JPCERT CC®

Javaセキュアコーディングセミナー東京

第3回
入出力(File, Stream)と例外時の動作

2012年11月11日(日)

JPCERTコーディネーションセンター
脆弱性解析チーム
戸田 洋三、久保 正樹

- **本資料について**
  - 本セミナーに使用するテキストの著作権はJPCERT/CCに帰属します。
  - 事前の承諾を受けた場合を除いて、本資料に含有される内容（一部か全部かを問わない）を複製・公開・送信・頒布・譲渡・貸与・使用許諾・転載・再利用できません。

- **本セミナーに関するお問い合わせ**
  - JPCERTコーディネーションセンター
  - セキュアコーディング担当
  - E-mail : secure-coding@jpcert.or.jp
  - TEL : 03-3518-4600

本セミナーについて

- 第1回 9月9日（日）
  - オブジェクトの生成と消滅におけるセキュリティ
- 第2回 10月14日（日）
  - 数値データの取扱いと入力値検査
- 第3回 11月11日（日）
  - 入出力(ファイル，ストリーム)と例外時の動作
- 第4回 12月16日
  - メソッドのセキュリティ

- 開発環境持参

今日の時間割

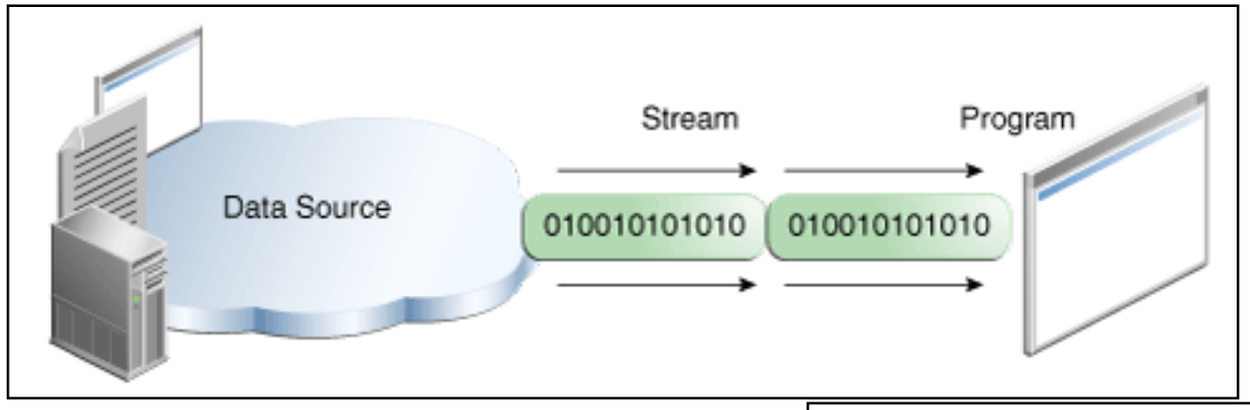
- 講義 (13:30--15:00)
  - 入出力
  - 例外時の動作
- ハンズオン (15:10--16:30)

