Comparing Checked and Unchecked Exceptions

Esteban Herrera
JAVA ARCHITECT

@eh3rrera http://eherrera.net

Overview

The secret about checked exceptions

Checked exceptions

Unchecked exceptions

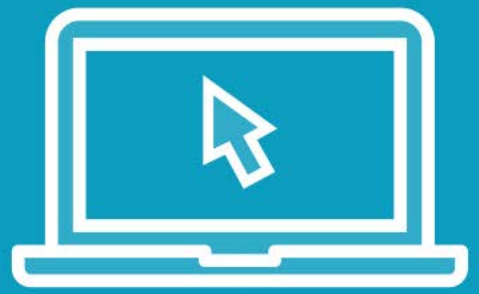
The rise of unchecked exceptions

Case study: The AWS SDK

The Secret About Checked Exceptions

For the Java Virtual Machine,
there's no concept
of checked exceptions.

Demo



Throwing a checked exception
as an unchecked exception

http://bit.ly/throwCheckedAsUnchecked

For the Java Virtual Machine,
there's no concept
of checked exceptions.

Checked Exceptions

Checked Exceptions' Advantages

Document a method.

```
public List<SalesData> processData(String path) {

    // ...

}
```

```
public List<SalesData> processData(String path)
    throws ParseException, IOException {

    // ...

}
```

Checked Exceptions' Advantages

Document a method.

Force to deal with the exception.

```java
public List<SalesData> processData(String path) {
    BufferedReader br = null;
    // ...
    br = new BufferedReader((new FileReader(path)));

    // ...
}
```

```
public List<SalesData> processData(String path) {
    BufferedReader br = null;
    // ...
    try {
        br = new BufferedReader((new FileReader(path)));
        // ...
    } catch(FileNotFoundException e) {
        // ...
    }
}
```

Checked Exceptions' Advantages

Document a method.

Force to deal with the exception.

Help write more robust programs.

```
public List<SalesData> processData(String path) {
    // ...
    try {
        // ...
    } catch(ParseException e) {
        // ...
    } catch(IOException e) {
        // ...
    }
}
```

```
public List<SalesData> processData(String path)
    throws ParseException, IOException {

    // ...

}
```

Checked Exceptions' Advantages

Document a method.

Force to deal with the exception.

Help write more robust programs.

Arguments Against Checked Exceptions

Expose implementation details.

Document the use of unchecked exceptions.

Cannot recover from the error.

Forget to handle anything other than what the compiler forces to handle.

Unchecked Exceptions

Unchecked Exceptions' Advantages

Catch only what you want.

```
public List<SalesData> processData(String path) {
    // ...

    data.setAmount(Double.parseDouble(items[1]));

    // ...
}
```

```
public List<SalesData> processData(String path) {
    // ...
    try {
        // ...
        data.setAmount(Double.parseDouble(items[1]));
        // ...
    } catch(NumberFormatException e) {
        // ...
    }
}
```

Unchecked Exceptions' Advantages

Catch only what you want.

Less cluttered code.

```
BufferedReader br = null;
try {
      br = new BufferedReader((new FileReader(path)));
      while ((String line = br.readLine()) != null) {
            String[] items = line.split(SEPARATOR);
            SalesData data = new SalesData();
            SimpleDateFormat sdf = new SimpleDateFormat(TIME_PATTERN);
            data.setDate(sdf.parse(items[0]));
            data.setAmount(Double.parseDouble(items[1]));
            list.add(data);
      }
} catch(ParseException  | IOException e) { }
finally {
      try {
         if(br != null) br.close();
      } catch (IOException e1) { }
}
```

```
BufferedReader br = new BufferedReader((new FileReader(path)));
while ((String line = br.readLine()) != null) {
    String[] items = line.split(SEPARATOR);
    SalesData data = new SalesData();
    SimpleDateFormat sdf = new SimpleDateFormat(TIME_PATTERN);
    data.setDate(sdf.parse(items[0]));
    data.setAmount(Double.parseDouble(items[1]));
    list.add(data);
}
br.close();
```

Unchecked Exceptions' Advantages

Catch only what you want.

