmt_{catch1}$
catch ($type_2$ $id$)
     $stmt_{catch2}$
// etc.
```

Copyright © 2005-2007 William C. Benton



$stmt_{try}$
exception
$stmt_{catch}$

When an exception is generated, Java transfers control to the nearest enclosing *catch block* that can handle that exception.

Copyright © 2005-2007 William C. Benton

When an exception is generated, Java transfers control to the nearest enclosing *catch block* that can handle that exception.

(It won’t always be in the same method.)

Copyright © 2005-2007 William C. Benton

The VM creates a catch block surrounding `main()`.

Copyright © 2005-2007 William C. Benton

```
throw ref;
```

Copyright © 2005-2007 William C. Benton

`throw` *ref*;

(*ref* must be a reference to an instance of a class that extends `Exception` or implements `Throwable`)

Copyright © 2005-2007 William C. Benton

Typically, you'll construct *ref* on the line in which you `throw` it.

```
throw new LameExampleException();
```

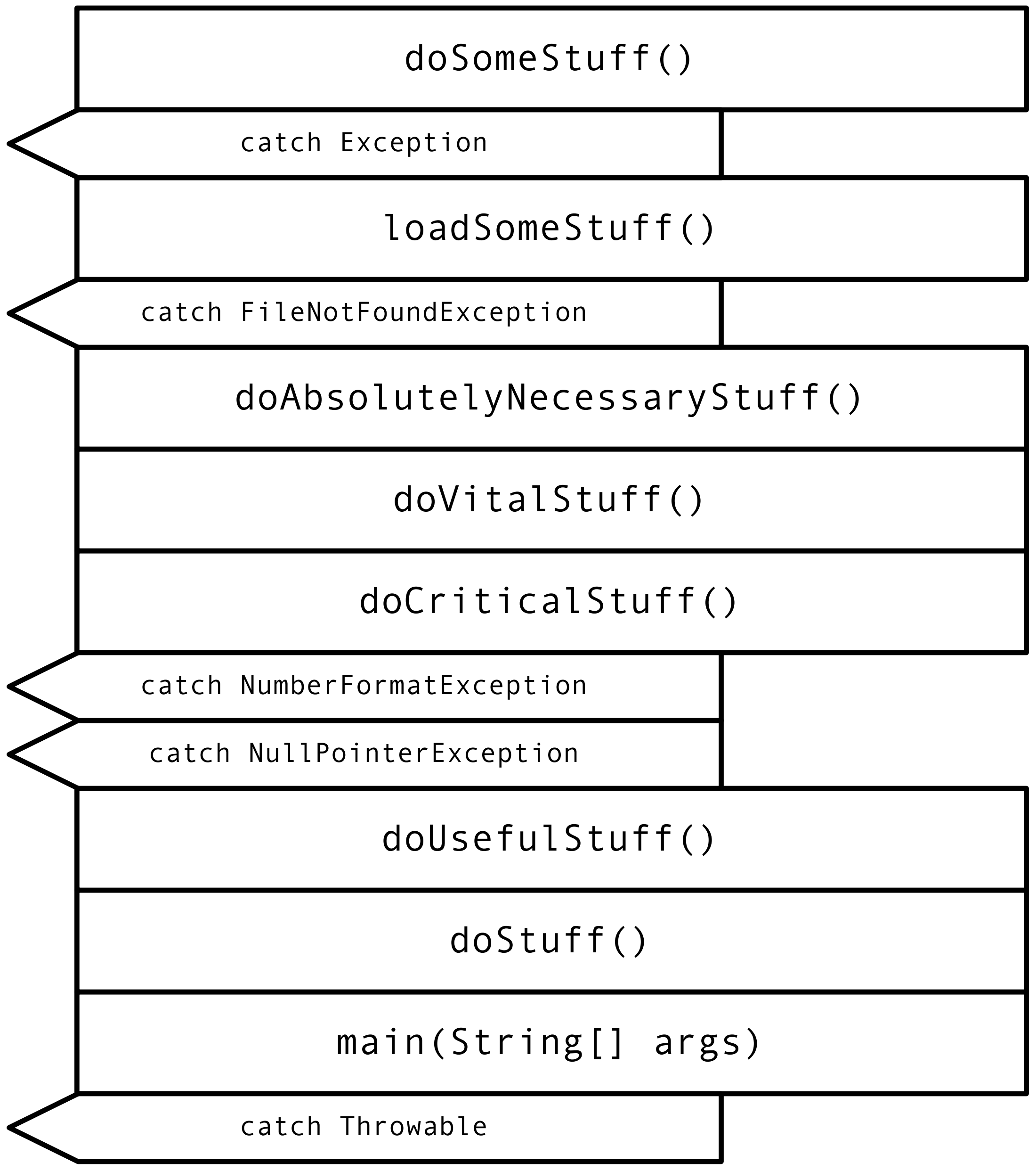
Copyright © 2005-2007 William C. Benton

Constructing an exception object is just like constructing other objects:

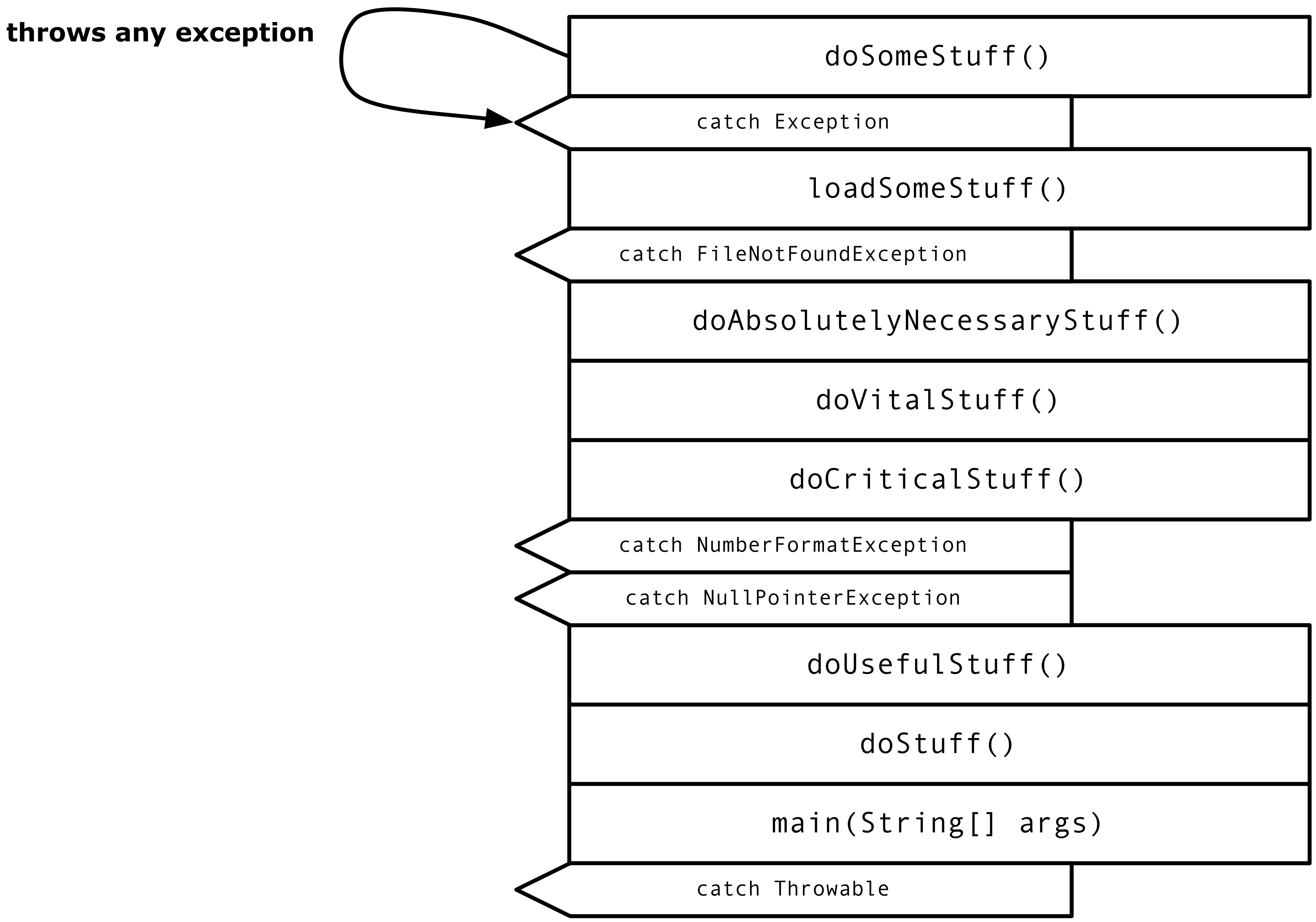
```
Exception e =
   new LameExampleException();
```

This does not cause `e` to be thrown. (Indeed, you can do all sorts of things with `e`.)

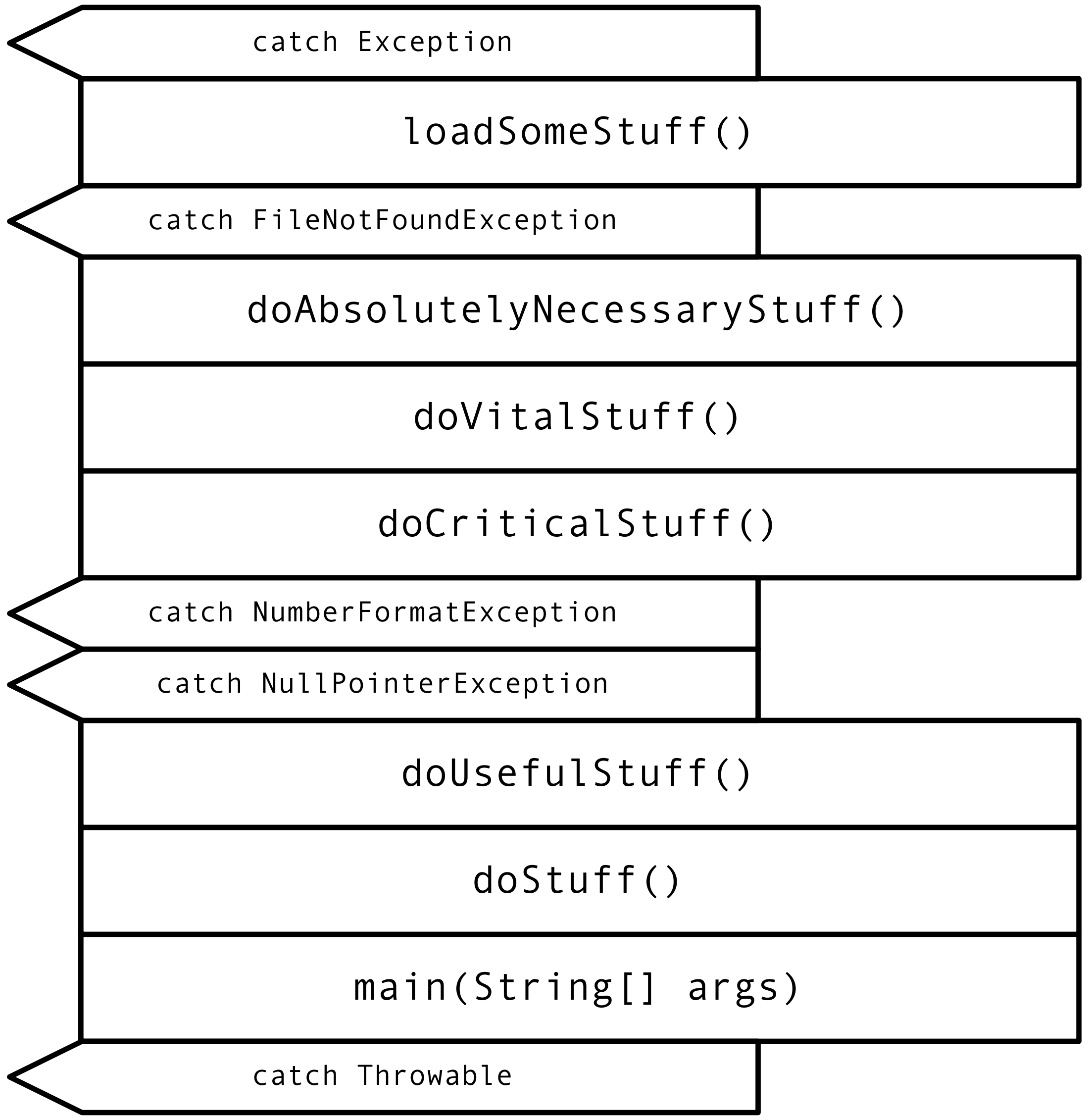
Copyright © 2005-2007 William C. Benton



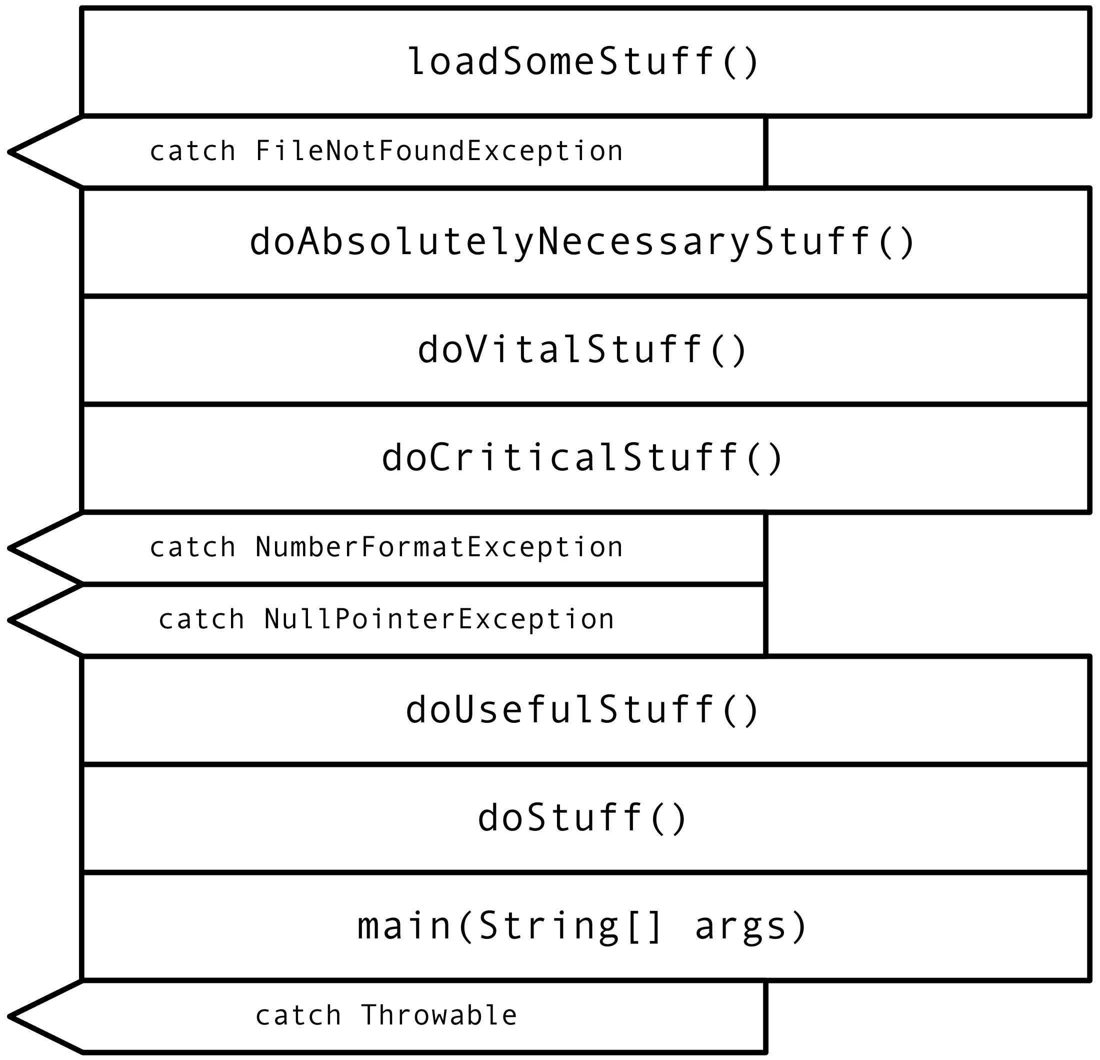
Copyright © 2005-2007 William C. Benton



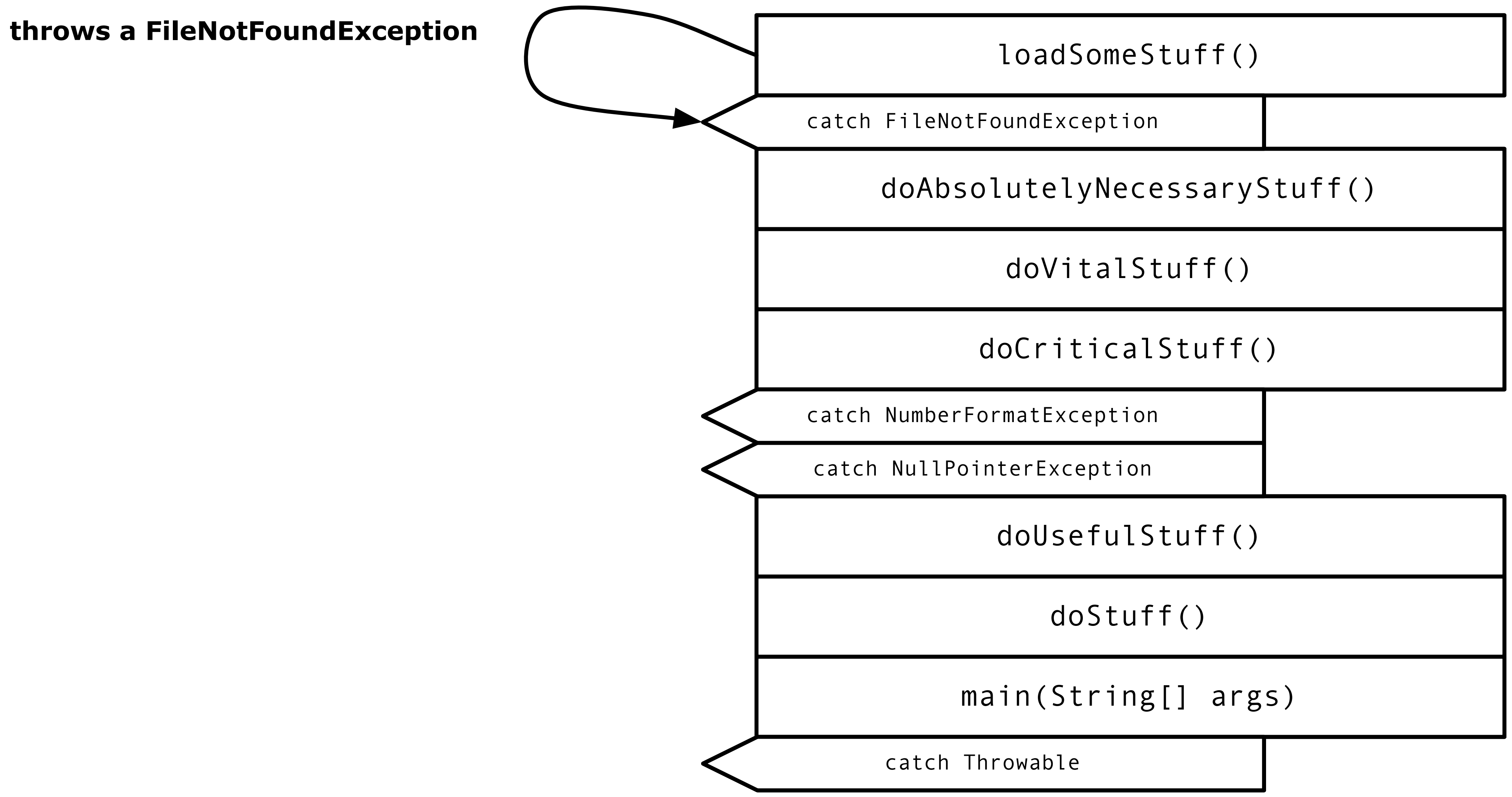
Copyright © 2005-2007 William C. Benton

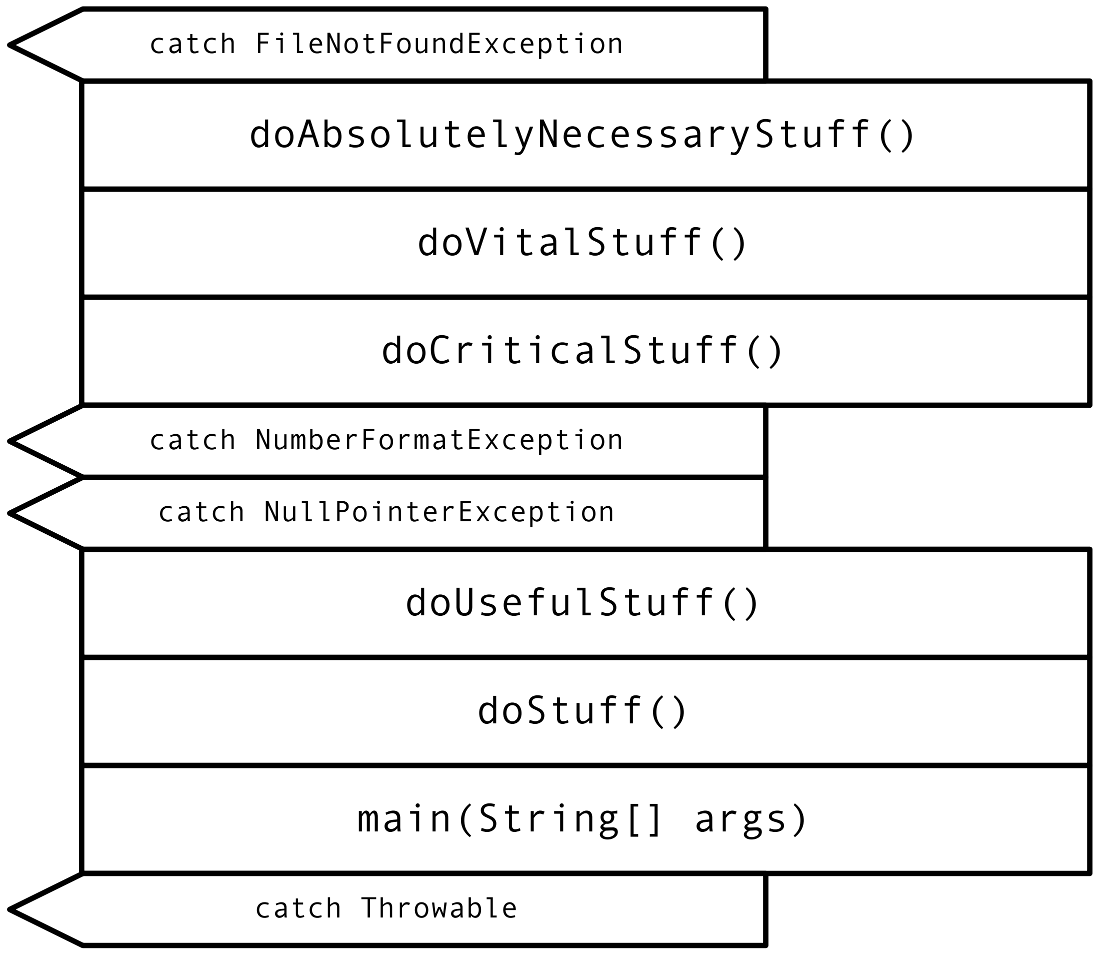


Copyright © 2005-2007 William C. Benton

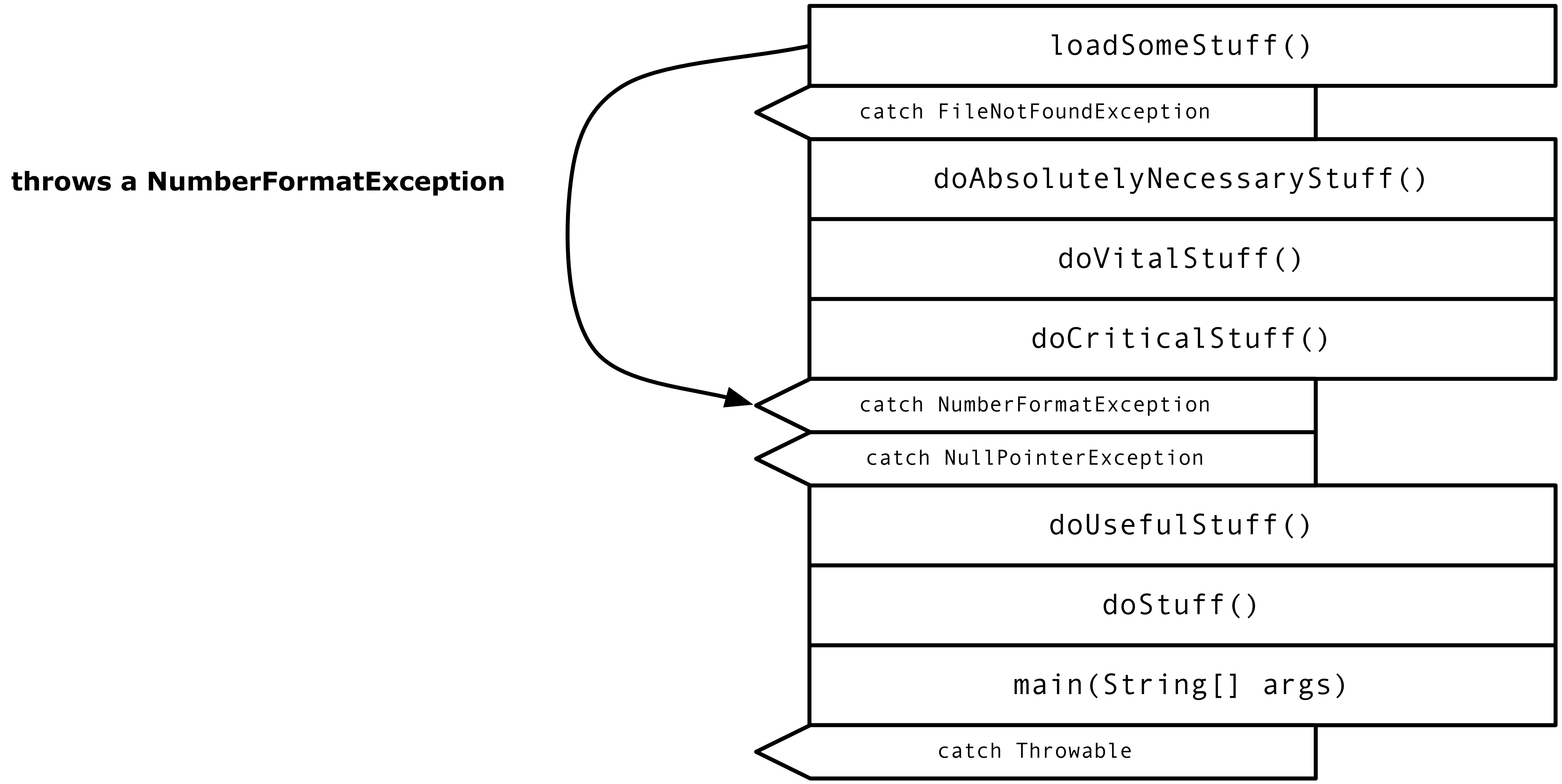

loadSomeStuff()
catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

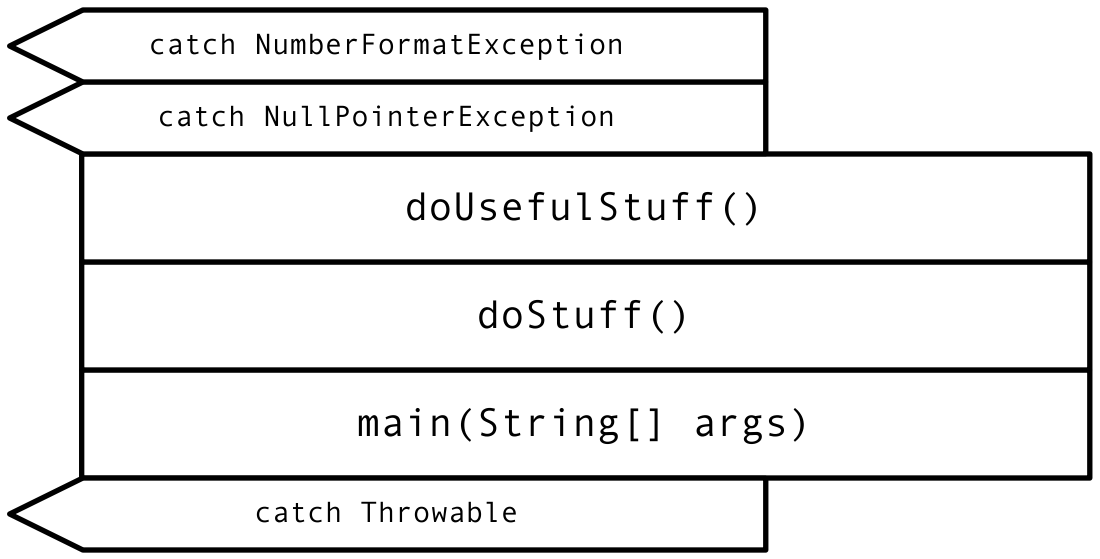
Copyright © 2005-2007 William C. Benton


throws a FileNotFoundException
loadSomeStuff()
catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

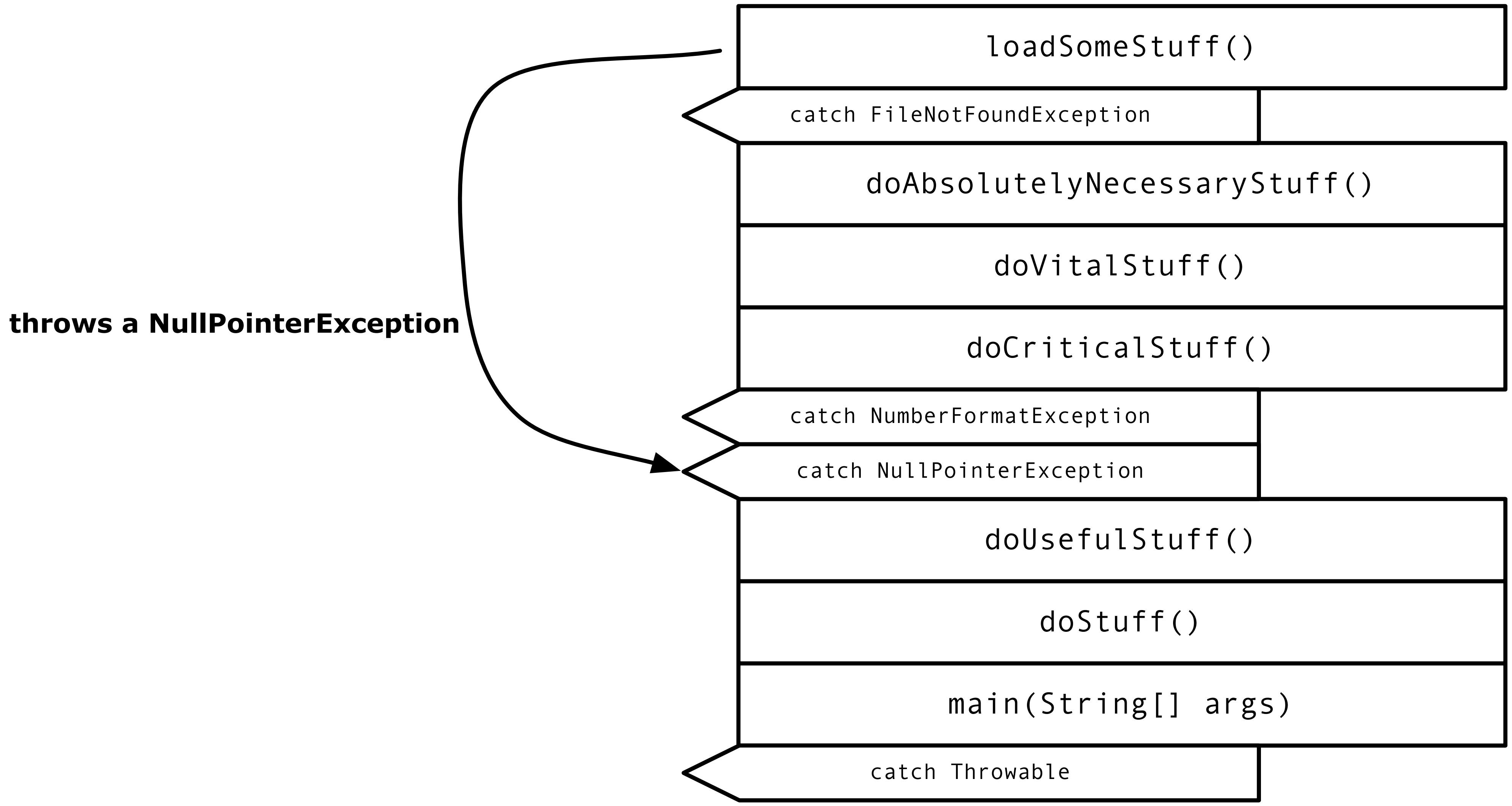

catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

Copyright © 2005-2007 William C. Benton


throws a NumberFormatException
loadSomeStuff()
catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

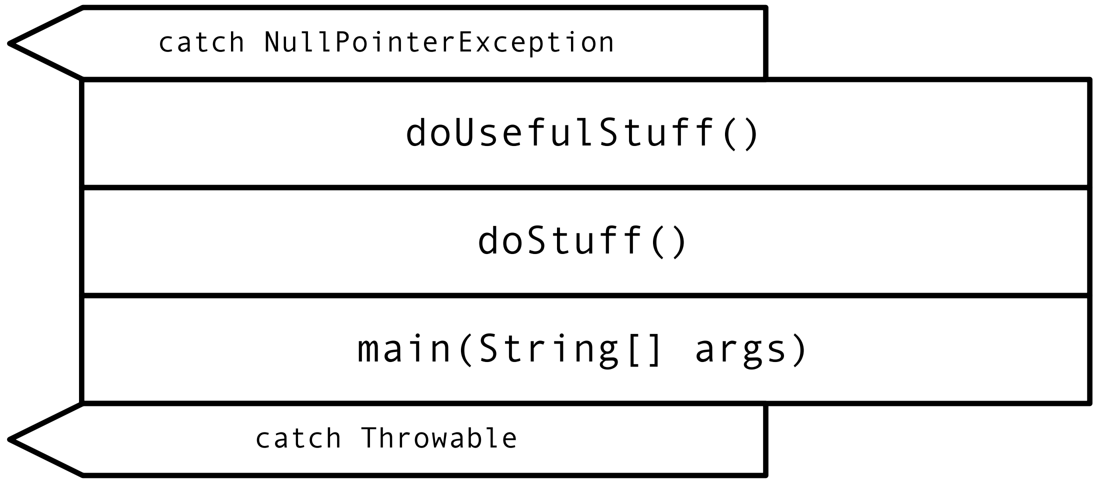


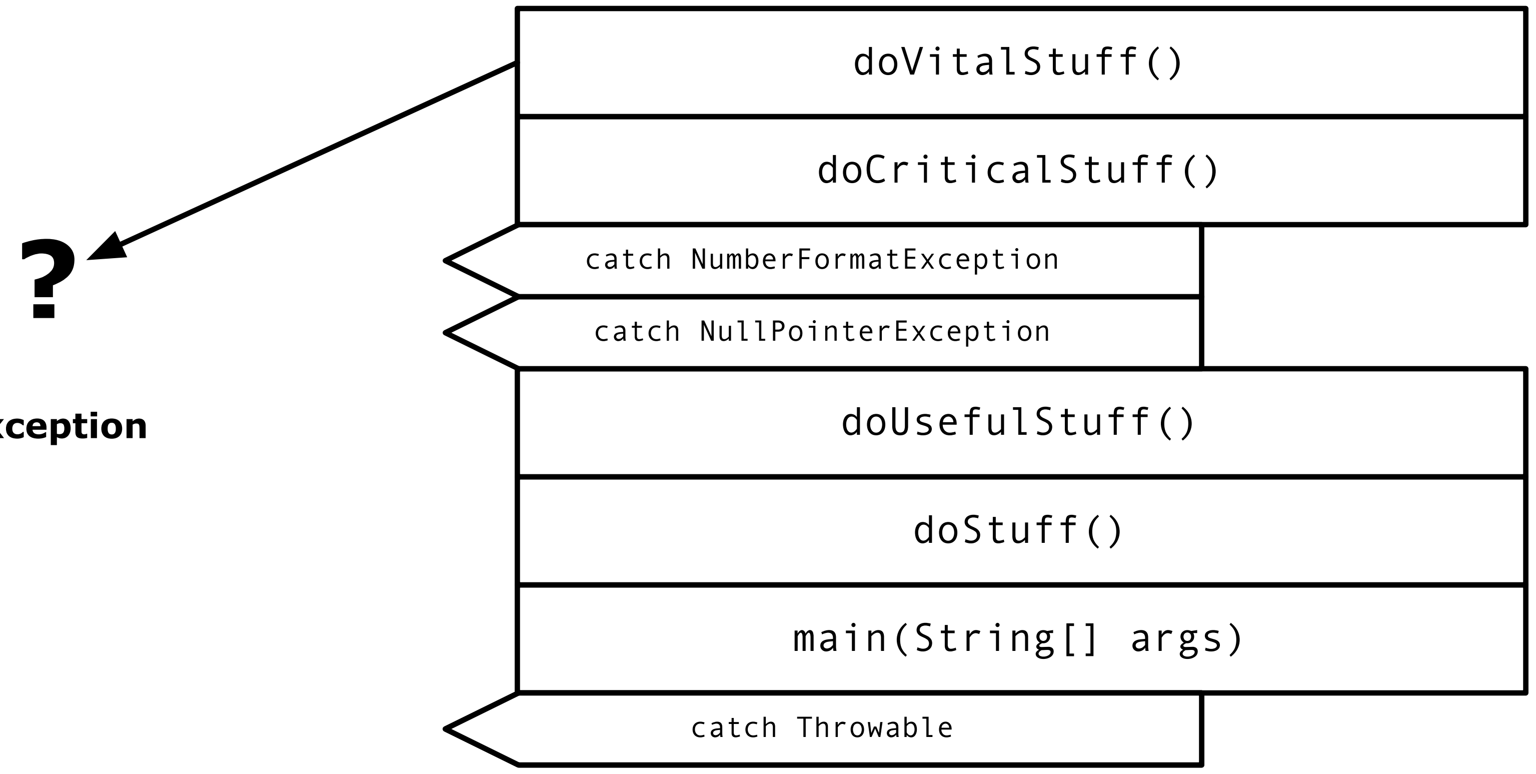
Copyright © 2005-2007 William C. Benton



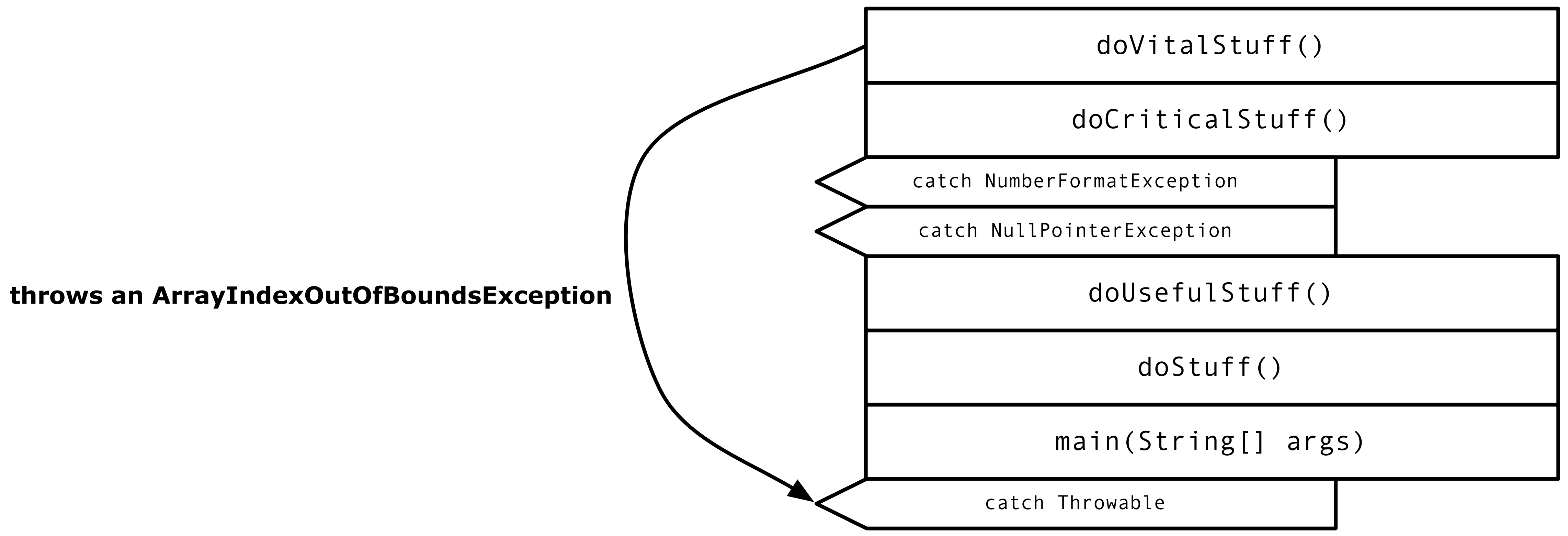
throws a NullPointerException
loadSomeStuff()
catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

Copyright © 2005-2007 William C. Benton


catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

