Abstraction-Safe Effect Handlers via Tunneling

Yizhou Zhang
Cornell University

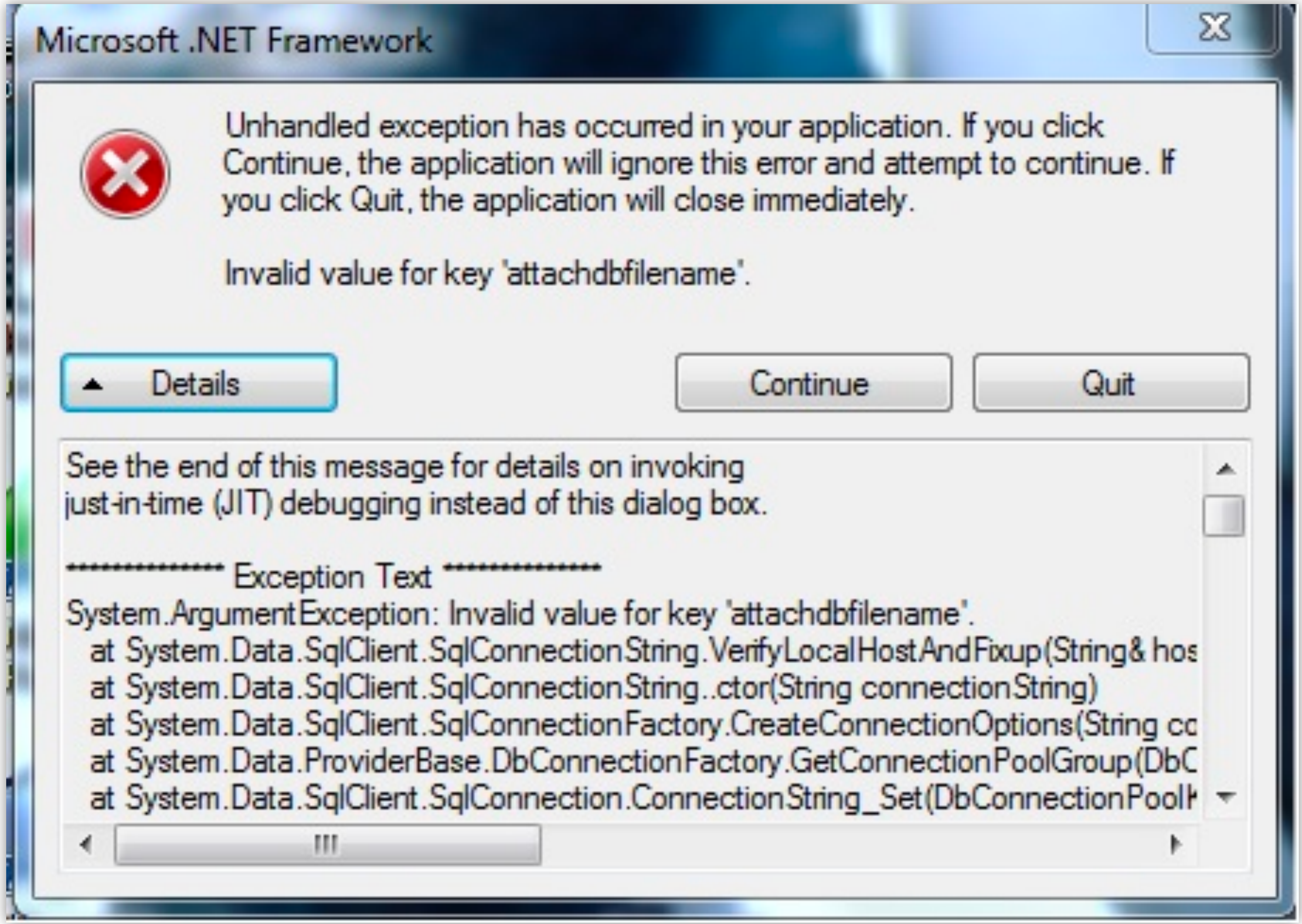
Exception handling

```
try {
    … // normal-case code
} catch (SomeException e) {
    … // exception-handling code
}
```

Separation of concerns

Programs crash when programmers forget to handle exceptions

⇨ Statically checking exceptions



To check, or not to check, that is the question…

Hacker News — https://news.ycombinator.com

To check exceptions

I quite like checked exceptions. They're actually one of the few things that keeps me writing java for more projects instead of switching more of my time to golang

Checked exceptions are great, but **unchecked exceptions are a huge disaster** only mitigated by

Merely that there's **a case for checked exceptions**, especially if you are actually trying to write "**reliable**" software.

4**. Unchecked exceptions are a production run-time nightmare**. My anecdotal but extensive experience is that

Not to check exceptions

Checked exceptions I love you, but you have to go

39 points | fogus | 9 years ago | 35 comments

Java's **Checked Exceptions Are Evil**? (2015)

55 points | tosh | 5 months ago | 86 comments | (https://blog.philipphauer.de**/checl**

Java's **checked exceptions were a mistake** (2003)

2 points | alexk | 9 years ago | 0 comments

Checked exceptions are indeed obnoxious and a major language design failure, which is why pretty

The big problem with Java checked exceptions was that the language was simply **not flexible enough in other areas to handle higher-order stuff**. For

To check, or not to check, that is the question…

exceptions

I quite like checked exceptions. They're actually one of the few things that keeps me writing java for more projects instead of switching more of my time to golang

Checked exceptions are great, but unchecked exceptions are a huge disaster only mitigated by

Merely that there's **a case for checked exceptions**, especially if you are actually trying to write "**reliable**" software.

4**. Unchecked exceptions are a production run-time nightmare**. My anecdotal but extensive experience is that

Checked exce
39 points | fogus

Java's Checked
55 points | tosh | 5 m

Java's checked
2 points | alexk | 9 ye

Checked exc
major langua

The big proble
that the langu
in other area

To check, or not to check, that is the question…

To check, or not to check, that is the question…

Checked exceptions are too rigid to work with higher-order programming!

Sat, 2011-02-05, 13:47

odersky

Re: Re: Add compiler warning for checked exception

The problem with checked exceptions is best demonstrated by the map method on lists:

def map[B](f: A => B): List[B]

How to annotate map with @throws? If map does not get a @throws annotation itself then presumably you cannot pass it any function t has a @throws. That would introduce cumbersome restrictions and distinctions for the ways in which map can be used. Things would

To check, or not to check, that is the question…

Checked exceptions are too rigid to work with higher-order programming!

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(File file) throws IOException {…}
```

Checked exceptions are too rigid

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> l, X→Y f) {…}
```

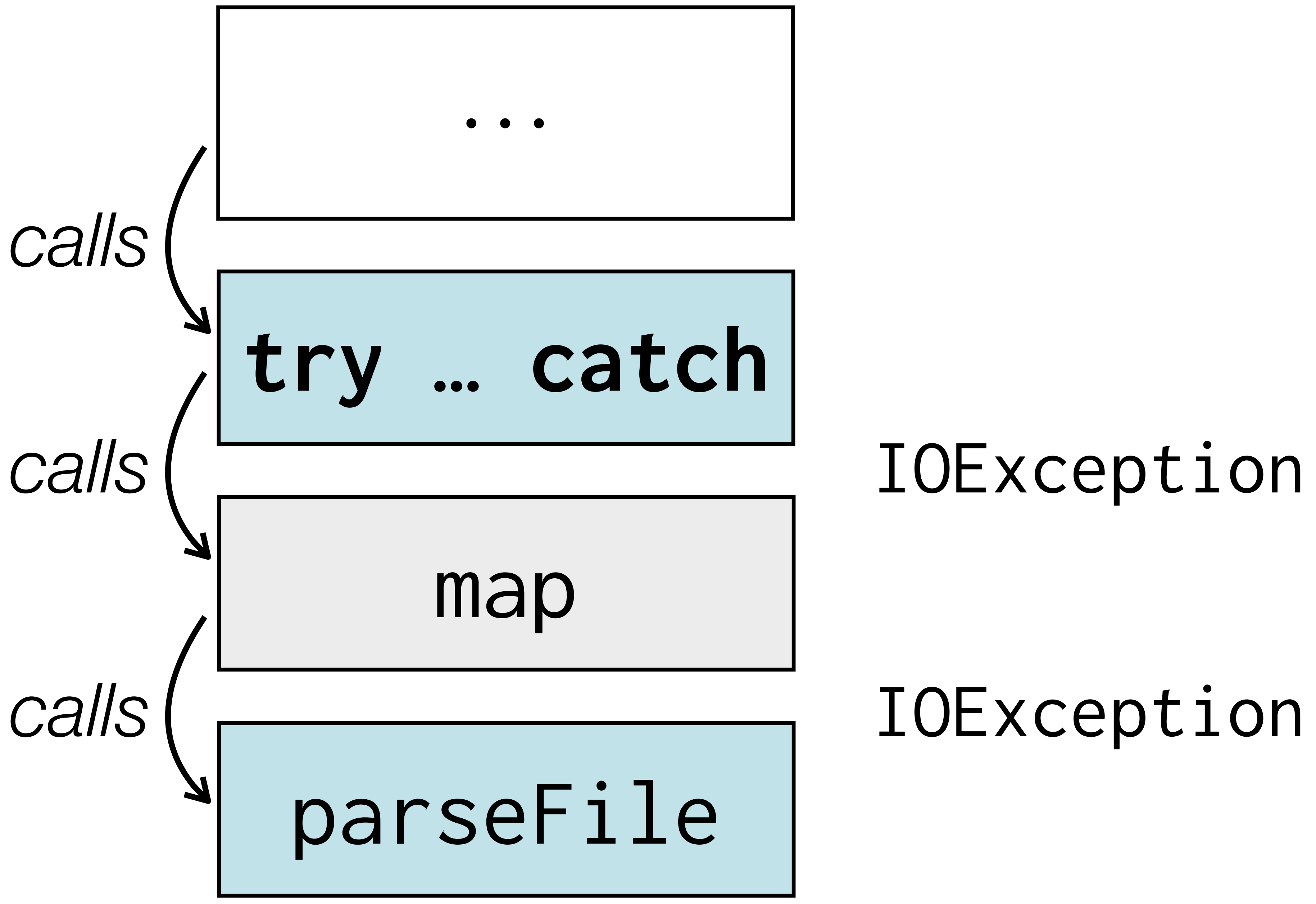
Raises `IOException` if the file does not exist

```
Tree parseFile(File file) throws IOException {…}
```

Client code

```
List<File> files = …;
List<Tree> trees;
try {
   trees = map(files, f->parseFile(f));
   …
} catch (IOException e) {
   …
}
```

Compile-time error



Checked exceptions are too rigid

A checked exception class in Java

```
class IOException extends Exception {…}
```

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> l, X→Y f
```

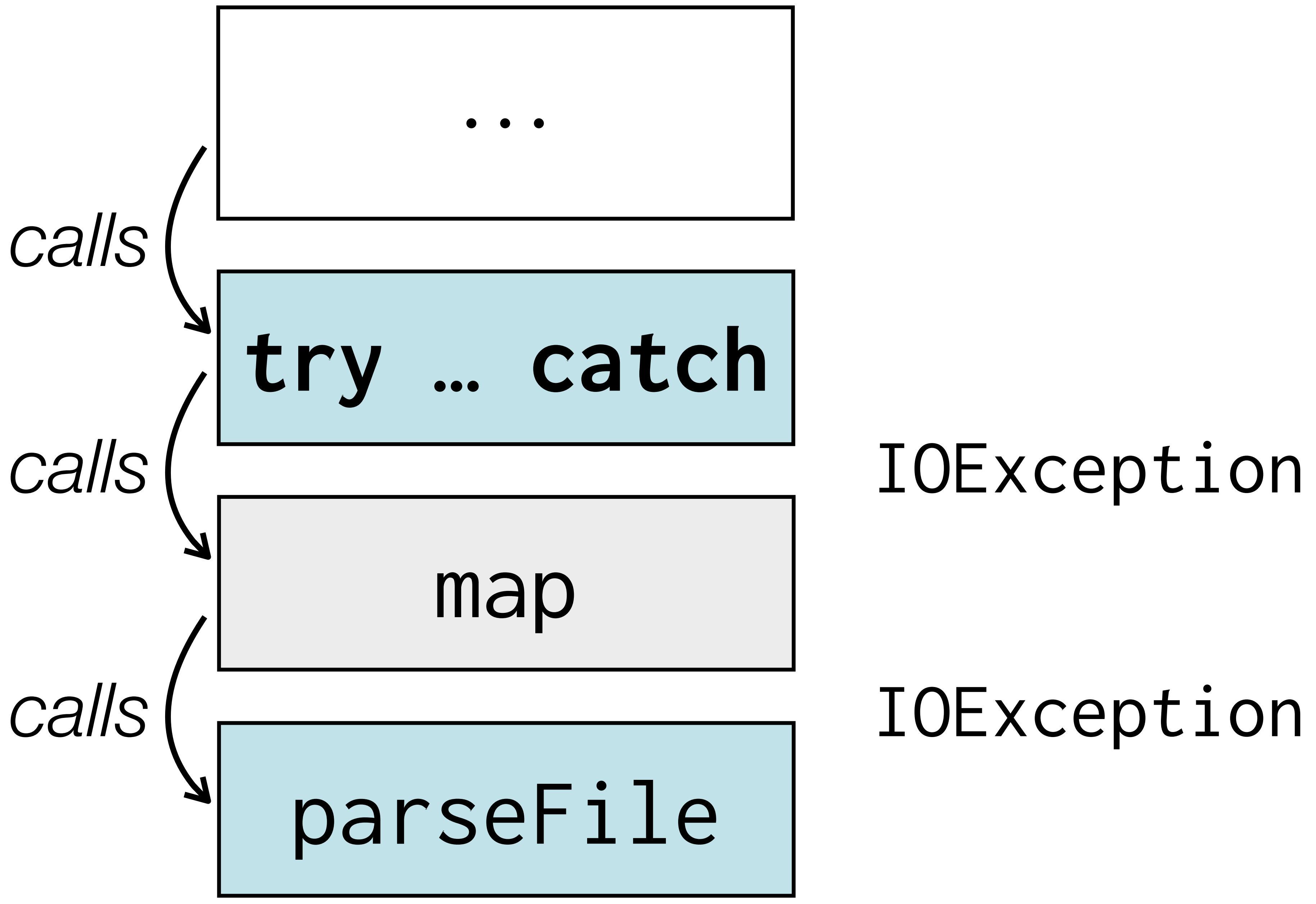
Raises `IOException` if the file does not exist

```
Tree parseFile(File file) throws IOException {…}
```

Client code

```
List<File> files = …;
List<Tree> trees;
try {
  trees = map(files, f->parseFile(f));
  …
} catch (IOException e) {
  …
}
```

Compile-time error



Checked exceptions are too rigid

A checked exception class in Java

```
class IOException extends Exception {…}
```

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> l, X→Y f
```

Raises `IOException` if the file does not exist

```
Tree parseFile(File file) throws IOException {…}
```

Client code

```
List<File> files = …;
List<Tree> trees;
try {
  trees = map(files, f->parseFile(f));
  …
} catch (IOException e) {
  …
}
```

Compile-time error: exception mismatch

Lambda-expression has type

`File → Tree throws IOException`

but a function of type

`File → Tree`

is expected.

Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

```
class UncheckedIOException extends RuntimeException {…}
```

A higher-order function i

```
<X,Y> List<Y> map(L
```

Raises `IOException` if the file does not exist

```
Tree parseFile(File file) throws IOException {…}
```

Client code

```
List<File> files = …;
List<Tree> trees;
try {
   trees = map(files, f->parseFile(f));
   …
} catch (IOException e) {
   …
}
```

Compile-time error: exception mismatch

Lambda-expression has type

`File → Tree throws IOException`

but a function of type

`File → Tree`

is expected.

Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

```
class UncheckedIOException extends RuntimeException {…}
```

```
List<File> files = …;
List<Tree> trees;
trees = map(files, f->parseFile(f));
…
```

Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

```
class UncheckedIOException extends RuntimeException {…}
```

```
List<File> files = …;
List<Tree> trees;
trees = map(files, f->{
  try { return parseFile(f); }
  catch (IOException e) {
    throw new UncheckedIOException(e);
  });
…
```

exception wrapping

Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

```
class UncheckedIOException extends RuntimeException {…}
```

```
List<File> files = …;
List<Tree> trees;
try {
  trees = map(files, f->{
    try { return parseFile(f); }
    catch (IOException e) {
      throw new UncheckedIOException(e);
    });
  …
} catch (UncheckedIOException u) {
  IOException e = u.getCause();
  …
}
```

exception wrapping

exception unwrapping

Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

```
class UncheckedIOException extends RuntimeException {…}
```

```
List<File> files = …;
List<Tree> trees;
try {
  trees = map(files, f->{
    try { return parseFile(f); }
    catch (IOException e) {
      throw new UncheckedIOException(e);
    });
  …
} catch (UncheckedIOException u) {
  IOException e = u.getCause();
  …
}
```

exception wrapping

exception unwrapping

Anti-pattern subverts static checking

An unchecked wrapper class introduced in Java 8

```
class UncheckedIOException extends RuntimeException {…}
```

```
List<File> files = …;
List<Tree> trees;
try {
  trees = map(files, f->{
    try { return parseFile(f); }
    catch (IOException e) {
      throw new UncheckedIOException(e);
    });
  …
} catch (UncheckedIOException u) {
  IOException e = u.getCause();
  …
}
```

exception wrapping

exception unwrapping

The tragedy of exceptions

Java

1990

2000

2010

The tragedy of exceptions

“More thinking is needed before we put some kind of checked exceptions mechanism in place.”
—Anders Hejlsberg—

