Java Fundamentals: Exception Handling

INTRODUCING ERROR HANDLING IN JAVA

Esteban Herrera
JAVA ARCHITECT

@eh3rrera http://eherrera.net

Things WILL fail

Overview

Introduction

Understanding Exceptions

Understanding the Exception Hierarchy

Working with Exceptions

Comparing Checked and Unchecked Exceptions

Creating Custom Exceptions

Java Fundamentals: The Java Language

by Jim Wilson

This course provides complete coverage of the Java programming language and serves as a strong foundation for all Java-based development environments including client-based, server-side, and Android.

Start Course

http://bit.ly/jwjavalang

Java Fundamentals, Part 1

by John Sonmez

This course is designed to teach you Java whether you are an experienced programmer or just getting started.

Start Course

http://bit.ly/jsjava1

Java Fundamentals, Part 2

by John Sonmez

This course is designed to teach some of the fundamentals of Java and cover slightly more advanced topics than what were covered in Java Fundamentals Part1.

Start Course

http://bit.ly/jsjava2

Download IntelliJ IDEA

Windows macOS Linux

Ultimate

Web, mobile and enterprise development

DOWNLOAD .EXE

Free 30-day trial, 497 MB

Community

Java, Groovy, Scala and Android development

DOWNLOAD .EXE

Free, open-source (Apache 2.0), 342 MB

Version: 2017.1.1

Build: 171.4073.35

Released: April 10, 2017

System requirements

Installation Instructions

Previous versions

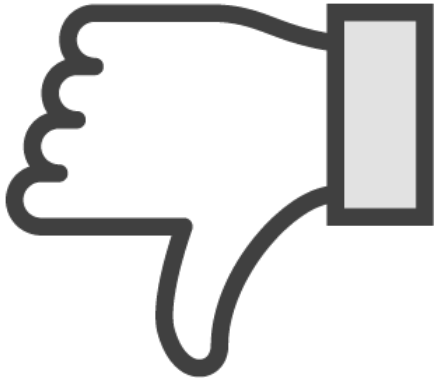
Get the ToolBox App to download IntelliJ IDEA and its future updates with ease

Not everything has to be black and white

During the course

Error Handling

Two Categories



May fail

Don’t expect to fail

Things WILL fail

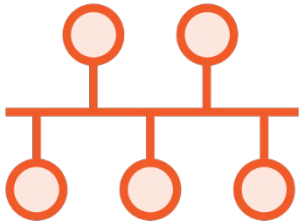
Source of Errors

Bugs

3rd-party Libraries

User Input

Hard Drive



Network

Database

WHAT

WHEN

HOW

Two Categories (revisited)

Expected Errors

Unexpected Errors

Error Handling

Any type of action to process any kind or error.