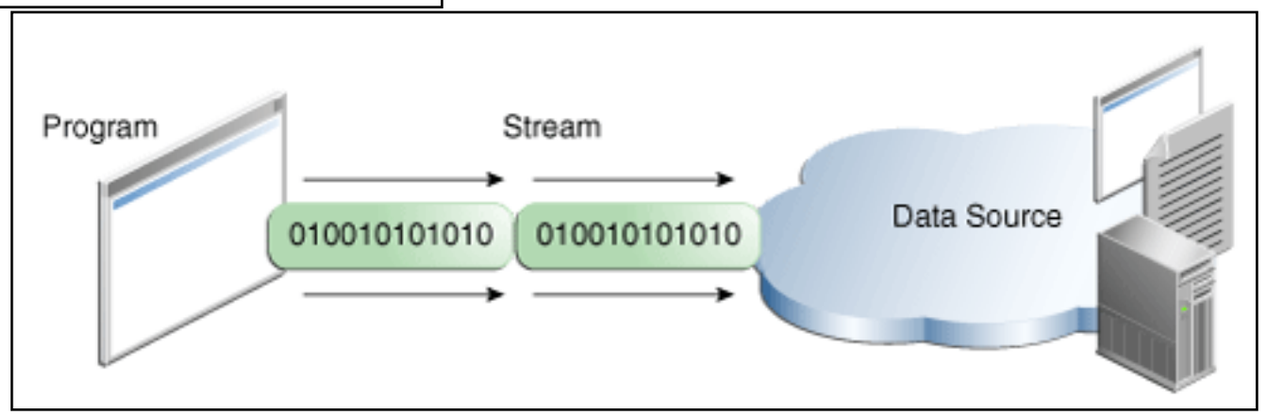
入出力

- **Java の入出力ストリーム**
- ファイルシステムの扱い
- 共有ディレクトリの危険性

I/O streams in Java

- Java 言語では, データの入出力を「**ストリーム**」という形で抽象化している
  - ディスクの読み書き
  - ネットワーク通信
  - 他のプログラムとのやりとりなど





from Basic I/O, The Java Tutorials

I/O streams in Java

- バイトデータのストリーム
  - **InputStream** や **OutputStream** など

- 文字データのストリーム
  - **Reader** や **Writer** など

- **Filter{In,Out}putStream** (他のストリームに機能を追加する)
  - **Buffered** ストリーム (バッファリング機能)
  - **Data** ストリーム (プリミティブ型や**String**型の値の入出力に使われる)
  - **Object** ストリーム (シリアライズの際, オブジェクトの入出力に使われる)

I/O streams in Java

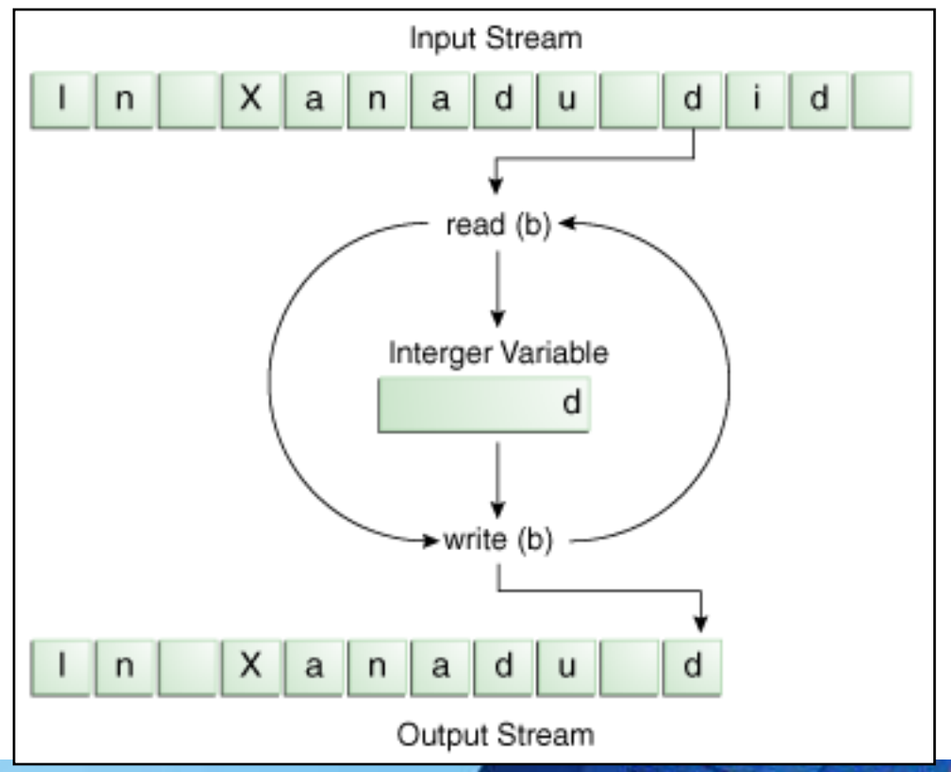
- バイトデータの入出力
  - など
- 文字
  - Fil
  - 型や
  - ズの