Java

C#

1990

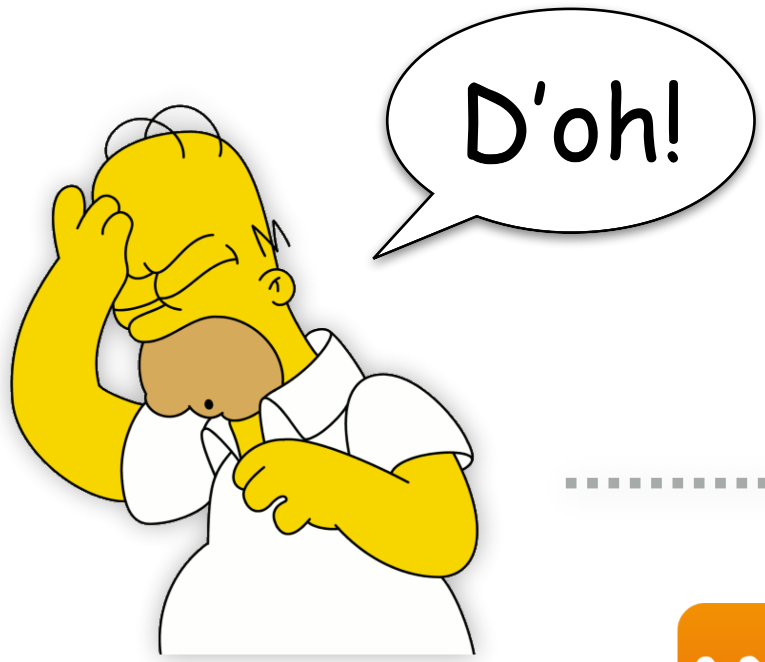
2000

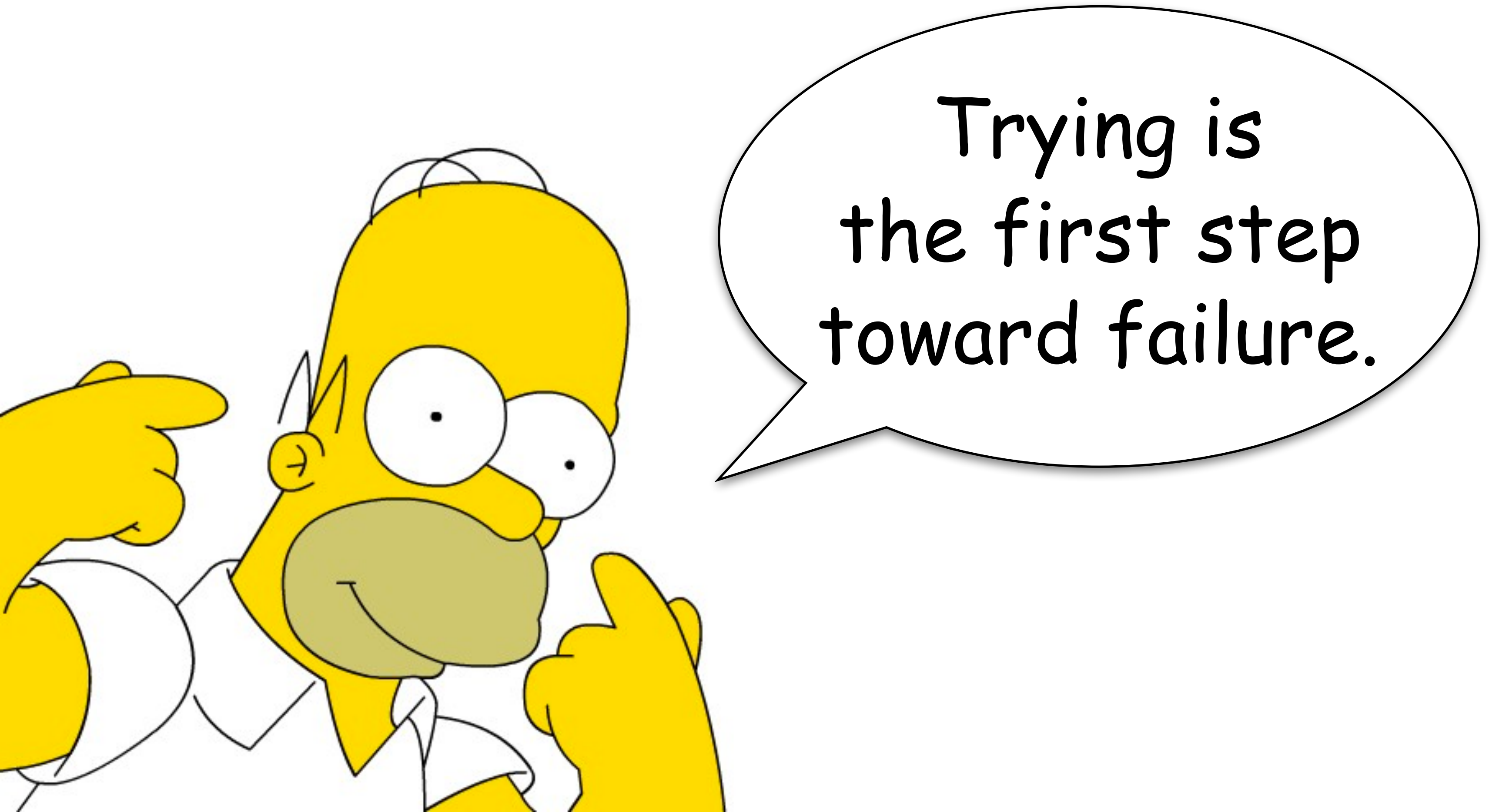
2010

The tragedy of exceptions

"More thinking is needed before we put some kind of checked exceptions mechanism in place."

—Anders Hejlsberg—





Modula-3

Haskell

1990

Java

OCaml

C#

2000

Scala

Ceylon

Dart

Go

2010

TS

Kotlin

Swift

The tragedy of exceptions

Unchecked exceptions lead to uncaught exceptions

Checked exceptions do not work with higher-order abstractions

Use patterns!

Anti-pattern subverts static checking

Unchecked exceptions can be handled by accident

Unchecked exceptions can be handled by accident

An unchecked exception

```
class NoSuchElementException extends RuntimeException {…}
```

Raises `NoSuchElementException` if there is no next element

```
interface Iterator<X> {
  X next() throws NoSuchElementException;
  boolean hasNext();
}
```

Unchecked exceptions can be handled by accident

An unchecked exception

```
class NSE extends RuntimeException {…}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  boolean hasNext();
}
```

Unchecked exceptions can be handled by accident

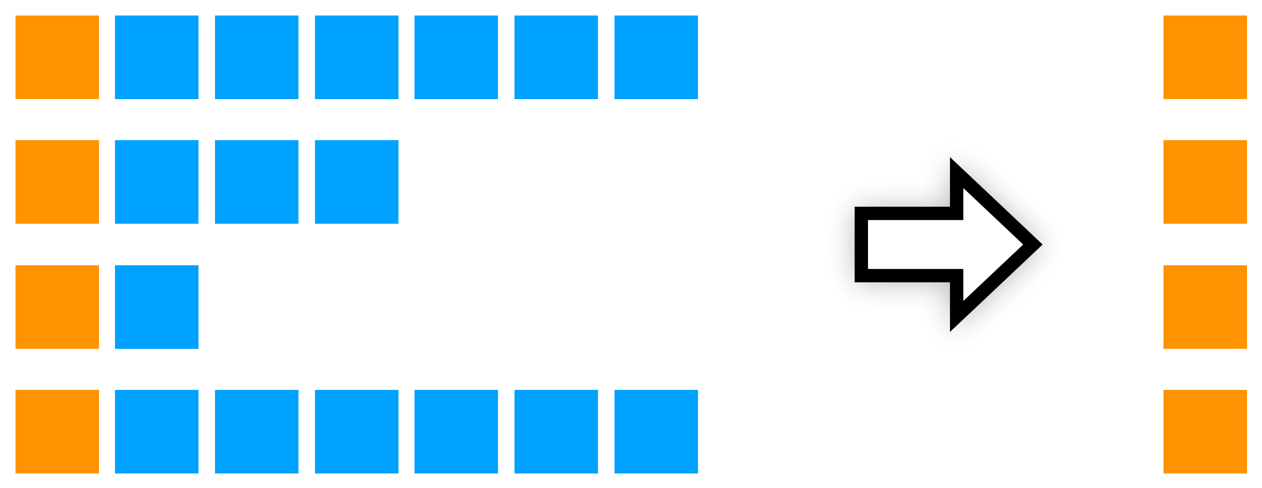
An unchecked exception

```
class NSE extends RuntimeException {…}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  boolean hasNext();
}
```

Programming task:
Transform a List<Iterator<Int>> into List<Int> by fetching the first Int from each Iterator<Int>.



Unchecked exceptions can be handled by accident

An unchecked exception

```
class NSE extends RuntimeException {…}
```

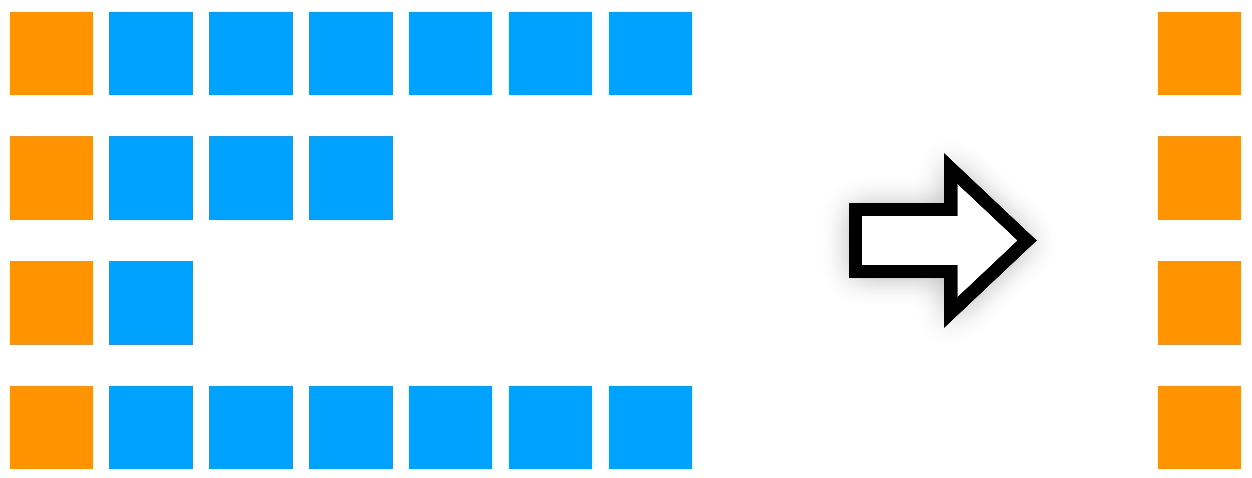
A higher-order function

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {…}
```

Raises `NSE` if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

Programming task:
Transform a `List<Iterator<Int>>` into `List<Int>` by fetching the first `Int` from each `Iterator<Int>`.



Unchecked exceptions can be handled by accident

An unchecked exception

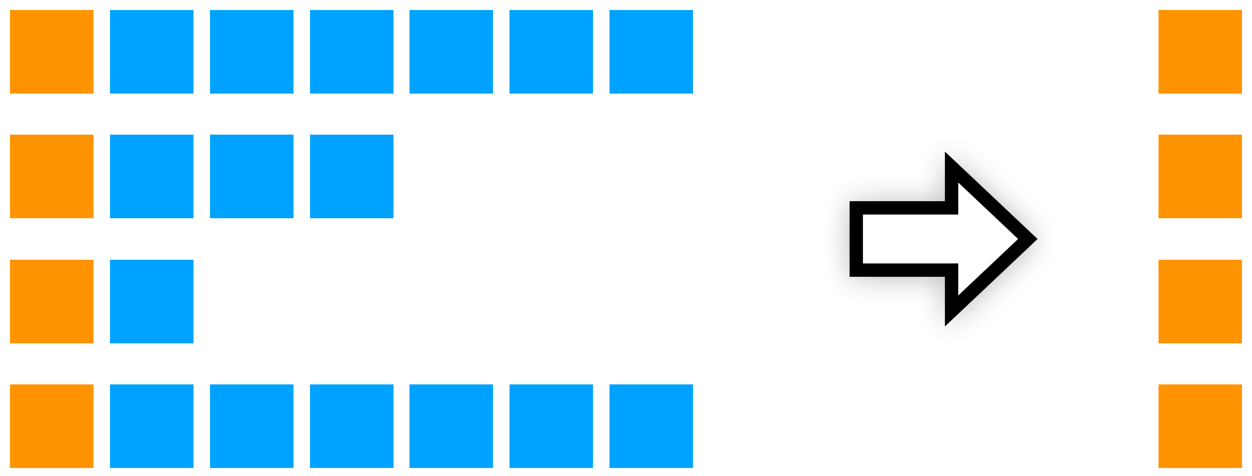
```
class NSE extends RuntimeException {…}
```

A higher-order function

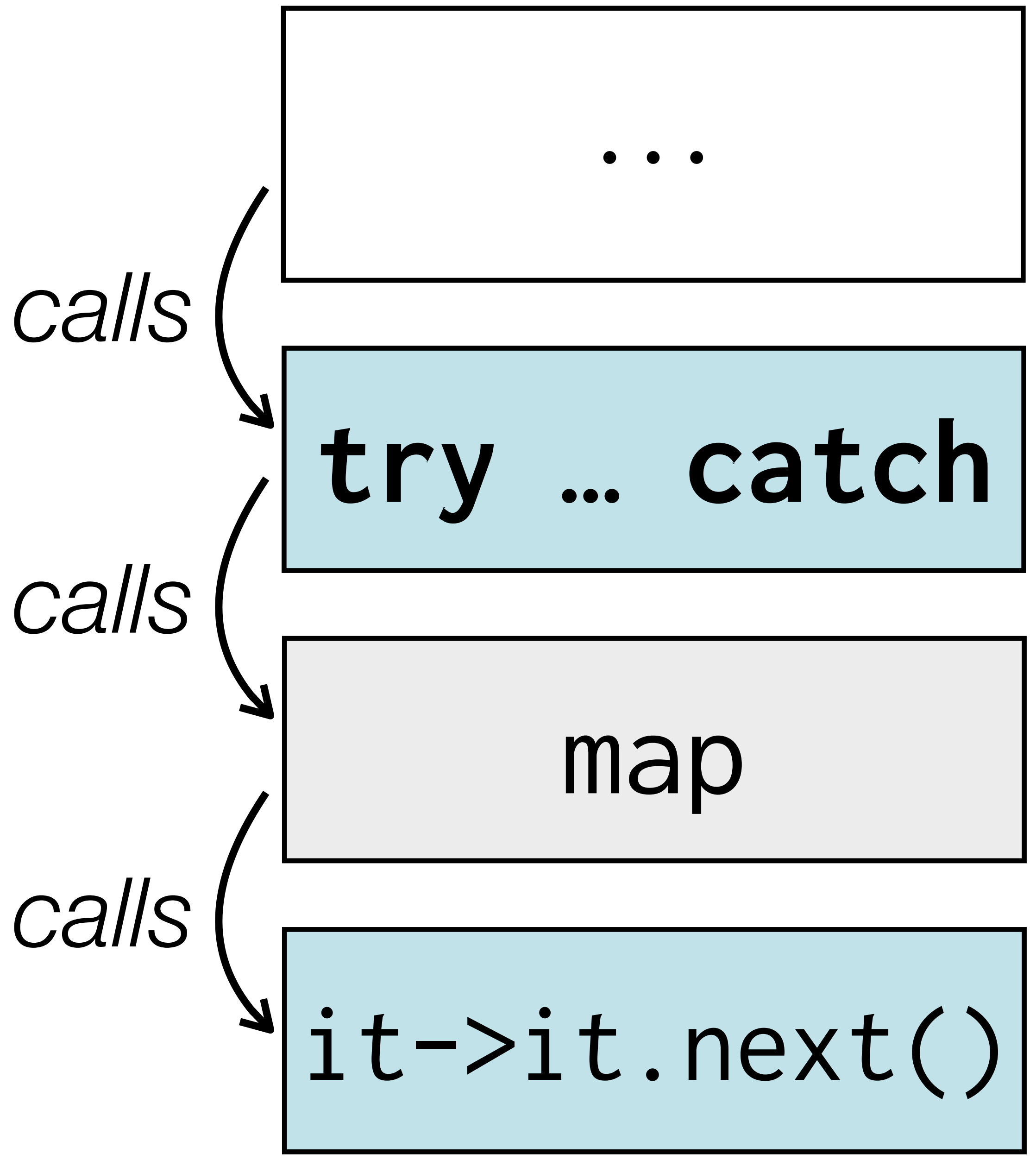
```
<X,Y> List<Y> map(List<X> xs, X→Y f) {…}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```



```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```



Unchecked exceptions can be handled by accident

An unchecked exception

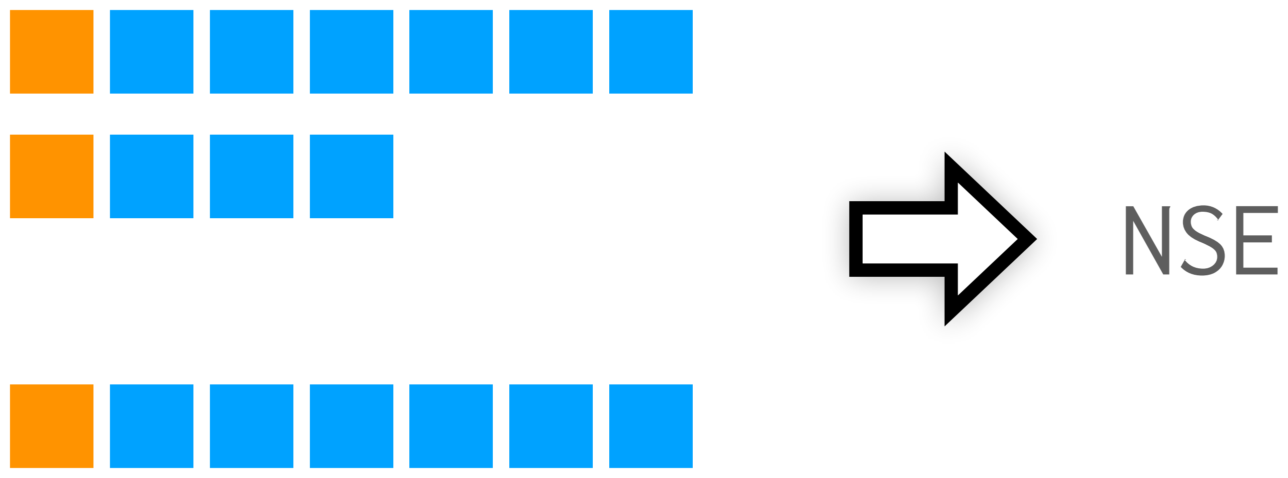
```
class NSE extends RuntimeException {…}
```

A higher-order function

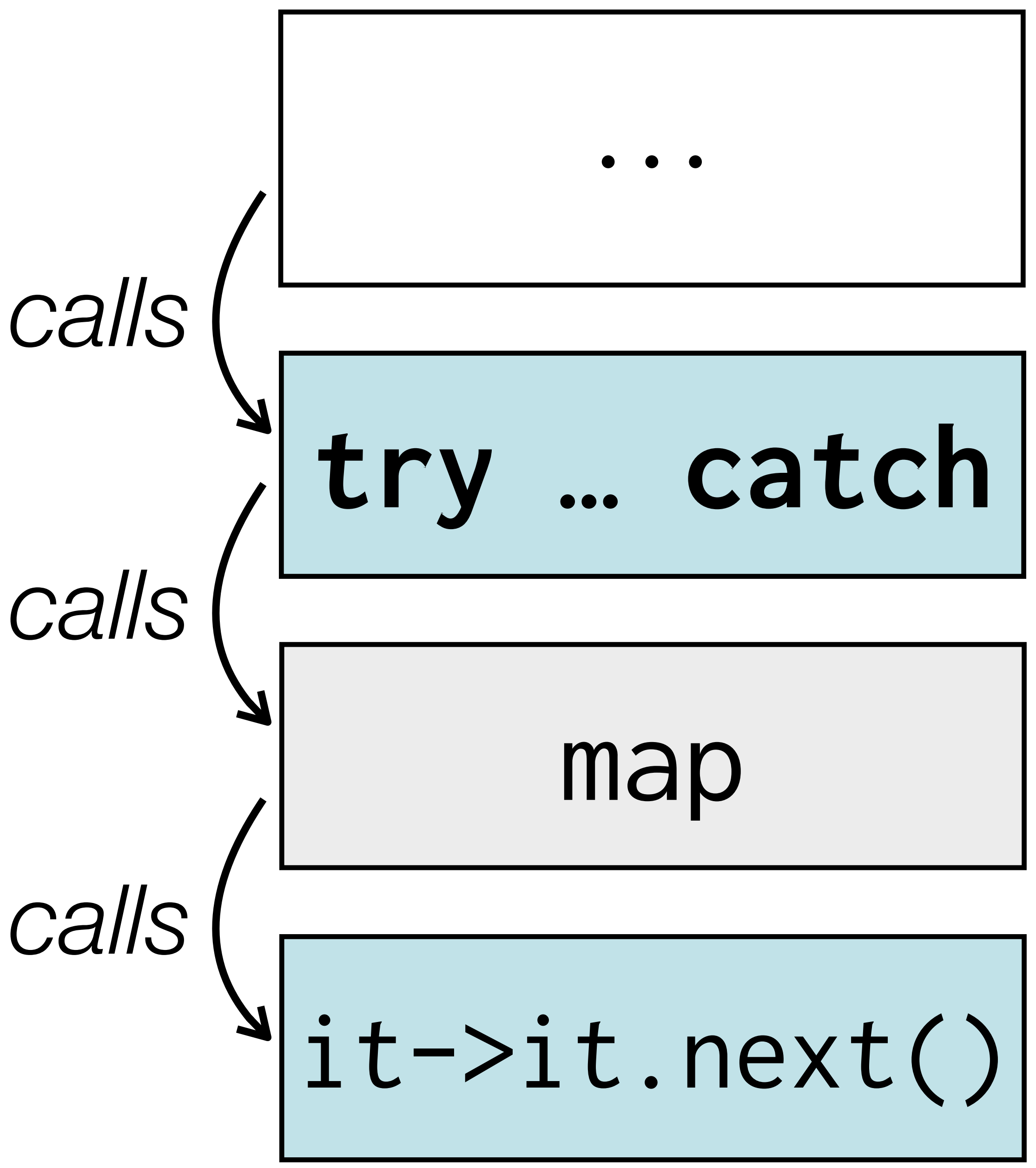
```
<X,Y> List<Y> map(List<X> xs, X→Y f) {…}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```




```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```



Unchecked exceptions can be handled by accident

An unchecked exception

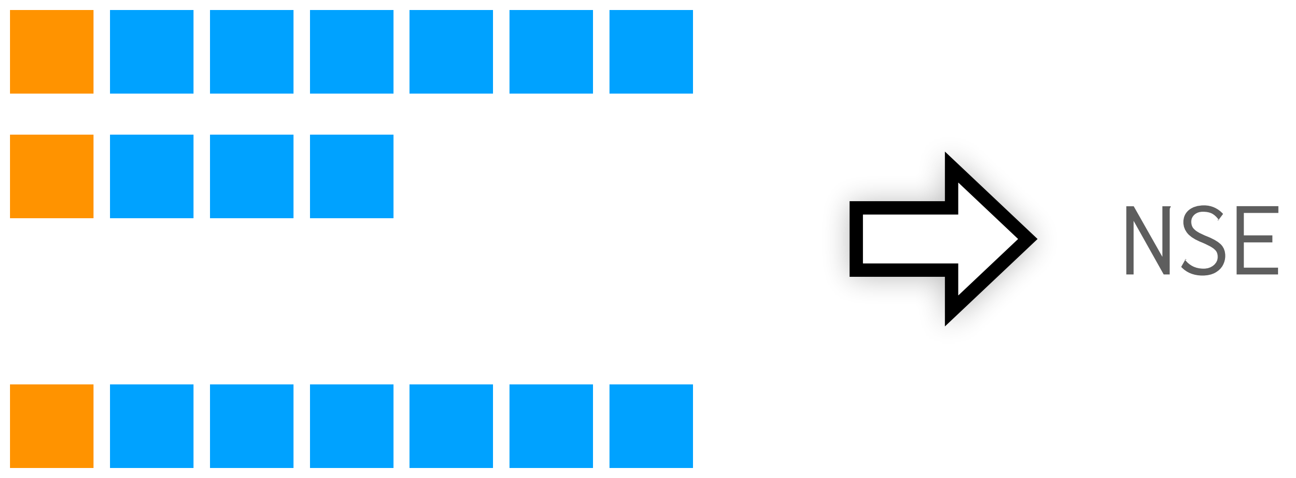
```
class NSE extends RuntimeException {…}
```

A higher-order function

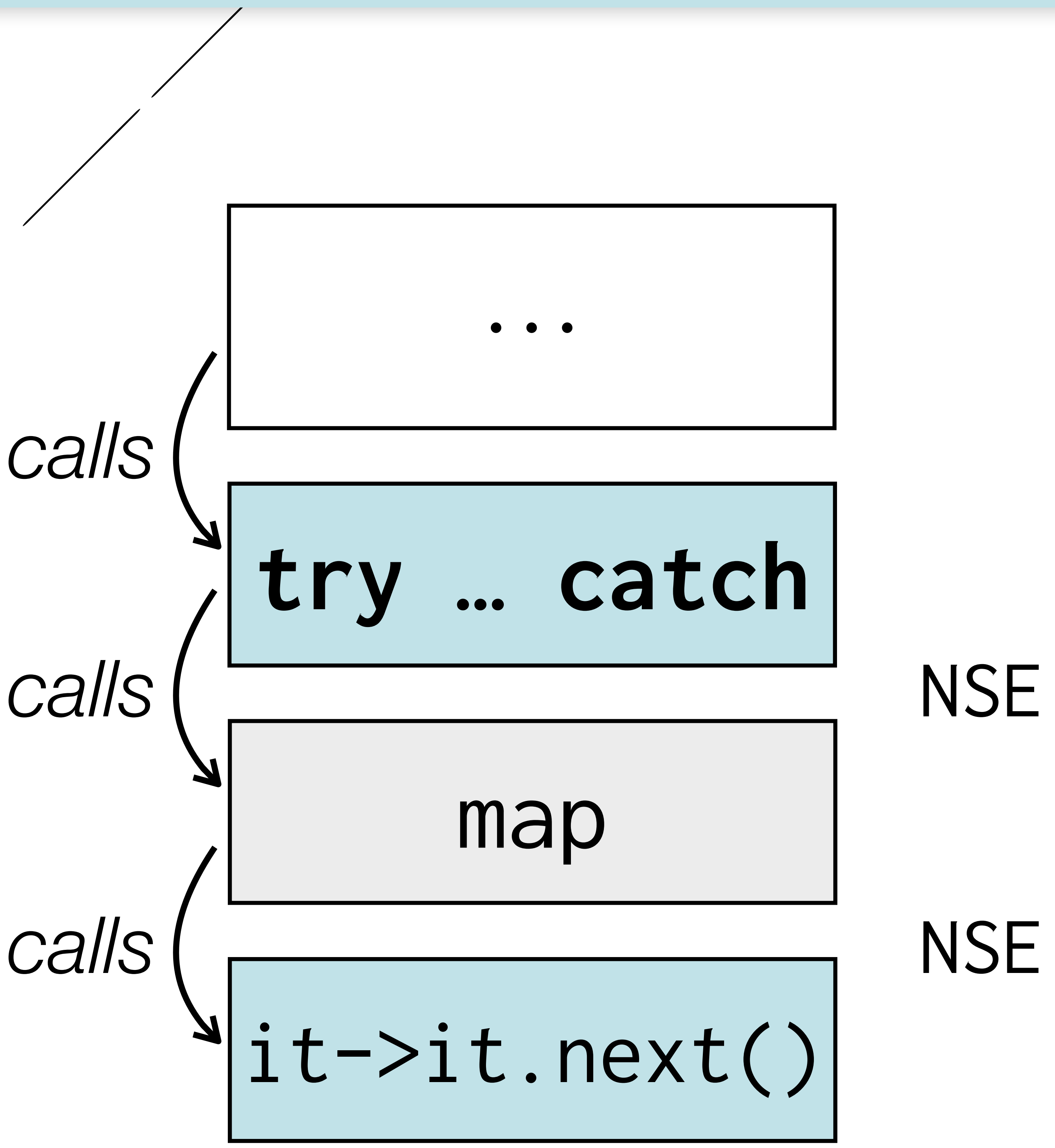
```
<X,Y> List<Y> map(List<X> xs, X→Y f) {…}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```




```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```



Unchecked exceptions can be handled by accident

An unchecked exception

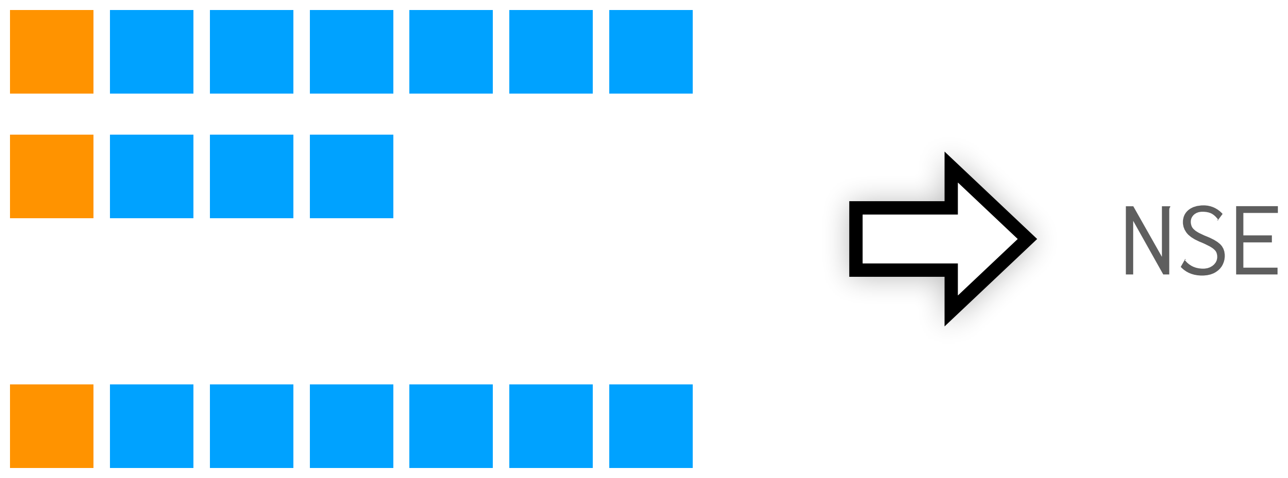
```
class NSE extends RuntimeException {…}
```

A higher-order function

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {…}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```