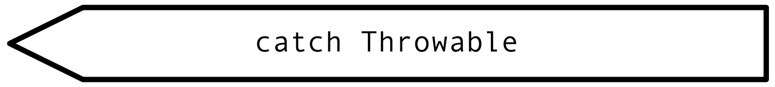


throws an ArrayIndexOutOfBoundsException


doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
throws an ArrayIndexOutOfBoundsException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

Copyright © 2005-2007 William C. Benton

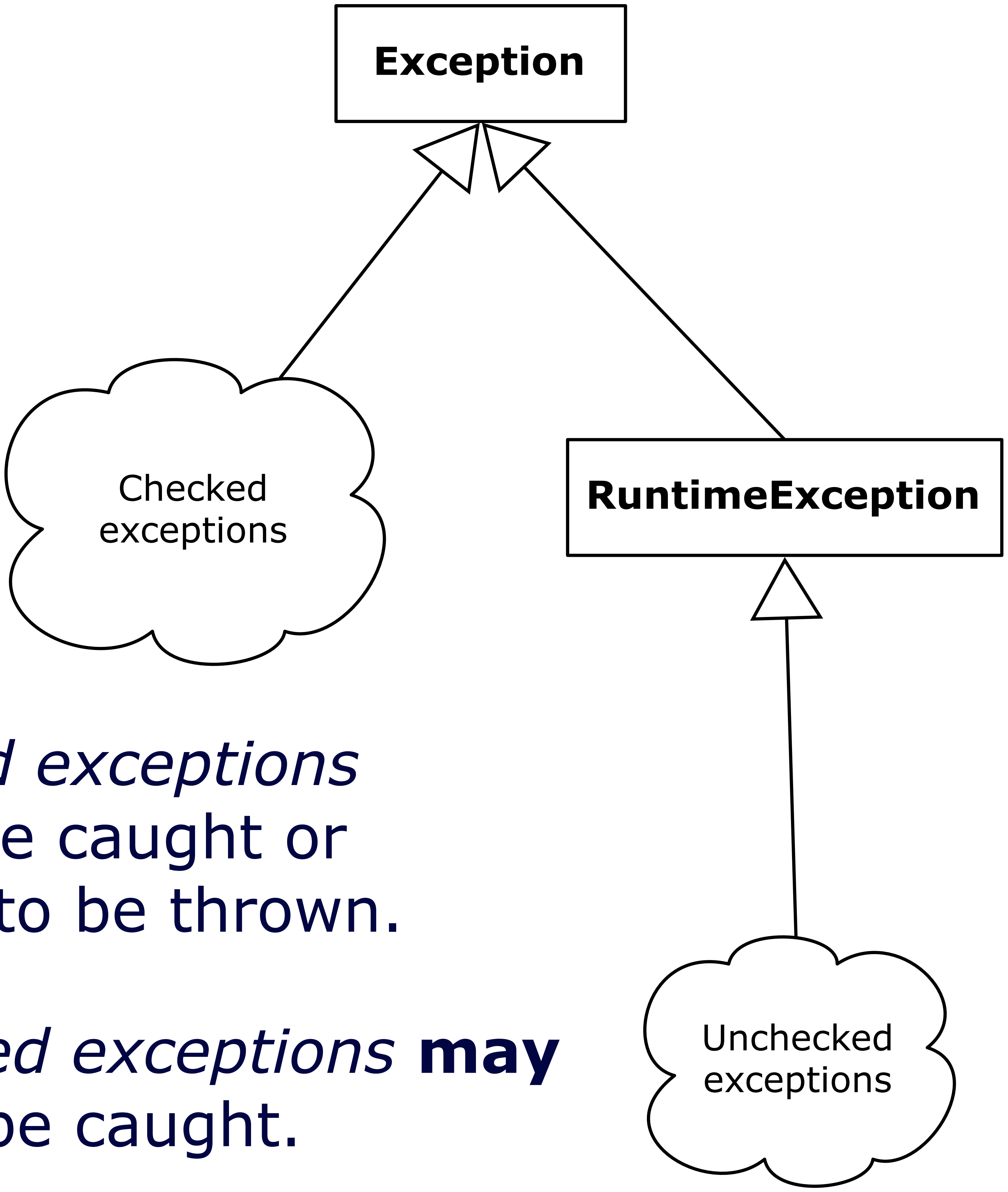
Exceptions are a one-way street: once you’ve exited main() (or any other method), you can’t go back.



(This program has terminated.)

There are two kinds of exceptions: *checked* and *unchecked*.

Copyright © 2005-2007 William C. Benton



Checked exceptions **must** be caught or declared to be thrown.

Unchecked exceptions **may** be caught.

Copyright © 2005-2007 William C. Benton

Why do we have this distinction?

Copyright © 2005-2007 William C. Benton

You **must** either `catch` checked exceptions or declare them in a *throws clause*.