- InputStream
  - AudioInputStream
  - ByteArrayInputStream
  - FileInputStream
    - InflaterInputStream
      - GZIPInputStream
      - ZipInputStream
        - JarInputStream
  - FilterInputStream
    - BufferedInputStream
    - DataInputStream
    - PushbackInputStream
    - CheckedInputStream
    - CipherInputStream
    - DeflaterInputStream
    - DigestInputStream
    - ProgressMonitorInputStream
  - ObjectInputStream
  - PipedInputStream
  - SequenceInputStream
- FileCacheImageInputStream
- FileImageInputStream
- MemoryCacheImageInputStream

- Reader
  - BufferedReader
    - LineNumberReader
  - CharArrayReader
  - FilterReader
    - PushbackReader
  - InputStreamReader
    - FileReader
  - PipedInputStream
  - StringReader
- AudioFileReader
- ImageReader
- MidiFileReader
- SoundbankReader

サンプルコード: xanadu.txt を outagain.txt にバイト単位でコピー

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
  public static void main(String[] args) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
      in = new FileInputStream(“xanadu.txt”);
      out = new FileOutputStream(“outagain.txt”);
      int c;
      while ((c = in.read()) != -1){
        out.write(c);
      }
    } finally {
      if (in != null){
        in.close();
      }
      if (out != null){
        out.close();
      }
    }
  }
}
```



FIO08-J: 文字やバイトを読み取るメソッドの返り値は int で受ける

```
byte[] c = new byte[1];
while ((c[0] = (byte)in.read()) != -1){
    out.write(c);
}
```

read() の返り値を byte 型にキャストするとどうなる?

FIO08-J: 文字やバイトを読み取るメソッドの返り値は int で受ける

- **`int read() throws IOException`**

入力ストリームからデータの次のバイトを読み込む.
値のバイトは 0 から 255 の範囲の int 型データ.
ストリームの終わりに達した場合は -1 が返される.

- **`read()`** ... **`0xff`** を読み込んだら返り値は **`0x000000ff`**
- **`(byte)read()`** ... **`0xff`**（下位1バイト)に切り捨てられる
- **`... != -1`** ... 比較演算で **`int`** 型の **`0xffffffff`** に格上げ

`byte` 型は符号付き整数型なので, `0xff` は `-1`(decimal) として扱われ, `0xffffffff` に格上げされ, end-of-stream と区別できなくなる
⇒ DoS 攻撃に悪用されるかも!

FIO04-J: 不要になったリソースは解放する

- データ入出力に使われるストリームは限りあるリソース
- 不適切なリソース管理はリソース枯渇攻撃に悪用される危険がある
- 不要になったら適切にクローズしましょう

サンプルコード: データベースとのやりとり

```
public void getResults(String sqlQuery) {
  try {
    Connection conn = getConnection();
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sqlQuery);
    processResults(rs);
    stmt.close();
    conn.close();
  } catch (SQLException e) { /* ハンドラに処理を移す */ }
}
```

executeQuery()やprocessResults()で例外が発生するとデータベース接続がクローズされないじゃないか!

サンプルコード: データベースとのやりとり

```
public void getResults(String sqlQuery) {
  Statement stmt = null;
  ResultSet rs = null;
  Connection conn = getConnection();
  try {
    stmt = conn.createStatement();
    rs = stmt.executeQuery(sqlQuery);
    processResults(rs);
  } catch(SQLException e) {
    // ハンドラに処理を移す
  } finally {
      if (rs != null) { rs.close(); }
      if (stmt != null) { stmt.close(); }
      if (conn !=null) { conn.close(); }
  }
}
```

close()メソッドで SQLException が発生したら finally 節は途中で終了しちゃうよ!

サンプルコード: データベースとのやりとり

