UNIVERSITÀ DELLA CALABRIA

il Campus per eccellenza

Handling failures securely

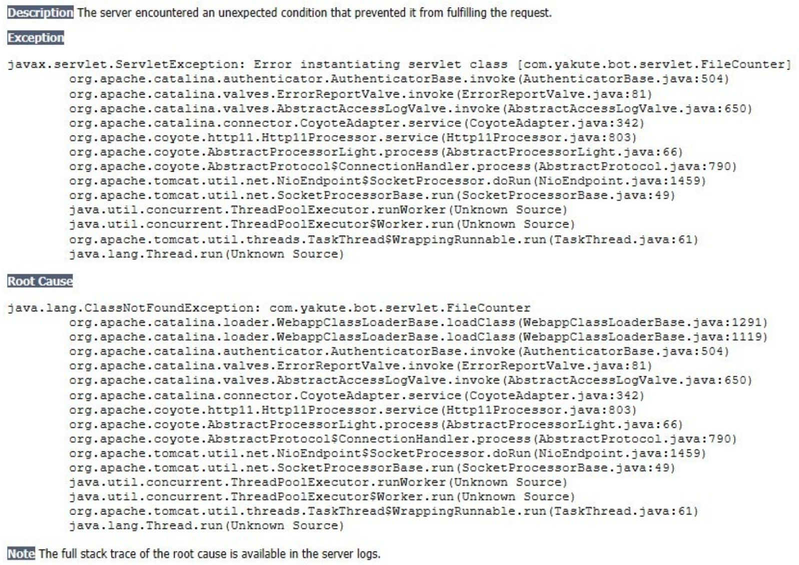
Mario Alviano

Consider failures, or fail

MAYBE YOU HAVEN'T SEEN THE WORLD, ELLIOT, BUT I HAVE, AND TRUST ME, IT'S GARBAGE. PEOPLE LIE, AND THEY HURT EACH OTHER. AND THEY WEAR THESE THINGS ON THEIR FEET CALLED CROCS.
RITA FARR
MAGICALQUOTE

- The real world is not perfect
- Nothing really goes as expected
- People may deviate from ordinary paths
- Please, consider **failure** when designing a system

Failure represented by exceptions



Description The server encountered an unexpected condition that prevented it from fulfilling the request.

Exception

```
javax.servlet.ServletException: Error instantiating servlet class [com.yakute.bot.servlet.FileCounter]
	org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:504)
	org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:81)
	org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:650)
	org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:342)
	org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:803)
	org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
	org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:790)
	org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1459)
	org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
	java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
	java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
	org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
	java.lang.Thread.run(Unknown Source)
```

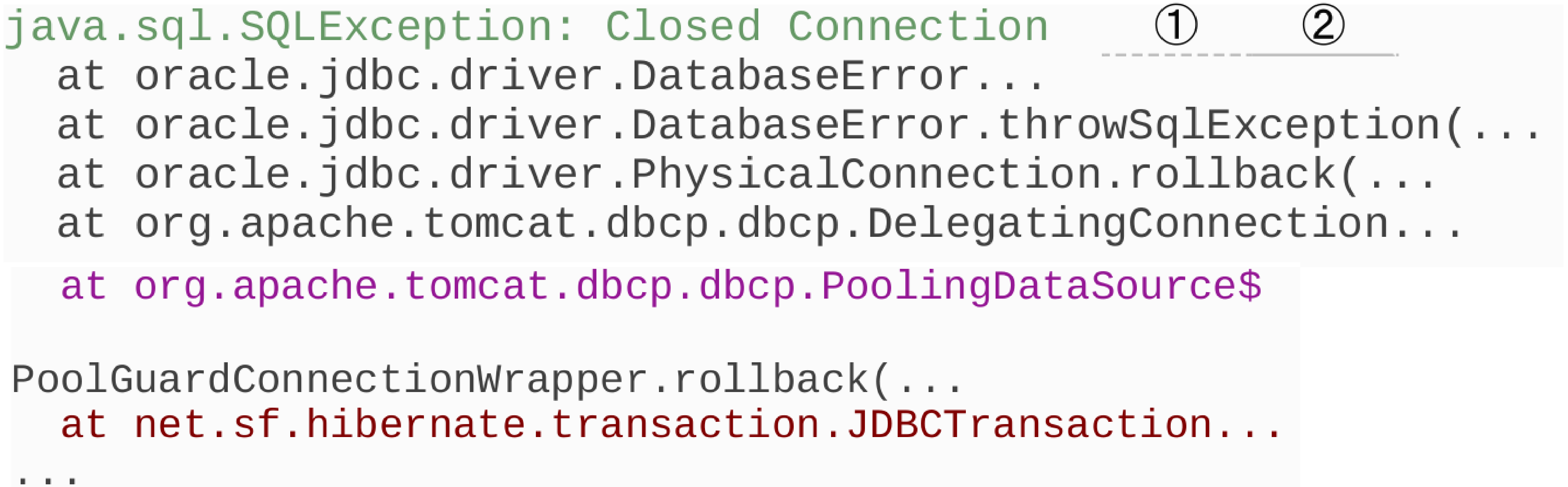
Root Cause

```
java.lang.ClassNotFoundException: com.yakute.bot.servlet.FileCounter
	org.apache.catalina.loader.WebappClassLoaderBase.loadClass(WebappClassLoaderBase.java:1291)
	org.apache.catalina.loader.WebappClassLoaderBase.loadClass(WebappClassLoaderBase.java:1119)
	org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:504)
	org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:81)
	org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:650)
	org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:342)
	org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:803)
	org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
	org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:790)
	org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1459)
	org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
	java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
	java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
	org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
	java.lang.Thread.run(Unknown Source)
```

Note The full stack trace of the root cause is available in the server logs.