```
public void doDangerousStuff()
  throws HazardException,
  ComputerCombustionException {

    // ...

}
```

Copyright © 2005-2007 William C. Benton

Signature vs. prototype

- Method signature: name of method + list of parameter types
- Method prototype: signature + throws clause
- Java compiler uses prototype to determine which method calls might throw exceptions

Copyright © 2005-2007 William C. Benton

Any questions?

Copyright © 2005-2007 William C. Benton

Rolling your own

- You can make your own exception types by extending `Exception` or `RuntimeException`
  - You can also implement `Throwable`
- What are some exceptions you might use in your programs?

Copyright © 2005-2007 William C. Benton

Exercise

- Say you need to develop an ice-cream cone class with an eat(int) method
- Write code to throw an exception if a class user attempts to eat a negative number of scoops
- Should this be checked or not?

Copyright © 2005-2007 William C. Benton

Sometimes, we want to execute some code after a block, whether or not it causes an exception.

Copyright © 2005-2007 William C. Benton

```
Scanner f = null;

try {
   f = new Scanner(“file.txt”);
} catch (IOException ioe) {
   if (f != null) f.close();
}

if (f != null) {
   try {
      f.close();
   } catch (Exception e) {}
}
```

Copyright © 2005-2007 William C. Benton

Statements in a *finally block* are executed whether or not the `try` block threw an exception.

Copyright © 2005-2007 William C. Benton

```
try
    stmt_try
finally
    stmt_finally
```

Copyright © 2005-2007 William C. Benton

Why are we interested in this capability?

Copyright © 2005-2007 William C. Benton

Review

What is an exception?

Copyright © 2005-2007 William C. Benton

How do we generate an exception?

Copyright © 2005-2007 William C. Benton