To process means

Catch

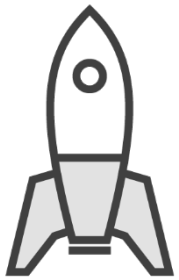
Recovery

Notify

Diagnostic

Spectrum

Everyday Software



Critical Software

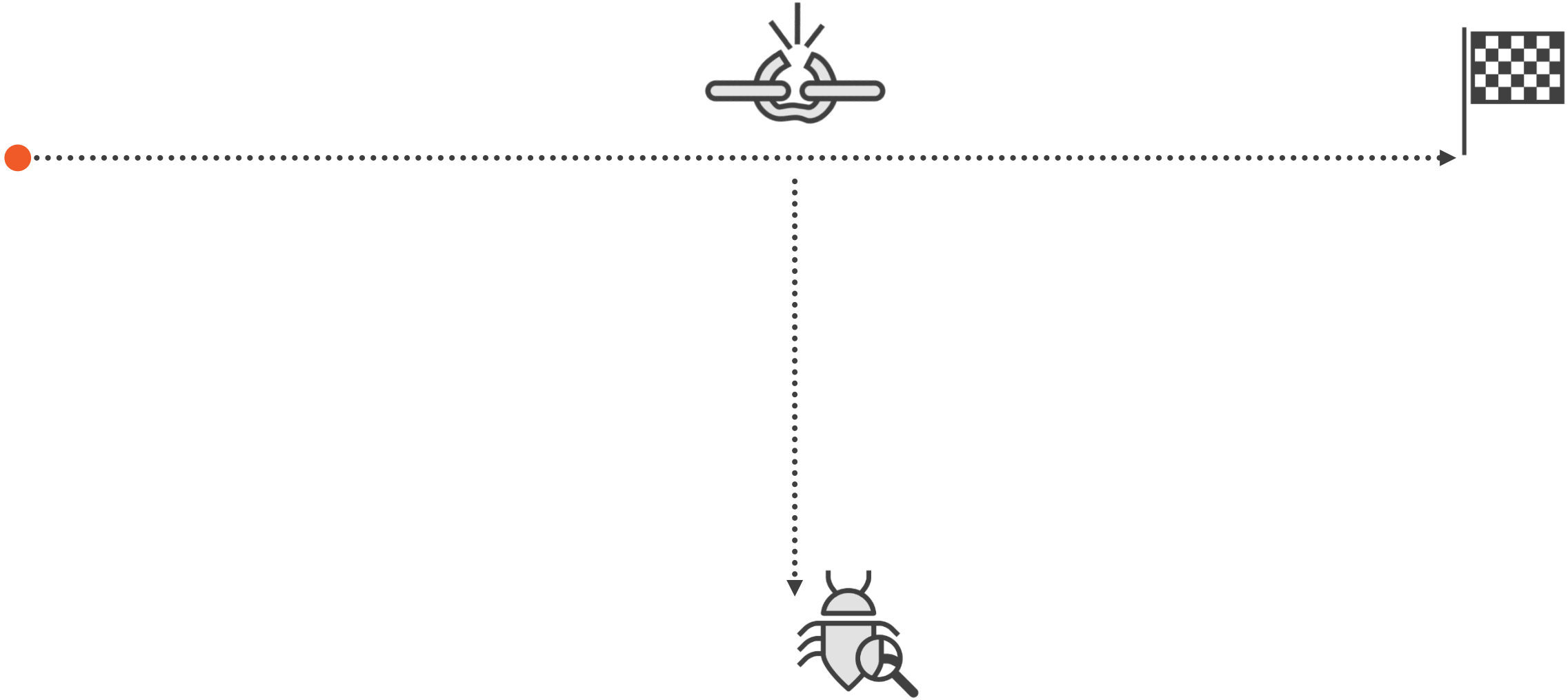
Exception

Atypical or exceptional condition that signals a piece of code could not execute normally.

NULL

```
String getFullName(User user) {
    String name = user.getName();
    String lastName = user.getLastName();

    return name + " " + lastName;
}
```

Program Execution Flow



```
String getFullName(User user) {
    String name = user.getName();
    String lastName = user.getLastName();

    return name + " " + lastName;
}
```

```
String getFullName(User user) {
    String name = user.getName();
    String lastName = user.getLastName();


    return name + " " + lastName;
}
```

Excecution flows up in the hierarchy

Check?

Error Codes

Error Codes

-1

EOF

NULL

Login

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    if (password.equals(plainPassword)) {
        code = 1;
    }
    return code;
}
```

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    if (password.equals(plainPassword)) {
        code = 1;
    }
    return code;
}
```

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    if (password.equals(plainPassword)) {
        code = 1;
    }
    return code;
}
```

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    if (password.equals(plainPassword)) {
        code = 1;
    }
    return code;
}
```

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    if (password.equals(plainPassword)) {
        code = 1;
    }
    return code;
}
```

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    if (user != null) {
        String plainPassword = decryptPassword(user.getPassword());
        if (password.equals(plainPassword)) {
            code = 1;
        }
    }
    return code;
}
```