- Often stack traces are shown to the end user
- Very bad!
- Why do this happen?

- Exceptions are often used to represent failures
- They disrupt the normal flow of a program
- Information on why and where the execution flow was disrupt
  - Why: the message
  - Where: the stack trace

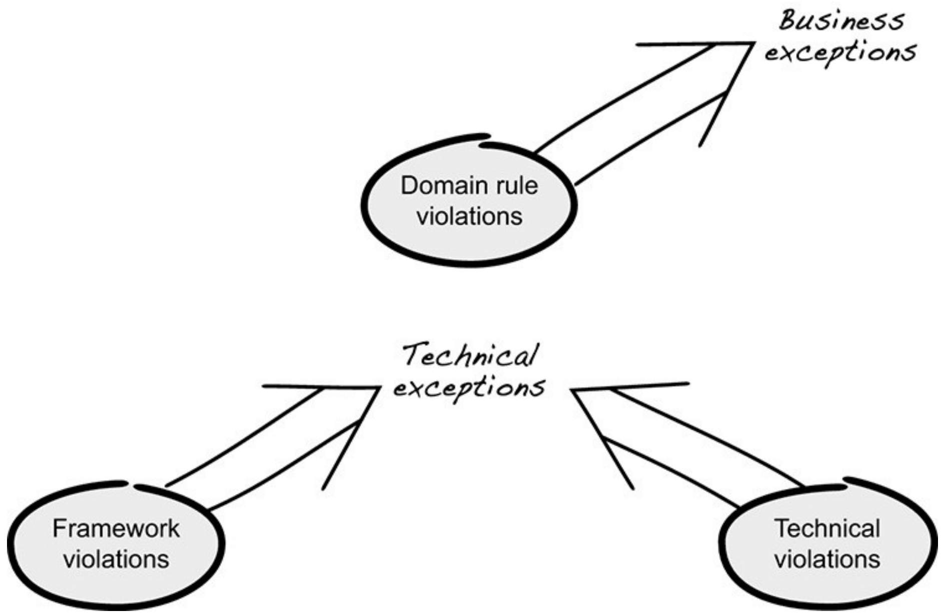


```
java.sql.SQLException: Closed Connection
  at oracle.jdbc.driver.DatabaseError...
  at oracle.jdbc.driver.DatabaseError.throwSqlException(...
  at oracle.jdbc.driver.PhysicalConnection.rollback(...
  at org.apache.tomcat.dbcp.dbcp.DelegatingConnection...
  at org.apache.tomcat.dbcp.dbcp.PoolingDataSource$

PoolGuardConnectionWrapper.rollback(...
  at net.sf.hibernate.transaction.JDBCTransaction...
...
```

```
java.sql.SQLException: Closed Connection    ①    ②
  at oracle.jdbc.driver.DatabaseError...
  at oracle.jdbc.driver.DatabaseError.throwSqlException(...
  at oracle.jdbc.driver.PhysicalConnection.rollback(...
  at org.apache.tomcat.dbcp.dbcp.DelegatingConnection...
  at org.apache.tomcat.dbcp.dbcp.PoolingDataSource$

PoolGuardConnectionWrapper.rollback(...
  at net.sf.hibernate.transaction.JDBCTransaction...
...
```

- Several leaks
  - Java is used, let’s check for Java vulnerabilities
  - SQL is used, data a stored in a RDBLP, let’s try SQLi
  - Tomcat is used
  - Hibernate is used

Reasons to raise exceptions



- Business exceptions prevent illegal actions from a domain perspective
  - withdrawing money from a bank account with insufficient funds
  - adding items to a paid order
- Technical exceptions aren’t concerned about domain rules
  - adding items to an order without enough memory allocated
- Better to separate business and technical exceptions
  - business exceptions are part of the domain

```
public Account fetchAccountFor(final Customer customer,
                               final AccountNumber
accountNumber) {
   notNull(customer);
   notNull(accountNumber);

   try {
      return accountDatabase
            .selectAccountsFor(customer)    ①
            .stream()
            .filter(account ->
account.number().equals(accountNumber))    ②
            .findFirst()    ③
            .orElseThrow(
              () -> new IllegalStateException(    ④
                      format("No account matching %s for
%s",
                             accountNumber.value(),
customer)));
   } catch (SQLException e) {    ⑤
      throw new IllegalStateException(    ⑤
            format("Unable to retrieve account %s for %s",
                   accountNumber.value(), customer), e);
   }
}
```

Mixing business and technical exceptions is bad

An exception is raised if no matching account is found (business exception) or if a database error occurs (technical exception)

Note that findFirst() is also a not-very-good choice here!

```
private final AccountRepository repository;

public Balance accountBalance(final Customer customer,
                              final AccountNumber
accountNumber) {
   notNull(customer);
   notNull(accountNumber);

   try {
      return repository.fetchAccountFor(customer,
accountNumber)
                       .balance();
   } catch (IllegalStateException e) {
      if (e.getMessage().contains("No account matching"))
{    ①
         return Balance.unknown(accountNumber);     ②
      }
      throw e;    ③
   }
}
```

Following the previous code,
how do we distinguish the two
exceptional cases if both are
represented by IllegalStateException?

