Accepting Blame for Safe Tunneled Exceptions

Yizhou Zhang*

Guido Salvaneschi†

Quinn Beightol*

Barbara Liskov‡

Andrew Myers*

**Cornell* †*TU Darmstadt* ‡*MIT*

www.cs.cornell.edu/projects/genus

Exceptions

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

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

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

But they are too rigid to support higher-order programming!

Checked exceptions are too rigid

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {…}
```

Checked exceptions are too rigid

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {…}
```

A method that declares a checked exception

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

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

Checked exceptions are too rigid

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {…}
```

A method that declares a checked exception

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

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

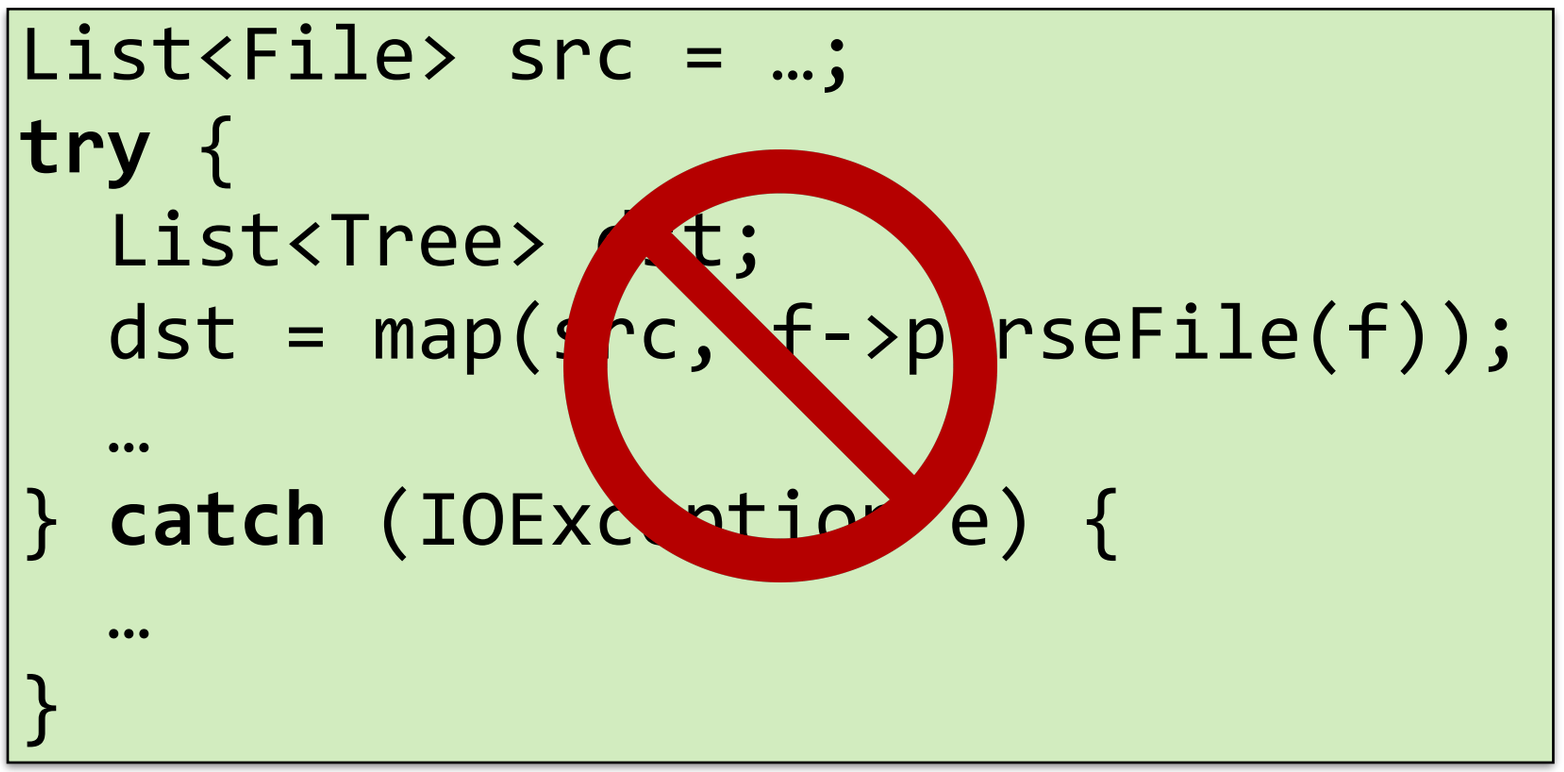
Checked exceptions are too rigid

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {…}
```

A method that declares a checked exception

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



Checked exceptions are too rigid

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {…}
```

A method that declares a checked exception

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

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

javac error: Exception mismatch between `parseFile` and `Function::apply`.

```
@FunctionalInterface
interface Function<T,R> {
    R apply(T t);
}
```

Checked exceptions are too rigid

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {…}
```

A method that declares a checked exception

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

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

javac error: Exception mismatch between `parseFile` and `Function::apply`.

javac error: `IOException` is never thrown in `try` block.

Checked exceptions are too rigid

An unchecked wrapper class introduced in Java 8

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

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

Checked exceptions are too rigid

An unchecked wrapper class introduced in Java 8

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

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

Checked exceptions are too rigid

An unchecked wrapper class introduced in Java 8

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

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

exception wrapping

Checked exceptions are too rigid

An unchecked wrapper class introduced in Java 8

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

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

exception wrapping

exception unwrapping

Checked exceptions are too rigid

An unchecked wrapper class introduced in Java 8

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

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

exception wrapping

exception unwrapping

Checked exceptions are too rigid

Java

1990

2000

2010

Checked exceptions are too rigid

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

—Anders Hejlsberg—

Java

C#

1990

2000

2010

Checked exceptions are too rigid

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

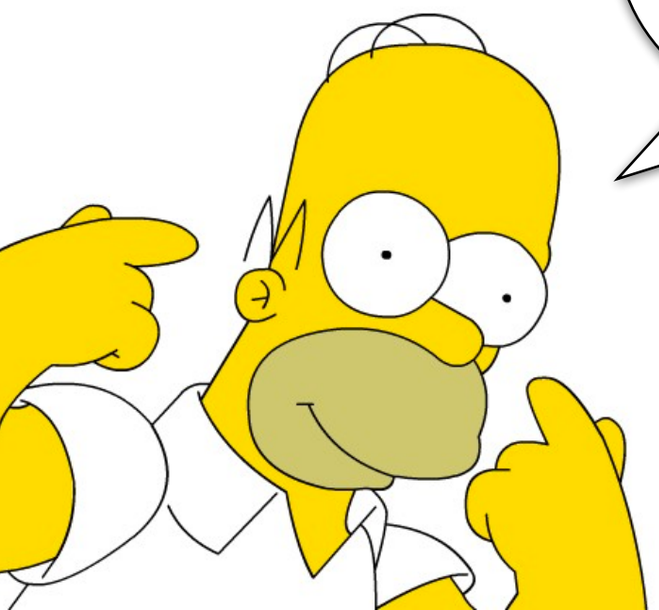
—Anders Hejlsberg—

1990

Java

C#

2000



Scala

F#

Go

2010

TypeScript

Ceylon

julia

Kotlin

Accidentally caught exceptions

Accidentally caught exceptions

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    dst.add(f.apply(i.next())); }
  }
  return dst;
}
```

Accidentally caught exceptions

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    dst.add(f.apply(i.next())); }
  }
  return dst;
}
```

```
interface Iterator[E] {
  // The exception indicates no more element.
  E next() throws NoSuchElementException;
  …
}
```

Accidentally caught exceptions

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

```
interface Iterator[E] {
  // The exception indicates no more element.
  E next() throws NoSuchElementException;
  …
}
```

Accidentally caught exceptions

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

The `catch` block is intended for exceptions from `i.next()`.

```
interface Iterator[E] {
  // The exception indicates no more element.
  E next() throws NoSuchElementException;
  …
}
```

Accidentally caught exceptions

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

`f` could be a function that throws `NoSuchElementException`.

The `catch` block is intended for exceptions from `i.next()`.

Accidentally caught exceptions

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

`f` could be a function that throws `NoSuchElementException`.

The `catch` block is intended for exceptions from `i.next()`.

Caught by accident!

Accidentally caught exceptions

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

`f` could be a function that throws `NoSuchElementException`.

The `catch` block is intended for exceptions from `i.next()`.

Caught by accident!

Bug: caller receives only the transformation of a list prefix

To check, or not to check:

Checked exception types

```
IOException
SQLException
ParseException
DataFormatException
…
```

Unchecked exception types

```
EmptyStackException
UncheckedIOException
NoSuchElementException
IndexOutOfBoundsException
…
```

To check, or not to check:

Checked exception types

`IOException`
`SQLException`
`ParseException`
`DataFormatException`
…

Unchecked exception types

`EmptyStackException`
`UncheckedIOException`
`NoSuchElementException`
`IndexOutOfBoundsException`
…

To check, or not to check:

Checked exception types

`IOException`
`SQLException`
`ParseException`
`DataFormatException`
…

Too rigid to work with.

Unchecked exception types

`EmptyStackException`
`UncheckedIOException`
`NoSuchElementException`
`IndexOutOfBoundsException`
…

Problems remain.

To check, or not to check:

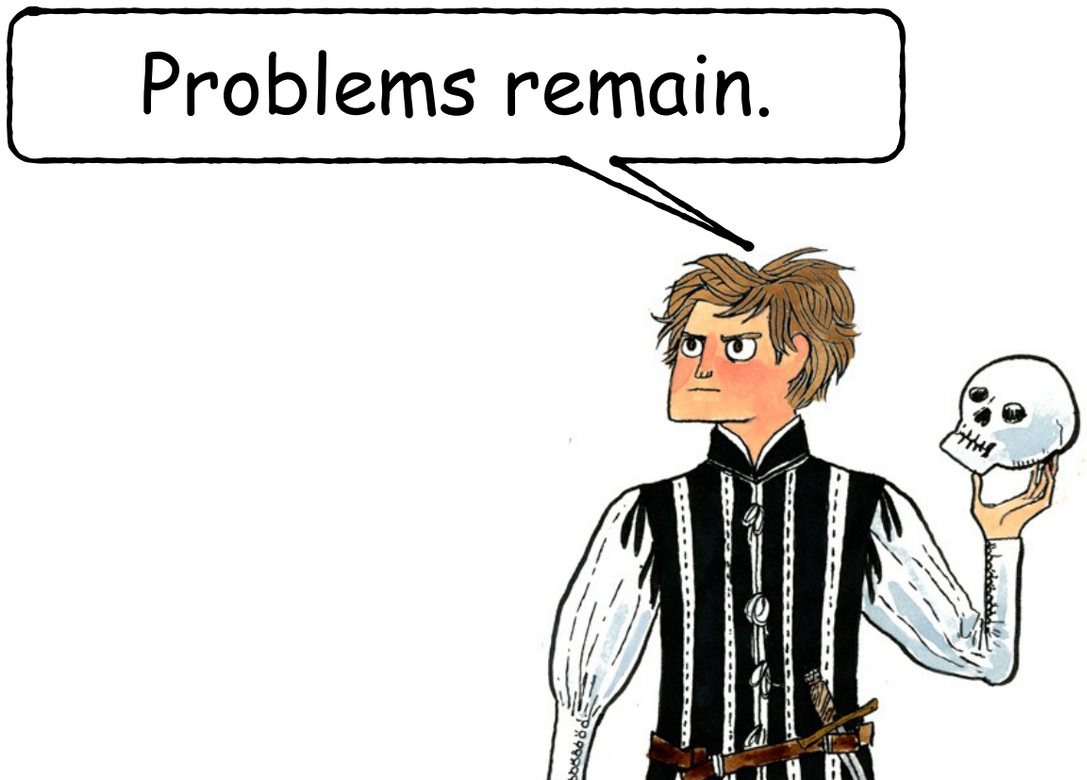
not a property of the exception type

Checked exception types

```
IOException
SQLException
ParseException
DataFormatException
...
```

Unchecked exception types

```
EmptyStackException
UncheckedIOException
NoSuchElementException
IndexOutOfBoundsException
...
```

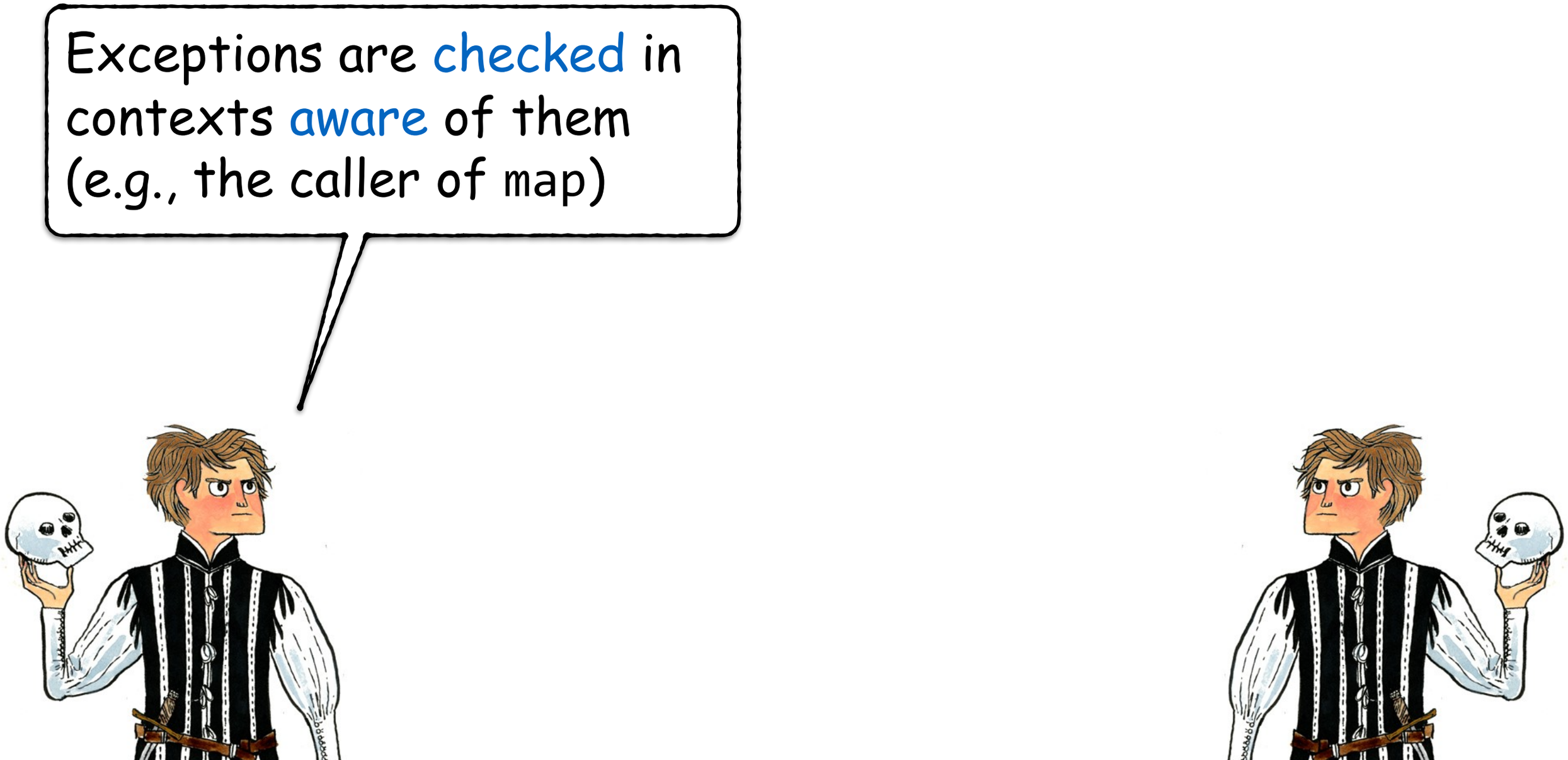


To check, or not to check:

not a property of the exception type,
but rather context-dependent.