```
public void getResults(String sqlQuery) {
  Statement stmt = null;
  ResultSet rs = null;
  Connection conn = getConnection();
  try {
    stmt = conn.createStatement();
    rs = stmt.executeQuery(sqlQuery);
    processResults(rs);
  } catch (SQLException e) { ... ハンドラに処理を移す ...
  } finally {
    try {
      if (rs != null) {rs.close();}
    } catch (SQLException e) { ... ハンドラに処理を移す ...
    } finally {
      try {
        if (stmt != null) {stmt.close();}
      } catch (SQLException e) { ... ハンドラに処理を移す ...
      } finally {
        try {
          if (conn != null) {conn.close();}
        } catch (SQLException e) { ... ハンドラに処理を移す ...
}}}}}
```

try-catch-finally を駆使して全ての close() メソッドをガードすればなんとか...

サンプルコード: データベースとのやりとり

```
public void getResults(String sqlQuery) {
  try (Connection conn = getConnection();
       Statement stmt = conn.createStatement();
       ResultSet rs = stmt.executeQuery(sqlQuery))
{
    processResults(rs);
  } catch (SQLException e) {
    // ハンドラに処理を移す
  }
}
```

Java7 の try-with-resources を使うと簡潔なコードになるよ!

ストリーム入出力に関連するコーディングルール

- FIO04-J. 不要になったらリソースを解放する
- FIO05-J. wrap() や duplicate() メソッドで作成したバッファを信頼できないコードにアクセスさせない
- FIO06-J. ひとつの InputStream に対して複数のバッファ付きラッパーを作成しない
- FIO07-J. 外部プロセスに IO バッファをブロックさせない
- FIO08-J. 文字やバイトを読み取るメソッドの返り値は int で受ける
- FIO09-J. 0 から 255 の範囲に収まらない整数値を出力するときには write() メソッドを信用しない
- FIO10-J. read() を使って配列にデータを読み込むときには配列への読み込みが意図した通りに行われたことを確認する
- FIO11-J. バイナリデータを文字データとして読み込もうとしない
- FIO12-J. リトルエンディアン形式のデータを読み書きするメソッドを用意する
- FIO13-J. センシティブな情報を信頼境界の外に記録しない
- FIO14-J. プログラムの終了時には適切なクリーンアップを行う

- Java の入出力ストリーム
- **ファイルシステムの扱い**
- 共有ディレクトリの危険性

ディレクトリトラバーサル

- ディレクトリトラバーサル攻撃
  - アクセス可能であることを意図していないファイルへのアクセスを行う攻撃
  - ファイル関連の脆弱性で典型的なもの
  - 前回, 東京セミナpart2で, ディレクトリトラバーサルの問題を作り込んだコード例を紹介

- 参考
  - https://www.owasp.org/index.php/Path_Traversal
  - http://ja.wikipedia.org/wiki/ディレクトリトラバーサル

サンプルプログラム: ディレクトリトラバーサル

```
class filecat {

  static private String BASEDIR="/home/yozo/tmp/";

  static public void cat(String s) throws IOException {
      BufferedReader bf =
          new BufferedReader(new FileReader(BASEDIR + s));
      String line;
      while (null != (line = bf.readLine())){
        System.out.println(line);
      }
  }

  public static void main(String[] args) throws IOException {
    String filename = args[0];
    cat(filename);
  }
}
```

対象とするディレクトリ

単純な文字列連結だけ行っている

ファイル名を期待

前回part2で紹介

サンプルプログラム: ディレクトリトラバーサル

```
$ java filecat choi
Hoihoi
$ java filecat ../Documents/WORK/SecureCoding/20121014.Tokyo_part2/filecat.java
import java.io.BufferedReader;
Import java.io.FileReader;
Import java.io.FileNotFoundException;
Import java.io.IOException;
Class filecat {

  static private String BASEDIR=“/Users/yozo/tmp”;