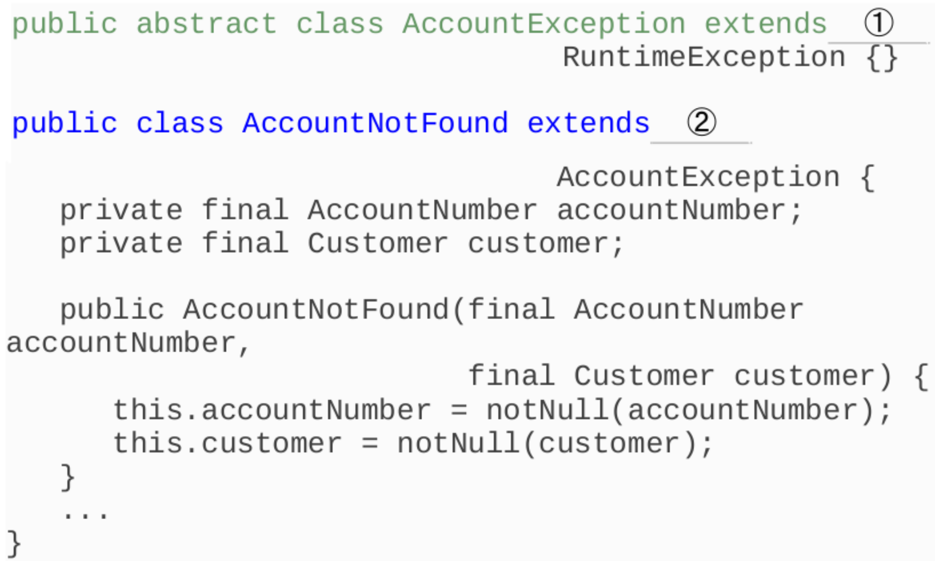
We can only rely on the message

A very fragile design

The message can change

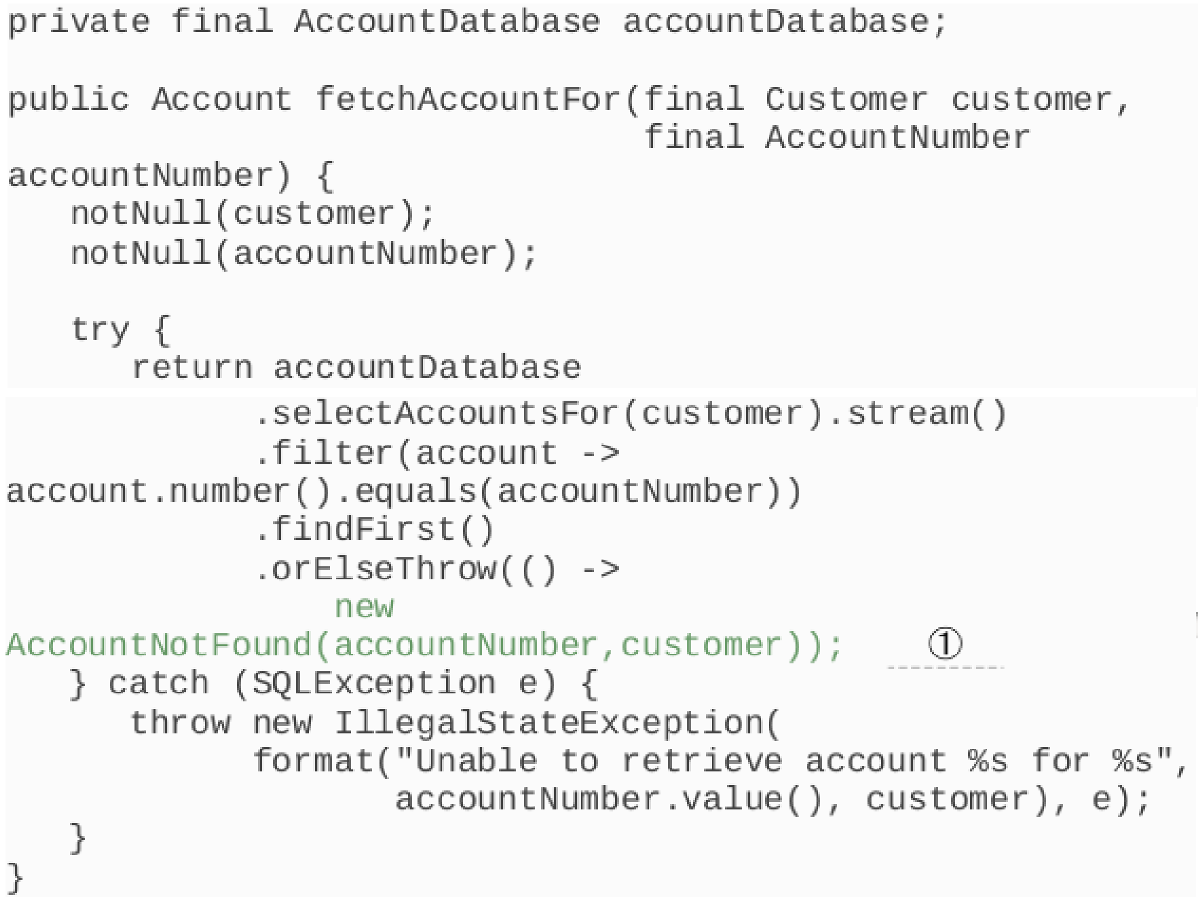
A new IllegalStateException may be added and escape the catch block