To check, or not to check:

not a property of the exception type, but rather context-dependent.



To check, or not to check:

not a property of the exception type, but rather context-dependent.

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)

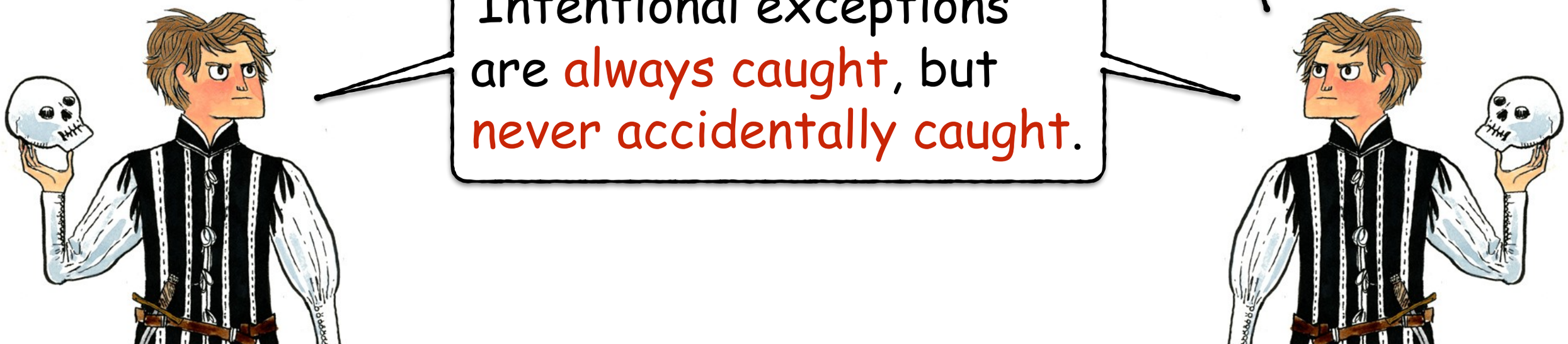
To check, or not to check:

not a property of the exception type, but rather context-dependent.

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)

Intentional exceptions are always caught, but never accidentally caught.



Genus

Genus

A variant of Java with lightweight, flexible, object-oriented generics

[Zhang et al. 2015]

Language constructs for exceptions

Inherits from Java `throw` statements, `throws` clauses, and `try`-`catch`-`finally` statements

Core ideas applicable to other languages

2 classes of exceptional events

2 classes of exceptional events

failure

violation of programmer assumptions

unusual condition

unusual, but planned-for

2 classes of exceptional events

failure

violation of programmer assumptions

- programmer mistakes
- running out of memory, etc

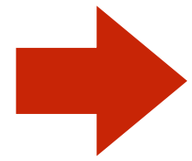
unusual condition

unusual, but planned-for

2 classes of exceptional events

failure	**violation of programmer assumptions**	- programmer mistakes - running out of memory, etc
unusual condition	**unusual, but planned-for**	such as an unusual case of an algorithm

Design goals *differ*

failure	violation of programmer assumptions	 stack trace as a debugging aid
unusual condition	**unusual, but planned-for**	such as an unusual case of an algorithm

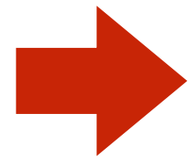
Design goals *differ*

failure	violation of programmer assumptions	stack trace as a debugging aid
unusual condition	unusual, but planned-for	static checking & efficiency

Design goals *differ*

failure

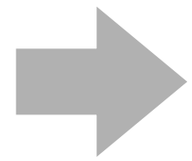
violation of programmer assumptions



stack trace as a debugging aid

unusual condition

unusual, but planned-for



static checking & efficiency

A failure event

```
// Returns: the first element.
// Precondition: the iterable is non-empty.
E first[E](Iterable[E] iterable)
{
  return iterable.iterator().next();
}
```

A failure event

```
// Returns: the first element.
// Precondition: the iterable is non-empty.
E first[E](Iterable[E] iterable)
{
  return iterable.iterator().next();
}
```

```
interface Iterator[E] {
  // The exception indicates no more
  // element.
   E next()
     throws NoSuchElementException;
  …
}
```

A failure event

```
// Returns: the first element.
// Precondition: the iterable is non-empty.
E first[E](Iterable[E] iterable)
{
  return iterable.iterator().next();
}
```

The exception is impossible under the precondition.

```
interface Iterator[E] {
  // The exception indicates no more
  // element.
  E next()
    throws NoSuchElementException;
  …
}
```

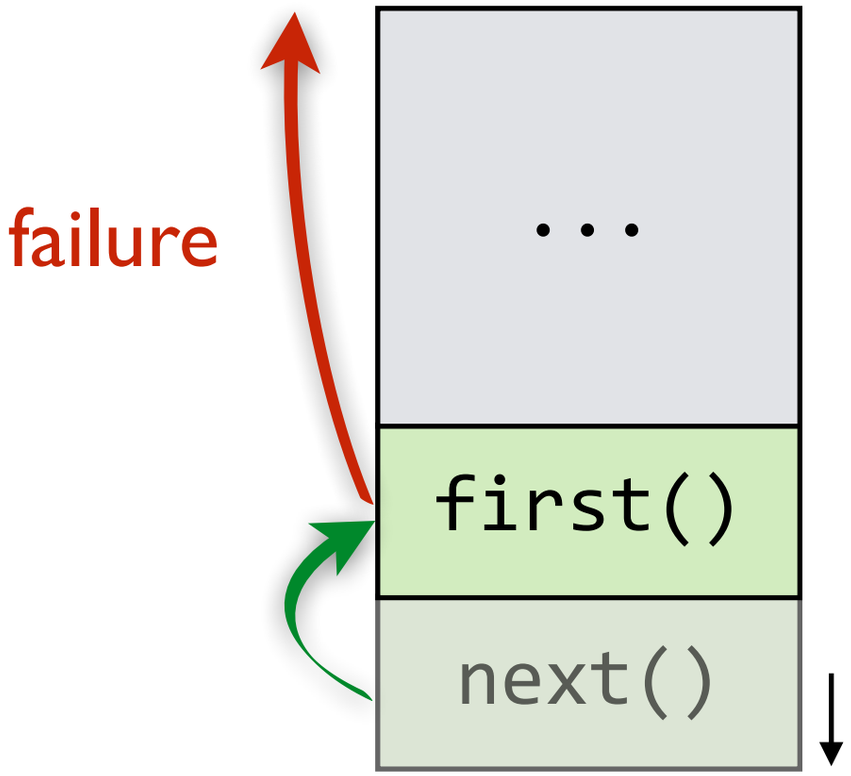
Indicating failure events with `fails` clauses

```
// Returns: the first element.
// Precondition: the iterable is non-empty.
E first[E](Iterable[E] iterable)
    fails NoSuchElementException {
  return iterable.iterator().next();
}
```

The exception is impossible under the precondition.

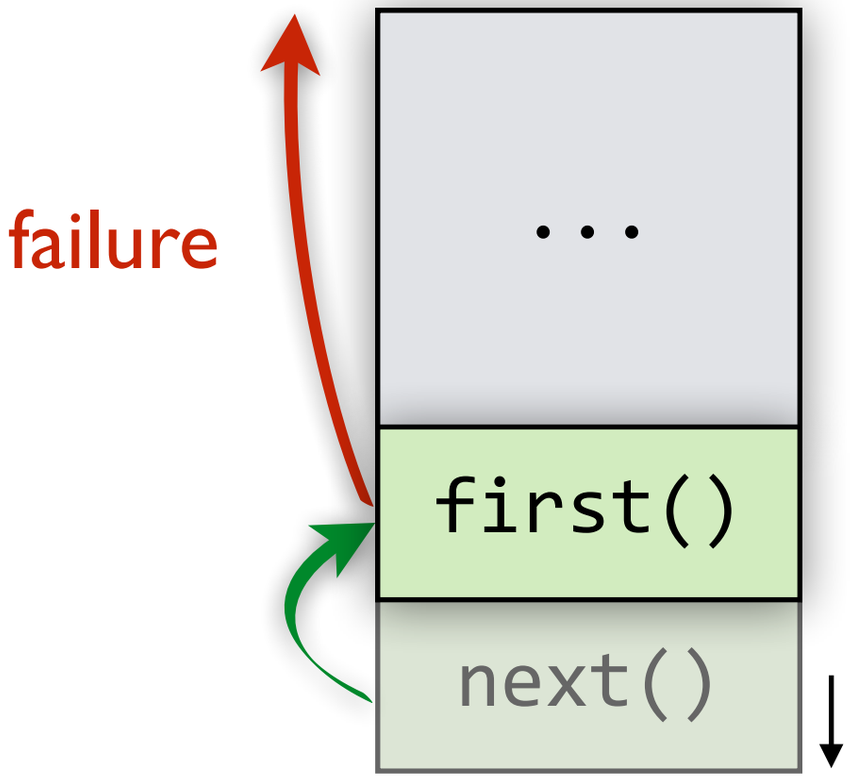
Failure exceptions crash the program

```
// Returns: the first element.
// Precondition: the iterable is non-empty.
E first[E](Iterable[E] iterable)
    fails NoSuchElementException {
  return iterable.iterator().next();
}
```



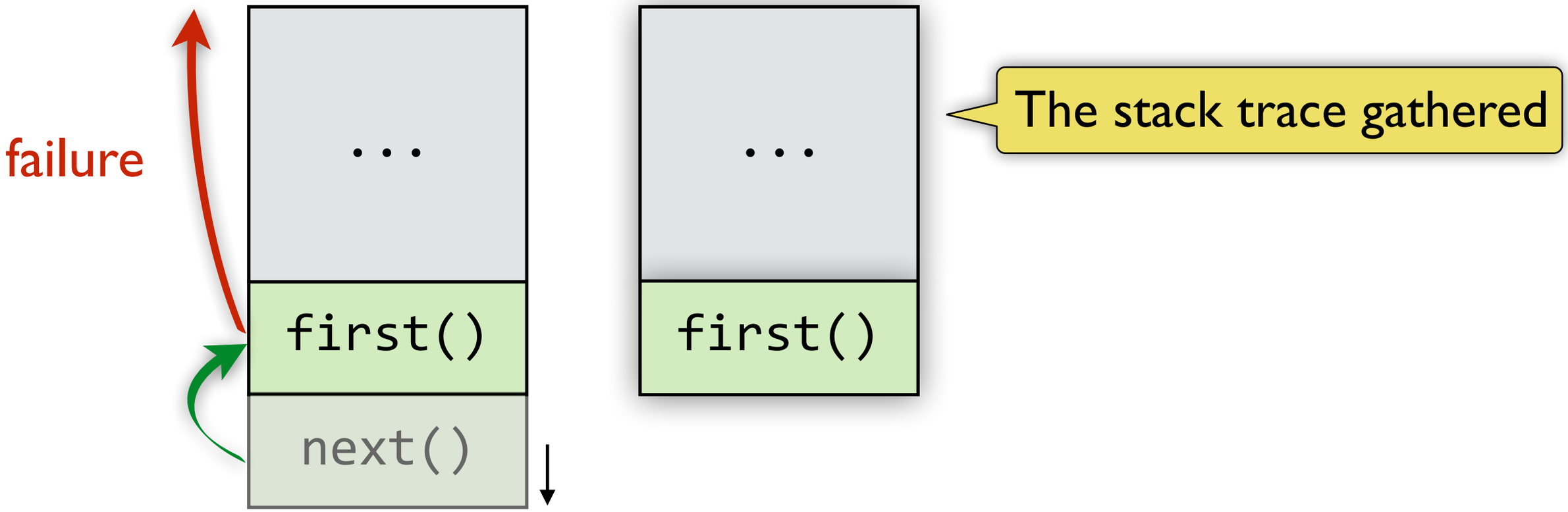
Failure exceptions gather a stack trace

```
// Returns: the first element.
// Precondition: the iterable is non-empty.
E first[E](Iterable[E] iterable)
    fails NoSuchElementException {
  return iterable.iterator().next();
}
```



Failure exceptions gather a stack trace

```
// Returns: the first element.
// Precondition: the iterable is non-empty.
E first[E](Iterable[E] iterable)
    fails NoSuchElementException {
  return iterable.iterator().next();
}
```



Design goals *differ*

failure

violation of programmer assumptions → stack trace as a debugging aid

unusual condition

unusual, but planned-for → **static checking & efficiency**

Recall…

A higher-order function in Java

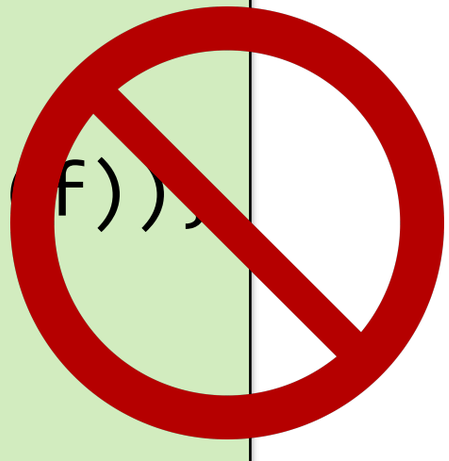
```
<T,R> List<R> map(List<T> src, Function<T,R> f) {…}
```

A method that declares a checked exception

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

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

javac error: Exception mismatch between `parseFile` and `Function::apply`.



javac error: `IOException` is never thrown in `try` block.