…
```

想定通りの動作

想定外の入力！

前回part2で紹介

サンプルプログラム修正例

```
class filecat1 {

  static private String BASEDIR="/home/yozo/tmp/";

  static public void cat(String s) throws IOException {
      File f = new File(BASEDIR + s);
      String fs = f.getCanonicalFile().toString();
      if (!fs.startsWith(BASEDIR)) {
            throw (new IllegalArgumentException(fs));
      }
      BufferedReader bf =
          new BufferedReader(new FileReader(f));
      String line;
      while (null != (line = bf.readLine())){
        System.out.println(line);
      }
  }

  public static void main(String[] args) throws IOException {
    String filename = args[0];
    cat(filename);
  }
}
```

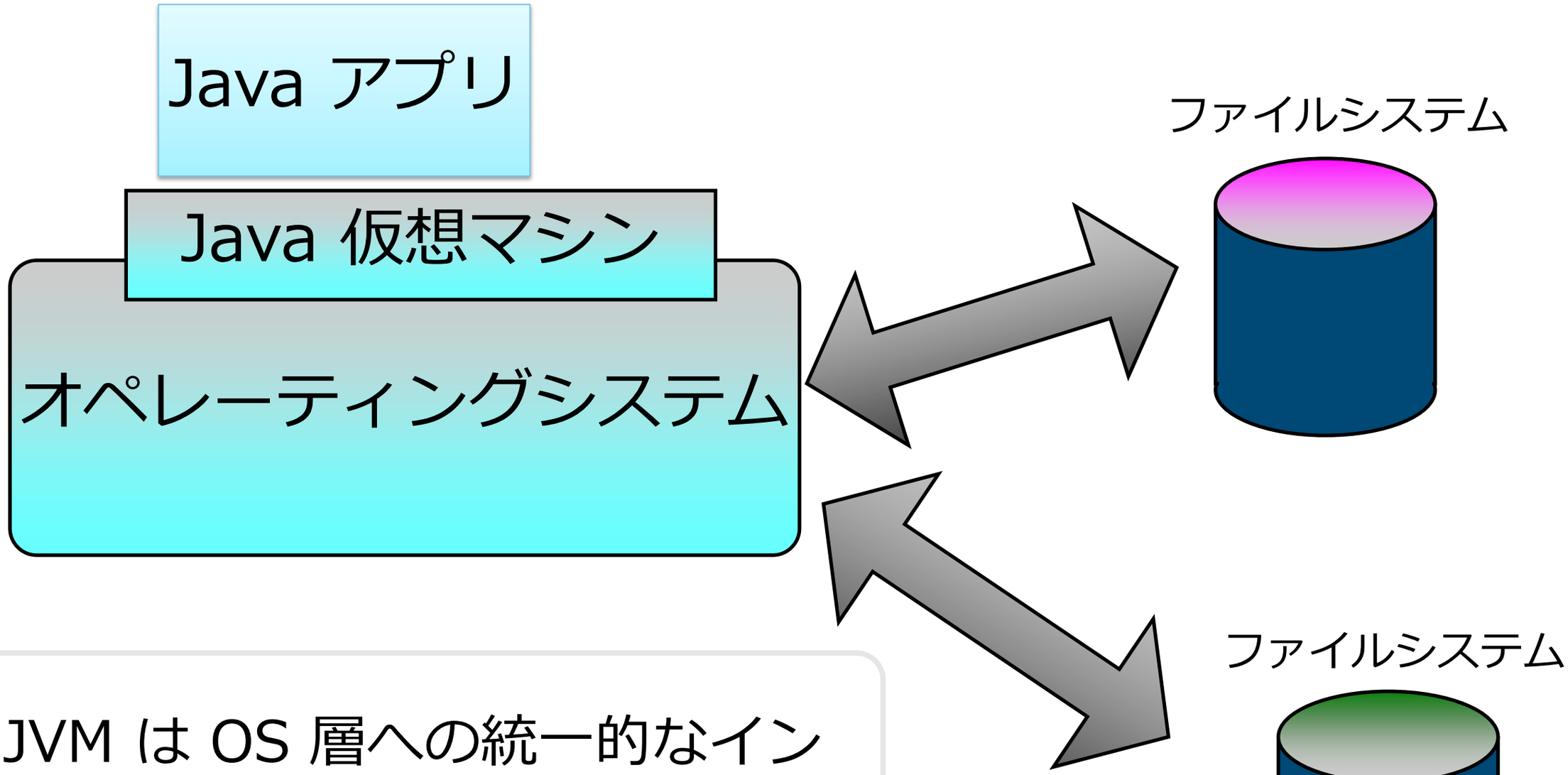
対策: canonicalパスで想定通りのディレクトリを指していることを確認する

File I/O

- 「最終的に、どんなOSでも実行できるコードになった。JVM（および、Java環境）が提供する抽象化を利用して、OSとファイルシステムの差異を隠蔽したおかげだ。」

Java The Good Parts, 6章 Java 仮想マシン

File I/O



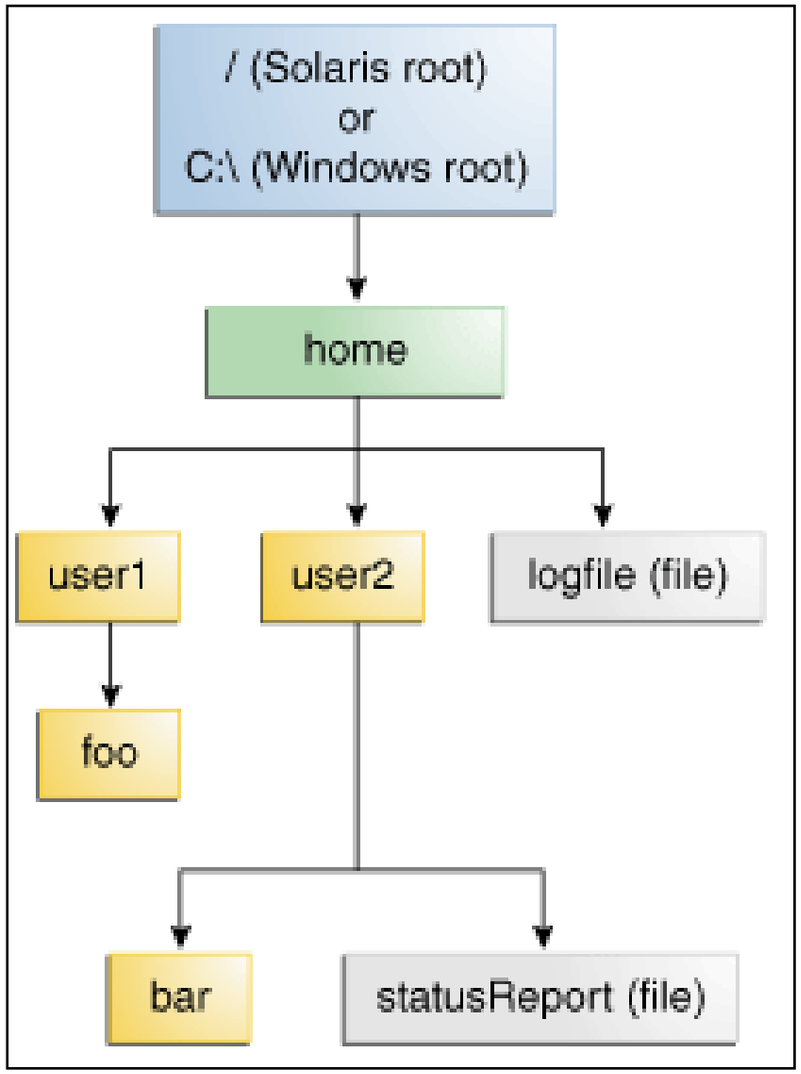
- JVM は OS 層への統一的なインタフェースを提供する
- OS 層はファイルシステムへの統一的なインタフェースを提供する

ファイルシステムにもいろいろ

- ファイルシステムの例
  - POSIX 系ファイルシステム
    - アクセスパーミッションによるアクセス制御
  - NFSv4, NTFS