Separate business exceptions and technical exceptions



```
public abstract class AccountException extends   ①
                                               RuntimeException {}

public class AccountNotFound extends   ②

                                              AccountException {
   private final AccountNumber accountNumber;
   private final Customer customer;

   public AccountNotFound(final AccountNumber
accountNumber,
                                    final Customer customer) {
      this.accountNumber = notNull(accountNumber);
      this.customer = notNull(customer);
   }
   ...
}
```

- All business exceptions extends AccountException
- Capture specific exceptions
- Handle AccountException by raising a technical exception (to be handled by a global exception handler)



```
private final AccountDatabase accountDatabase;

public Account fetchAccountFor(final Customer customer,
                               final AccountNumber
accountNumber) {
   notNull(customer);
   notNull(accountNumber);

   try {
      return accountDatabase
            .selectAccountsFor(customer).stream()
            .filter(account ->
account.number().equals(accountNumber))
            .findFirst()
            .orElseThrow(() ->
                new
AccountNotFound(accountNumber,customer));    ①
   } catch (SQLException e) {
      throw new IllegalStateException(
            format("Unable to retrieve account %s for %s",
                   accountNumber.value(), customer), e);
   }
}
```

The type of the exception already clarifies the reason of the failure

No need for a message

Technical exceptions stay separate from business exceptions

```
private final AccountRepository repository;

public Balance accountBalance(final Customer customer,
                              final AccountNumber
accountNumber) {
   notNull(customer);
   notNull(accountNumber);
   try {
      return repository.fetchAccountFor(customer,
accountNumber)
                       .balance();
   }
   catch (AccountNotFound e) {    ①
      return Balance.unknown(accountNumber);
   }
   catch (AccountException e) {    ②
      throw new IllegalStateException(    ③
            format("Unhandled domain exception: %s",
                   e.getClass().getSimpleName()));
   }
}
```

Handle known business exceptions

Unknown business exceptions should not exist, but just in case raise a technical exception

Be aware that this way your application may still leak sensitive data

```
catch (SQLException e) {
   throw new IllegalStateException(
         format("Unable to retrieve account %s for %s",
             accountNumber.value(), customer), e);    ①
}
```

Is the customer name a sensible data?

Is it sensible in another context?

Because at that point you really don’t know
which contexts this information will traverse

You may end up logging private data that
should not be accessed by developers,
who usually have to access log files