Rewriting in Genus…

A higher-order function in Genus

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {…}
```

A method that declares an exception

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

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

Rewriting in Genus…

A higher-order function in Genus

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {…}
```

A method that declares an exception

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

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

Tunneling through higher-order functions

A higher-order function in Genus

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {…}
```

A method that declares an exception

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

Exception mismatch is allowed.

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

Tunneling through higher-order functions

A higher-order function in Genus

exception-oblivious

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {…}
```

A method that declares an exception

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

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

Tunneling through higher-order functions

A higher-order function in Genus

exception-oblivious

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {…}
```

A method that declares an exception

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

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

exception-aware

Tunneling through higher-order functions

A higher-order function in Genus

exception-oblivious

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {…}
```

A method that declares an exception

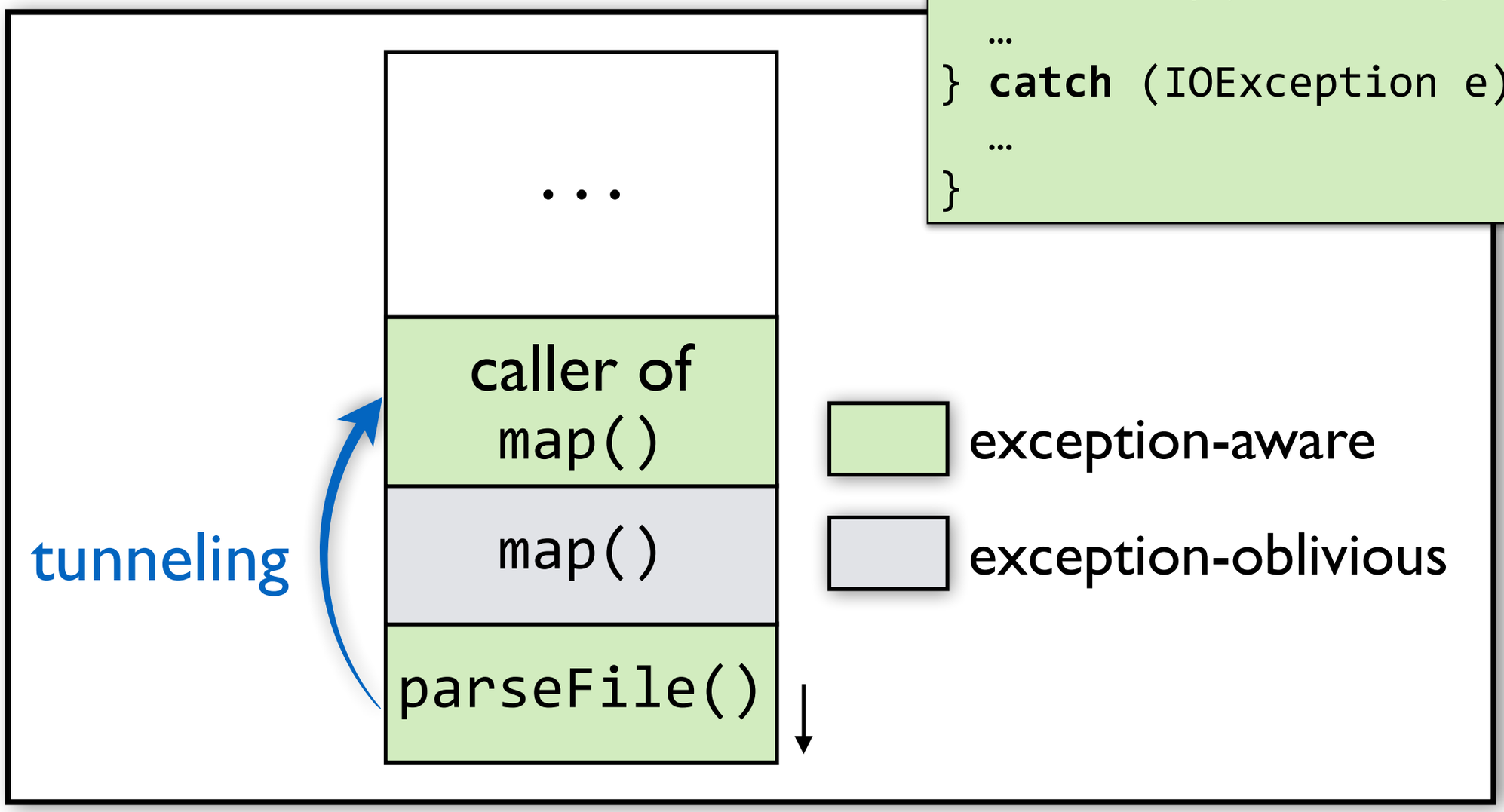
```
Tree parseFile(File f) throws IOException {…}
```

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

exception-aware

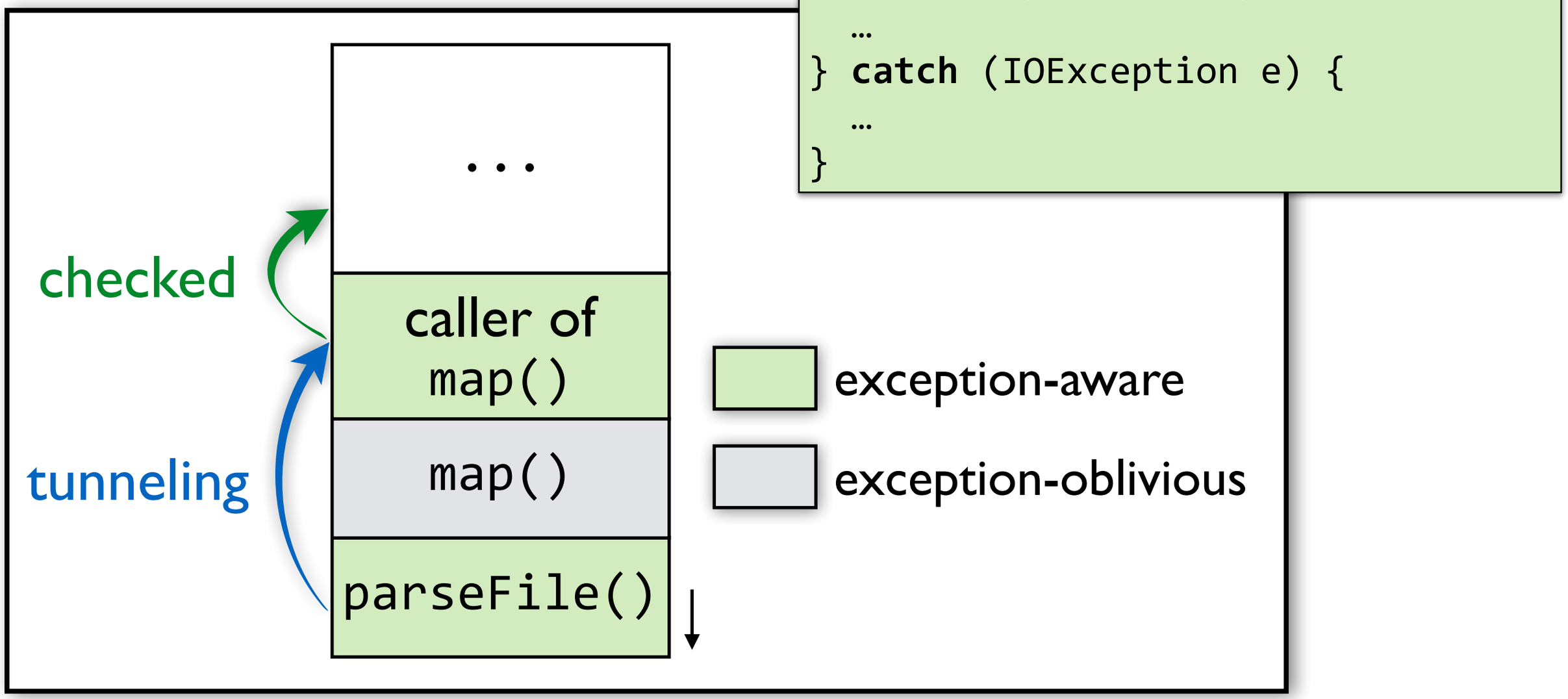
Tunneling through higher-order functions

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



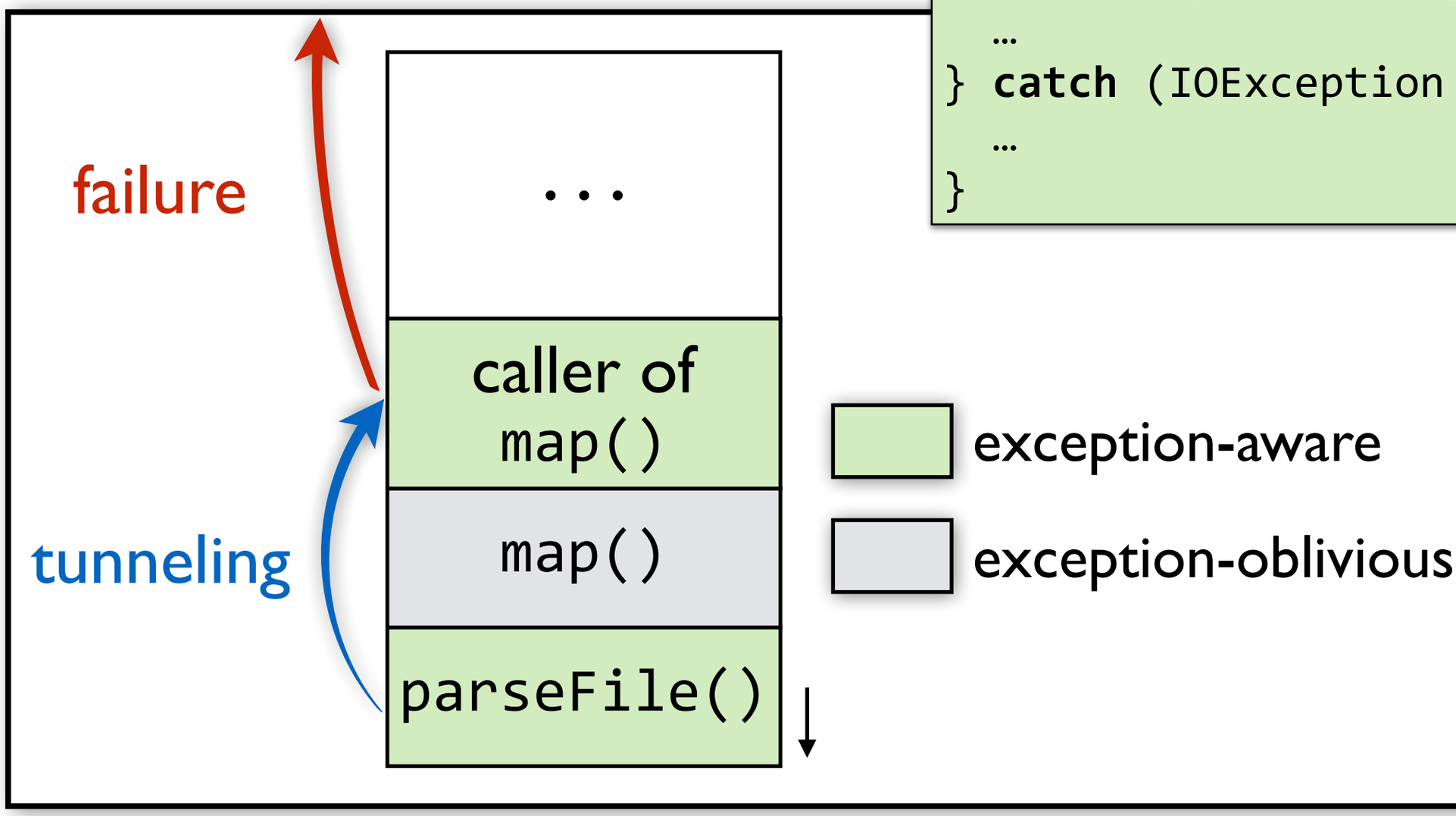
Tunneling through higher-order functions

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



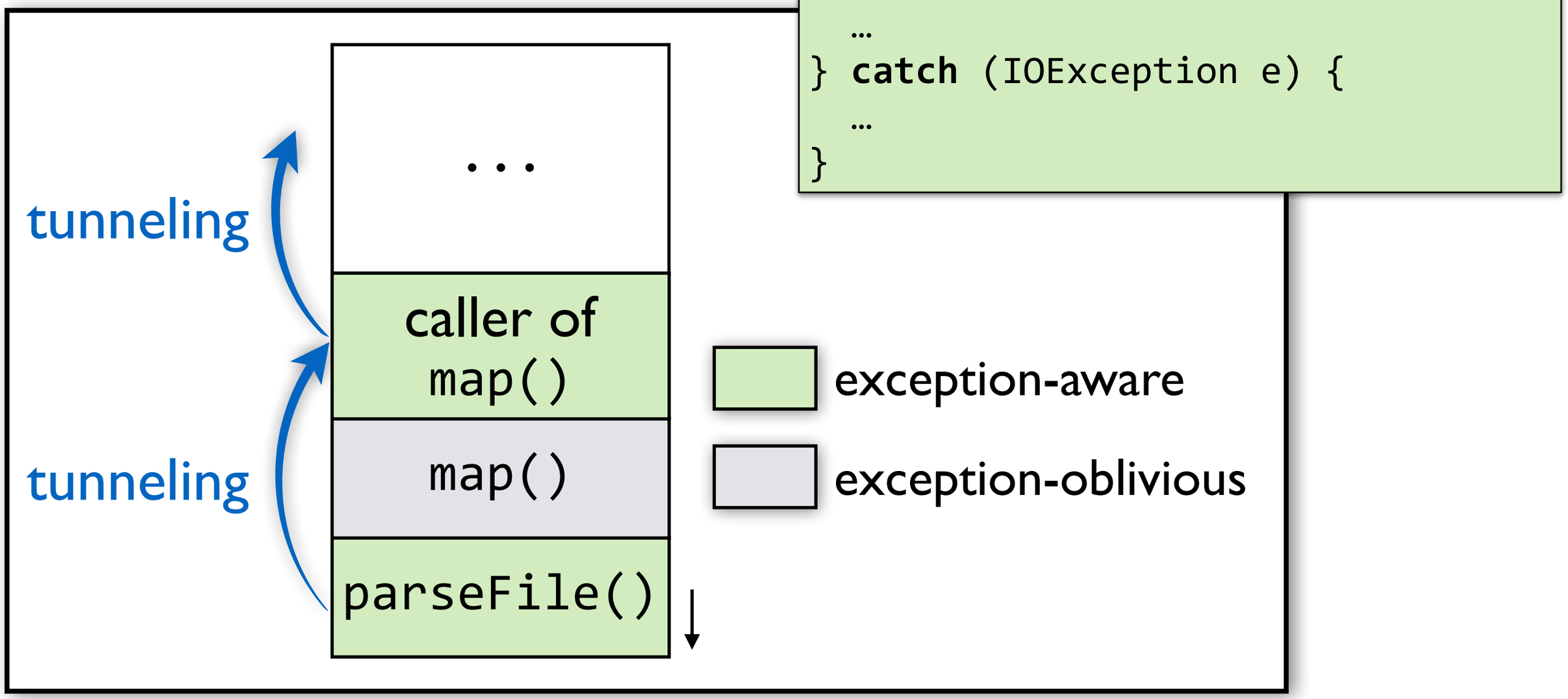
Tunneling through higher-order functions