```
throw ref;
```

Copyright © 2005-2007 William C. Benton

`throw` *ref*;

(*ref* must be a reference to an instance of a class that extends `Exception` or implements `Throwable`)

Copyright © 2005-2007 William C. Benton

How do we handle exceptional conditions?

Copyright © 2005-2007 William C. Benton

```
try
    stmt_try
catch (type id)
    stmt_catch
```

Copyright © 2005-2007 William C. Benton

```
try
    stmt_try
catch (type id)
    stmt_catch
```

(*type* must be an exception type; i.e. compatible with `Exception` or `Throwable`)

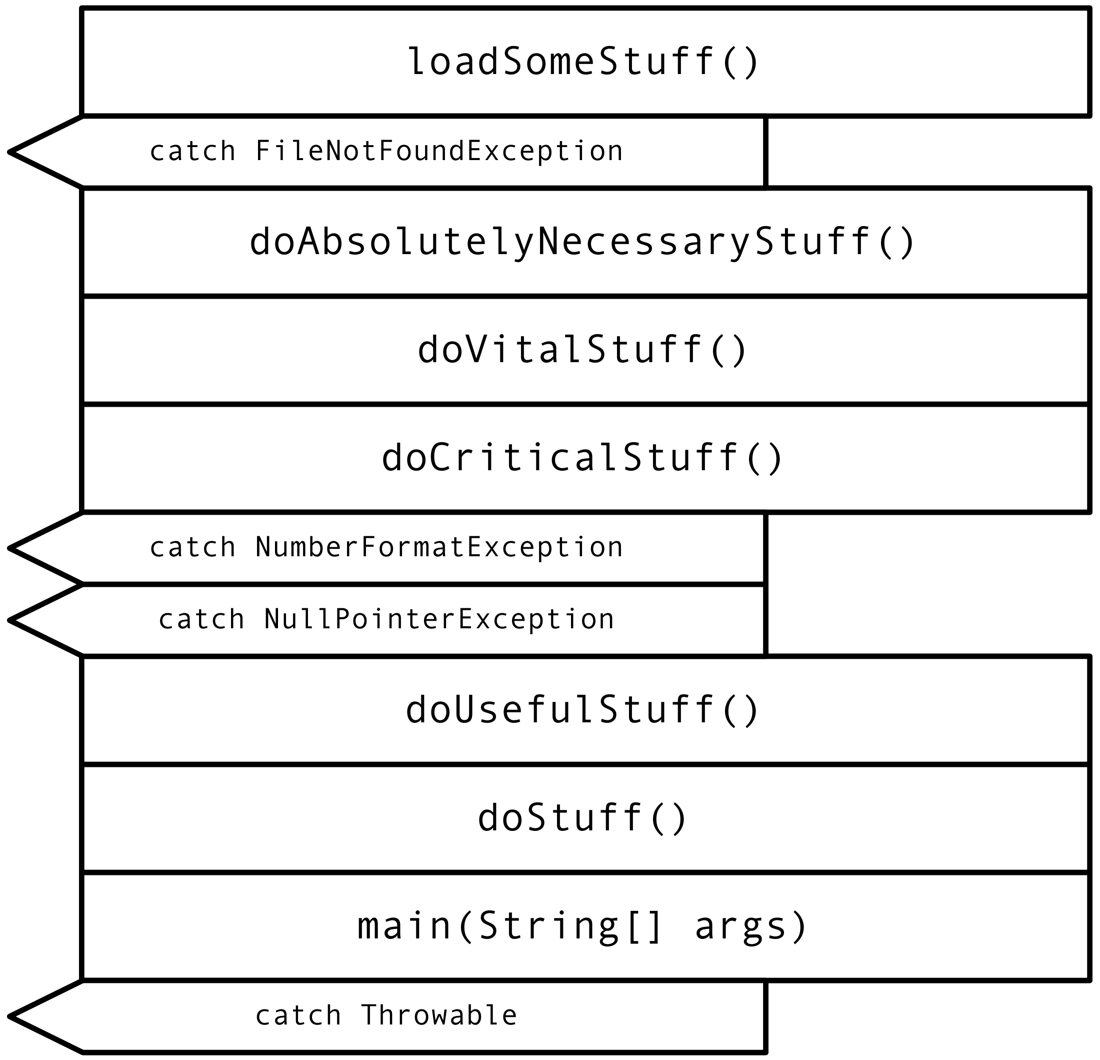
Copyright © 2005-2007 William C. Benton

What happens when an exception is thrown?

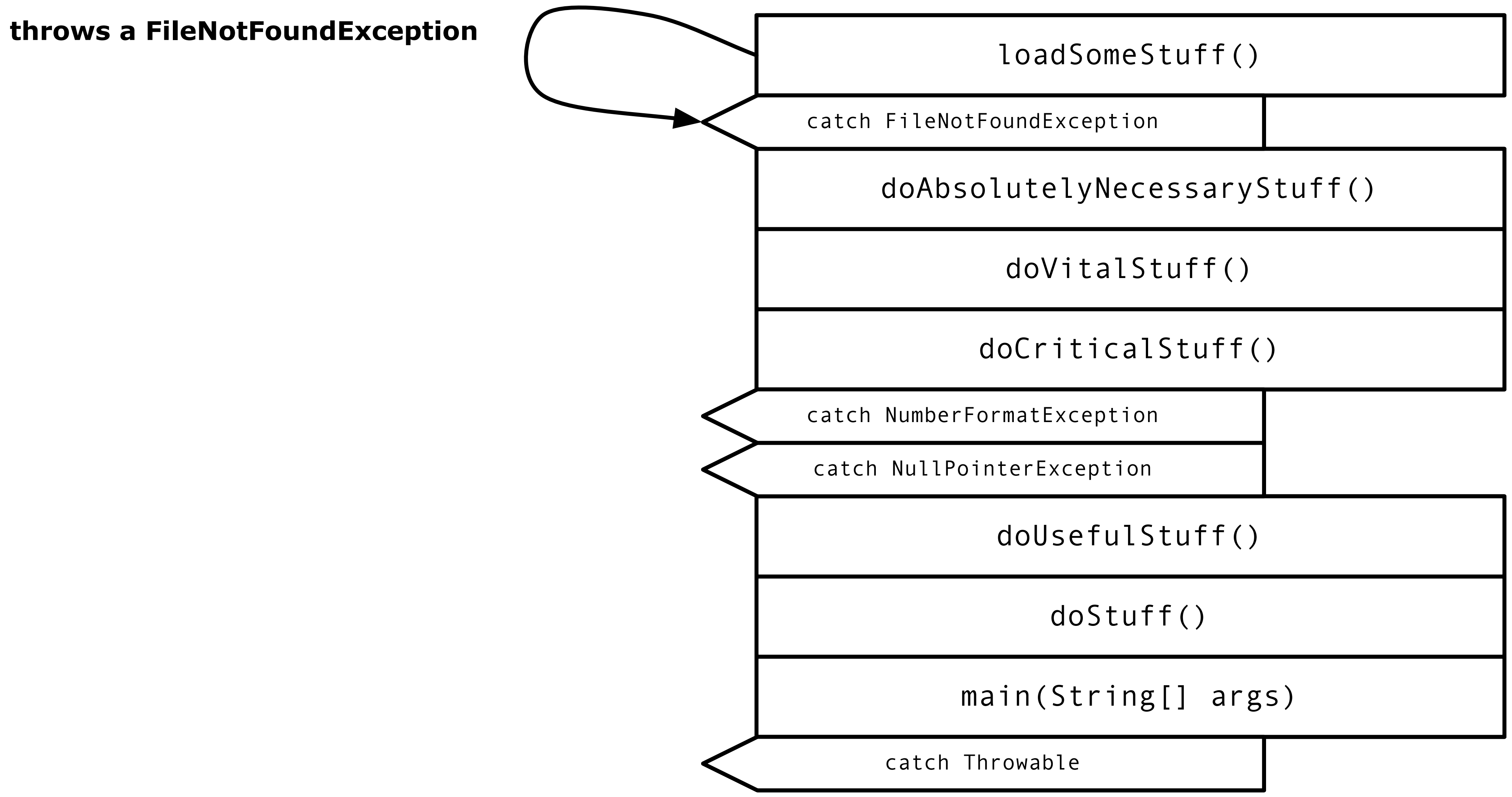
Copyright © 2005-2007 William C. Benton

Control transfers to the nearest enclosing *catch block* that can catch that type of exception.

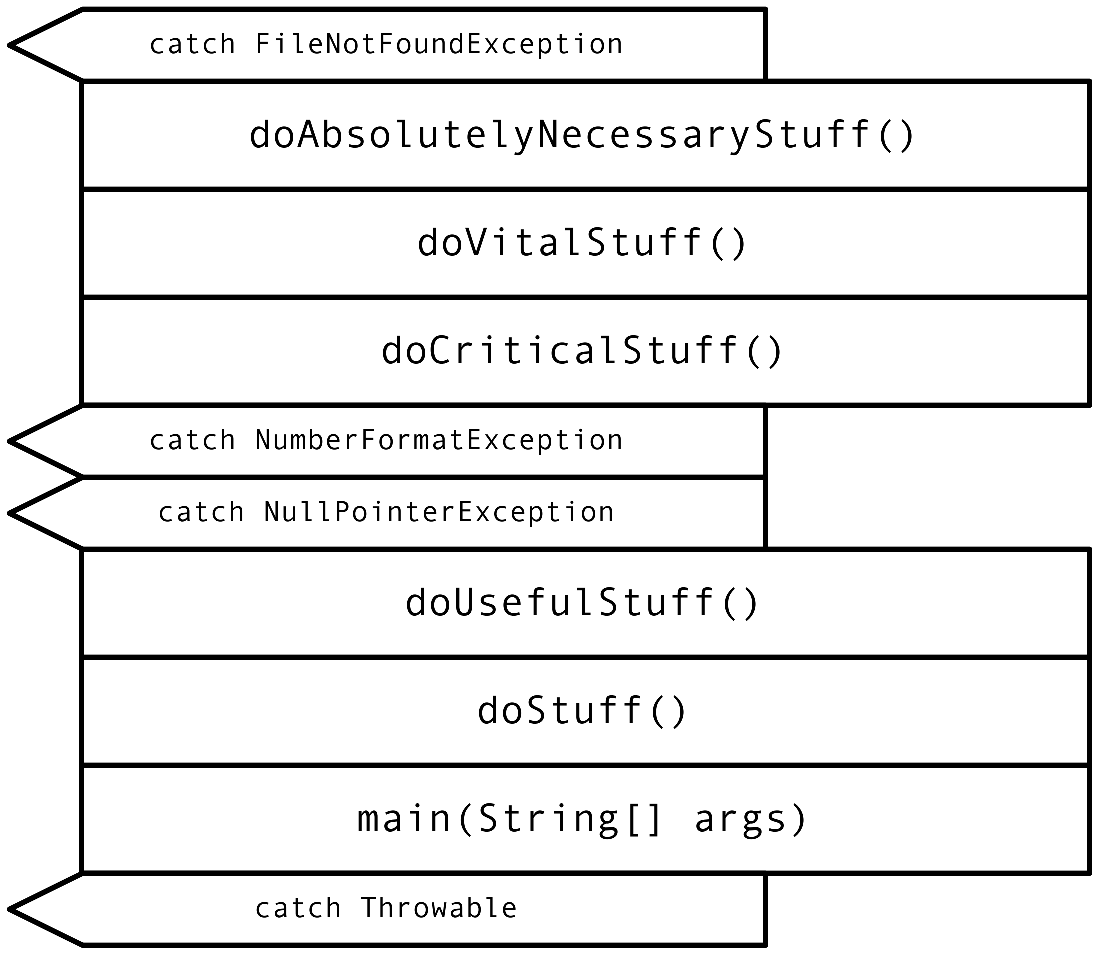
Copyright © 2005-2007 William C. Benton


loadSomeStuff()
catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

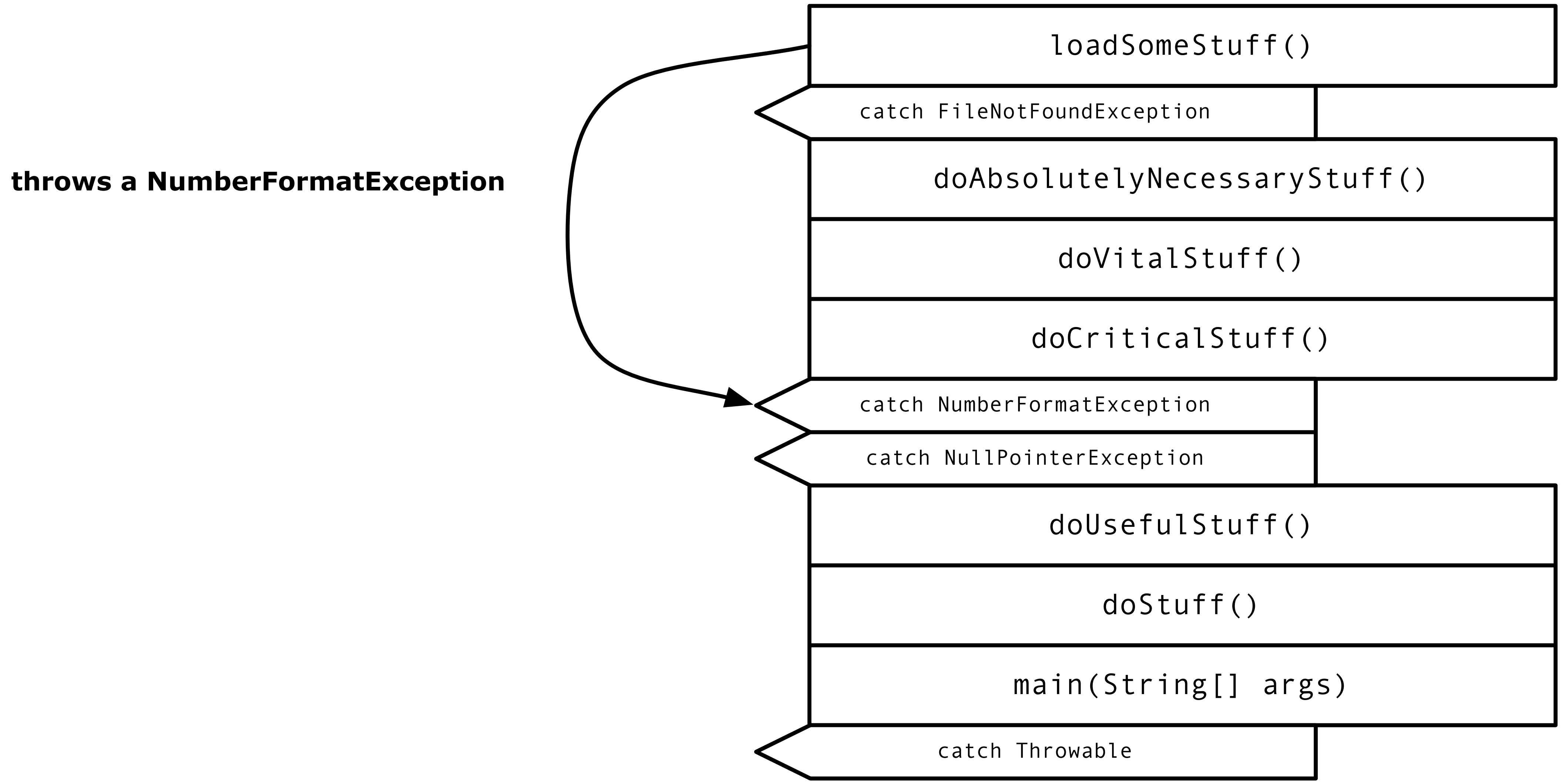
Copyright © 2005-2007 William C. Benton

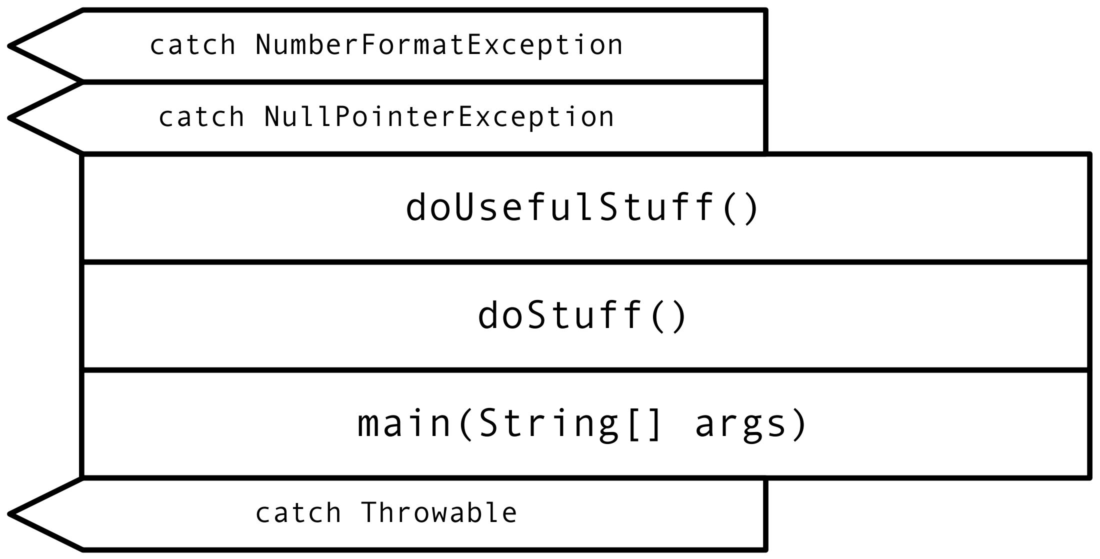

throws a FileNotFoundException
loadSomeStuff()
catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

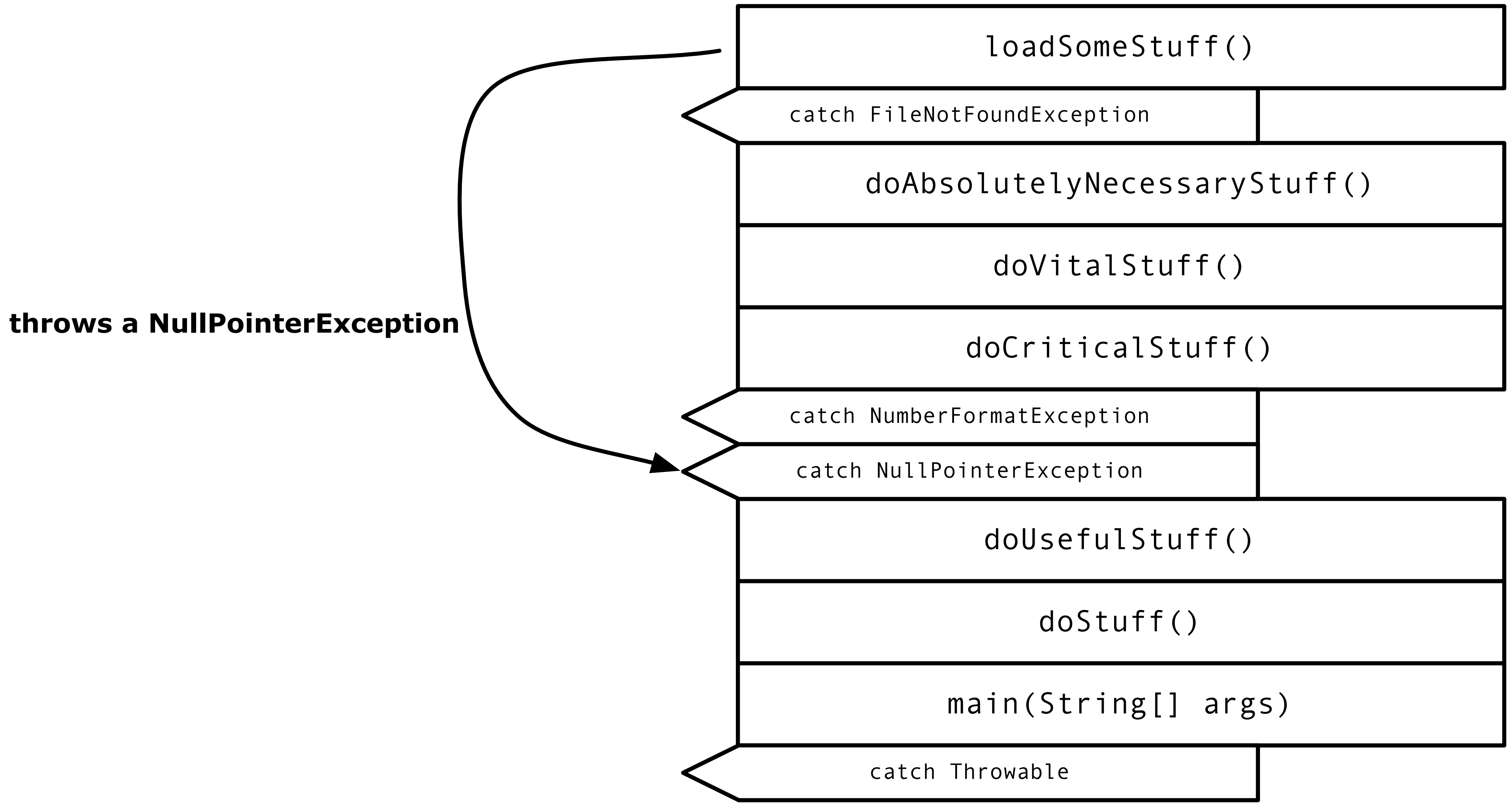
Copyright © 2005-2007 William C. Benton

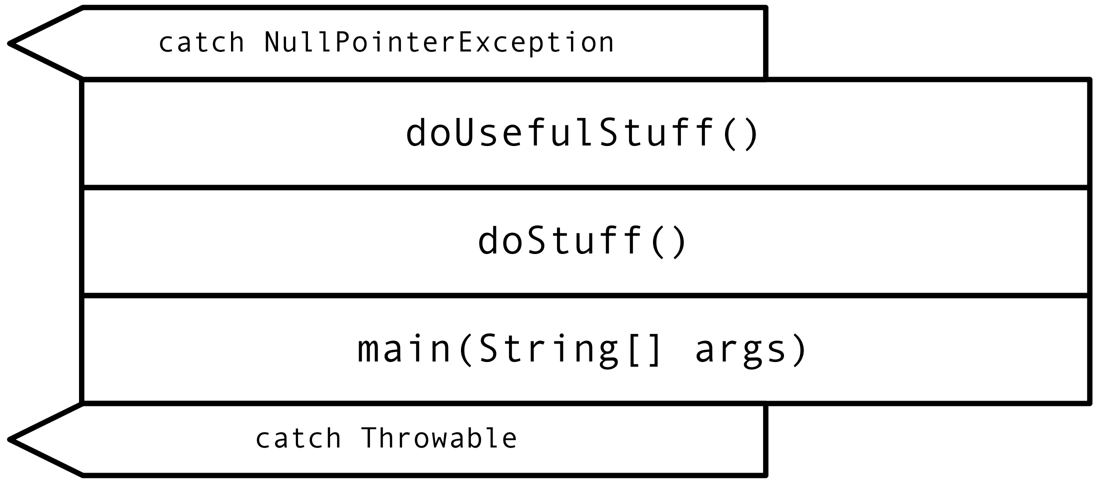

catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

Copyright © 2005-2007 William C. Benton


throws a NumberFormatException
loadSomeStuff()
catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

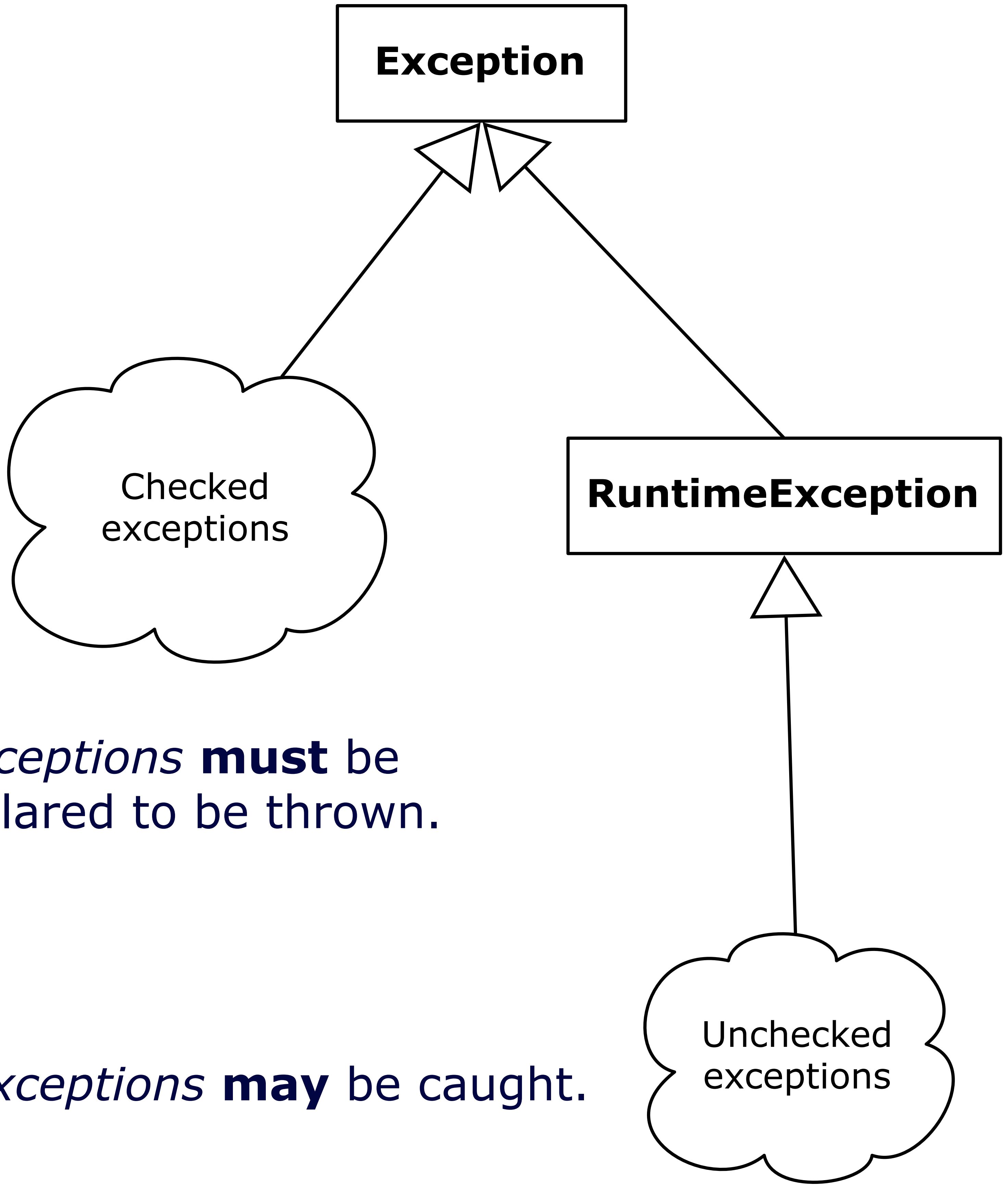

catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable


loadSomeStuff()
catch FileNotFoundException
doAbsolutelyNecessaryStuff()
doVitalStuff()
throws a NullPointerException
doCriticalStuff()
catch NumberFormatException
catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable


catch NullPointerException
doUsefulStuff()
doStuff()
main(String[] args)
catch Throwable

What are the two kinds of exceptions?

Copyright © 2005-2007 William C. Benton



Checked exceptions **must** be caught or declared to be thrown.

Unchecked exceptions **may** be caught.

Copyright © 2005-2007 William C. Benton

Rolling your own

- You can make your own exception types by extending `Exception` or `RuntimeException`
  - You can also implement `Throwable`
- What are some exceptions you might use in your programs?

Copyright © 2005-2007 William C. Benton

Exercise

- Say you need to develop an ice-cream cone class with an eat(int) method
- Write code to throw an exception if a class user attempts to eat a negative number of scoops
- Should this be checked or not?

Copyright © 2005-2007 William C. Benton

Sometimes, we want to execute some code after a block, whether or not it causes an exception.

Copyright © 2005-2007 William C. Benton

```
Scanner f = null;

try {
   f = new Scanner("file.txt");
} catch (IOException ioe) {
   if (f != null) f.close();
}

if (f != null) {
   try {
      f.close();
   } catch (Exception e) {}
}

/* what happens if some other
exception is thrown? */
```

Copyright © 2005-2007 William C. Benton

Statements in a *finally block* are executed whether or not the `try` block threw an exception.

Copyright © 2005-2007 William C. Benton

```
try
  stmt_try
finally
  stmt_finally
```

```
try
  stmt_try
catch (type id)
  stmt_catch
finally
  stmt_finally
```

Copyright © 2005-2007 William C. Benton

Why are we interested in this capability?

Copyright © 2005-2007 William C. Benton