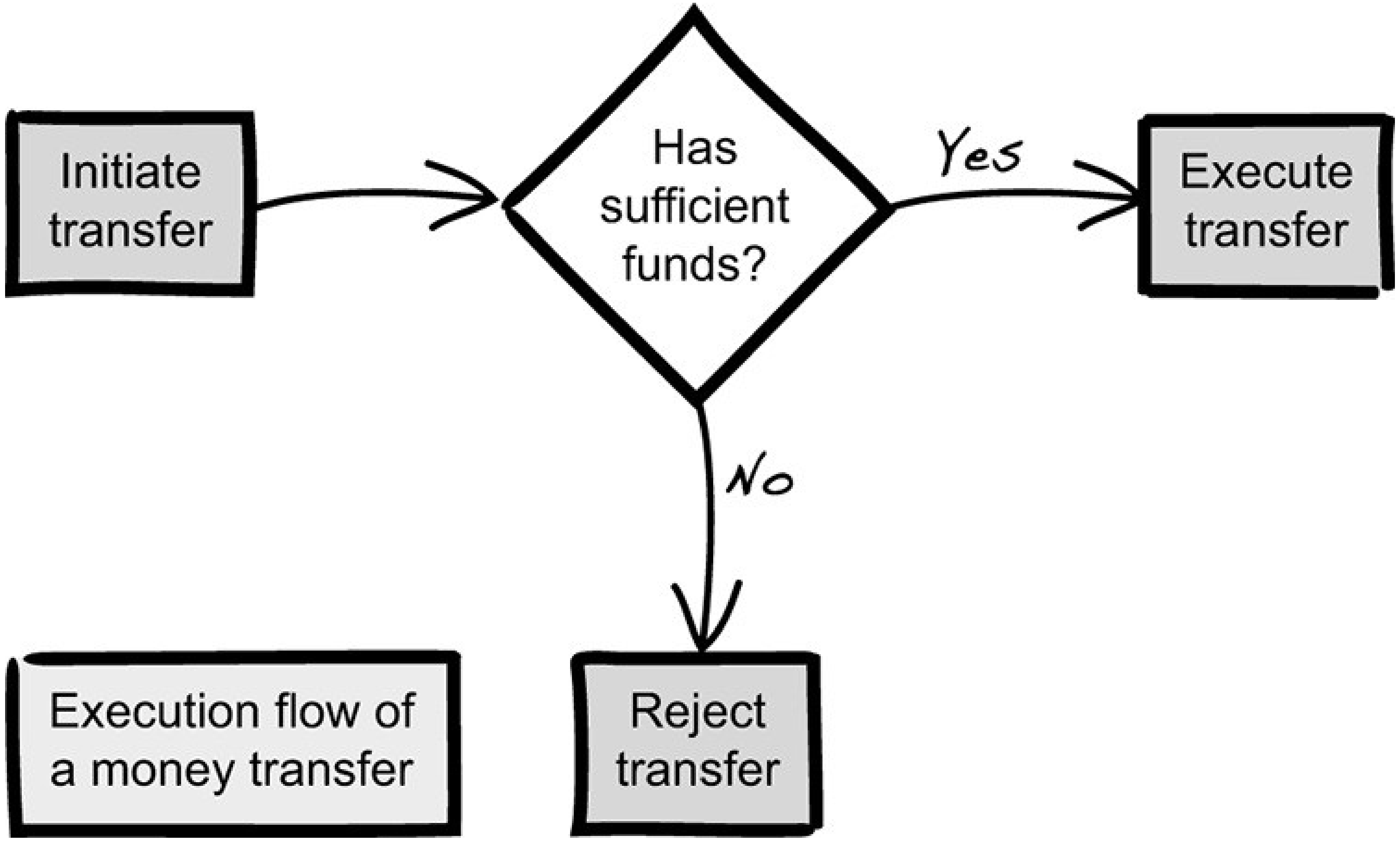
Never include sensitive data in exceptions!

Failure is not exceptional

- Failures are a natural and expected outcome of anything we do
- Does it make sense to model them as exceptions?
- A method usually has multiple outcomes
  - It can succeed
  - It can fail
- If failures are designed as unexceptional outcomes, many problems are solved
  - no ambiguity between domain and technical exceptions
  - impossible to inadvertently leak sensitive information

Example

Money transfer between bank accounts



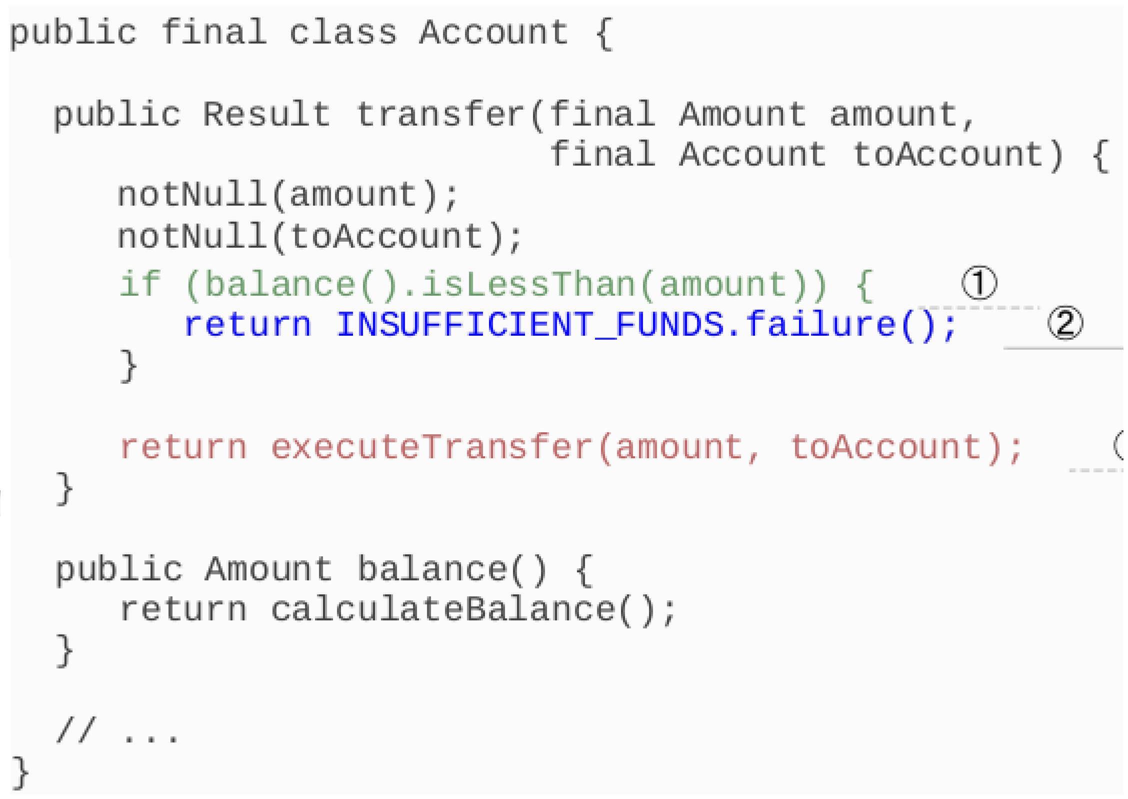
Execution flow of a money transfer

```
public final class Account {

  public void transfer(final Amount amount,
                       final Account toAccount)
        throws InsufficientFundsException {
    notNull(amount);
    notNull(toAccount);

    if (balance().isLessThan(amount)) {      ①
      throw new InsufficientFundsException();
    }

    executeTransfer(amount, toAccount);      ③
  }

  public Amount balance() {
      return calculateBalance();
  }

  // ...
}
```

```
public final class Amount implements Comparable<Amount> {
   private final long value;

   public Amount(final long value) {
      isTrue(value >= 0, "A price cannot be negative");
      this.value = value;
   }

   @Override
   public int compareTo(final Amount that) {
      notNull(that);
      return Long.compare(value, that.value);
   }

   public boolean isLessThan(final Amount that) {
      return compareTo(that) < 0;
   }

   // ...
}
```