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



Tunneling through higher-order functions

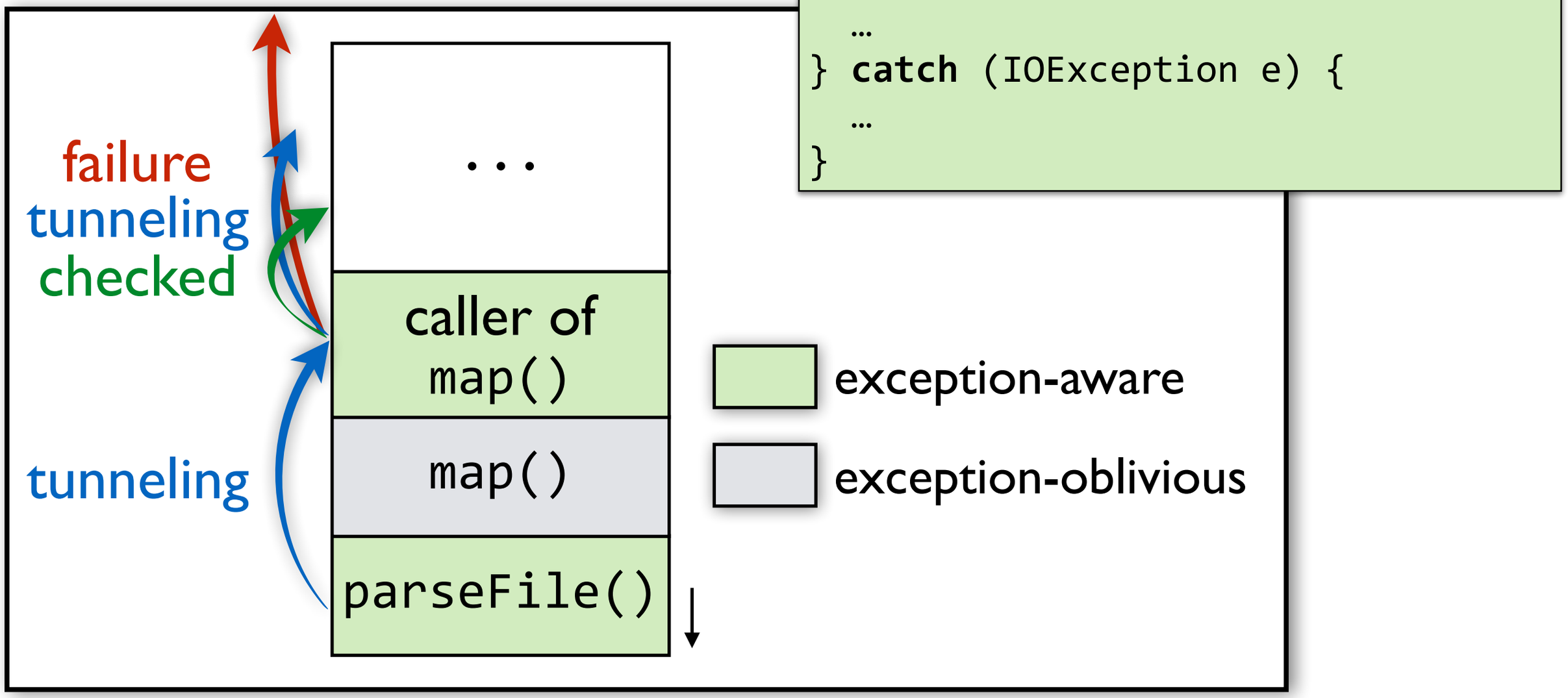
```
List[File] src = …;
try {
  List[Tree] dst;
  dst = map(src, f->parseFile(f));
  …
} catch (IOException e) {
  …
}
```



Tunneling through higher-order functions

- **Static checking** enforced in exception-aware contexts

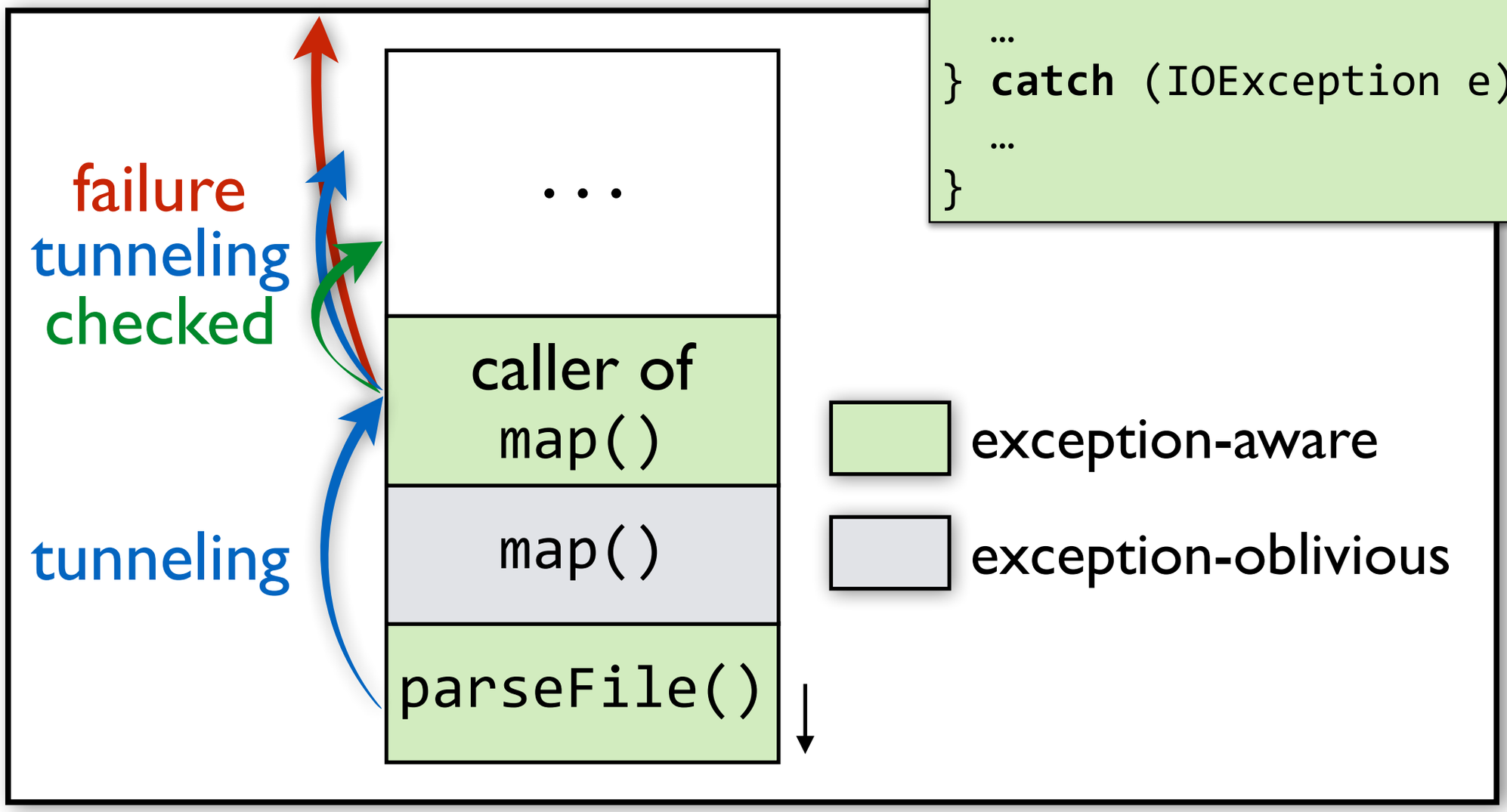
```
List[File] src = …;
try {
  List[Tree] dst;
  dst = map(src, f->parseFile(f));
  …
} catch (IOException e) {
  …
}
```



Tunneling through higher-order functions

- **Static checking** enforced in exception-aware contexts
- **Efficient** because of no stack-trace collection

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



Accidentally caught exceptions

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

`f` could be a function that throws `NoSuchElementException`.

The `catch` block is intended for exceptions from `i.next()`.

Caught by accident!

Bug: caller receives only the transformation of a list prefix

The “exception capture” problem

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

`f` could be a function that throws `NoSuchElementException`.

The `catch` block is intended for exceptions from `i.next()`.

Caught by accident!

Bug: caller receives only the transformation of a list prefix

The “exception capture” problem

A higher-order function in Java

```
<T,R> List<R> map(List<T> src, Function<T,R> f) {
  List<R> dst = new ArrayList<R>();
  for (Iterator<T> i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

`f` could be a function that throws `NoSuchElementExcept`

The `catch` block is intended for exceptions from `i.next()`.

Exception identifiers collide.

(exception identifier = exception type)

Context-awareness addresses exception capture

A higher-order function in Genus

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

(exception identifier = exception type + context)

Context-awareness addresses exception capture

A higher-order function in Genus

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

Oblivious to exceptions raised by `f`.

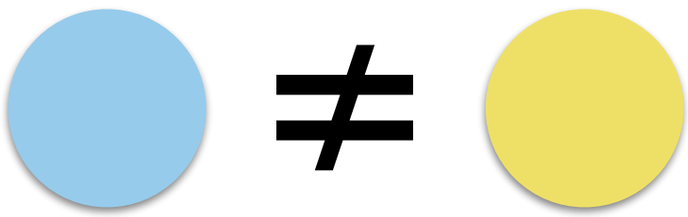
(exception identifier = exception type + context)

Context-awareness addresses exception capture

A higher-order function in Genus

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

Oblivious to exceptions raised by `f`.



Exception identifiers differ.

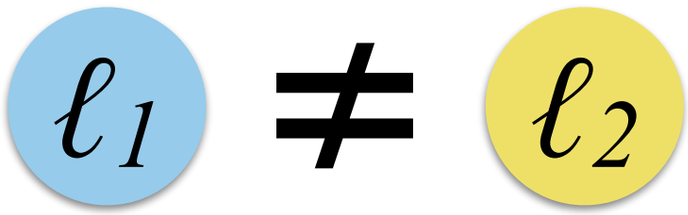
(exception identifier = exception type + context)

Context-awareness addresses exception capture

A higher-order function in Genus

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

Oblivious to exceptions raised by `f`.



Exception identifiers differ.

(exception identifier = exception type + context)

Blame labels as a notion of context

Gradual typing

Language

type mismatch

Compiler

blame label assigned

Runtime

type-cast failure attributed to the proper context

Blame labels as a notion of context

	Language	Compiler	Runtime
Gradual typing	type mismatch	blame label assigned	type-cast failure attributed to the proper context
Genus	exception mismatch	blame label assigned	exception propagated to the proper context

Blame labels as a notion of context

	Language	Compiler	Runtime
Gradual typing	type mismatch	blame label assigned	type-cast failure attributed to the proper context
Genus	exception mismatch	blame label assigned	exception propagated to the proper context

Blame labels impose a static responsibility on the context to handle corresponding exceptions.

Blame labels as a notion of context

	Language	Compiler	Runtime
Gradual typing	type mismatch	blame label assigned	type-cast failure attributed to the proper context
Genus	exception mismatch	blame label assigned	exception propagated to the proper context

Blame labels impose a **static responsibility** on the context to handle corresponding exceptions.

Blame labels as a notion of context

	Language	Compiler	Runtime
Gradual typing	type mismatch	blame label assigned	type-cast failure attributed to the proper context
Genus	exception mismatch	blame label assigned	exception propagated to the proper context

Blame labels impose a **static responsibility** on the context to handle **corresponding** exceptions.

Augmenting exception identifiers with implicit blame labels prevents capture

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

```
void g() fails NoSuchElementException {
  … map(src, Iterator::next) …
}
```

Genus

exception mismatch → blame label assigned → exception propagated to the proper context

Augmenting exception identifiers with implicit blame labels prevents capture

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

```
void g() fails NoSuchElementException {
  … map(src, Iterator::next) …
}
```

Genus

exception mismatch → blame label assigned → exception propagated to the proper context

Augmenting exception identifiers with implicit blame labels prevents capture

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

```
void g() fails NoSuchElementException {
  … map(src, Iterator::next) …
}
```

ℓ_1

Genus

exception mismatch → blame label assigned → exception propagated to the proper context

Augmenting exception identifiers with implicit blame labels prevents capture

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

```
void g() fails NoSuchElementException {
  … map(src, Iterator::next) …
}
```

ℓ_1

Genus

exception mismatch → blame label assigned → exception propagated to the proper context

Augmenting exception identifiers with implicit blame labels prevents capture

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

```
void g() fails NoSuchElementException {
  … map(src, Iterator::next) …
}
```

ℓ_1

Genus

exception mismatch → blame label assigned → exception propagated to the proper context

Augmenting exception identifiers with implicit blame labels prevents capture

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

ℓ_2

```
void g() fails NoSuchElementException {
  … map(src, Iterator::next) …
}
```

ℓ_1

Genus

exception mismatch → blame label assigned → exception propagated to the proper context

Augmenting exception identifiers with implicit blame labels prevents capture

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

ℓ_2

```
void g() fails NoSuchElementException {
  … map(src, Iterator::next) …
}
```

ℓ_1

Genus

exception mismatch → blame label assigned → exception propagated to the proper context

Augmenting exception identifiers with implicit blame labels prevents capture

```
List[R] map[T,R] (List[T] src, Function[T,R] f) {
  List[R] dst = new ArrayList[R]();
  for (Iterator[T] i = src.iterator();;) {
    try { dst.add(f.apply(i.next())); }
    catch (NoSuchElementException e) { break; }
  }
  return dst;
}
```

ℓ_2