```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

Unchecked exceptions can be handled by accident

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {
  List<Y> ys = new ArrayList<Y>();
  Iterator<X> it = xs.iterator();
  while (true) {
    ys.add(f(it.next()));
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

Client code

```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

Unchecked exceptions can be handled by accident

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {
  List<Y> ys = new ArrayList<Y>();
  Iterator<X> it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Client code

```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

Unchecked exceptions can be handled by accident

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {
  List<Y> ys = new ArrayList<Y>();
  Iterator<X> it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

Client code

```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

Unchecked exceptions can be handled by accident

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {
  List<Y> ys = new ArrayList<Y>();
  Iterator<X> it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

Client code

```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

Unchecked exceptions can be handled by accident

A higher-order function in Java

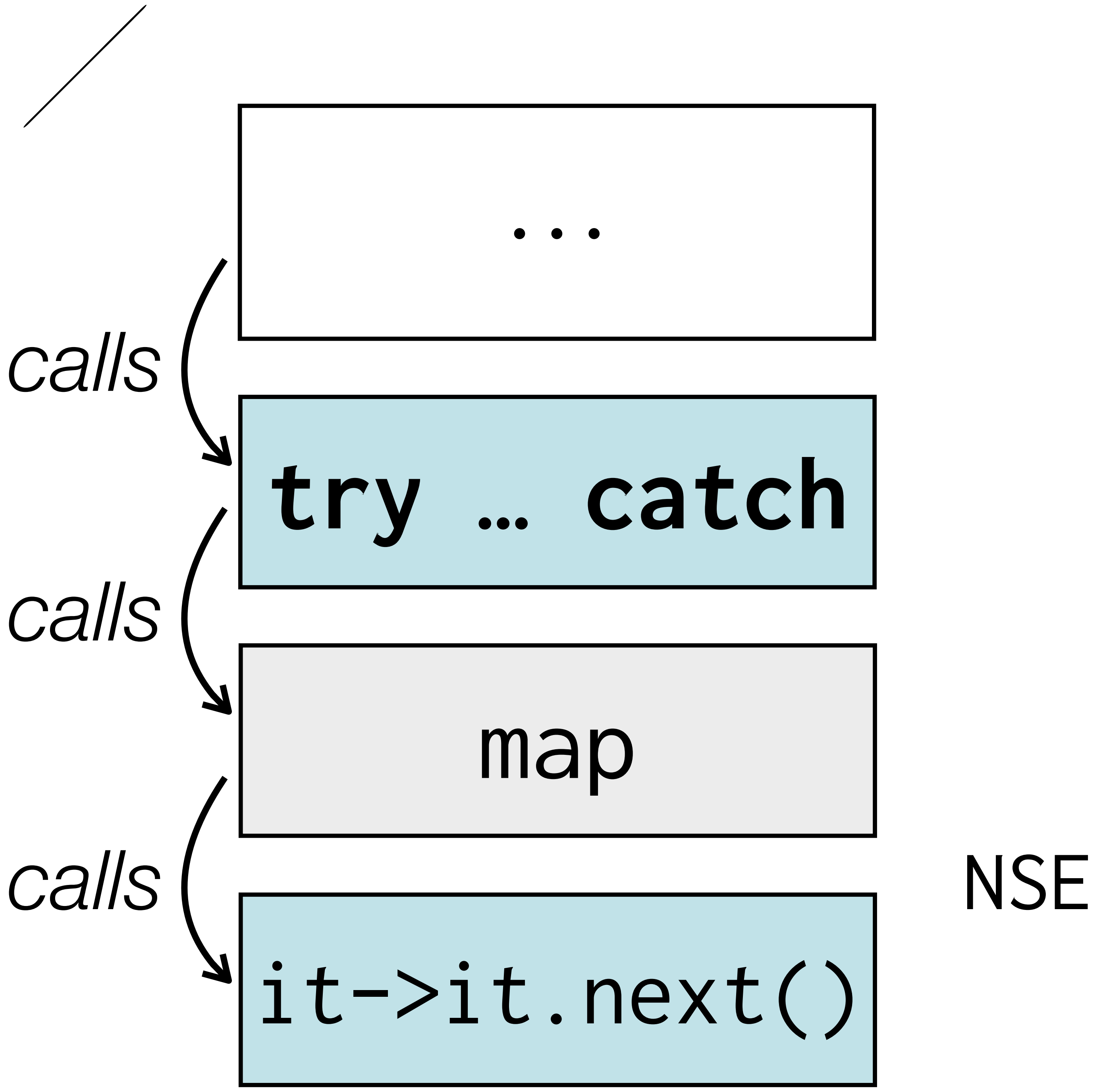
```
<X,Y> List<Y> map(List<X> xs, X→Y f) {
  List<Y> ys = new ArrayList<Y>();
  Iterator<X> it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Client code

```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```



Unchecked exceptions can be handled by accident

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {
  List<Y> ys = new ArrayList<Y>();
  Iterator<X> it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

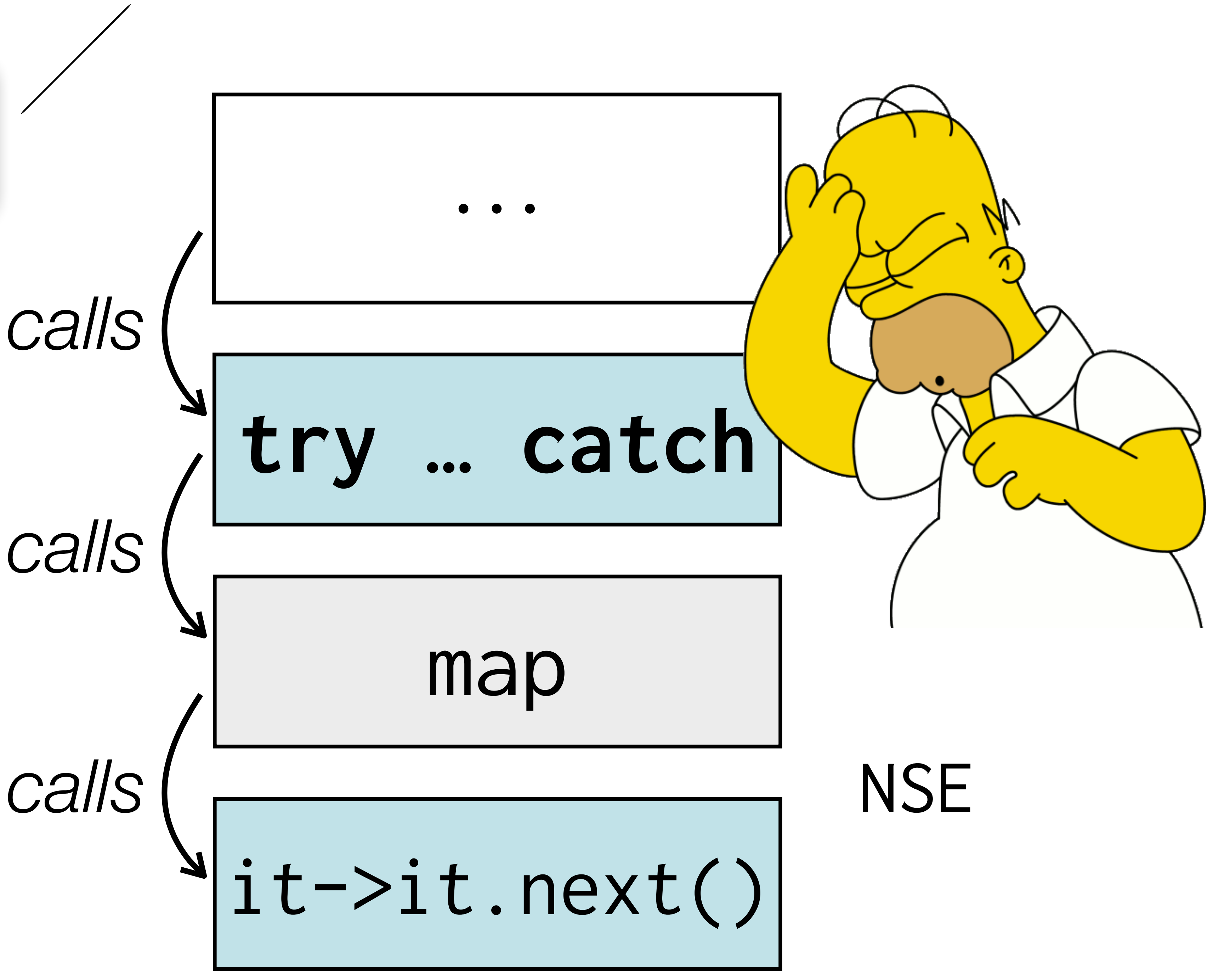
Handler intercepts exceptions raised by f!

Raises NSE if there is no next element

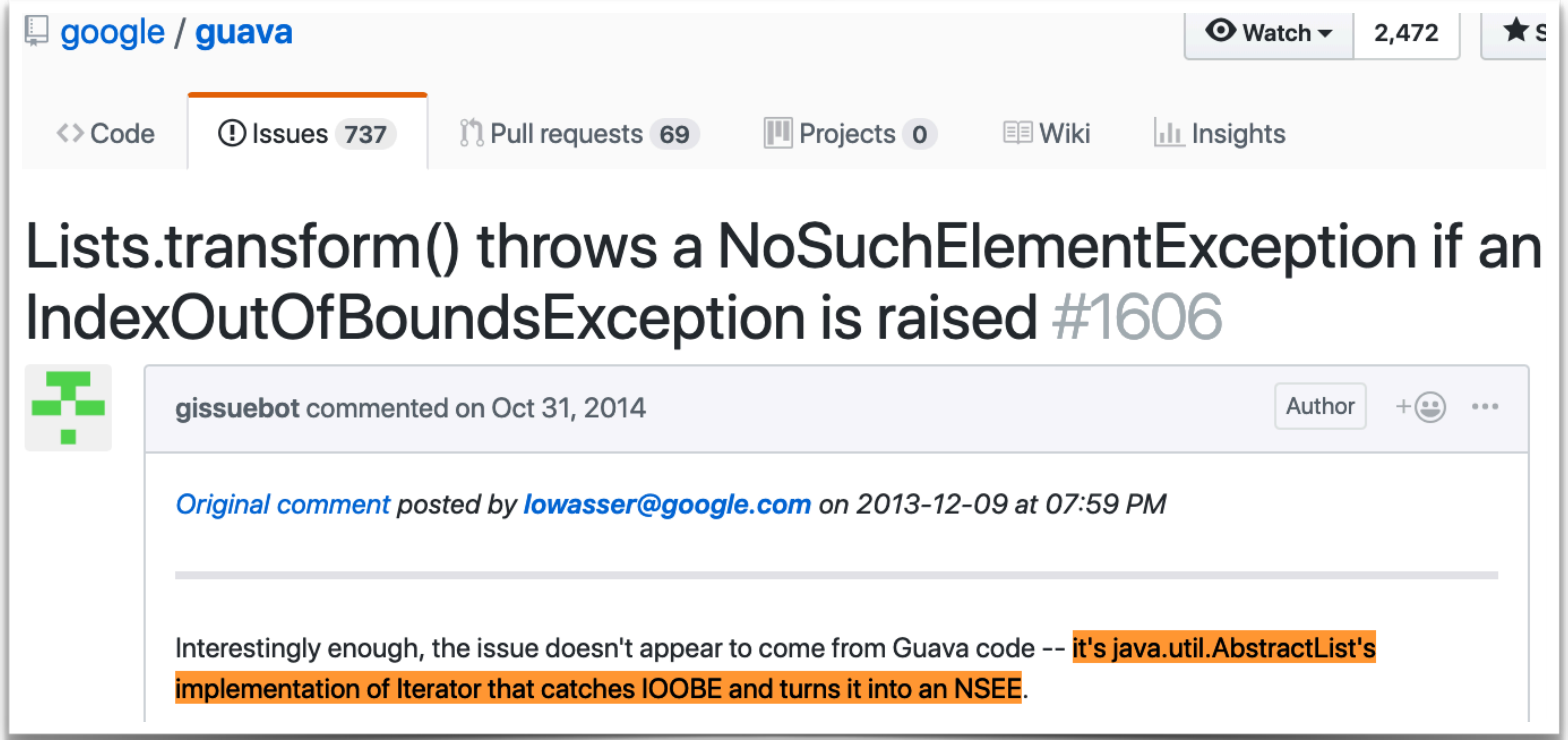
```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

Client code

```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```



Accidentally handled exceptions are real



An `IndexOutOfBoundsException` is accidentally handled by `java.util.AbstractList`

To check, or not to check, that is the question…

Checked exception types

```
IOException
SQLException
DataFormatException
…
```

To check, or not to check, that is the question…

Checked exception types

IOException
SQLException
DataFormatException
…

Unchecked exception types

UncheckedIOException
NoSuchElementException
NumberFormatException
…

Checked exceptions are too rigid.