Use exceptions to control flow of a program is odd

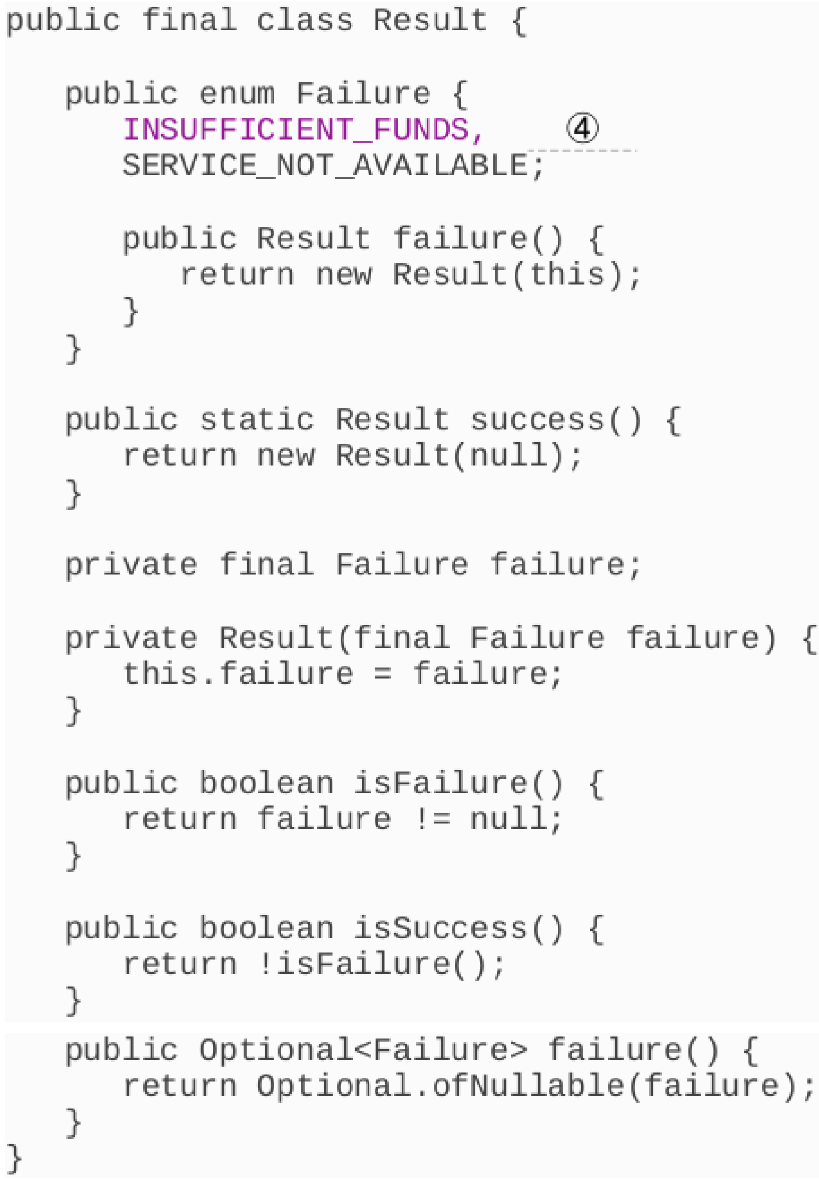
An insufficient balance is not exceptional



```
public final class Account {

  public Result transfer(final Amount amount,
                         final Account toAccount) {
     notNull(amount);
     notNull(toAccount);
     if (balance().isLessThan(amount)) {     ①
        return INSUFFICIENT_FUNDS.failure();    ②
     }

     return executeTransfer(amount, toAccount);    (
  }

  public Amount balance() {
     return calculateBalance();
  }

  // ...
}
```