```
void g() fails NoSuchElementException {
  … map(src, Iterator::next) …
}
```

ℓ_1

Genus

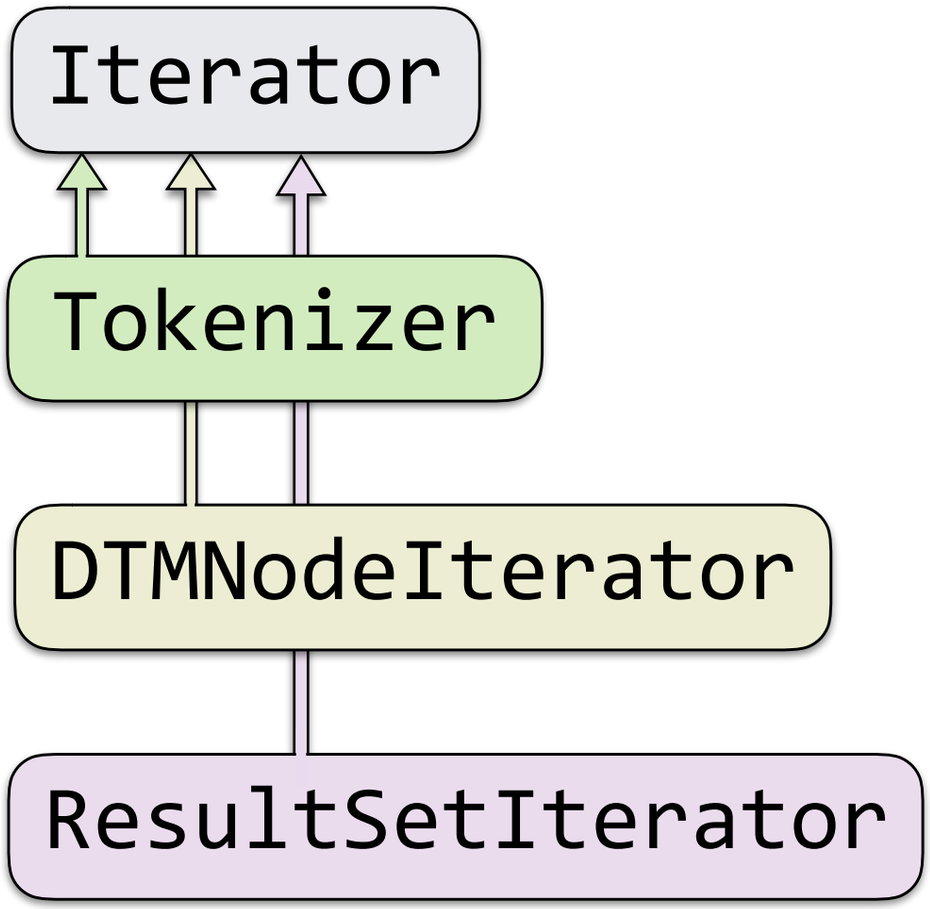
exception mismatch → blame label assigned → exception propagated to the proper context

A complement to behavioral subtyping

{ Function
Iterator
Visitor
Factory
... }

Subtypes may want to throw new exceptions.

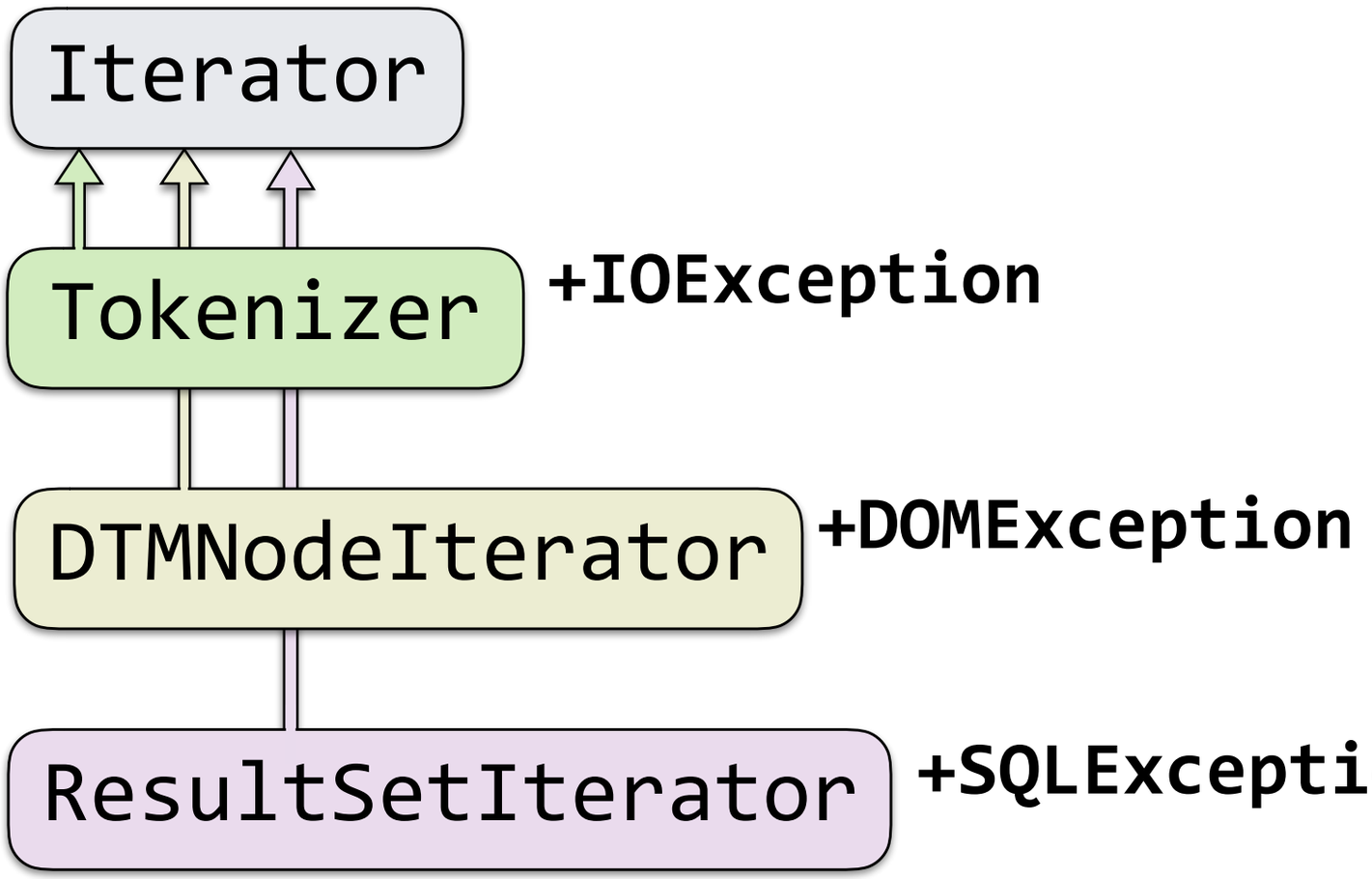
A complement to behavioral subtyping



{ Function
Iterator
Visitor
Factory
... }

Subtypes may want to throw new exceptions.

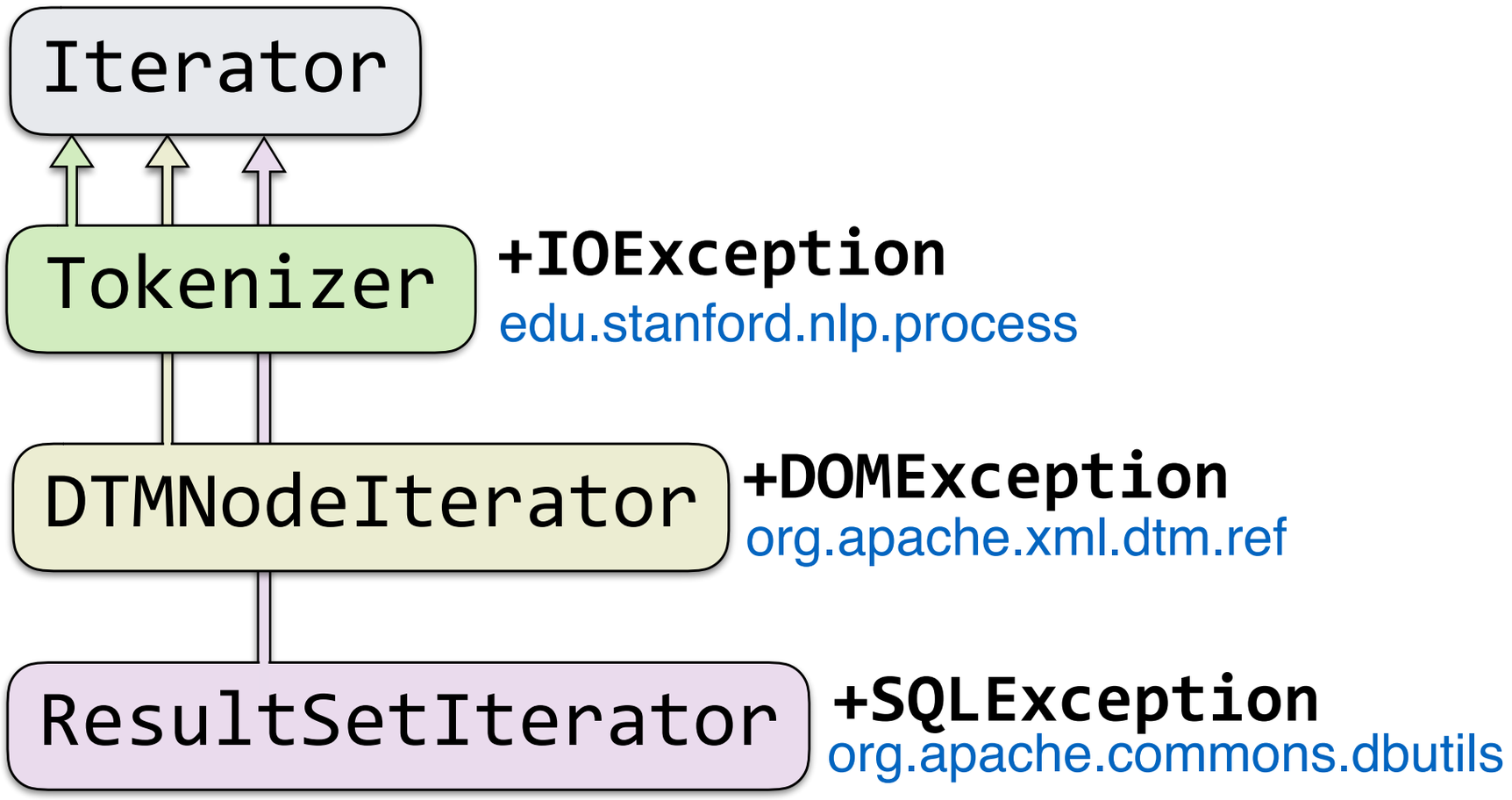
A complement to behavioral subtyping



{ Function
Iterator
Visitor
Factory
... }

Subtypes may want to throw new exceptions.

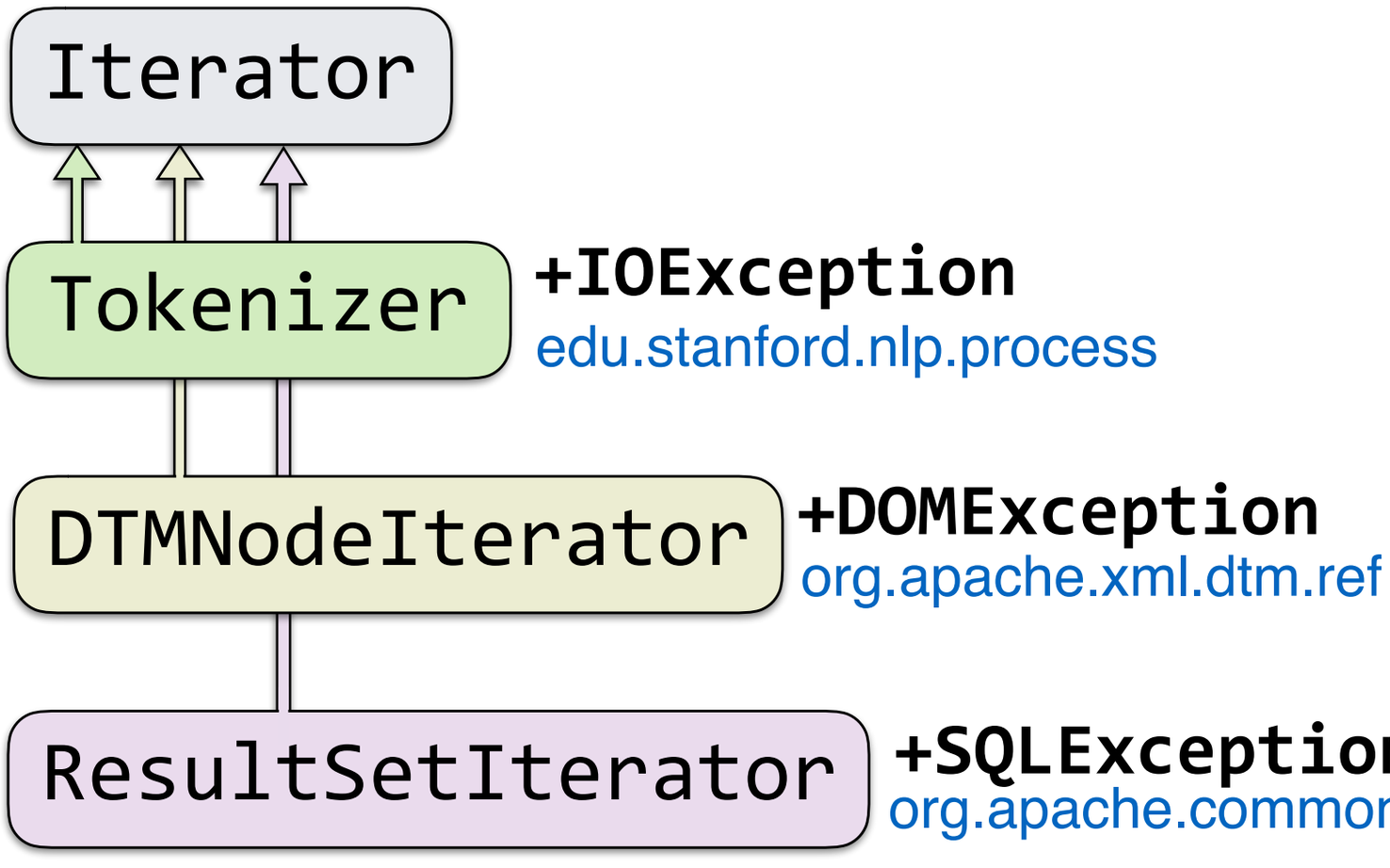
A complement to behavioral subtyping



{ Function
Iterator
Visitor
Factory
... }

Subtypes may want to throw new exceptions.

A complement to behavioral subtyping



{ Function
Iterator
Visitor
Factory
... }

Subtypes may want to throw new exceptions.

Behavioral subtyping: subtypes cannot add new exceptions.

A complement to behavioral subtyping

{ Function
Iterator
Visitor
Factory
... }

Subtypes may want to throw new exceptions.

Behavioral subtyping: subtypes cannot add new exceptions, **unless the supertype is weak.**

A complement to behavioral subtyping

Indicates mismatch is allowed

```
weak interface Factory[T] { T make(); }
```

{ Function
Iterator
Visitor
Factory
... }

Subtypes may want to throw new exceptions.

Behavioral subtyping: subtypes cannot add new exceptions, **unless the supertype is weak.**

A complement to behavioral subtyping

Indicates mismatch is allowed

```
weak interface Factory[T] { T make(); }
```

```
class ConnectionFactory
implements Factory[Connection] {
  Connection make() throws SQLException {…}
}
```

{ Function Iterator Visitor Factory ... }

Subtypes may want to throw new exceptions.

Behavioral subtyping: subtypes cannot add new exceptions, **unless the supertype is weak.**

A complement to behavioral subtyping

Indicates mismatch is allowed

```
weak interface Factory[T] { T make(); }
```

```
class ConnectionFactory
implements Factory[Connection] {
  Connection make() throws SQLException {…}
}
```

An exception mismatch

```
Factory[Connection] f =
  new ConnectionFactory();
try { Connection conn = f.make(); … }
catch (SQLException e) {…}
```

{ Function, Iterator, Visitor, Factory, ... }

Subtypes may want to throw new exceptions, **but the extra exceptions are checked.**

Behavioral subtyping: subtypes cannot add new exceptions, **unless the supertype is weak.**

A complement to behavioral subtyping

Indicates mismatch is allowed

```
weak interface Factory[T] { T make(); }
```

```
class ConnectionFactory
implements Factory[Connection] {
  Connection make() throws SQLException {…}
}
```

An exception mismatch

```
Factory[Connection] f =
  new ConnectionFactory();
try { Connection conn = f.make(); … }
catch (SQLException e) {…}
```

Accounts for the mismatch

Behavioral subtyping: subtypes cannot add new exceptions, **unless the supertype is weak.**

{ Function Iterator Visitor Factory ... }

Subtypes may want to throw new exceptions, **but the extra exceptions are checked.**

A mismatch analysis

Infers mismatch inherent in weak-ly typed local variables.

```
Factory[Connection] f =
  new ConnectionFactory();
try { Connection conn = f.make(); … }
catch (SQLException e) {…}
```

A mismatch analysis

Infers mismatch inherent in weak-ly typed local variables.

```
Factory[Connection]⟨ℓ⟩ f =
  new ConnectionFactory();
try { Connection conn = f.make(); … }
catch (SQLException e) {…}
```

assignment to f induces a constraint

$$\texttt{Factory[Connection]}\langle \texttt{make} : \texttt{SQLException} \rangle \leq \texttt{Factory[Connection]}\langle \ell \rangle$$

A mismatch analysis

Infers mismatch inherent in weak-ly typed local variables.