Unchecked exceptions lead to uncaught and accidentally caught exceptions.

To check, or not to check, that is the question...

Checked exception types

~~IOException~~
~~SQLException~~
~~DataFormatException~~
~~...~~

Unchecked exception types

~~UncheckedIOException~~
~~NoSuchElementException~~
~~NumberFormatException~~
~~...~~

Checked exceptions are too rigid.

Unchecked exceptions lead to uncaught and accidentally caught exceptions.

To check, or not to check, that depends on the context the exception is passing through

To check, or not to check, that depends on the context the exception is passing through

Exceptions are checked in contexts aware of them (e.g., the caller of `map`)

Exceptions are “unchecked” in contexts oblivious to them (e.g., the definition of `map`)

To check, or not to check, that depends on the context the exception is passing through

Exceptions are checked in contexts aware of them (e.g., the caller of `map`)

Exceptions are "unchecked" in contexts oblivious to them (e.g., the definition of `map`)

Guarantees

Exceptions are always caught, but never accidentally caught.

Genus

An object-oriented language with powerful generics [Zhang & al. 2015]

Syntax is close to Java, but uses [] instead of <>

The new exception mechanism can be applied to many other languages

Guarantees

Exceptions are always caught, but never accidentally caught.

Recall this Java program doesn't type-check…

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List<File> files = …;
List<Tree> trees;
try {
  trees = map(files, f->parseFile(f));
  …
} catch (IOException e) {…}
```

Compile-time error: exception mismatch

Lambda-expression has type

`File → Tree throws IOException`

but a function of type

`File → Tree`

is expected.

This Genus program type-checks!

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List[File] files = …;
List[Tree] trees;
try {
  trees = map(files, f->parseFile(f));
  …
} catch (IOException e) {…}
```

This Genus program type-checks!

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List[File] files = …;
List[Tree] trees;
try {
  trees = map(files, f->parseFile(f));
  …
} catch (IOException e) {…}
```

Exception mismatch is allowed

This Genus program type-checks!

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List[File] files = …;
List[Tree] trees;
try {
  trees = map(files, f->parseFile(f));
  …
} catch (IOException e) {…}
```

Exception mismatch is allowed

aware of exceptions raised by the lambda expression

This Genus program type-checks!

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

oblivious to exceptions raised by f

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

NSE-oblivious NSE-aware

Client code

```
List[File] files = …;
List[Tree] trees;
try {
   trees = map(files, f->parseFile(f));
   …
} catch (IOException e) {…}
```

Exception mismatch is allowed

aware of exceptions raised by the lambda expression

Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

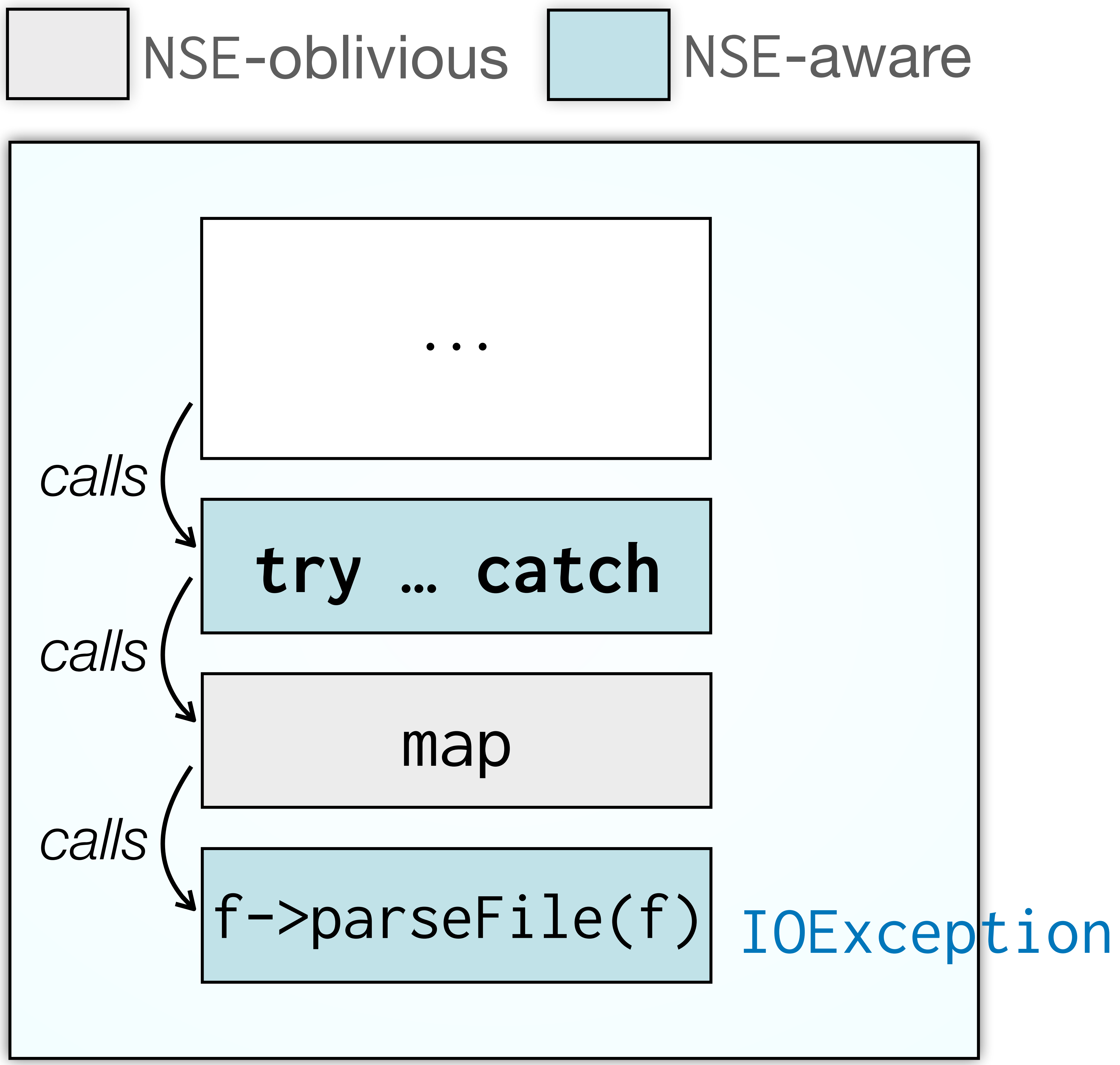
```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List[File] files = …;
List[Tree] trees;
try {
  trees = map(files, f->parseFile(f));
  …
} catch (IOException e) {…}
```



Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

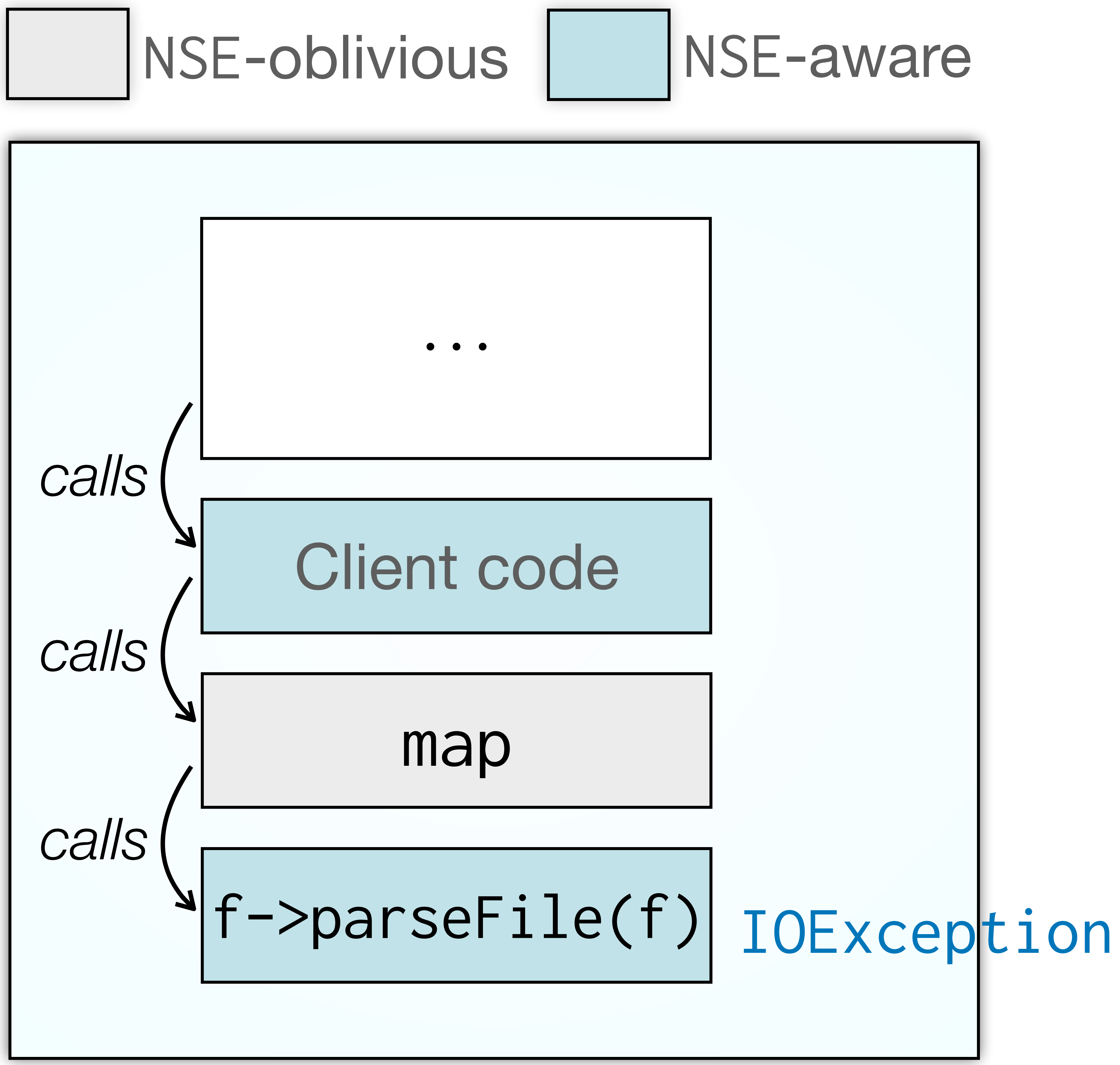
```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List[File] files = …;
List[Tree] trees;
trees = map(files, f->parseFile(f));
…
```



Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

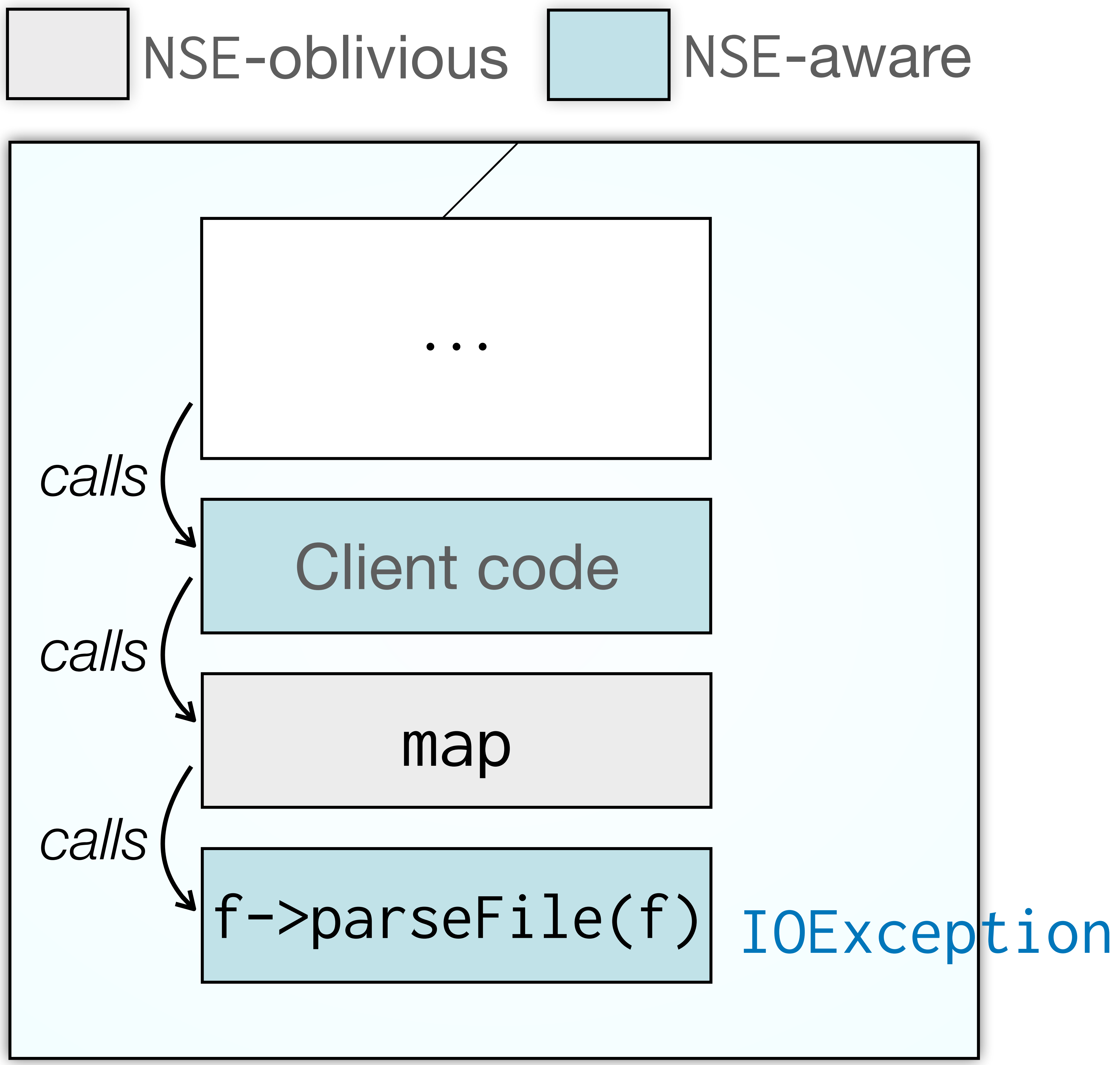
```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List[File] files = …;
List[Tree] trees;
trees = map(files, f->parseFile(f));
…
```



Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

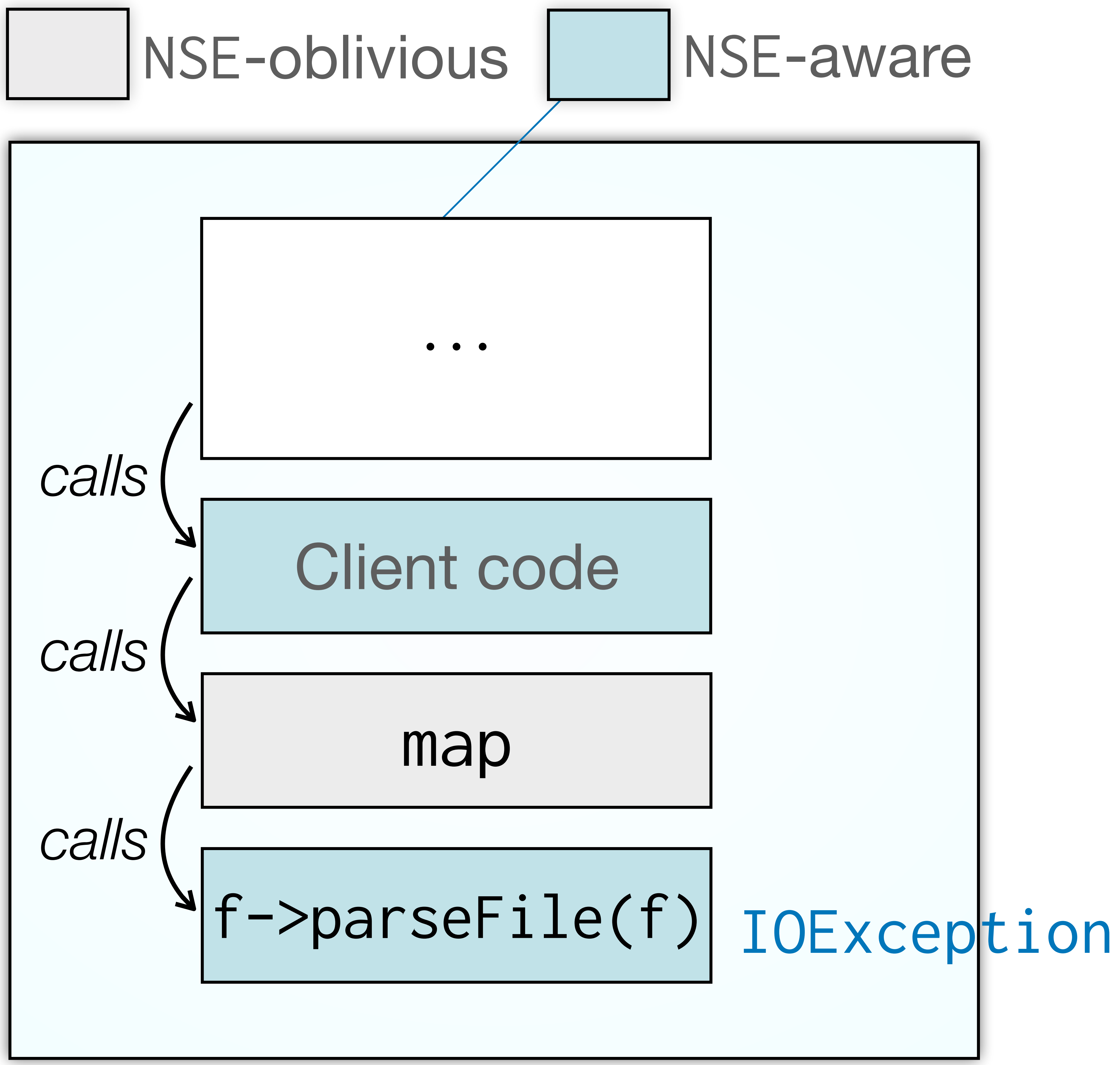
```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List[File] files = …;
List[Tree] trees;
trees = map(files, f->parseFile(f));
…
```



Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

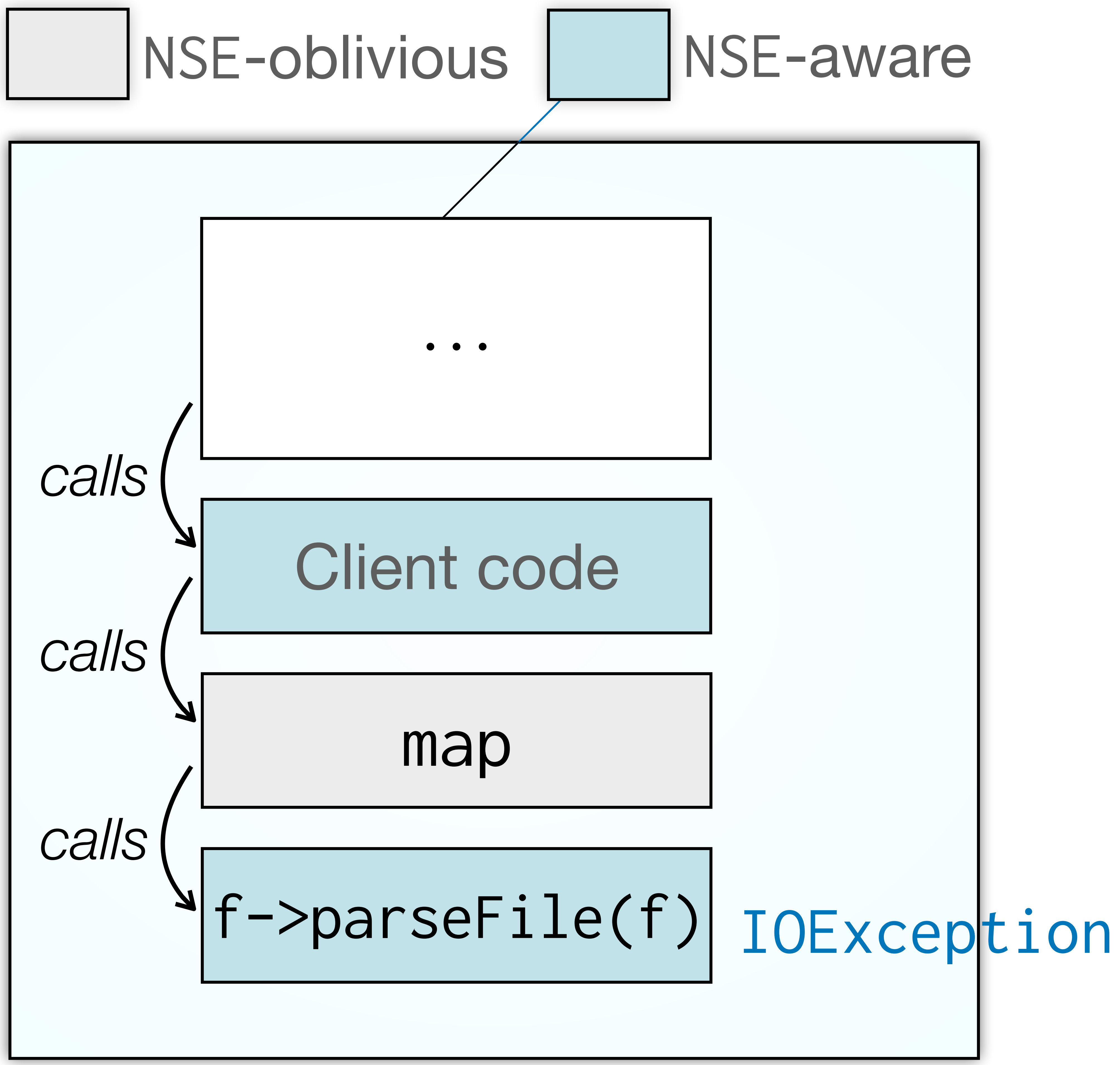
```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(String filename)
throws IOException {…}
```

Client code

```
List[File] files = …;
List[Tree] trees;
trees = map(files, f->parseFile(f));
…
```



Recall this Java program handles exceptions by accident…

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {
  List<Y> ys = new ArrayList<Y>();
  Iterator<X> it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

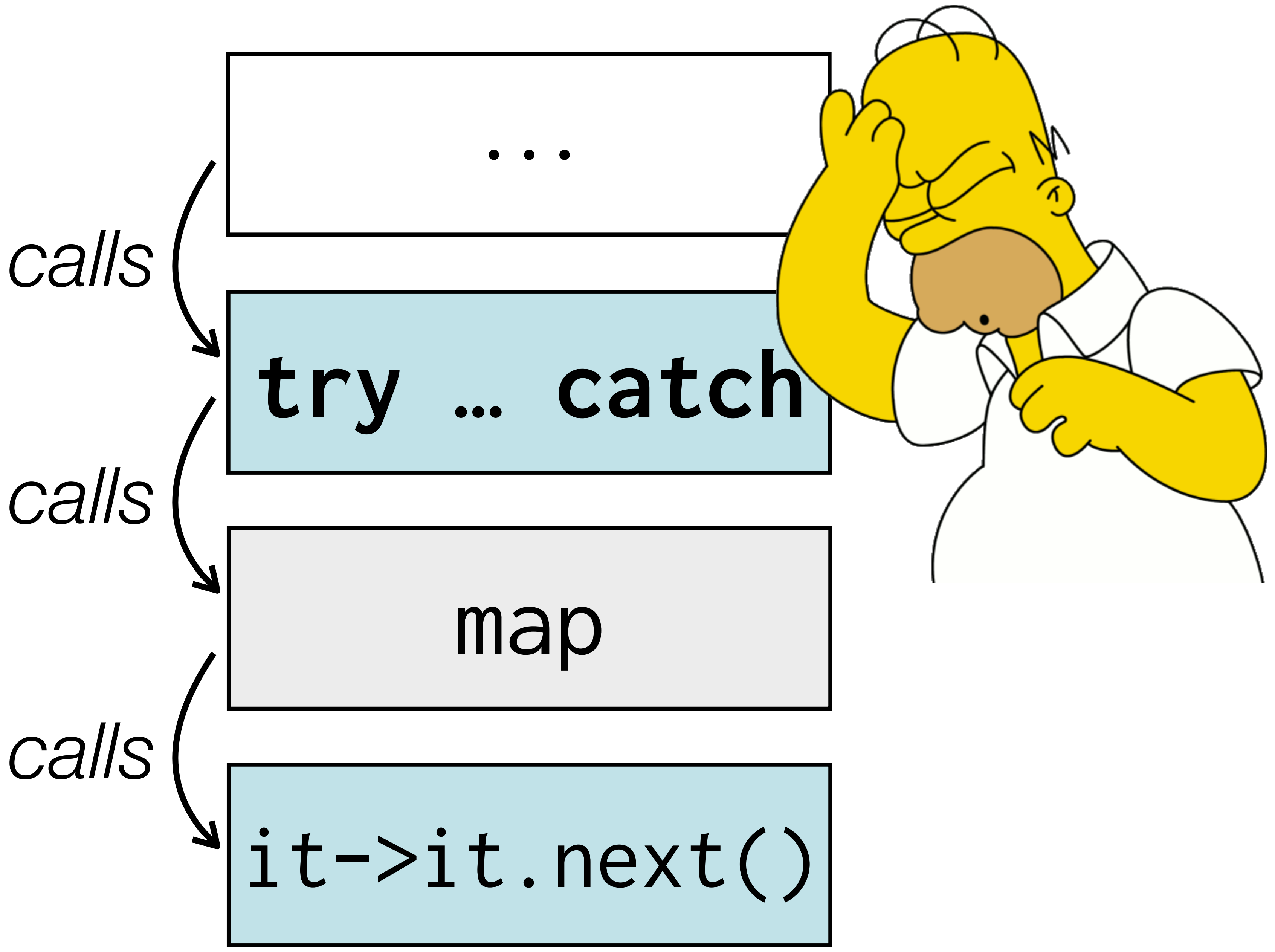
Handler intercepts exceptions raised by *f*!

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

Client code

```
List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```



Genus prevents accidental handling

A higher-order function in Genus

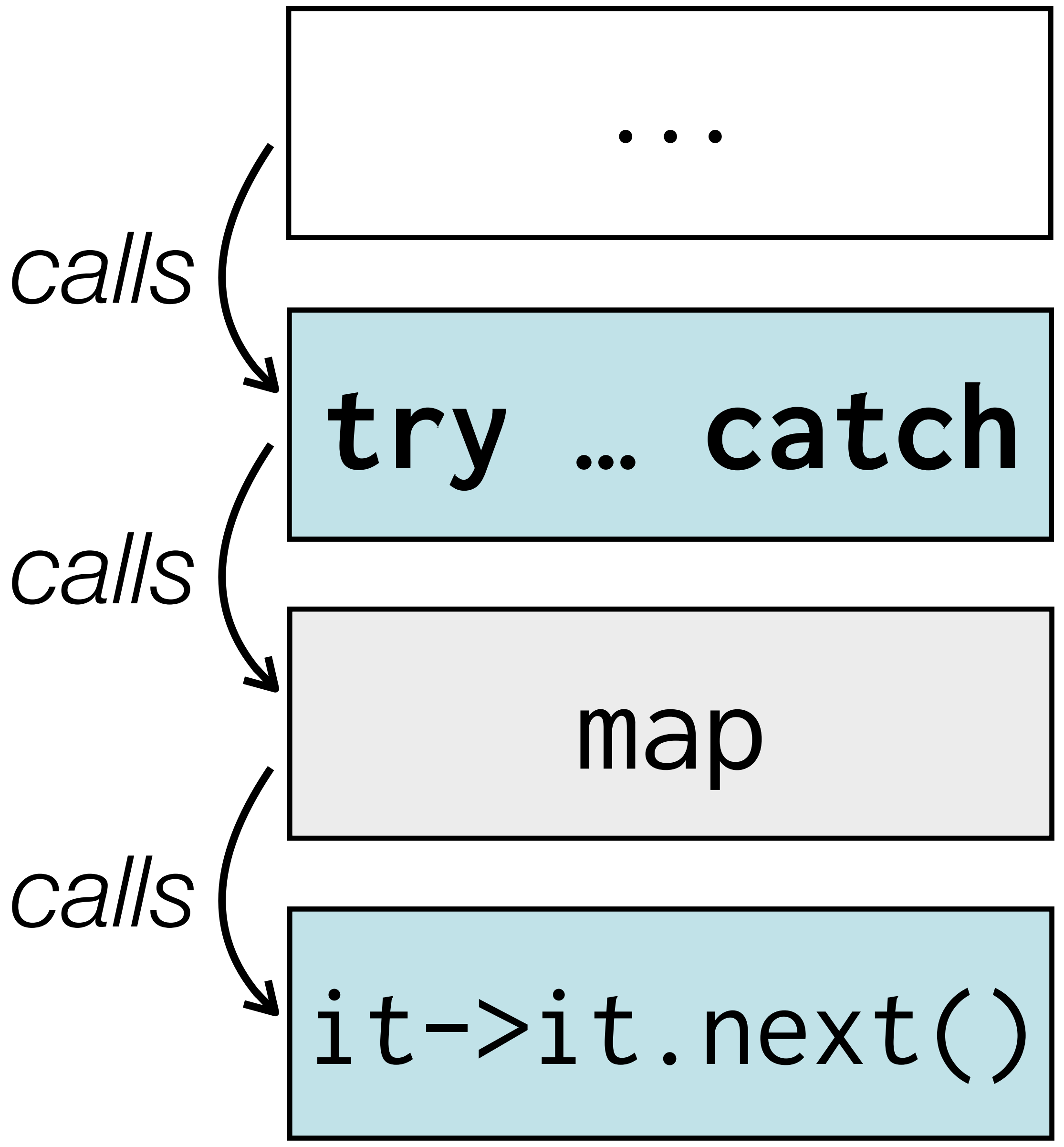
```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