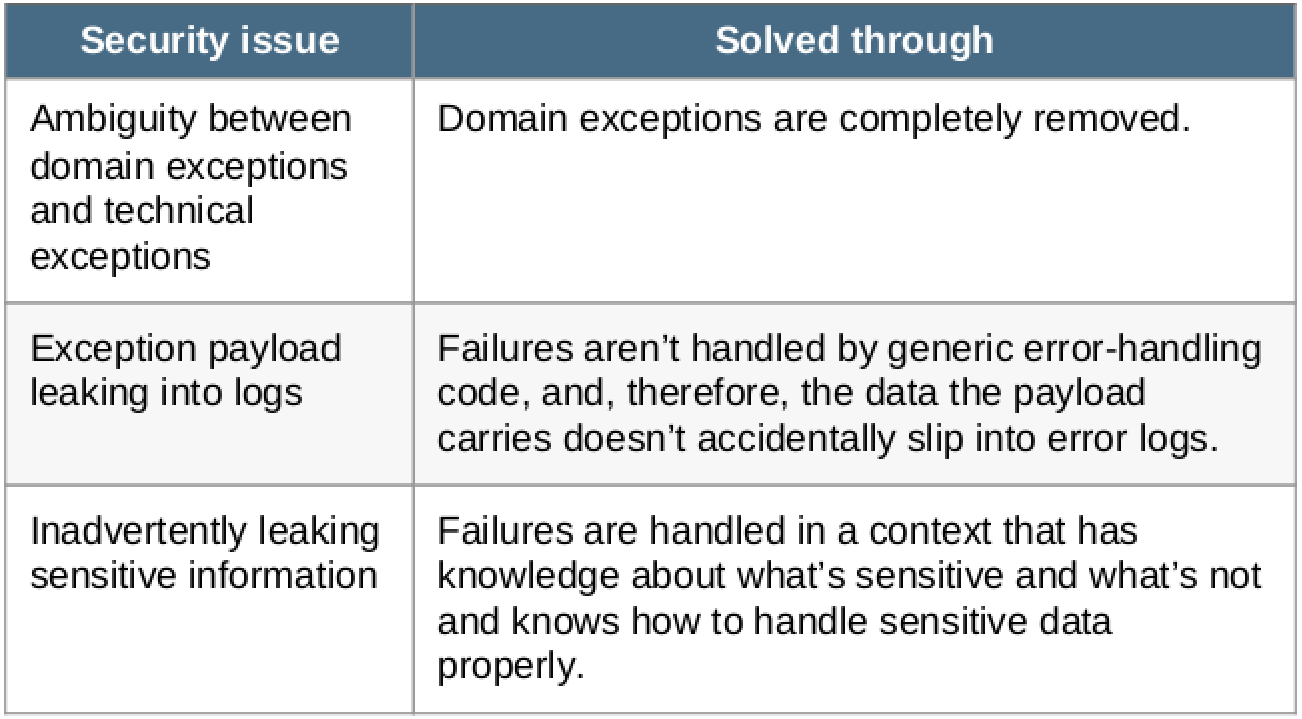
Define Result objects for your methods

Their design is part of the business model



```
public final class Result {

   public enum Failure {
      INSUFFICIENT_FUNDS,     ④
      SERVICE_NOT_AVAILABLE;

      public Result failure() {
         return new Result(this);
      }
   }

   public static Result success() {
      return new Result(null);
   }

   private final Failure failure;

   private Result(final Failure failure) {
      this.failure = failure;
   }

   public boolean isFailure() {
      return failure != null;
   }

   public boolean isSuccess() {
      return !isFailure();
   }
   public Optional<Failure> failure() {
      return Optional.ofNullable(failure);
   }
}
```

Some advatantages of designing failures as expected and unexceptional outcomes

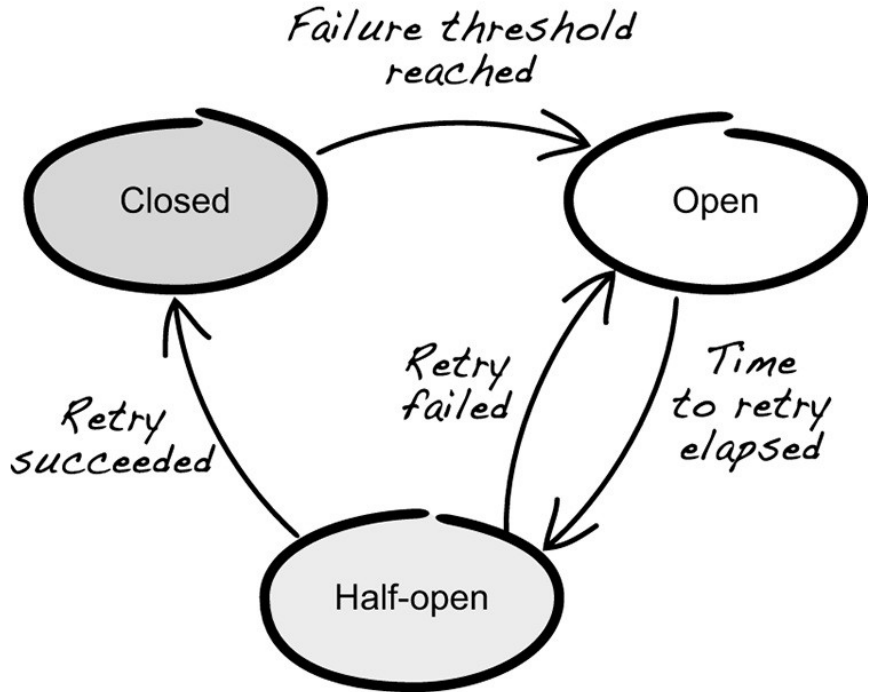


Security issue	Solved through
Ambiguity between domain exceptions and technical exceptions	Domain exceptions are completely removed.
Exception payload leaking into logs	Failures aren't handled by generic error-handling code, and, therefore, the data the payload carries doesn't accidentally slip into error logs.
Inadvertently leaking sensitive information	Failures are handled in a context that has knowledge about what's sensitive and what's not and knows how to handle sensitive data properly.