Less cluttered code.

Provide more flexibility.

```java
public interface IDataService {
    List<SalesData> processData(String path);
}



class SalesDataService implements IDataService {
    public List<SalesData> processData(String path) {
        // ...
    }
}
```

```java
public interface IDataService {
    List<SalesData> processData(String path);
}



class SalesDataService implements IDataService {
    public List<SalesData> processData(String path) throws RuntimeException {
        // ...
    }
}
```

```
public interface IDataService {
    List<SalesData> processData(String path);
}
```

Error

```
class SalesDataService implements IDataService {
    public List<SalesData> processData(String path) throws IOException {
        // ...
    }
}
```

```java
public interface IDataService {
    List<SalesData> processData(String path) throws IOException;
}
```

Not recommended

```java
class SalesDataService implements IDataService {
    public List<SalesData> processData(String path) throws IOException {
        // ...
    }
}
```

Unchecked Exceptions' Advantages

Catch only what you want.

Less cluttered code.

Provide more flexibility.

Arguments Against Unchecked Exceptions

Overlook errors.

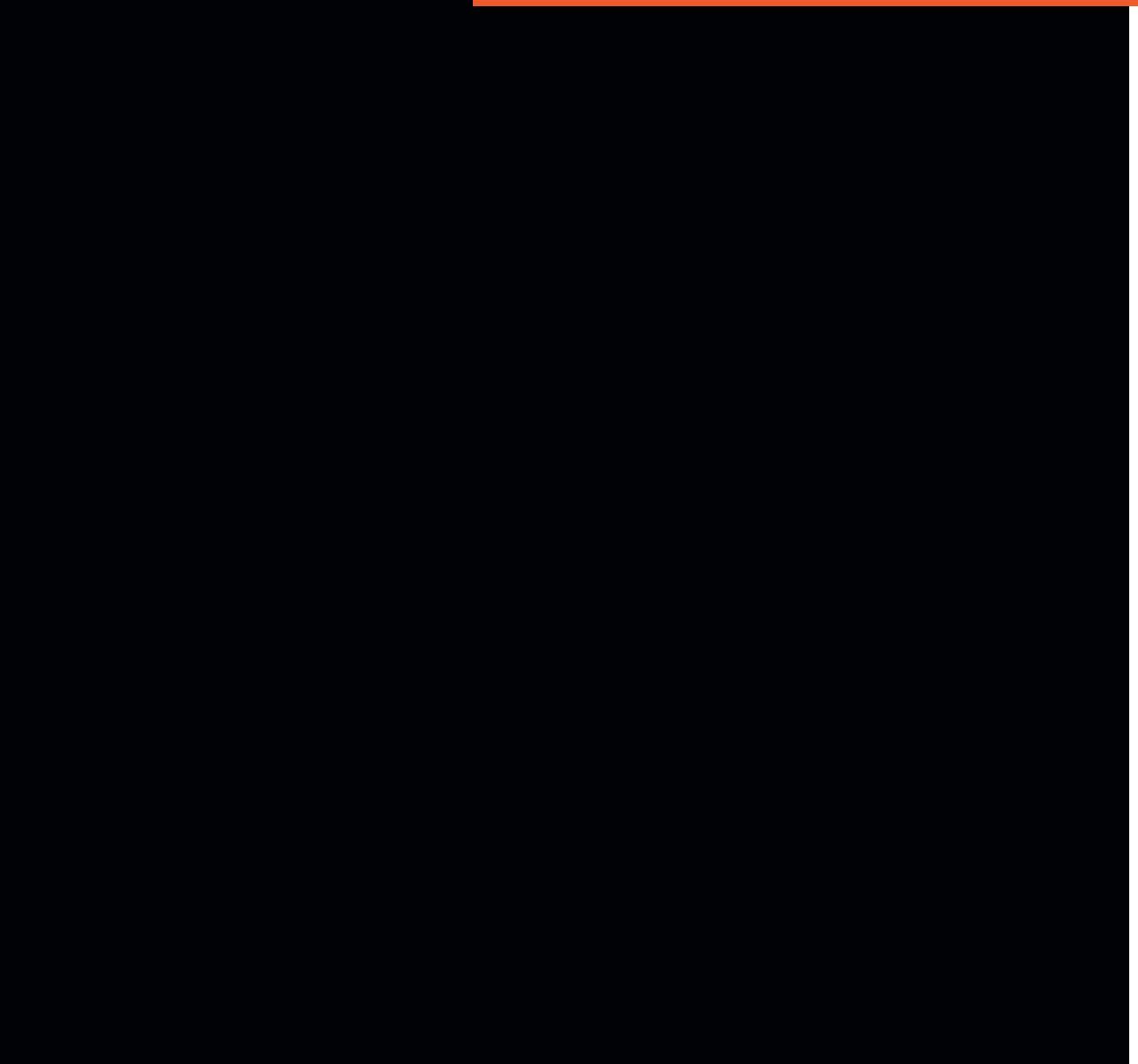
Unchecked exceptions are kind of invisible.