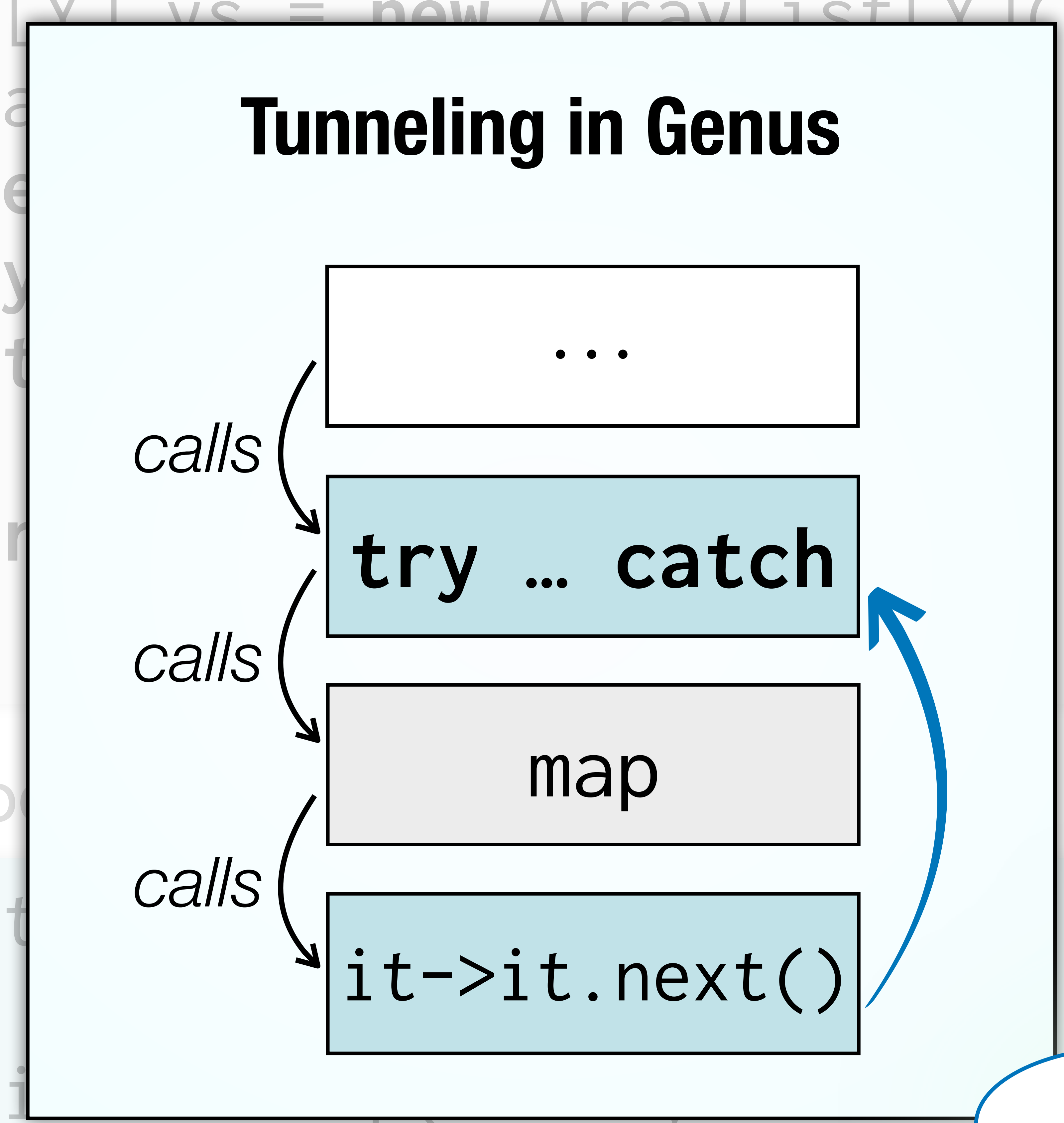


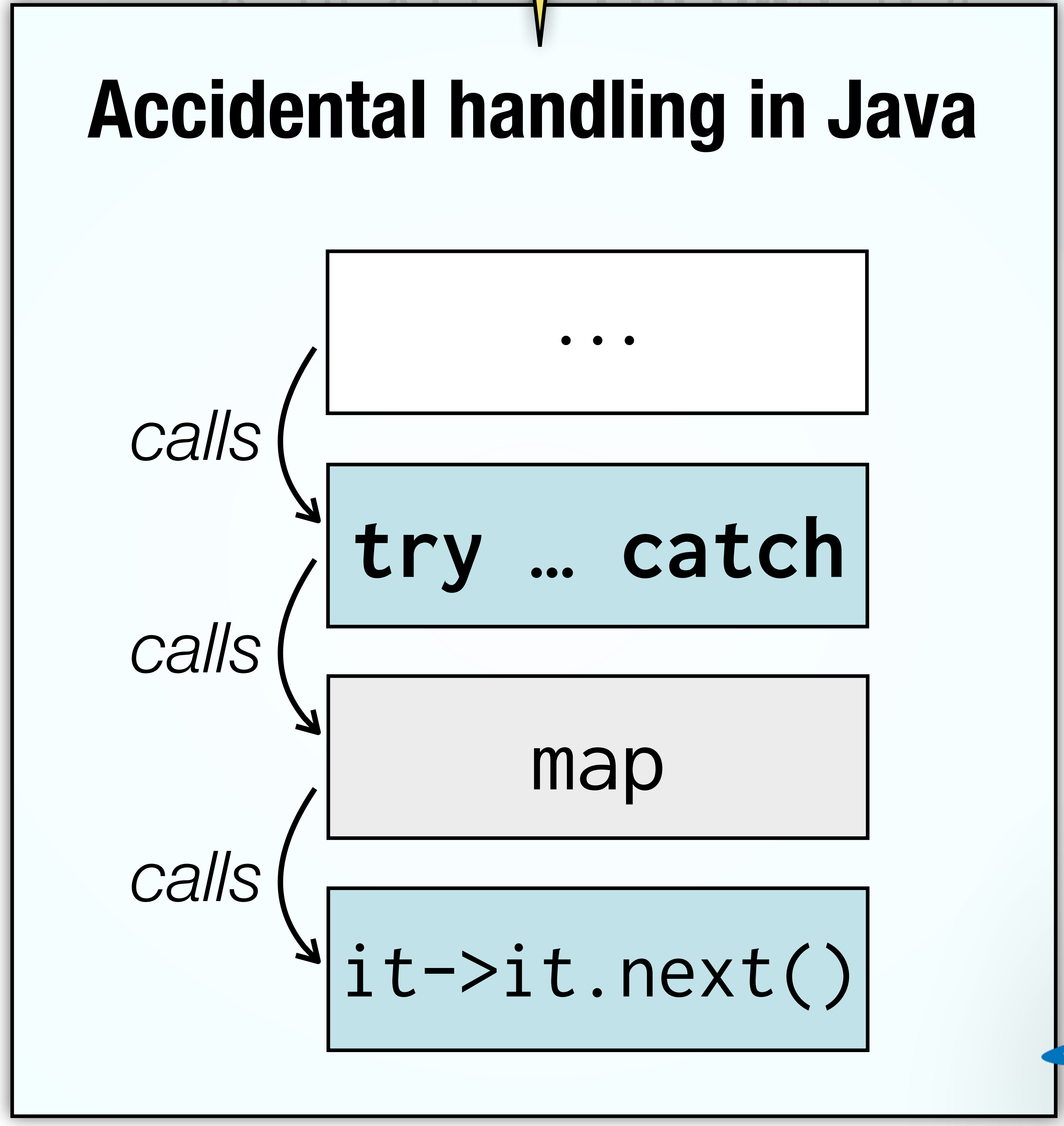
Genus prevents accidental handling

A higher-order function in Genus

Raises NSE if there is no next element

Identifier of an exception: **type**





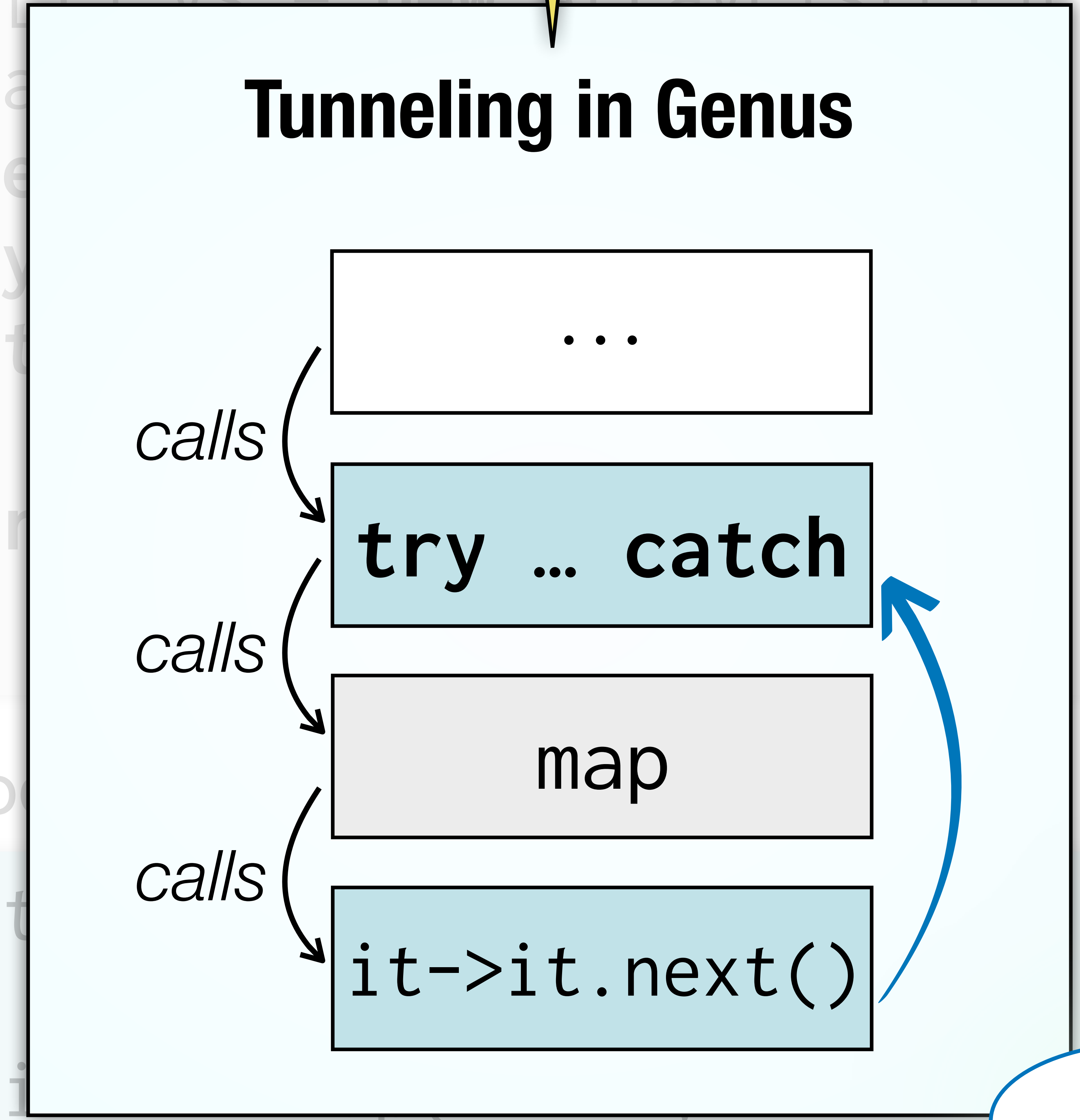

```
List[Y] map[X,Y](List[X] xs, X→Y f){
  ...
}catch (NSE e) {…}
```

Genus prevents accidental handling

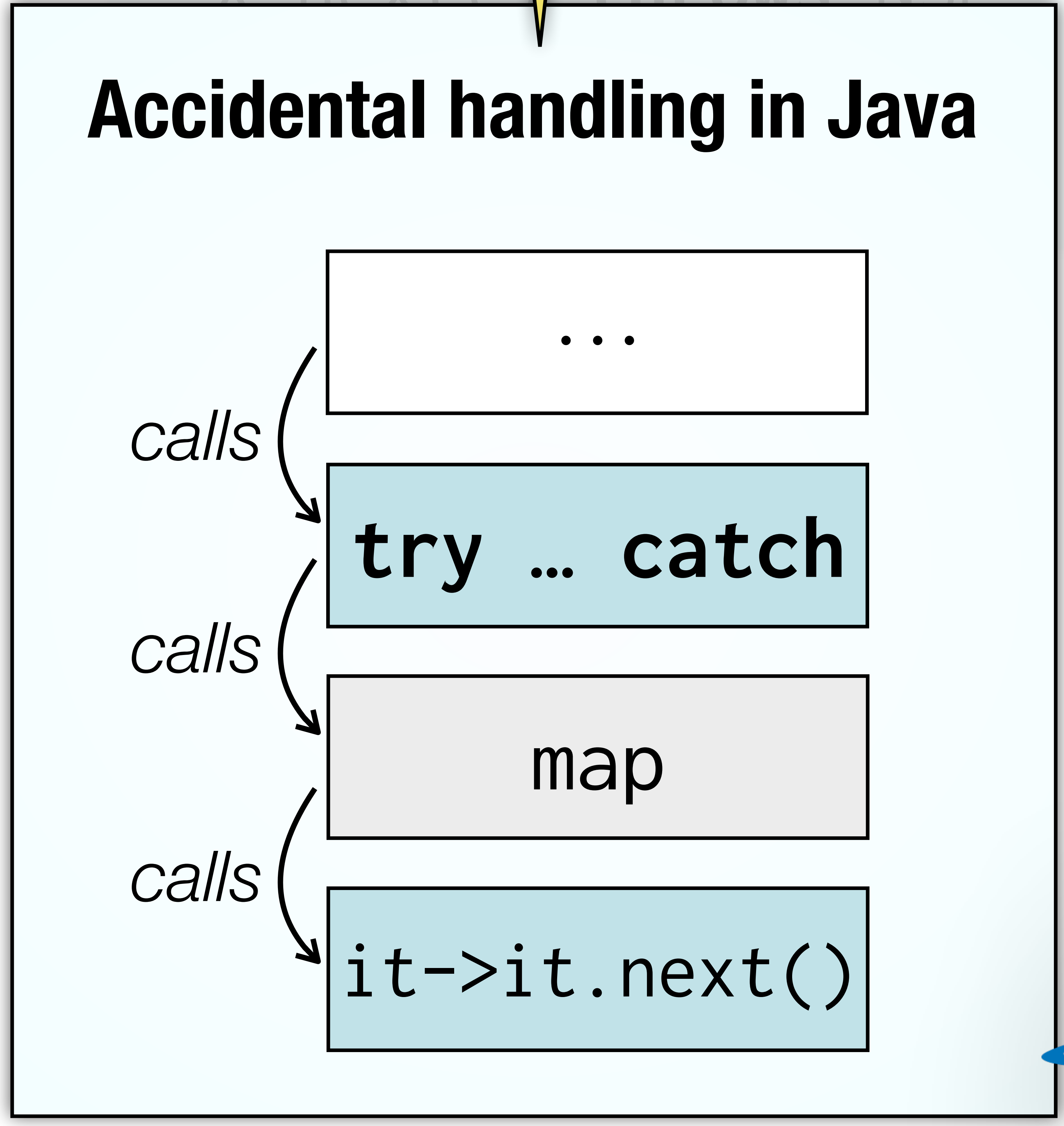
A higher-order function in Genus

Raises NSE if there is no next element

Identifier of an exception: **type** and **blame label**



Identifier of an exception: **type**



Client co

```
} catch (NSE e) {…}
```

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Client code

```
List[Iterator[int]] iters = …;
var ints = map(iters, it->it.next());
…
```

Program Exception mismatch

Compiler Blame label created

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Client code

```
List[Iterator[int]] iters = …;
var ints = map(iters, it->it.next());
…
```

B1

Program Exception mismatch

Compiler Blame label created

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE·B1 e) {…}
```

B1

Program Exception mismatch

Compiler Blame label created

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE·B1 e) {…}
```

B1

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    ys.add(f(it.next()));
  }
  return ys;
}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE·B1 e) {…}
```

B1

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    ys.add(f(it.next()));
  }
  return ys;
}
```

B2

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE·B1 e) {…}
```

B1

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE·B2 e) { break; }
  }
  return ys;
}
```

B2

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  ……
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE·B1 e) {…}
```

B1

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE·B2 e) { break; }
  }
  return ys;
}
```

B2

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE·B1 e) {…}
```

B1

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE·B2 e) { break; }
  }
  return ys;
}
```

B2

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE·B1 e) {…}
```

B1

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next(b) throws NSE·b;
  …
}
```

blame parameter

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next(B2))); }
    catch (NSE·B2 e) { break; }
  }
  return ys;
}
```

B2

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next(b) throws NSE·b;
  …
}
```

blame parameter

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE·B1 e) {…}
```

B1

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next(B2))); }
    catch (NSE·B2 e) { break; }
  }
  return ys;
}
```

B2

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next(B1));
  …
} catch (NSE·B1 e) {…}
```

B1

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next(b) throws NSE·b;
  …
}
```

blame parameter

Program Exception mismatch

↓

Compiler Blame label created

↓

Runtime Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }; }
  }
  return ys;
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

No need to write blame labels out

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }; }
  }
  return ys;
}
```

Client code

```
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next() throws NSE;
  …
}
```

Program Exception mismatch

Compiler Blame label created

Runtime Exceptions tunneled to contexts at fault

Blame labels as a notion of context

	Gradual typing	Exc. tunneling
Program	Type mismatch	Exception mismatch
Compiler	Blame label created	Blame label created
Runtime	Cast failures attributed to context at fault	Exceptions tunneled to context at fault

Blame labels as a notion of context

	Gradual typing	Exc. tunneling
Program	Type mismatch	Exception mismatch
Compiler	Blame label created	Blame label created
Runtime	Cast failures attributed to context at fault	Exceptions tunneled to context at fault

Blame labels impose **static obligations** on blameful contexts to handle **exceptions so labeled**.

A theorem

Theorem Well-typed programs handle their exceptions.

Type system checks that exception mismatches do not escape.

A theorem

Theorem Well-typed programs handle their exceptions.

Type system checks that exception mismatches do not escape.

```
List[Y] map[X,Y](List[X] l, X→Y f) {…}
```

Function `f` should not escape `map`.

A theorem, and a compiler too

Theorem	Well-typed programs handle their exceptions.

Compiler	An extension of the base Genus compiler 39,000 LoC + 6,000 LoC

Experience: cleaner yet safer code

Porting Java code that uses anti-patterns

Examples come from various sources: the javac compiler, Apache Commons, etc.

Genus version restores static checking of exceptions

Genus version eliminates ~200 (out of ~1,000) LoC in a visitor class in javac

Experience: cleaner yet safer code

Porting Java code that uses anti-patterns

Examples come from various sources: the javac compiler, Apache Commons, etc.

Genus version restores static checking of exceptions

Genus version eliminates ~200 (out of ~1,000) LoC in a visitor class in javac

```
    public void visitReturn(JCReturn tree) {
        try {
            print("return");
            if (tree.expr != null) {
                print(" ");
                printExpr(tree.expr);
            }
            print(";");
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }
```

exception unwrapping in javac

Experience: cleaner yet safer code

Porting Java code that uses anti-patterns

Examples come from various sources: the javac compiler, Apache Commons, etc.

Genus version restores static checking of exceptions

Genus version eliminates ~200 (out of ~1,000) LoC in a visitor class in javac

Porting the Java Collections Framework (all general-purpose implementations)

Eliminated risks of accidental handling

Experience: cleaner yet safer code

Porting Java code that uses anti-patterns

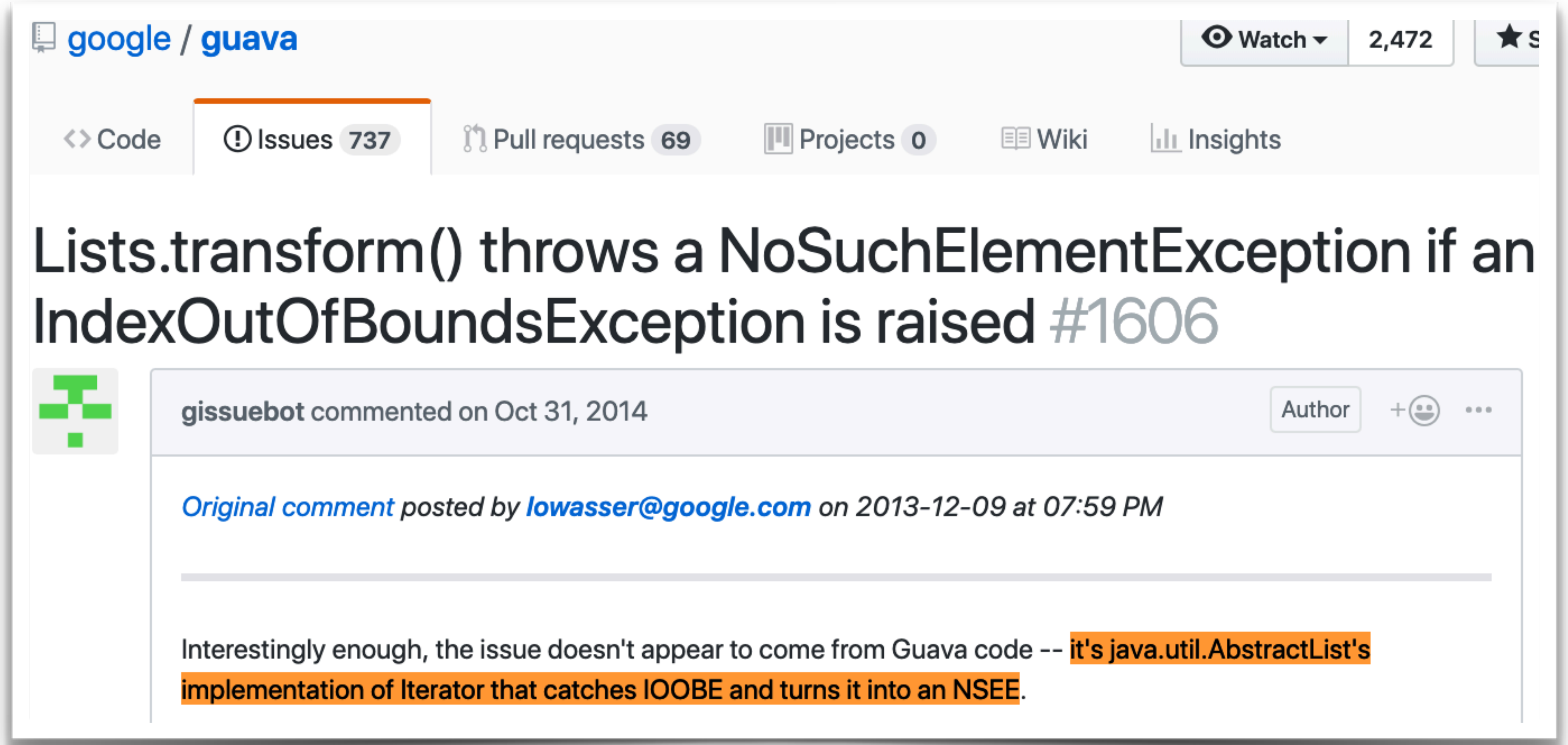
Examples come from various sources: the javac compiler, Apache Commons, etc.

Genus version restores static checking of exceptions

Genus version eliminates ~200 (out of ~1,000) LoC in a visitor class in javac

Porting the Java Collections Framework (all general-purpose implementations)

Eliminated risks of accidental handling



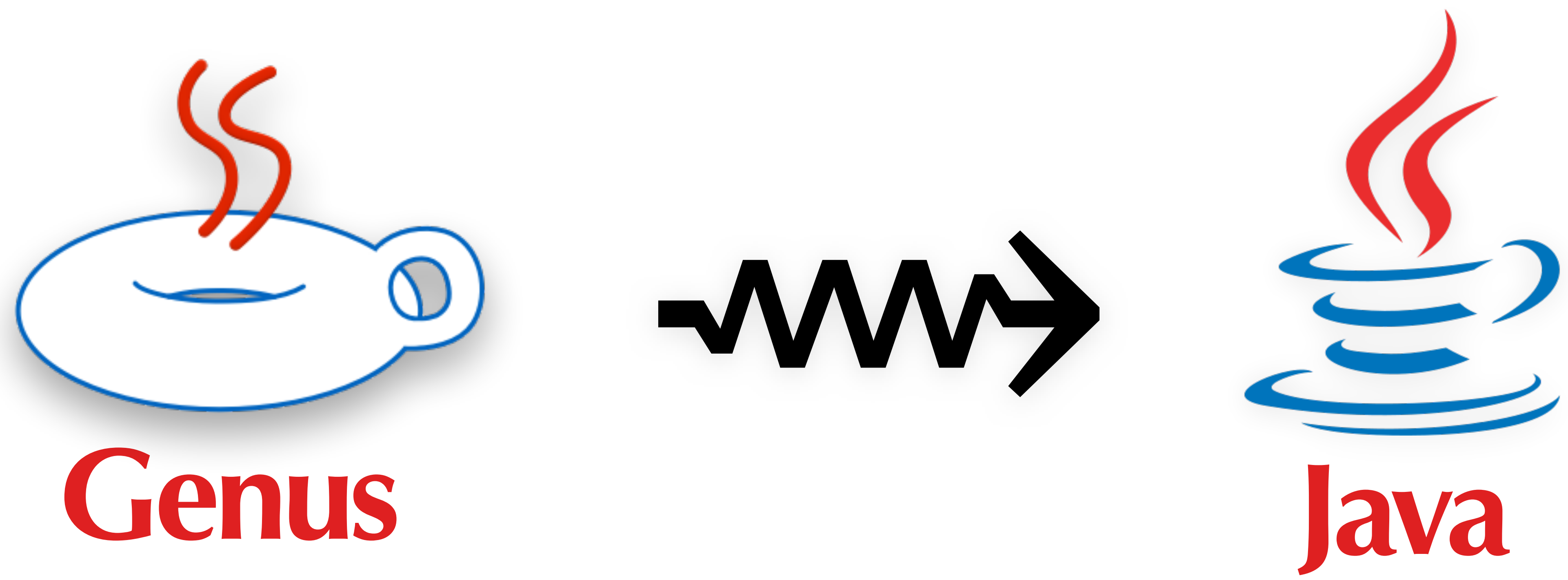
Compilation

Genus

Java

Translation reifies blame labels so that exceptions can acquire new identities

Compilation

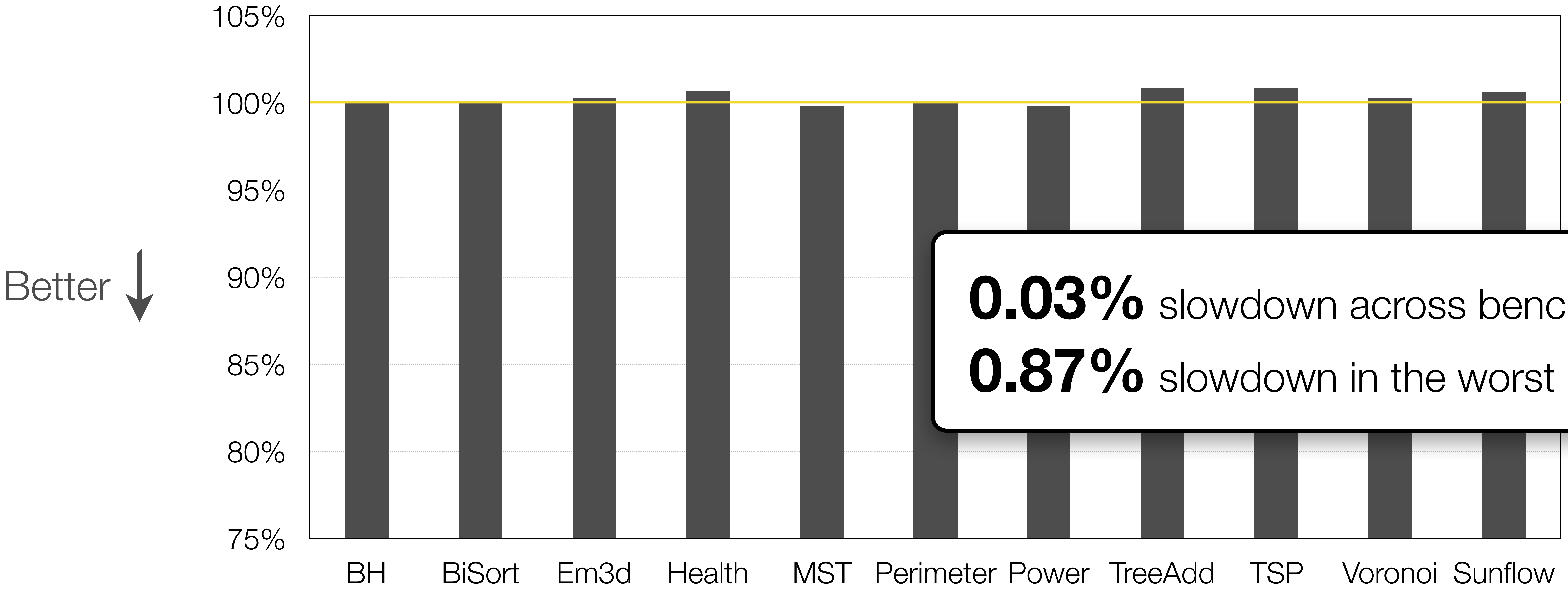


Translation reifies blame labels so that exceptions can acquire new identities

Imposes a cost on normal control flow even when exceptions are not raised.

Performance of exception-infrequent code

$$\text{y-axis} = \frac{\text{running time of Genus using Java's exception mechanism}}{\text{running time of Genus using the new exception mechanism}}$$

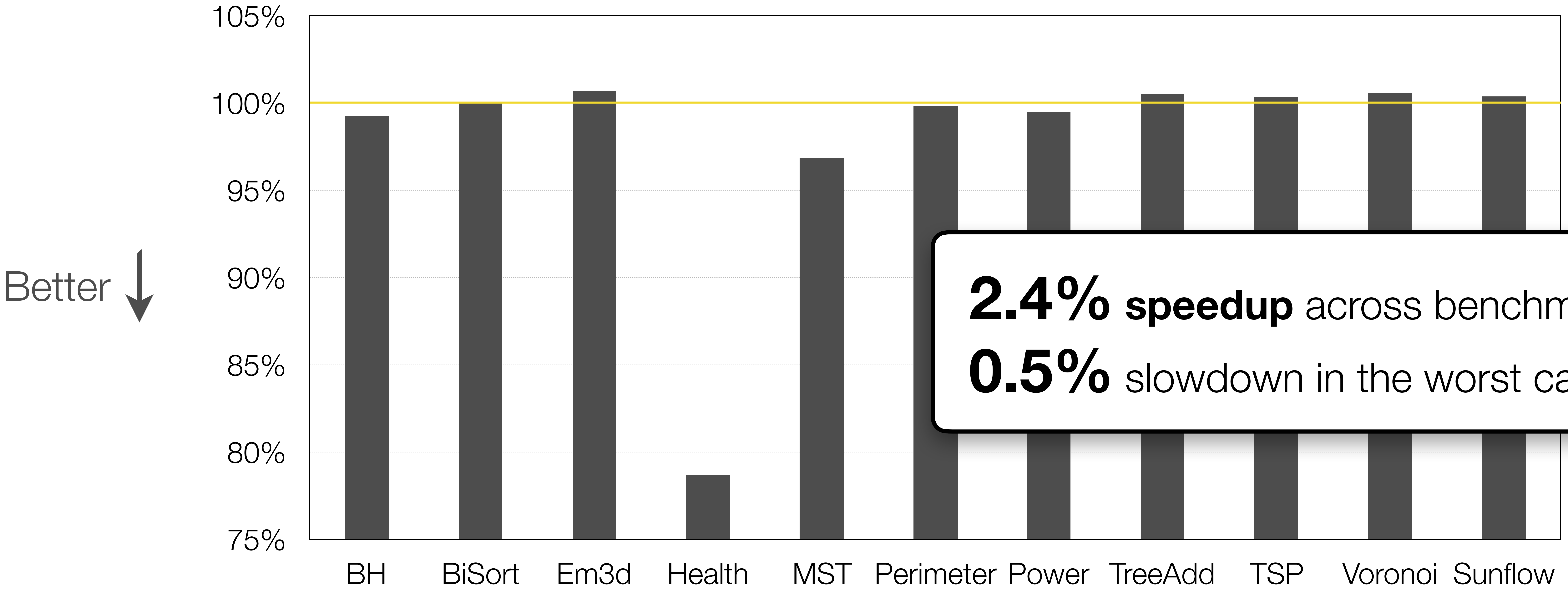


0.03% slowdown across benchmarks
0.87% slowdown in the worst case

(Benchmarks come from JOlden and DaCapo suites)

Performance of exception-infrequent code

$$\text{y-axis} = \frac{\text{running time of Genus using Java's exception mechanism}}{\text{running time of Genus using the new exception mechanism and using NSE to speed up for-each loops}}$$



2.4% speedup across benchmarks
0.5% slowdown in the worst case

(Benchmarks come from JOlden and DaCapo suites)

Using NSE to speed up for-each loops