Designing for availability

- You don’t want your application or service to be unavailable
- Yet, you cannot pretend to serve all request
- There is always a physical limit
- Better to inform the user that the system is busy than to let they wait forever
- Implement queues

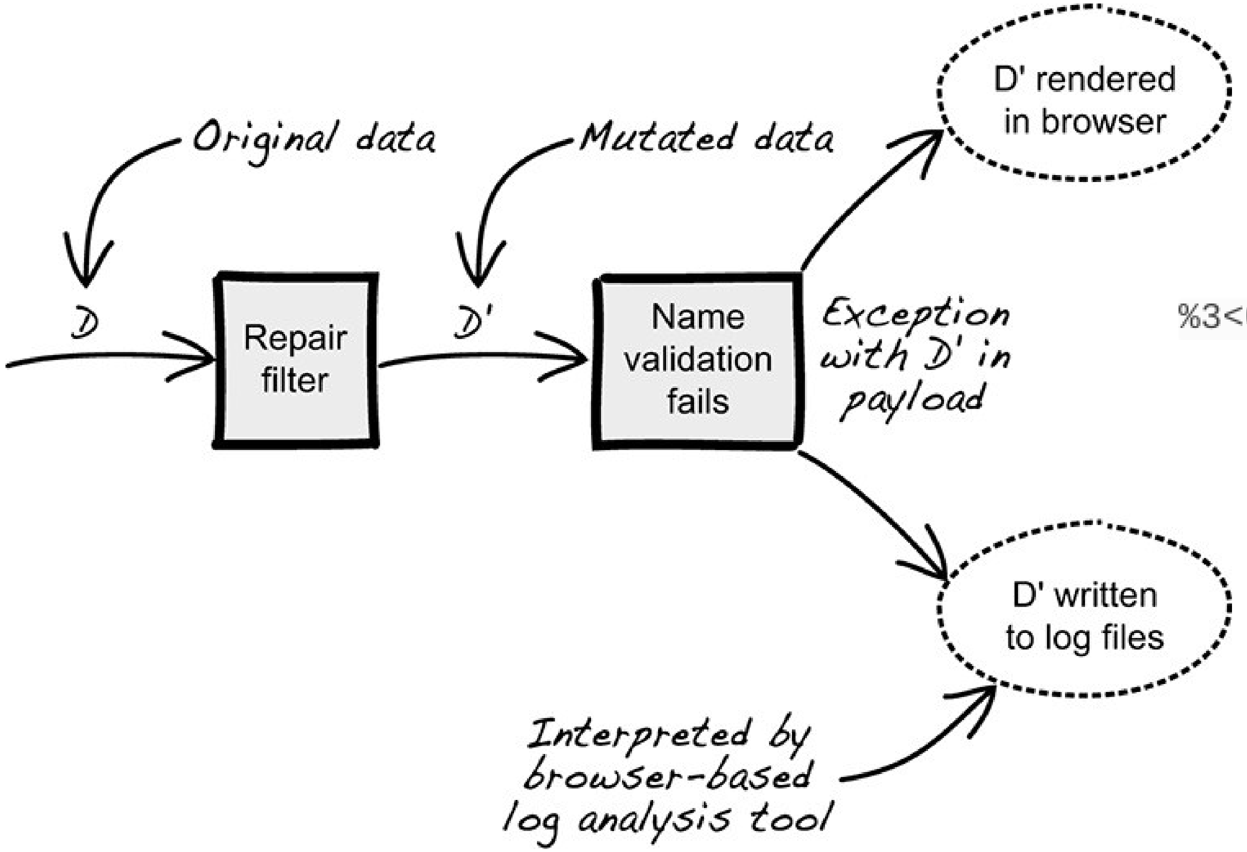
Circuit breakers



- Start with closed circuit
  - All requests are processed
  - Count failures
- Open the circuit when too many failures
  - Discard requests
- After some time, half-open the circuit
  - Process some requests
  - If they succeed, close the circuit
  - Otherwise, open the circuit

Handling bad data

- Data is often dirty
  - Spaces here and there
  - Missing characters
  - Special characters
- Don’t try to repair the input
  - Injection flows
  - Second-order attacks (the vulnerability arises on another system, like the log viewer)



```
%3<Cscript%3>Ealert("XSS")%3<C/script%3>E
```

Remove
<
and
>

```
%3Cscript%3Ealert("XSS")%3C/script%3E
```

Do not echo input verbatim, never, not even in log files!

Google Form

- https://forms.gle/VHn7SuPw5J8W6ryT8

Questions