Less cluttered code is not always useful.

More flexibility isn't always good.

The Rise of Unchecked Exceptions

Not everything has to be black and white



Unchecked Exceptions

Java's Checked Exception Model

getSingleResult

```
Object getSingleResult()
```

Execute a SELECT query that returns a single untyped result.

Returns:

the result

Throws:

NoResultException - if there is no result

NonUniqueResultException - if more than one result

IllegalStateException - if called for a Java Persistence query language UPDATE or DELETE statement

QueryTimeoutException - if the query execution exceeds the query timeout value set and only the statement is rolled back

TransactionRequiredException - if a lock mode other than NONE has been set and there is no transaction or the persistence context has not been joined to the transaction

PessimisticLockException - if pessimistic locking fails and the transaction is rolled back

LockTimeoutException - if pessimistic locking fails and only the statement is rolled back

PersistenceException - if the query execution exceeds the query timeout value set and the transaction is rolled back

executeUpdate

```
int executeUpdate()
```

Execute an update or delete statement.

Returns:

the number of entities updated or deleted

Throws:

Is the Use of Exceptions Justified?

What if the row is not in the database?

Why it throws unchecked exceptions?

```
try {
    // ...
} catch(Exception e) {
   // Empty
}
```

```
try {
    // ...
} catch(Exception e) {
    throw new RuntimeException(e);
}
```

Class UncheckedIOException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.io.UncheckedIOException

All Implemented Interfaces:

Serializable

```
public class UncheckedIOException
extends RuntimeException
```

Wraps an IOException with an unchecked exception.

Since:

1.8

See Also:

Serialized Form

Constructor Summary

Constructors

Constructor and Description
UncheckedIOException(**IOException** cause) Constructs an instance of this class.
UncheckedIOException(**String** message, **IOException** cause) Constructs an instance of this class.

```
void processFile(String path) throws IOException {
    Files.lines(Paths.get(path))
        .forEach(new Consumer<String>() {
            public void accept(String line) {
                processData(line);
            }
        });
}
```

```
void processFile(String path) throws IOException {
    Files.lines(Paths.get(path))
        .forEach(line -> processData(line));
}
```

```java
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

```
void processFile(String path) throws IOException {
    Files.lines(Paths.get(path))
        .forEach(line -> processData(line));
}
```