```
for (T x : c) {
  … // loop body
}
```

conventional
translation

```
Iterator<T> it = c.iterator();
while (it.hasNext()) {
  T x = it.next();
  … // loop body
}
```

N+1 calls to hasNext()

Using NSE to speed up for-each loops

```
for (T x : c) {
  … // loop body
}
```

conventional translation

```
Iterator<T> it = c.iterator();
while (it.hasNext()) {
  T x = it.next();
  … // loop body
}
```

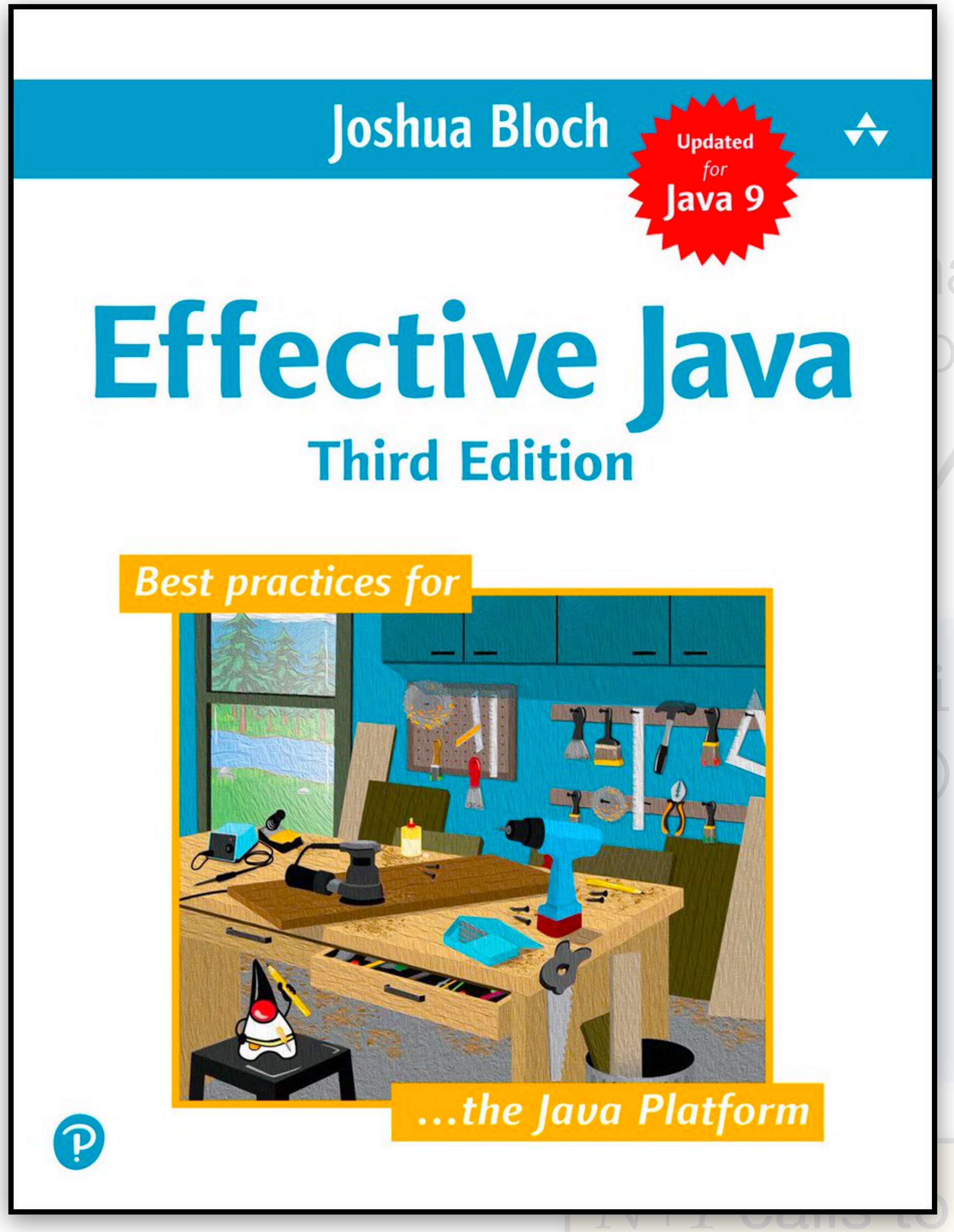
N+1 calls to hasNext()

exception-based translation

```
Iterator[T] it = c.iterator();
while (true) {
  try {
    T x = it.next();
    … // loop body
  }
  catch (NSE e) { break; }
}
```

one **throw**-and-**catch**

Exception-based translation is frowned upon…




```
// Do not use this hideous code for iteration over a collection!
try {
    Iterator<Foo> i = collection.iterator();
    while(true) {
        Foo foo = i.next();
        ...
    }
} catch (NoSuchElementException e) {
}
```

The NSE is guaranteed to be handled

⇨ Optimization is cost-effective

```
for (T x : c) {
  … // loop body
}
```

conventional translation

```
Iterator<T> it = c.iterator();
while (it.hasNext()) {
  T x = it.next();
  … // loop body
}
```

$N{+}1$ calls to hasNext()

exception-based translation

```
Iterator[T] it = c.iterator();
while (true) {
  try {
    T x = it.next();
    … // loop body
  }
  catch (NSE e) { break; }
}
```

one **throw**-and-**catch**

The NSE is guaranteed not to be accidentally handled

⇨ Optimization is correct

```
for (T x : c) {
  … // loop body
}
```

conventional translation

```
Iterator<T> it = c.iterator();
while (it.hasNext()) {
  T x = it.next();
  … // loop body
}
```

N+1 calls to hasNext()

exception-based translation

```
Iterator[T] it = c.iterator();
while (true) {
  try {
    T x = it.next();
    … // loop body
  }
  catch (NSE e) { break; }
}
```

one **throw**-and-**catch**

Algebraic effects subsume exceptions

```
effect NSE {
  void throw();
}
```

Algebraic effects subsume exceptions

⇨ Effects can be accidentally handled if not checked !

```
effect NSE {
  void throw();
}
```

Effect polymorphism

```
List[Y] map[X,Y](List[X] xs, X → Y f) {…}
```

Effect polymorphism

effect variable

```
List[Y] map[X,Y,E](List[X] xs, X→Y throws E f) throws E {…}
```

Effect polymorphism

effect variable

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {…}
```

Effect polymorphism

effect variable

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {…}
```

throws E

throws E

Effect polymorphism

⇨ Statically checked effects can be accidentally handled*!*

effect variable

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {…}
```

throws E

throws E

Effect polymorphism

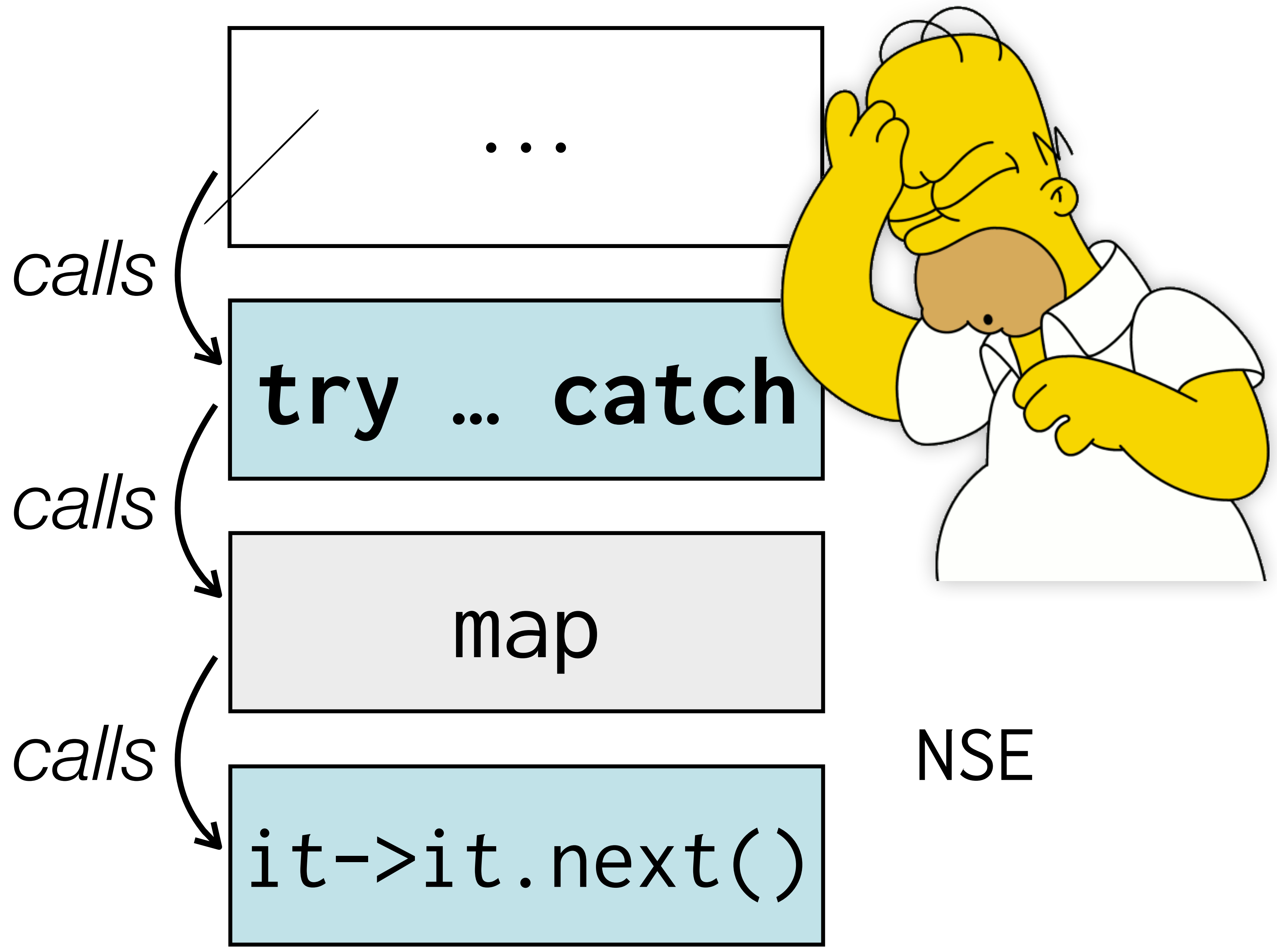
⇨ Statically checked effects can be accidentally handled*!*

An effect-polymorphic, higher-order function

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch NSE { void throw() { break; } }
  }
  return ys;
}
```

Client code

```
try {
  var ints = map(iters, it->it.next());
  …
} catch NSE { void throw() { … } }
```



Effect polymorphism, but with tunneling

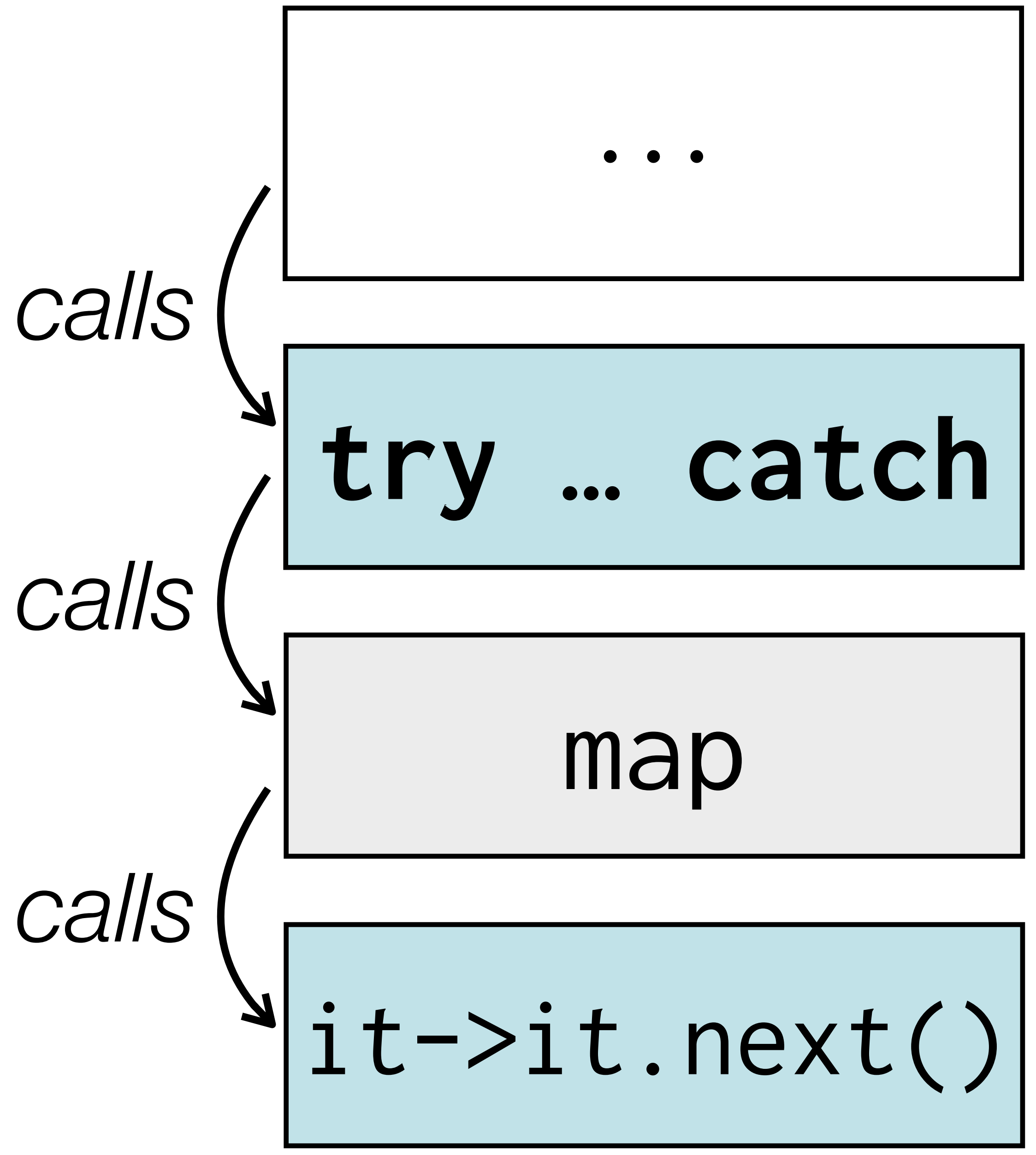
⇨ Accidental handling is prevented

An effect-polymorphic, higher-order function

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch NSE { void throw() { break; } }
  }
  return ys;
}
```