```
Factory[Connection]⟨ℓ⟩ f =
  new ConnectionFactory();
try { Connection conn = f.make(); … }
catch (SQLException e) {…}
```

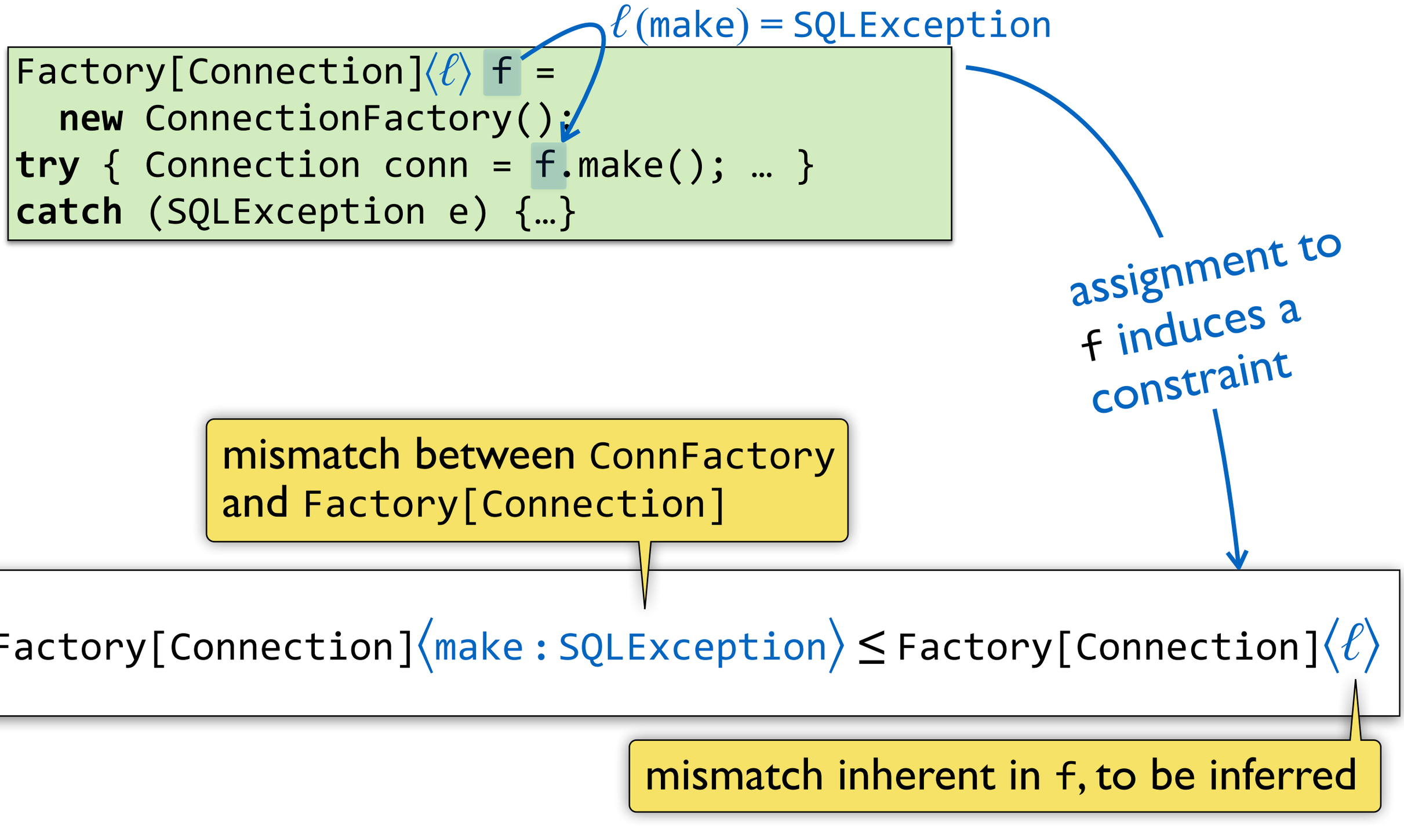
assignment to `f` induces a constraint

$$\texttt{Factory[Connection]}\langle \texttt{make} : \texttt{SQLException} \rangle \leq \texttt{Factory[Connection]}\langle \ell \rangle$$

mismatch inherent in `f`, to be inferred

A mismatch analysis

Infers mismatch inherent in `weak`-ly typed local variables.



Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

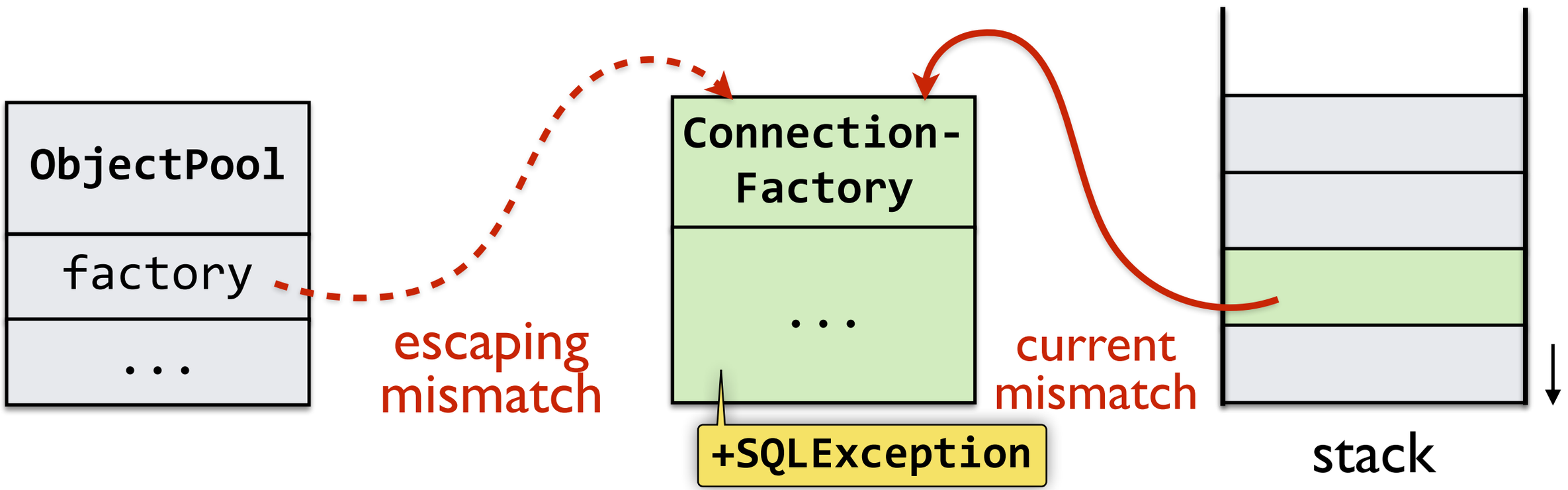
```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