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    if (user != null) {
        String plainPassword = decryptPassword(user.getPassword());
        if (password.equals(plainPassword)) {
            code = 1;
        } else {
            code = 2
        }
    }
    return code;
}
```

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    if (user != null) {
        String plainPassword = decryptPassword(user.getPassword());
        if (password.equals(plainPassword)) {
            code = 1;
            if (!containsNumbers(password)) {
                return 3;
            }
        } else {
            code = 2;
        }
    }
    return code;
}
```

Error Codes Disadvantages

- Check and track codes
- Easy to ignore
- Ambiguous meaning

Semipredicate problem

Occurs when a method uses an otherwise valid return value to indicate failure.

Division

$$3 / 0$$

Division

$$3 / 0 = 0?$$

Division

$$0 / 3 = 0$$

Checking the input?

Either:

- Adds more code
- Violates encapsulation

In complicated operations:

- Invalid input
- Costly performance

```
if (“abc".indexOf("a") != -1) {
    // ...
}
```

Exceptions to the Semipredicate problem

And yet...

Disadvantages

Remember specific values

Different implementations, different values

What if we want to communicate more info?

Error codes and exceptions can coexist

Favor exceptions over error codes

Historical Perspective

“ […] Languages like Modula-3 really pushed things like exception mechanisms. […]”

James Gosling

http://bit.ly/jginterv

Timeline

Early 1960s
LISP 1.5
ERRSET

Late 1960s
MacLisp
ERR

Early 1970s
MacLisp
CATCH and THROW

mid-late 1970s
NIL
UNWIND-PROTECT

mid-late 1970s
CLU
Specify exceptions, *failure*

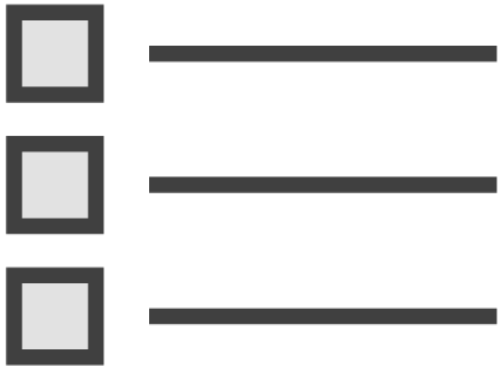
late 1980s
Modula-3
TRY-EXCEPT block

```
try {

    if (error)
        throws Exception

} catch (Exception) {

}
```

- Code that can raise an exception
- Throw
  (Create exception and transfer control)
- Catch
  (where execution is transferred and exception handled)

Two Categories of Exceptions

Checked



Unchecked

Advantages of Exceptions

Exception Advantages

Clean up code

```
int login(email, password) {
    int code = 0;
    User user = getUser(email);
    if (user != null) {
        String plainPassword = decryptPassword(user.getPassword());
        if (password.equals(plainPassword)) {
            code = 1;
            if (!containsNumbers(password)) {
                return 3;
            }
        } else {
            code = 2;
        }
    }
    return code;
}
```

```
User login(email, password) {
    User user = null;
    try {
        user = getUser(email);
        String plainPassword = decryptPassword(user.getPassword());
        user.verifyPassword(plainPassword);
    } catch (UserNotFoundException e) { /* Do something */ }
      catch (InvalidPasswordException e) { /* Do something */ }
      catch (PasswordDidNotMatchException e) { /* Do something */ }
    return user;
}
```

```
User login(email, password) throws
        UserNotFoundException,
         InvalidPasswordException,
         PasswordDidNotMatchException {
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    user.verifyPassword(plainPassword);
    return user;
}
```

```
User login(email, password) {
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    user.verifyPassword(plainPassword);
    return user;
}
```