```
void processFile(String path) throws IOException {
    Files.lines(Paths.get(path))
        .forEach(line -> processData(line));
}

void processData(String l) {
    try {
        throw new IOException();
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}
```

Most people believe that checked exceptions are a failed experiment.

Case Study: The AWS SDK

Frameworks must pay special attention to design

Class SQLException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.sql.SQLException

All Implemented Interfaces:

Serializable, Iterable<Throwable>

Direct Known Subclasses:

BatchUpdateException, RowSetWarning, SerialException, SQLClientInfoException, SQLNonTransientException, SQLRecoverableException, SQLTransientException, SQLWarning, SyncFactoryException, SyncProviderException

---

```
public class SQLException
extends Exception
implements Iterable<Throwable>
```

An exception that provides information on a database access error or other errors.

Each SQLException provides several kinds of information:

- a string describing the error. This is used as the Java Exception message, available via the method getMesasge.
- a "SQLstate" string, which follows either the XOPEN SQLstate conventions or the SQL:2003 conventions. The values of the SQLState string are described in the appropriate spec. The DatabaseMetaData method getSQLStateType can be used to discover whether the driver returns the XOPEN type or the SQL:2003 type.
- an integer error code that is specific to each vendor. Normally this will be the actual error code returned by the underlying database.
- a chain to a next Exception. This can be used to provide additional error information.
- the causal relationship, if any for this SQLException.

See Also:

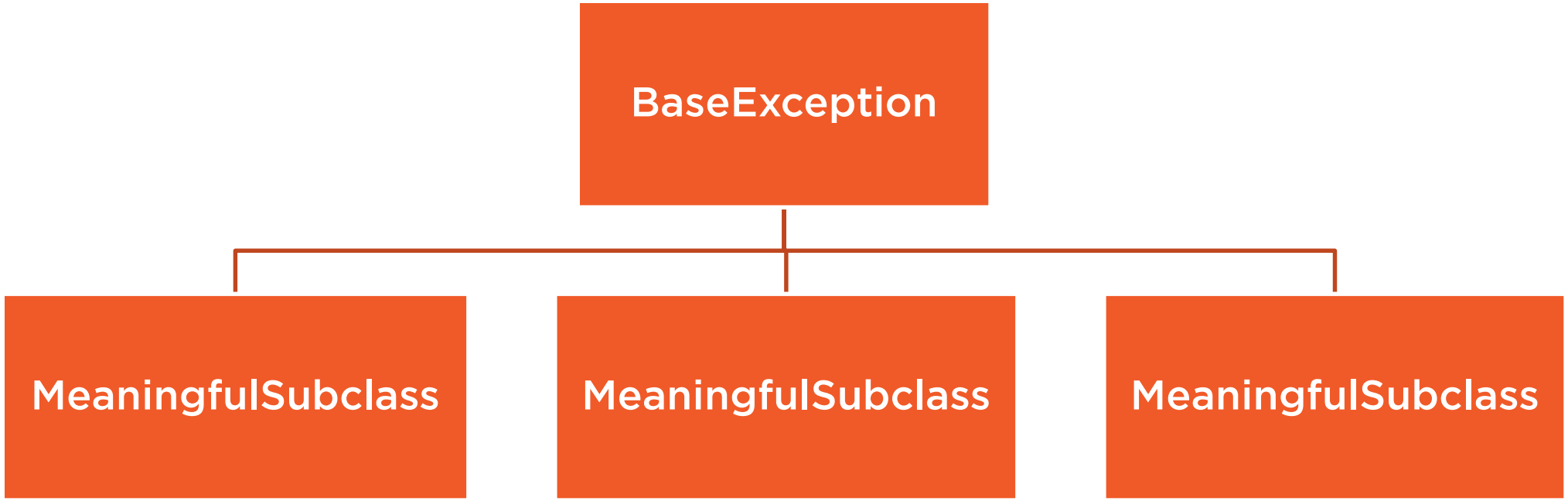
Serialized Form

Constructor Summary

Spring Framework

Hibernate

The Strategy



AWS SDK for Java

AmazonServiceException

AmazonClientException

Summary

The secret about checked exceptions

Checked exceptions

Unchecked exceptions

The rise of unchecked exceptions

Case study: The AWS SDK