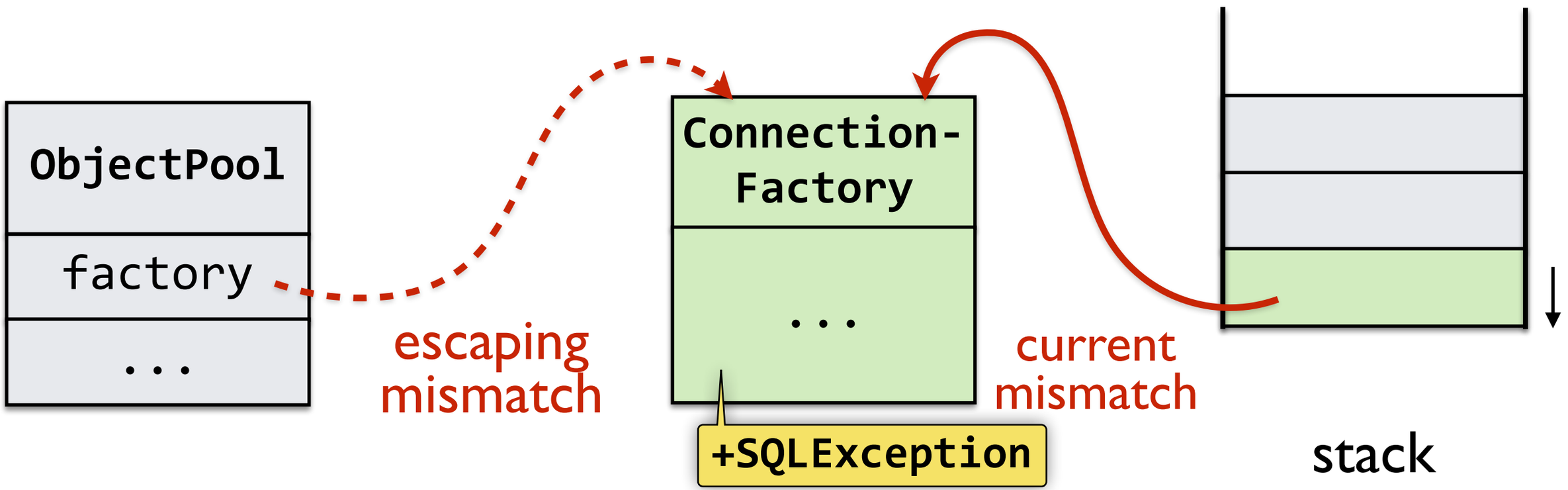


Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

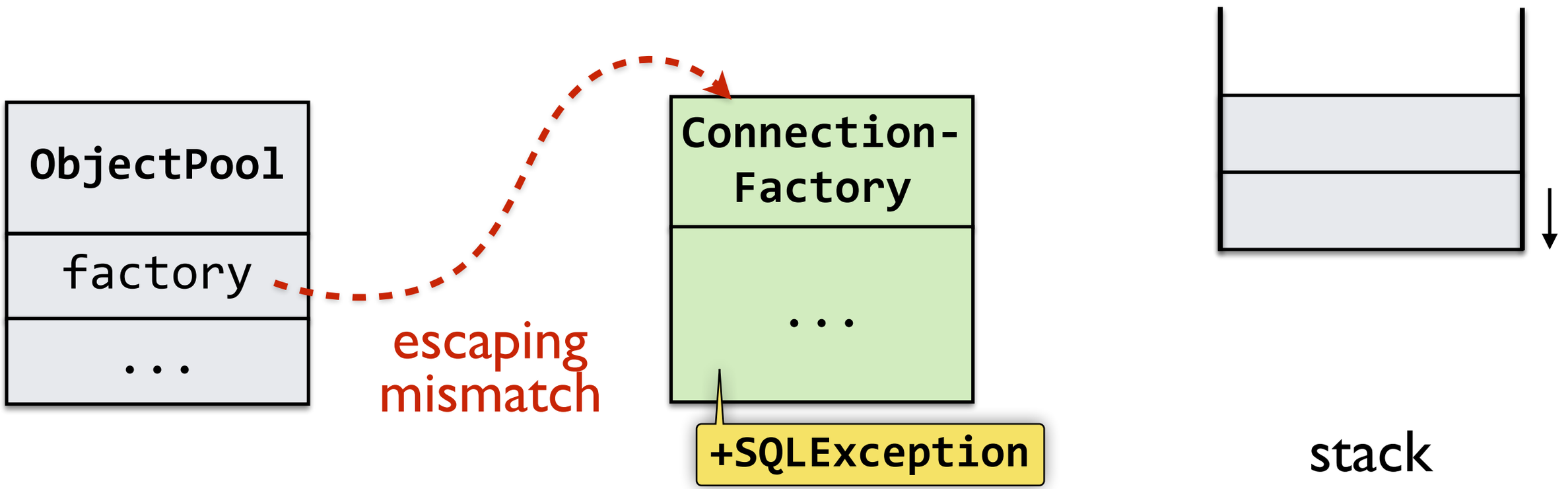


Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

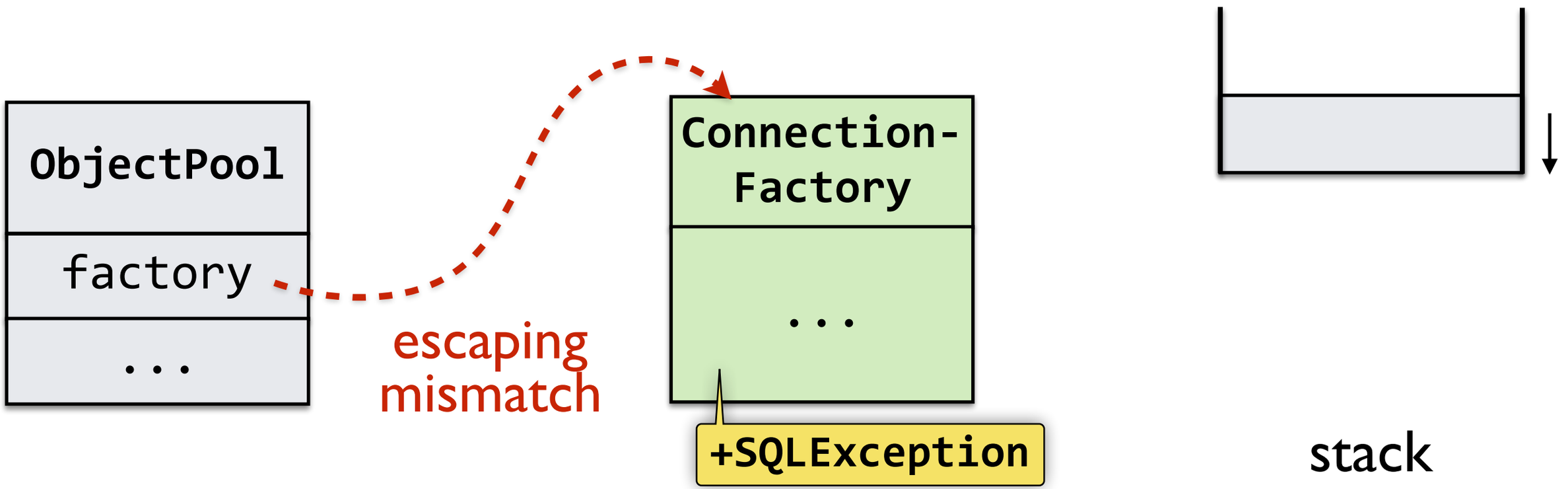


Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

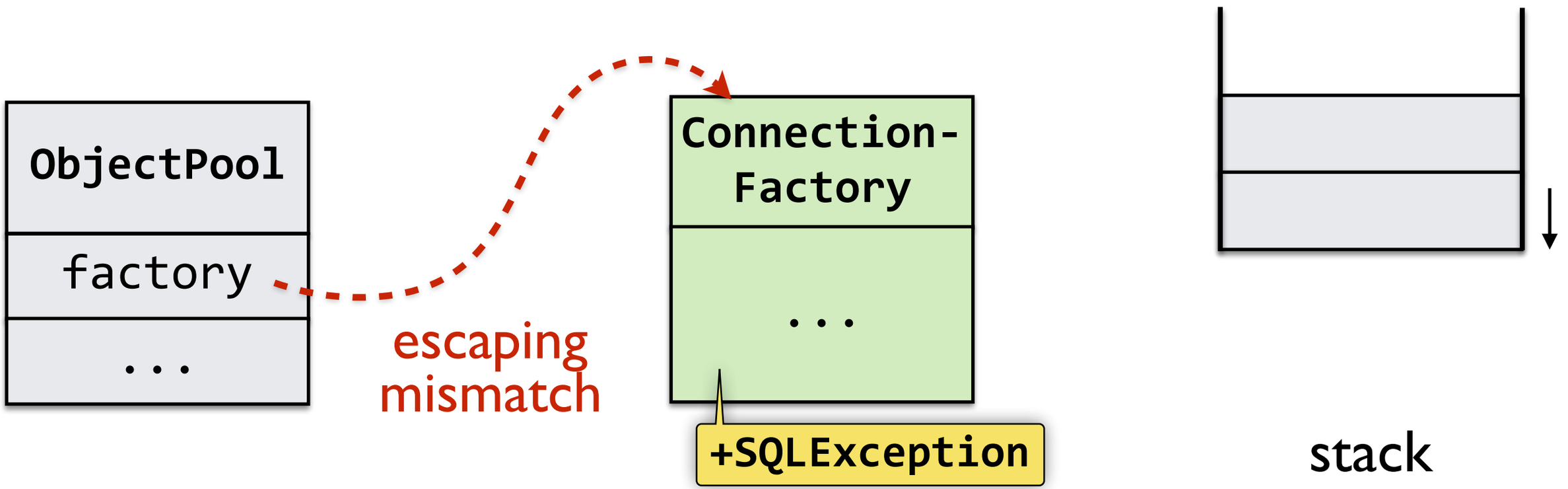


Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

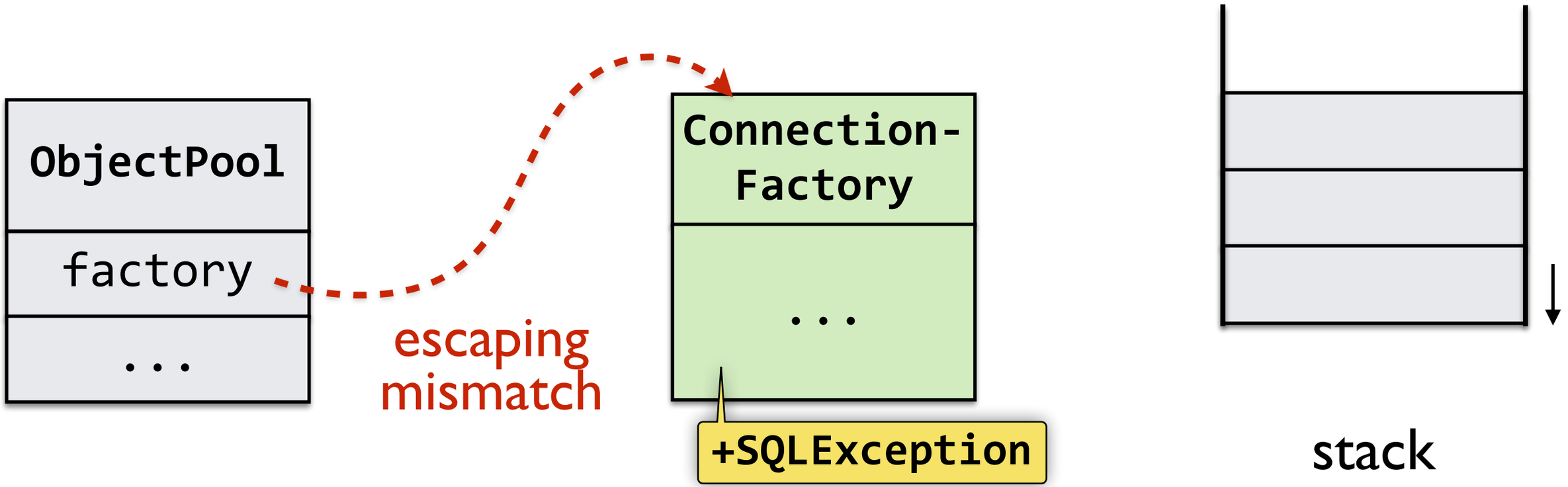


Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

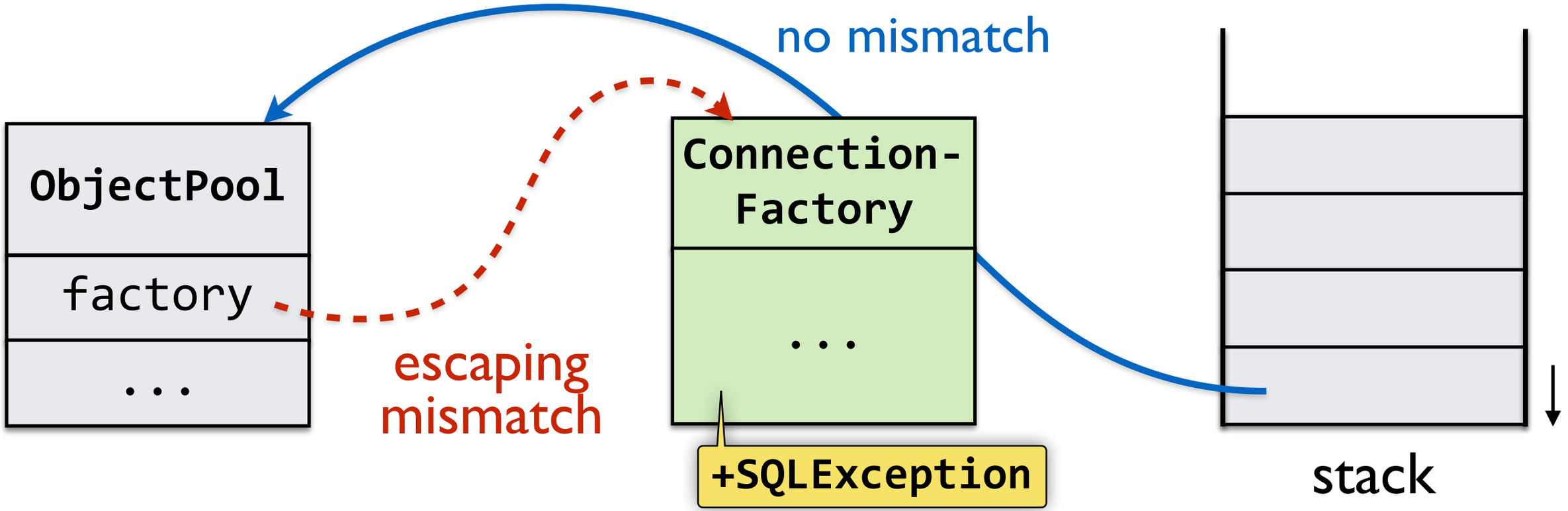


Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```



Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(Factory[T]⟨ℓ⟩ f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

assignment to `iter` induces a constraint

Argument `f` can have arbitrary mismatch.

$$\forall \ell.\ \texttt{Factory[T]}\langle \ell \rangle \leq \texttt{Factory[T]}\langle \bot \rangle$$

Field `factory` does not allow mismatch.

Preventing mismatches from escaping

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

A warning is issued.

```
class ObjectPool[T] {
  ObjectPool(Factory[T]⟨ℓ⟩ f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

constraint unsatisfiable

assignment to `iter` induces a constraint

Argument `f` can have arbitrary mismatch.

$$\forall \ell.\ \texttt{Factory[T]}\langle \ell \rangle \leq \texttt{Factory[T]}\langle \bot \rangle$$

Field `factory` does not allow mismatch.

Use @exact to indicate no mismatch

Blame labels are scoped

⇒ Mismatches cannot escape their scopes

```
class ObjectPool[T] {
  ObjectPool(@exact Factory[T] f) { factory = f; }
  Factory[T] factory;
  …
  T borrow() { … factory.make() … }
}
```

assignment to `iter` induces a constraint

`@exact` indicates no mismatch.

$$\texttt{Factory[T]}\langle\bot\rangle \leq \texttt{Factory[T]}\langle\bot\rangle$$

Field `factory` does not allow mismatch.

Use generics to indicate polymorphic mismatch

Use generics to indicate polymorphic mismatch

```
constraint Factory[T] {
  static T make();
}
```

Use generics to indicate polymorphic mismatch

```
constraint Factory[T] {
  static T make();
}
```

```
class ObjectPool[T]
    where Factory[T] {
  T borrow()
    { … f.make() … }
  …
}
```

Use generics to indicate polymorphic mismatch

```
constraint Factory[T] {
  static T make();
}
```

```
model ConnectionFactory
for Factory[Connection] {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool[T]
    where Factory[T] {
  T borrow()
    { … f.make() … }
  …
}
```

Use generics to indicate polymorphic mismatch

```
constraint Factory[T] {
  static T make();
}
```

```
model ConnectionFactory
for Factory[Connection] {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool[T]
    where Factory[T] {
  T borrow()
    { … f.make() … }
  …
}
```

```
// Using code
try {
  ObjectPool[Connection] pool =
    new ObjectPool[Connection]();
  Connection c = pool.borrow();
  …
} catch (SQLException e) { … }
```

Use generics to indicate polymorphic mismatch

```
constraint Factory[T] {
  static T make();
}
```

```
model ConnectionFactory
for Factory[Connection] {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool[T]
    where Factory[T] {
  T borrow()
    { … f.make() … }
  …
}
```

The model is inferred, and is part of the type.

```
// Using code
try {
  ObjectPool[Connection] pool =
    new ObjectPool[Connection]();
  Connection c = pool.borrow();
  …
} catch (SQLException e) { … }
```

Use generics to indicate polymorphic mismatch

```
constraint Factory[T] {
  static T make();
}
```

```
model ConnectionFactory
for Factory[Connection] {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool[T]
    where Factory[T] {
  T borrow()
    { … f.make() … }
  …
}
```

The model is inferred, and is part of the type.

```
// Using code
try {
  ObjectPool[Connection] pool =
    new ObjectPool[Connection]();
  Connection c = pool.borrow();
  …
} catch (SQLException e) { … }
```

SQLException is tracked precisely.

Object Pool pattern in Java

```
interface Factory<T> {
  T make() throws Exception;
}
```

```
class ConnectionFactory
implements Factory<Connection> {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool<T> {
  Factory<T> f;
  T borrow() throws Exception
    { … f.make() … }
  …
}
```

```
// Using code
try {
  ObjectPool<Connection> pool =
    new ObjectPool<Connection>(cf);
  Connection c = pool.borrow();
  …
} catch (Exception e) { … }
```

Object Pool pattern in Java

A simplification of the `apache.commons.pool` package

```
interface Factory<T> {
  T make() throws Exception;
}
```

```
class ConnectionFactory
implements Factory<Connection> {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool<T> {
  Factory<T> f;
  T borrow() throws Exception
    { … f.make() … }
  …
}
```

```
// Using code
try {
  ObjectPool<Connection> pool =
    new ObjectPool<Connection>(cf);
  Connection c = pool.borrow();
  …
} catch (Exception e) { … }
```

Object Pool pattern in Java

A simplification of the `apache.commons.pool` package

```
interface Factory<T> {
  T make() throws Exception;
}
```

```
class ConnectionFactory
implements Factory<Connection> {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool<T> {
  Factory<T> f;
  T borrow() throws Exception
    { … f.make() … }
  …
}
```

```
// Using code
try {
  ObjectPool<Connection> pool =
    new ObjectPool<Connection>(cf);
  Connection c = pool.borrow();
  …
} catch (Exception e) { … }
```

Object Pool pattern in Java

A simplification of the `apache.commons.pool` package

```
interface Factory<T> {
  T make() throws Exception;
}
```

```
class ConnectionFactory
implements Factory<Connection> {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool<T> {
  Factory<T> f;
  T borrow() throws Exception
    { … f.make() … }
  …
}
```

```
// Using code
try {
  ObjectPool<Connection> pool =
    new ObjectPool<Connection>(cf);
  Connection c = pool.borrow();
  …
} catch (Exception e) { … }
```

Object Pool pattern in Java

A simplification of the `apache.commons.pool` package

```
interface Factory<T> {
  T make() throws Exception;
}
```

```
class ConnectionFactory
implements Factory<Connection> {
  Connection make()
    throws SQLException { … }
}
```

```
class ObjectPool<T> {
  Factory<T> f;
  T borrow() throws Exception
    { … f.make() … }
  …
}
```

Catching "`Exception`" raises the risk of exception capture.