Client code

```
try {
  var ints = map(iters, it->it.next());
  …
} catch NSE { void throw() { … } }
```



Effect polymorphism, but with tunneling

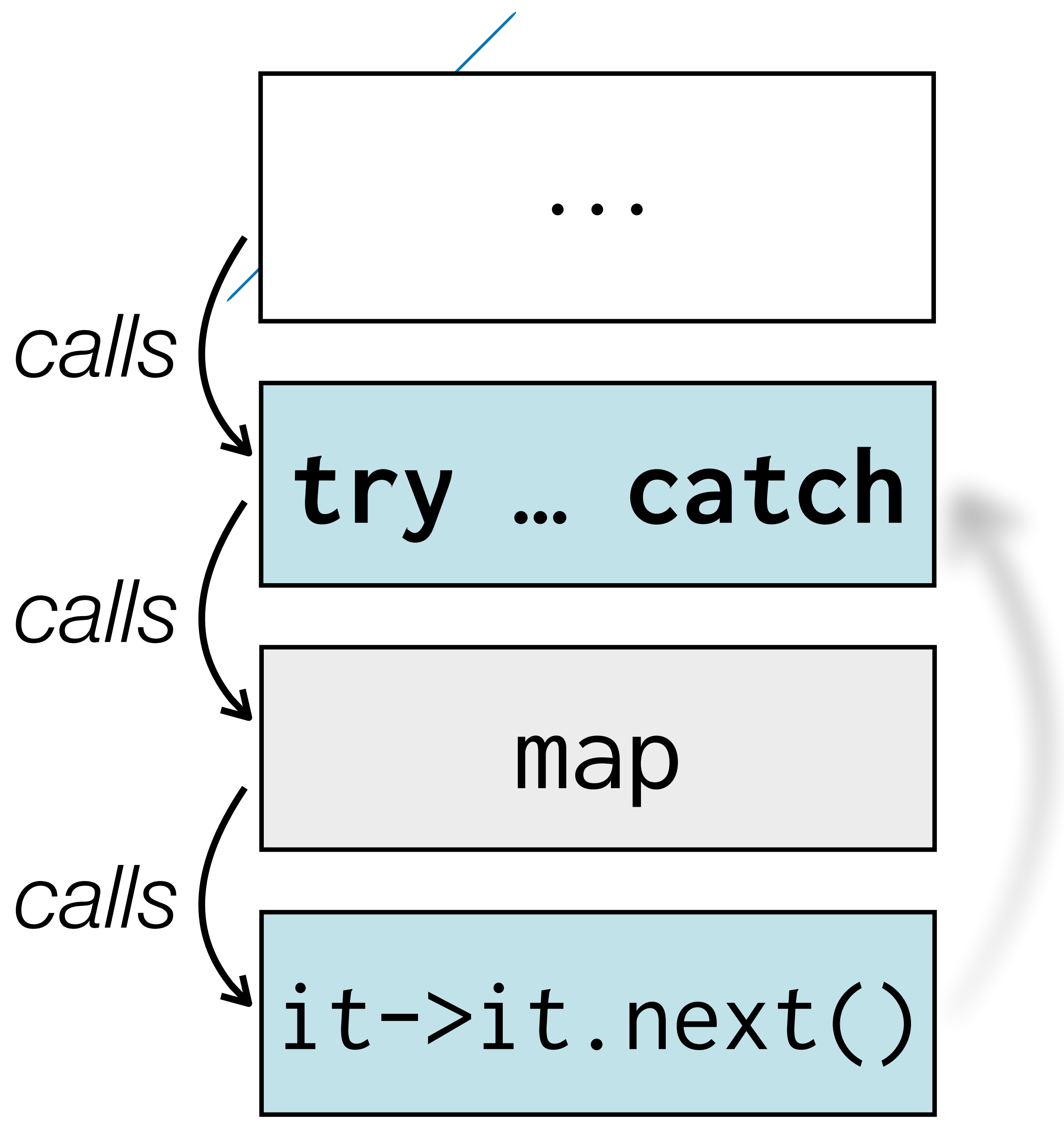
⇨ Accidental handling is prevented

An effect-polymorphic, higher-order function

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch NSE { void throw() { break; } }
  }
  return ys;
}
```

Client code

```
try {
  var ints = map(iters, it->it.next());
  …
} catch NSE { void throw() { … } }
```



Effect polymorphism, but with tunneling

⇨ Accidental handling is prevented

Effects tunnel through contexts polymorphic to them

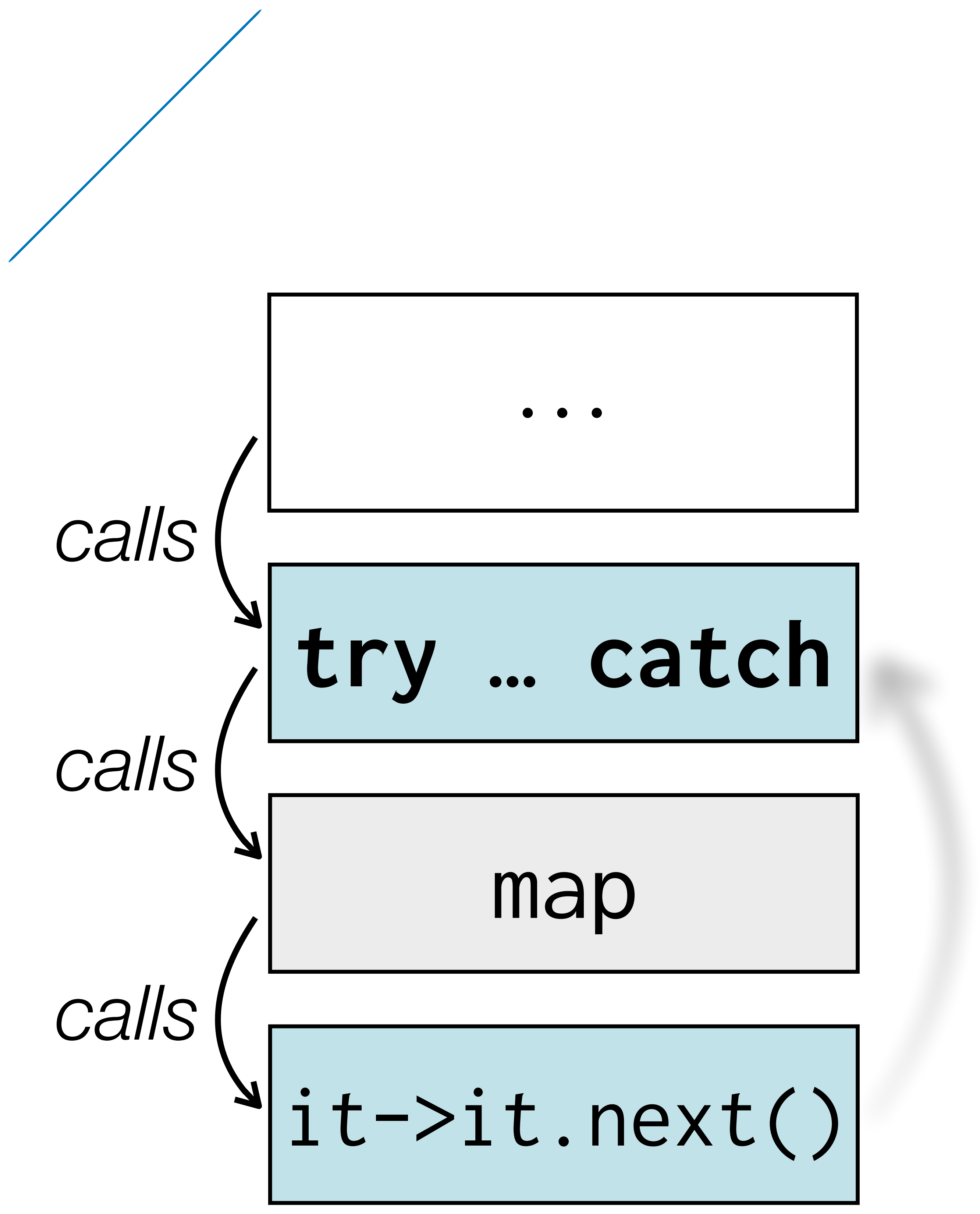
(cf. exception-oblivious contexts in Genus)

Compiler (or programmer) assigns unique names to effect handlers

(cf. unique blame labels in Genus)

Handlers have lexical regions

(cf. local escape analysis in Genus)



The theorem we proved

No unhandled exceptions

Theorem Well-typed programs handle their exceptions.

One more theorem to prove

No unhandled exceptions

Theorem Well-typed programs handle their exceptions.

No accidentally handled exceptions

Theorem ?

One more theorem to prove

No unhandled exceptions

A theorem about type safety Well-typed programs handle their exceptions.

No accidentally handled exceptions

Theorem **?**

One more theorem to prove

No unhandled exceptions

A theorem about type safety

Well-typed programs handle their exceptions.

No accidentally handled exceptions

A theorem beyond type safety

?

One more theorem to prove

No accidentally handled exceptions

A theorem beyond type safety

?

Accidental handling causes implementation details to leak through abstraction boundaries.

One more theorem to prove

No accidentally handled exceptions

A theorem beyond type safety

?

Accidental handling causes implementation details to leak through abstraction boundaries.

The new tunneling semantics fixes this leakage and preserves abstraction.

One more theorem to prove

No accidentally handled exceptions

A theorem about abstraction safety

?

Accidental handling causes implementation details to leak through abstraction boundaries.

The new tunneling semantics fixes this leakage and preserves abstraction.

One more theorem to prove

No accidentally handled exceptions

A theorem
about
abstraction safety

?

[Zhang & Myers 2019]

Accidental handling causes implementation details to leak through abstraction boundaries.

The new tunneling semantics fixes this leakage and preserves abstraction.

One more theorem to prove

No accidentally handled **algebraic effects**

A theorem about abstraction safety

?

[Zhang & Myers 2019]

Accidental handling causes implementation details to leak through abstraction boundaries.

The new tunneling semantics fixes this leakage and preserves abstraction.

Accidental handling is a violation of abstraction

An effect-polymorphic, higher-order abstraction

```
List[Y]/E map[X,Y,E](List[X] xs, X → Y/E f)
```

Accidental handling is a violation of abstraction

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (it.hasNext()) {
    ys.add(f(it.next())); }
  }
  return ys;
}
```

N+1 calls to hasNext()

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch NSE { void throw() { break; } }
  }
  return ys;
}
```

one **throw**-and-**catch**

Accidental handling is a violation of abstraction

Two supposedly equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // one throw-and-catch
}
```

Accidental handling is a violation of abstraction

Two supposedly equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // one throw-and-catch
}
```

Client makes distinct observations

iters contains an empty iterator

```
List[Iterator[int]] iters = …
try {
  ints = map(iters, it->it.next());
  …
} catch NSE { void throw() { … } }
```

Accidental handling is a violation of abstraction

Two supposedly equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // one throw-and-catch
}
```

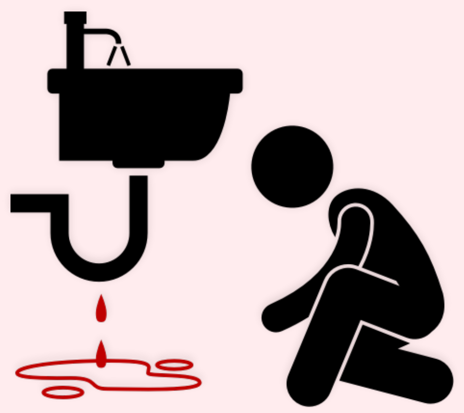
Client makes distinct observations

map raises NSE

map returns normally

iters contains an empty iterator

```
List[Iterator[int]] iters = …
try {
  ints = map(iters, it->it.next());
  …
} catch NSE { void throw() { … } }
```



Accidental handling causes implementation details to leak through abstraction boundaries.

Two supposedly equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // one throw-and-catch
}
```

Client makes distinct observations

map raises NSE

map returns normally

The new tunneling semantics fixes the leakage and preserves abstraction.

Two supposedly equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f){
   … // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f){
   … // one throw-and-catch
}
```

Client makes distinct observations

map raises NSE
if `iters` contains an empty iterator

map returns normally
if `iters` contains an empty iterator

The new tunneling semantics fixes the leakage and preserves abstraction.

Two equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // one throw-and-catch
}
```

Client makes distinct observations

map raises NSE
if `iters` contains an empty iterator

map returns normally
if `iters` contains an empty iterator

The new tunneling semantics fixes the leakage and preserves abstraction.

Two equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // one throw-and-catch
}
```

No client can distinguish between the two implementations

contextual equivalence [Morris 1968]

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

No client can distinguish between the two implementations

contextual equivalence [Morris 1968]

Definition of the logical relation

Algebraic effects render the core language Turing-complete

⇨ Make the logical relation step-indexed
[Appel and McAllester 2001]

Reduction is dependent on surrounding evaluation contexts

⇨ Make the logical relation biorthogonal
[Pitts and Stark 1998]

Runtime allocates fresh names

⇨ Make the logical relation indexed by (degenerate) possible worlds

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

Properties of the logical relation

Abstraction Theorem (aka **Parametricity**)

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

Properties of the logical relation

Abstraction Theorem (aka **Parametricity**)

Abstraction Theorem
for type polymorphism
[Reynolds 1983]

Abstraction Theorem
for effect polymorphism
[Zhang & Myers 2019]

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

Properties of the logical relation

Abstraction Theorem (aka **Parametricity**)

Abstraction Theorem
for type polymorphism
[Reynolds 1983]

```
List[Y] map[X,Y](List[X] l, X → Y f) {   }
```

Should **not** make decisions based on the run-time instantiations of type variables X and Y

Abstraction Theorem
for effect polymorphism
[Zhang & Myers 2019]

```
List[Y]/E map[X,Y,E](List[X] l, X → Y/E f) {   }
```

The abs
in abstra

1 Define c
2 Define t
3 Show th
implies c
4 Profit

Properties of the logical relation

Abstraction Theorem (aka **Parametricity**)

Abstraction Theorem
for type polymorphism
[Reynolds 1983]

```
List[Y] map[X,Y](List[X] l, X → Y f) {   }
```

Should **not** make decisions based on the run-time instantiations of type variables X and Y

Abstraction Theorem
for effect polymorphism
[Zhang & Myers 2019]

```
List[Y]/E map[X,Y,E](List[X] l, X → Y/E f) {   }
```

Should **not** make decisions based on the run-time instantiation of effect variable E

The abs
in abstra

1 Define c

2 Define t

3 Show th
implies c

4 Profit

Properties of the logical relation

Abstraction Theorem (aka **Parametricity**)

Abstraction Theorem
for effect polymorphism
[Zhang & Myers 2019]

```
List[Y]/E map[X,Y,E](List[X] l, X → Y/E f) {   }
```

Should **not** make decisions based on the run-time instantiation of effect variable E

Handling the run-time instantiation of effect variable E breaks abstraction !

The abs
in abstra

1 Define c

2 Define t

3 Show th
implies

4 Profit

Properties of the logical relation

Abstraction Theorem (aka **Parametricity**)

Sound w.r.t. contextual equivalence

Coq mechanization

https://github.com/yizhouzhang/abseff-coq

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

Properties of the logical relation

Abstraction Theorem (aka **Parametricity**)

Sound w.r.t. contextual equivalence

Coq mechanization

`https://github.com/yizhouzhang/abseff-coq`

Results apply to the core language of Genus too.

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

Deriving equivalence results

Abstraction Theorem (aka **Parametricity**)

Sound w.r.t. contextual equivalence

No accidentally handled algebraic effects

Theorem **?**

The abs
in abstra

1 Define c

2 Define t

3 Show th
implies c

4 Profit

Deriving equivalence results

No accidentally handled algebraic effects

Theorem

Abstraction Theorem (aka **Parametricity**)

Sound w.r.t. contextual equivalence

The abs
in abstra

1. Define c
2. Define t
3. Show th
   implies c
4. Profit

Deriving equivalence results

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // one throw-and-catch
}
```

No accidentally handled algebraic effects

Theorem

Abstraction Theorem (aka **Parametricity**)

Sound w.r.t. contextual equivalence

The abstraction story, in abstract

1 Define contextual equivalence

2 Define the logical relation

3 Show that logical relatedness implies contextual equivalence

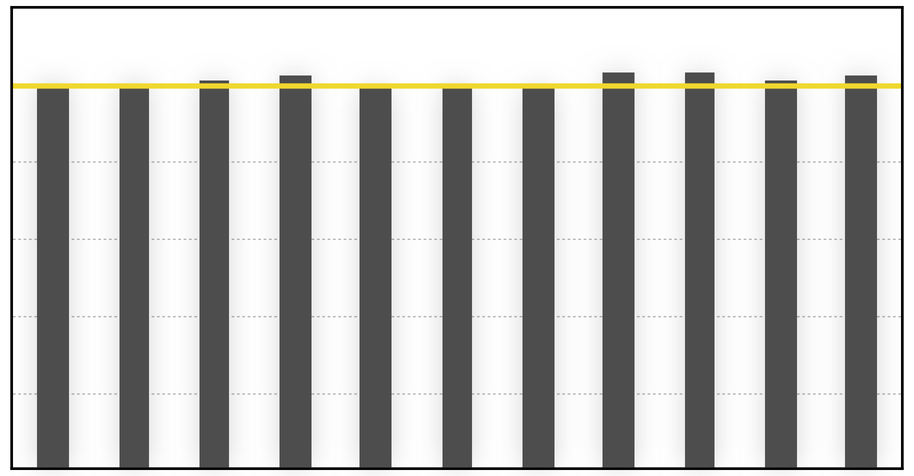
4 Profit

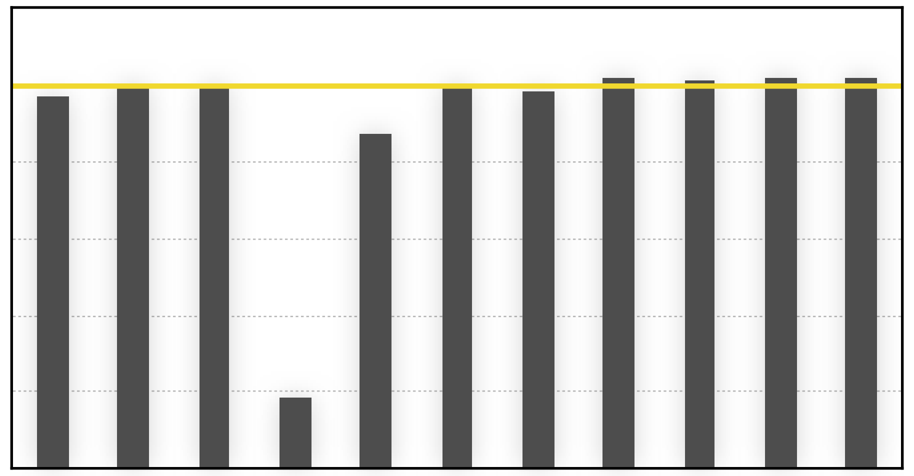
Deriving equivalence results

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // N+1 calls to hasNext()
}
```

⇩ compiler optimization

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  … // one throw-and-catch
}
```





No accidentally handled algebraic effects

Theorem

Abstraction Theorem (aka **Parametricity**)

Sound w.r.t. contextual equivalence

The abs
in abstra

1 Define c

2 Define t

3 Show th
implies c

4 Profit

Abstraction-Safe Effect Handlers via Tunneling

Yizhou Zhang

Cornell University