    - Access Control List によるアクセス制御
  - FAT (MS-DOS)
    - file attribute によるアクセス制御

File I/O

ファイルシステムには**ファイル**と**ディレクトリ**が収められている. それらに対する**アクセス制御の機能**が実装されている.



ファイルシステムにはその実装形態によってそれぞれ固有の特徴があり, それらを理解して**適切に**扱う必要がある.

file separator, ファイル名の最大長, ディレクトリ階層の深さ, ファイル名に使える文字種, ...

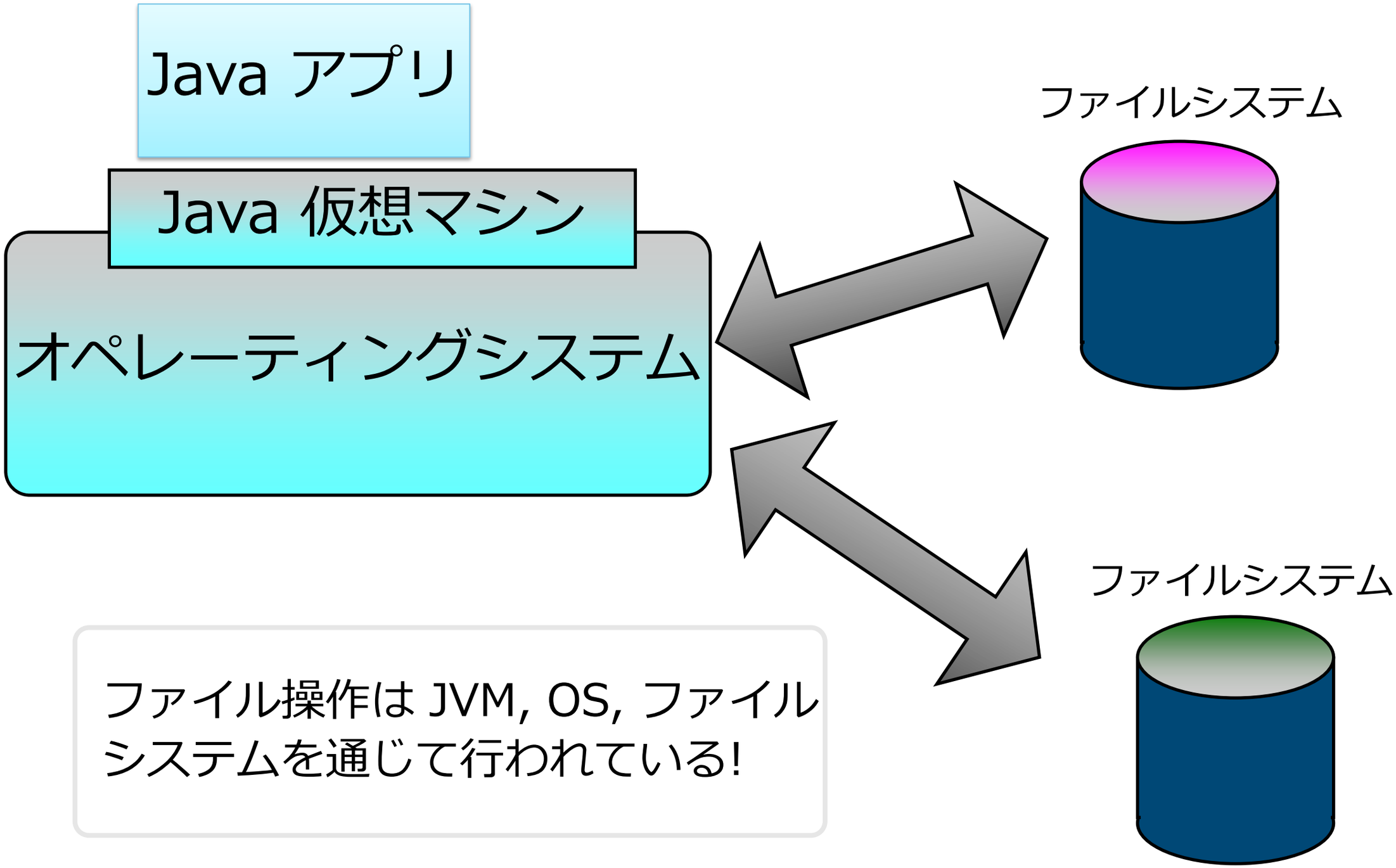
File I/O

- ポータビリティに関する Java のスローガン
  - ***“Write once, run anywhere”***
    - JRE(Java実行環境)は OS に依らない API をJava アプリに提供してくれる

しかし

セキュアなファイル操作を行うためには, ファイルシステムの詳細を理解しておく必要がある

File I/O



ファイル操作は JVM, OS, ファイルシステムを通じて行われている!

File I/O

POSIX