Exception Advantages

Clean up code

Propagate Errors

```
try {
    User user = login(email, password);
} catch (UserNotFoundException e) { /* Do something */ }
catch (PasswordDidNotMatchException e) { /* Do something */ }
catch (InvalidPasswordException e) { /* Do something */ }
```

```
User login(email, password) {
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    user.verifyPassword(plainPassword);
    return user;
}
```

```
class ExceptionHandler { /* Deal with all uncaught exceptions */ }
```

```
User login(email, password) {
    User user = getUser(email);
    String plainPassword = decryptPassword(user.getPassword());
    user.verifyPassword(plainPassword);
    return user;
}
```

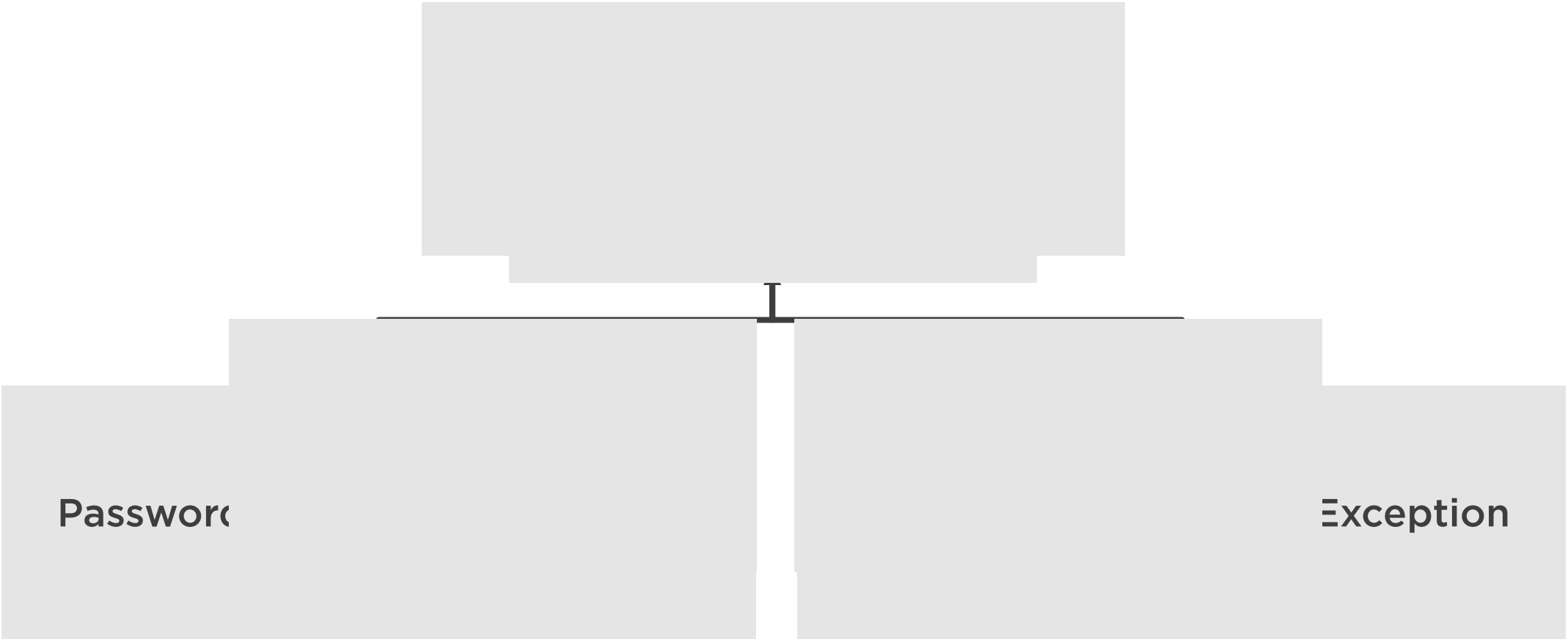
Exception Advantages

Clean up code

Propagate Errors

Group Error Types

Hierarchy of Exceptions

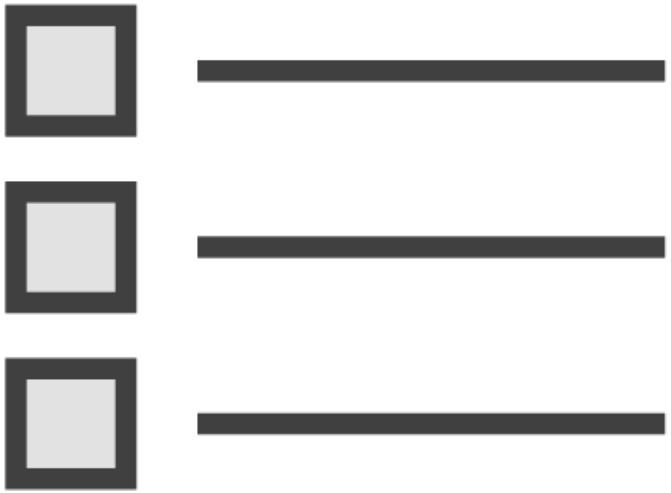


Java Exception Handling

The Problem with Exceptions in Java

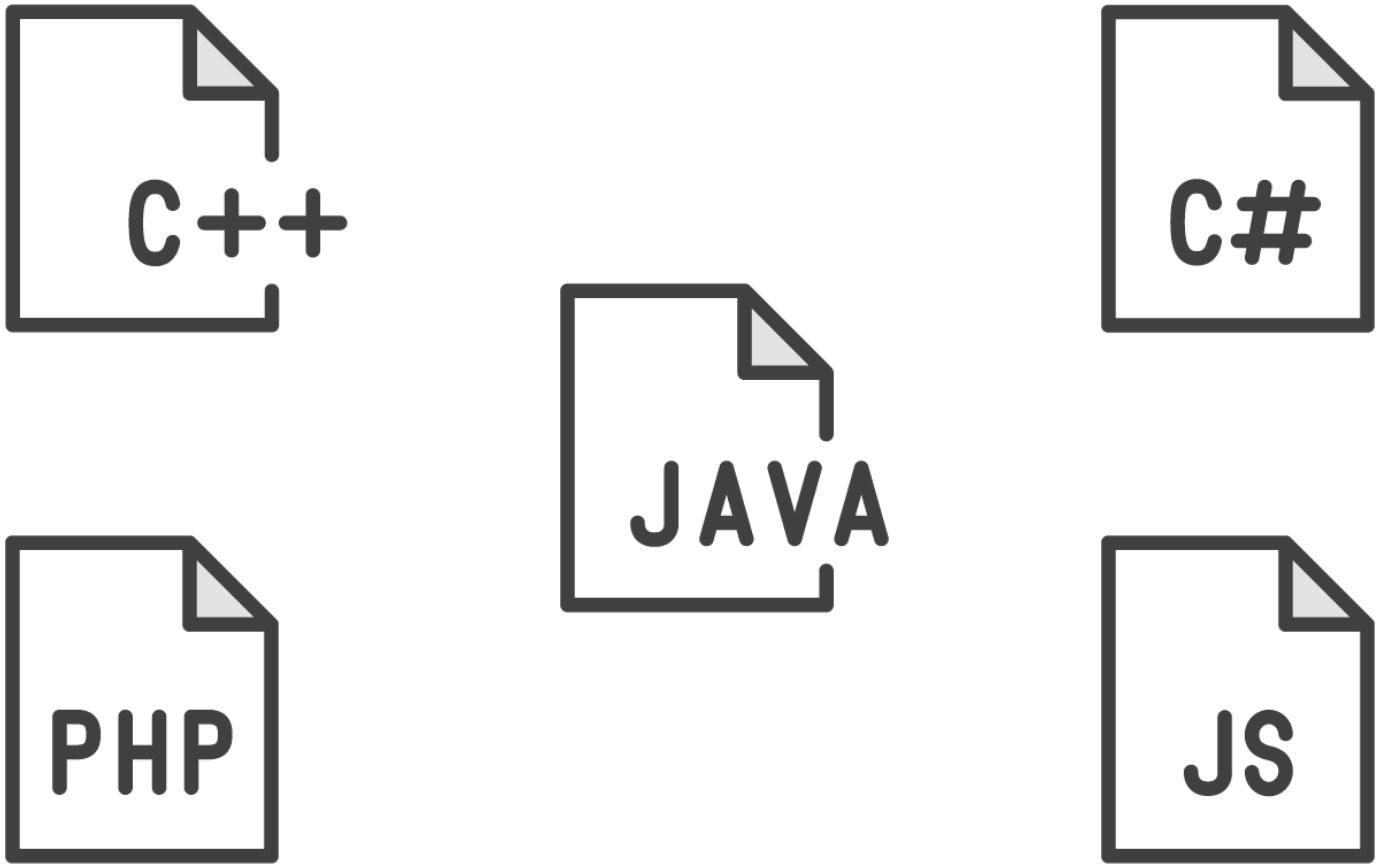
Two Categories of Exceptions

Checked

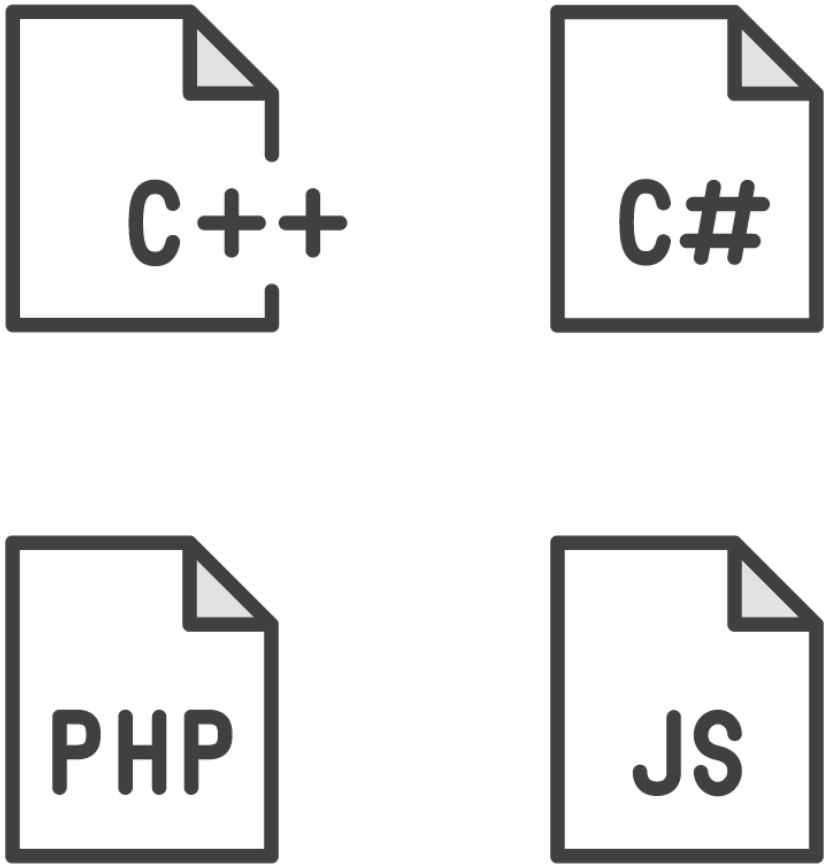


Unchecked

Exceptions



Checked Exceptions



"A programming language can't solve all the problems. A language can't guarantee that no matter how screwed up the environment gets the program will survive. But anything the language can do to increase the probability that programs will be reasonably graceful under fire is a good thing. For example, just making people at least willfully ignore return codes helps. In Java you can ignore exceptions, but you have to willfully do it. You can't accidentally say, *I don't care*. You have to explicitly say, *I don't care*."