```
// Using code
try {
  ObjectPool<Connection> pool =
    new ObjectPool<Connection>(cf);
  Connection c = pool.borrow();
  …
} catch (Exception e) { … }
```

Using code is littered with the overly general "`Exception`" type.

Formalizing the language

- An extension of the simply typed λ-calculus
  Models exception tunneling and mismatch analysis.

- A translation from surface syntax into a kernel calculus

- **Theorem** *Well-typed programs handle their exceptions.*

Implementing the language

- An extension of the base Genus compiler [Zhang et al.'15]
  ~6,000 LoC

- A translation from Genus to Java

Key challenge:

(a) preventing exception capture (by reifying blame labels)

(b) … without slowing down normal control flow

Experience: safer and cleaner code

Porting the Java Collections Framework

(All general-purpose implementations)

- Static checking found "unreachable" `catch` blocks
- Prevents **exception capture** bugs

Porting exception-tunneling code

- Examples come from various sources:

  `javac` compiler, Apache Commons, etc.

- Restores static checking
- Eliminates ~200 (out of ~1,000) LoC in a `javac` visitor

Performance evaluation

1. Does reifying blame labels slow down normal-case code?

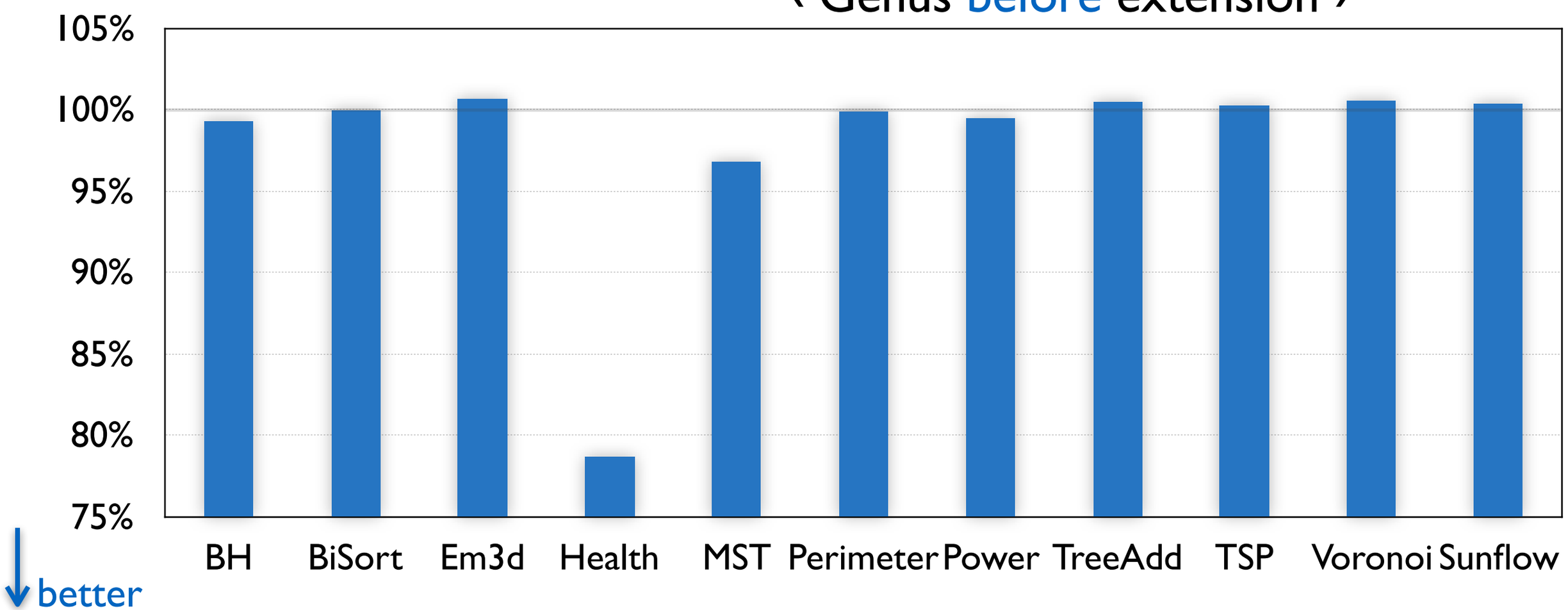
2. What speedup is achieved for exception-heavy code by not gathering a stack trace?

Performance with normal-case code

Benchmarks: JOlden & SunFlow (from DaCapo, ~21,000 LoC)

- 2.4% speedup across benchmarks
- Less than 0.5% slowdown in the worse case

Ratio of running times $\left(\frac{\text{Genus after extension}}{\text{Genus before extension}}\right)$

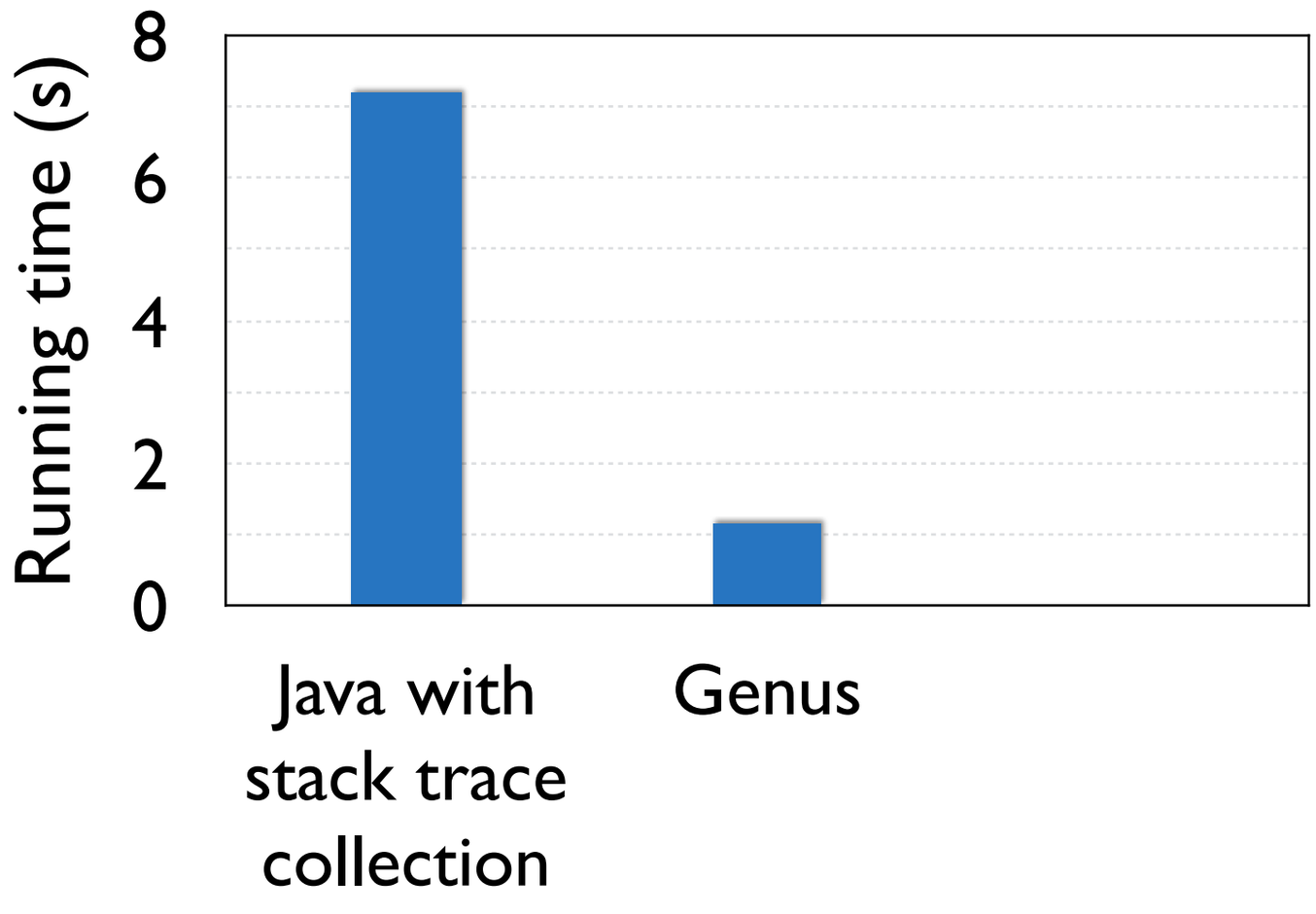


Performance with exception-heavy code

Benchmark: EasyIO (a text parsing package)

~6,200,000 exceptions raised per run

6× speedup by not collecting stack traces

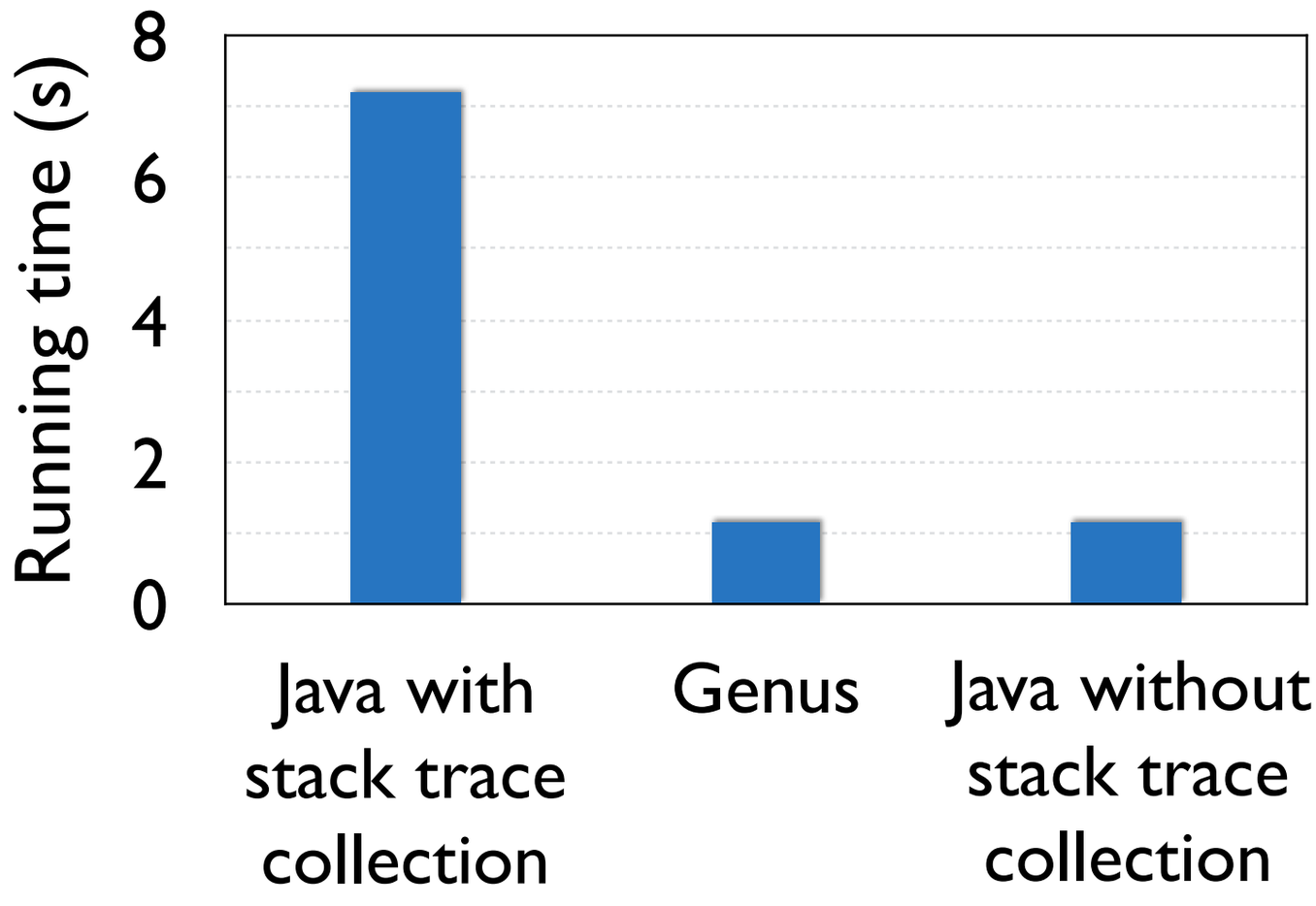


Performance with exception-heavy code

Benchmark: EasyIO (a text parsing package)

~6,200,000 exceptions raised per run

6× speedup by not collecting stack traces



Related work

Exception Analysis	SML [Fähndrich, Foster, Aiken, & Cu '98]
	OCaml [Leroy & Pessaux '00]
	Java [Robillard and Murphy '03]
	Java [Bravenboer & Smaragdakis '09]
	Others
Exception Polymorphism	Java [van Dooren & Steegmans '05]
	Scala [Rytz, Odersky, & Haller '12]
	Koka [Leijen '14]
	Swift 2 [Apple Inc. '15]

Blame Tracking	Higher-order contracts [Findler & Felleisen '02]
	Blame Calculus [Wadler & Findler '09]
Type Inference	Gradual type inference [Rastogi, Chaudhuri, & Hosmer '12]
	Region inference [Grossman et al. '02]

Related work

Exception Analysis	SML [Fähndrich, Foster, Aiken, & Cu '98]
	OCaml [Leroy & Pessaux '00]
	Java [Robillard and Murphy '03]
	Java [Bravenboer & Smaragdakis '09]
	Others
Exception Polymorphism	Java [van Dooren & Steegmans '05]
	Scala [Rytz, Odersky, & Haller '12]
	Koka [Leijen '14]
	Swift 2 [Apple Inc. '15]

- **Trade-off** between performance and precision
- **Global reasoning** does not help code understanding

Blame Tracking	Higher-order contracts [Findler & Felleisen '02]
	Blame Calculus [Wadler & Findler '09]
Type Inference	Gradual type inference [Rastogi, Chaudhuri, & Hosmer '12]
	Region inference [Grossman et al. '02]

Related work

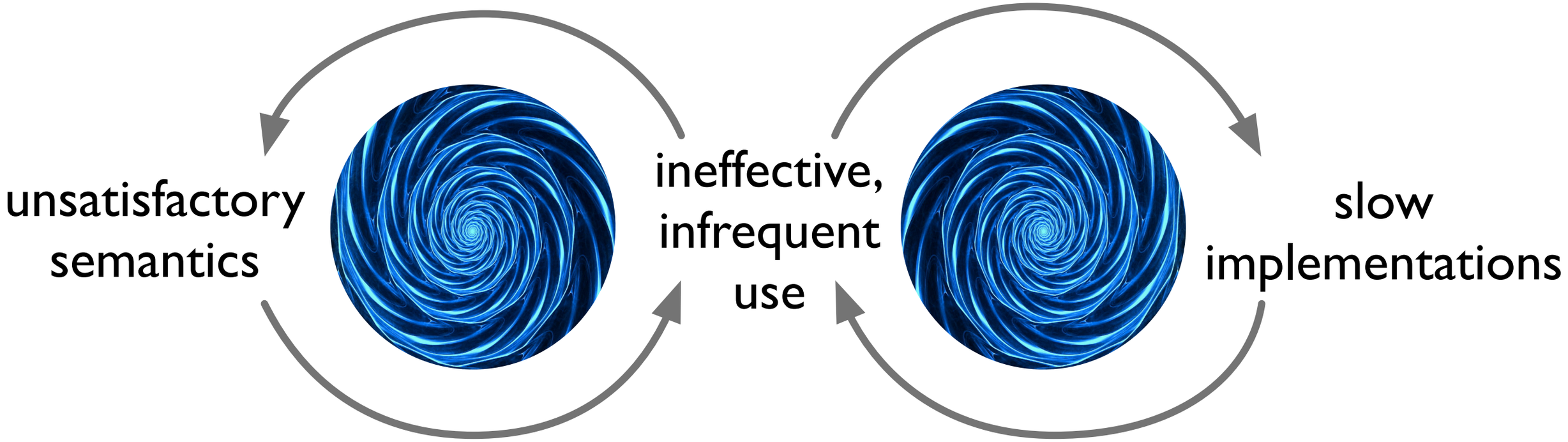
Exception Analysis	SML [Fähndrich, Foster, Aiken, & Cu '98]
	OCaml [Leroy & Pessaux '00]
	Java [Robillard and Murphy '03]
	Java [Bravenboer & Smaragdakis '09]
	Others
Exception Polymorphism	Java [van Dooren & Steegmans '05]
	Scala [Rytz, Odersky, & Haller '12]
	Koka [Leijen '14]
	Swift 2 [Apple Inc. '15]

- **Trade-off** between performance and precision
- **Global reasoning** does not help code understanding

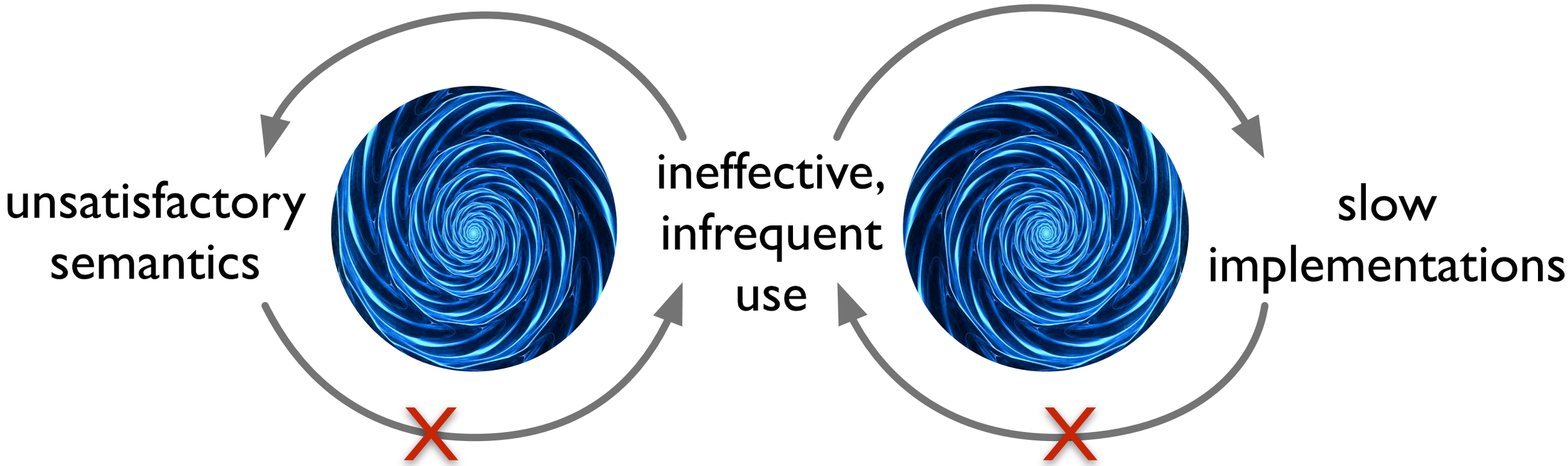
- Notional burden
- Exception capture

Blame Tracking	Higher-order contracts [Findler & Felleisen '02]
	Blame Calculus [Wadler & Findler '09]
Type Inference	Gradual type inference [Rastogi, Chaudhuri, & Hosmer '12]
	Region inference [Grossman et al. '02]

Breaking vicious cycles in language design



Breaking vicious cycles in language design



A safe, flexible, and efficient approach to exceptions

A safe, flexible, and efficient approach to exceptions

www.cs.cornell.edu/projects/genus

Yizhou Zhang*
Guido Salvaneschi†
Quinn Beightol*
Barbara Liskov‡
Andrew Myers*

**Cornell* †*TU Darmstadt* ‡*MIT*

Exception-based desugaring of foreach loops

```
for (E e : c) {
  print(e);
}
```

Java's desugaring

```
Iterator<E> it = c.iterator();
while (it.hasNext()) {
  E e = it.next();
  print(e);
}
```

Genus' desugaring

```
Iterator[E] it = c.iterator();
while (true) {
  E e;
  try { e = it.next(); }
  catch (NoSuchElementException stop) { break; }
  print(e);
}
```