James Goslin

http://bit.ly/jginterv2

Compile-time

Run-time

compact1, compact2, compact3

java.io

Class IOException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

ChangedCharSetException, CharacterCodingException, CharConversionException, ClosedChannelException, EOFException, FileLockInterruptionException, FileNotFoundException, FilerException, FileSystemException, HttpRetryException, IIOException, InterruptedByTimeoutException, InterruptedIOException, InvalidPropertiesFormatException, JMXProviderException, JMXServerErrorException, MalformedURLException, ObjectStreamException, ProtocolException, RemoteException, SaslException, SocketException, SSLException, SyncFailedException, UnknownHostException, UnknownServiceException, UnsupportedDataTypeException, UnsupportedEncodingException, UserPrincipalNotFoundException, UTFDataFormatException, ZipException

---

```
public class IOException
extends Exception
```

Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.

Since:

JDK1.0

See Also:

```
void method() {

}
```

```
void method() {
    Connection conn = getConnection();



}
```

```
void method() {
    Connection conn = getConnection();
    Statement statement = null;
    ResultSet rs = null;



}
```

```
void method() {
    Connection conn = getConnection();
    Statement statement = null;
    ResultSet rs = null;
    try {





    } catch (SQLException e) { }



}
```

```java
void method() {
    Connection conn = getConnection();
    Statement statement = null;
    ResultSet rs = null;
    try {
        statement = conn.createStatement();
        rs = statement.executeQuery("SELECT * FROM dummy");
        /* Do something */
        rs.close();
        statement.close();
    } catch (SQLException e) { /* Do something */ }



}
```

```java
void method() {
    Connection conn = getConnection();
    Statement statement = null;
    ResultSet rs = null;
    try {
        statement = conn.createStatement();
        rs = statement.executeQuery("SELECT * FROM dummy");
        /* Do something */
        rs.close();
        statement.close();
    } catch (SQLException e) { /* Do something */ }
    finally {
        conn.close();
    }
}
```

```java
void method() {
    Connection conn = getConnection();
    Statement statement = null;
    ResultSet rs = null;
    try {
        statement = conn.createStatement();
        rs = statement.executeQuery("SELECT * FROM dummy");
         /* Do something */
    } catch (SQLException e) { /* Do something */ }
    finally {
        try {
            rs.close();
            statement.close();
            conn.close();
        } catch (SQLException e) { }
    }
}
```

```java
void method() {
    Connection conn = getConnection();
    Statement statement = null;
    ResultSet rs = null;
    try {
        statement = conn.createStatement();
        rs = statement.executeQuery("SELECT * FROM dummy");
         /* Do something */
    } catch (SQLException e) { /* Do something */ }
    finally {
        if (rs != null) try { rs.close(); } catch (SQLException e) {}
        if (statement != null) try { statement.close(); } catch (SQLException e) {}
        if (conn != null) try { conn.close(); } catch (SQLException e) {}
    }
}
```

```
void method() {
    Connection conn = getConnection();
    Statement statement = null;
    ResultSet rs = null;
    try {
        statement = conn.createStatement();
        try {
            rs = statement.executeQuery("SELECT * FROM dummy");
            try {
                /* Do something */
            }  finally {
                rs.close();
            }
        } finally {
            statement.close();
        }
    } catch (SQLException e) { /* Do something */ }
    finally {
        try { conn.close(); } catch (SQLException e) {}
    }
}
```

Java 7 try-with-resources

```
try (Connection conn = getConnection();
     PreparedStatement st = conn.createStatement();
     ResultSet rs = st.executeQuery("SELECT * FROM dummy")
) {
   // Do something
   // All resources will be cleaned up at the end
} catch (SQLException e) {
    /* Do something */
}
```

Java's Checked Exception Model

Most people believe that checked exceptions are a failed experiment.

Not everything has to be black and white

Summary

- What is Error Handling
- Error codes vs Exceptions
- Advantages of Exceptions
- Problems with Exceptions in Java