```
$ ls -al showAttributesPosix.java
-rw-r--r--  1 yozo  staff  764 Nov 9 14:51 showAttributesPosix.java
$
```

NTFS

```
C:>dir /q showAttributesACL.java
......
2012/11/09  15:07    611  earlgrey¥yozo   showAttributesACL.java
......
C:>icacls showAttributesACL.java
showAttributesACL.java  NT AUTHORITY¥SYSTEM:(I)(F)
                        BUILTIN¥Administrators:(I)(F)
                        earlgrey¥yozo:(I)(F)

......
C:>
```

File I/O

POSIX

```
class showAttributesPosix {
  public static void main(String[] args)
        throws IOException {
    if (args.length <= 0){ return; }
    Path p = FileSystems.getDefault().getPath(args[0]);
    PosixFileAttributes attrs =
        Files.getFileAttributeView(p, PosixFileAttributeView.class)
          .readAttributes();
    System.out.format("%s %s%n",
            attris.owner().getName(),
            PosixFilePermissions.toString(attrs.permissions()));
  }
}
```

```
$ java showAttributesPosix showAttributesPosix.java
yozo -rw-r--r—
$
```

File I/O

ACL

```
class showAttributesACL {
  public static void main(String[] args) throws IOException {
    if (args.length <= 0){ return; }
    Path p = FileSystems.getDefault().getPath(args[0]);
    List<AclEntry> acls =
        Files.getFileAttributeView(p, AclFileAttributeView.class)
          .getAcl();
    for(AclEntry acl : acls) {
      System.out.println(acl.toString());
    }
  }
}
```

```
C:> java showAttributesACL showAttributesACL.java
NT AUTHORITY¥SYSTEM:
READ_DATA/WRITE_DATA/APPEND_DATA/READ_NAMED_ATTRS/WRITE_NAMED_ATTRS/EXECUTE/DELETE_CHI
LD/READ_ATTRIBUTES/WRITE_ATTRIBUTES/DELETE/READ_ACL/WRITE_ACL/WRITE_OWNER/SYNCHRONIZE:
ALLOW
BUILTIN¥Administrators:READ_DATA/WRITE_DATA/APPEND_DATA/READ_NAMED_ATTRS/WRITE_NAMED_A
TTRS/EXECUTE/DELETE_CHILD/READ_ATTRIBUTES/WRITE_ATTRIBUTES/DELETE/READ_ACL/WRITE_ACL/W
RITE_OWNER/SYNCHRONIZE:ALLOW
earlgrey¥yozo:READ_DATA/WRITE_DATA/APPEND_DATA/READ_NAMED_ATTRS/WRITE_NAMED_ATTRS/EXEC
UTE/DELETE_CHILD/READ_ATTRIBUTES/WRITE_ATTRIBUTES/DELETE/READ_ACL/WRITE_ACL/WRITE_OWNE
R/SYNCHRONIZE:ALLOW
C:>
```

File I/O

- OSとファイルシステムの差異を隠蔽したって言っても、隠しきれないものもある。
- ポータビリティを確保するのは難しい。

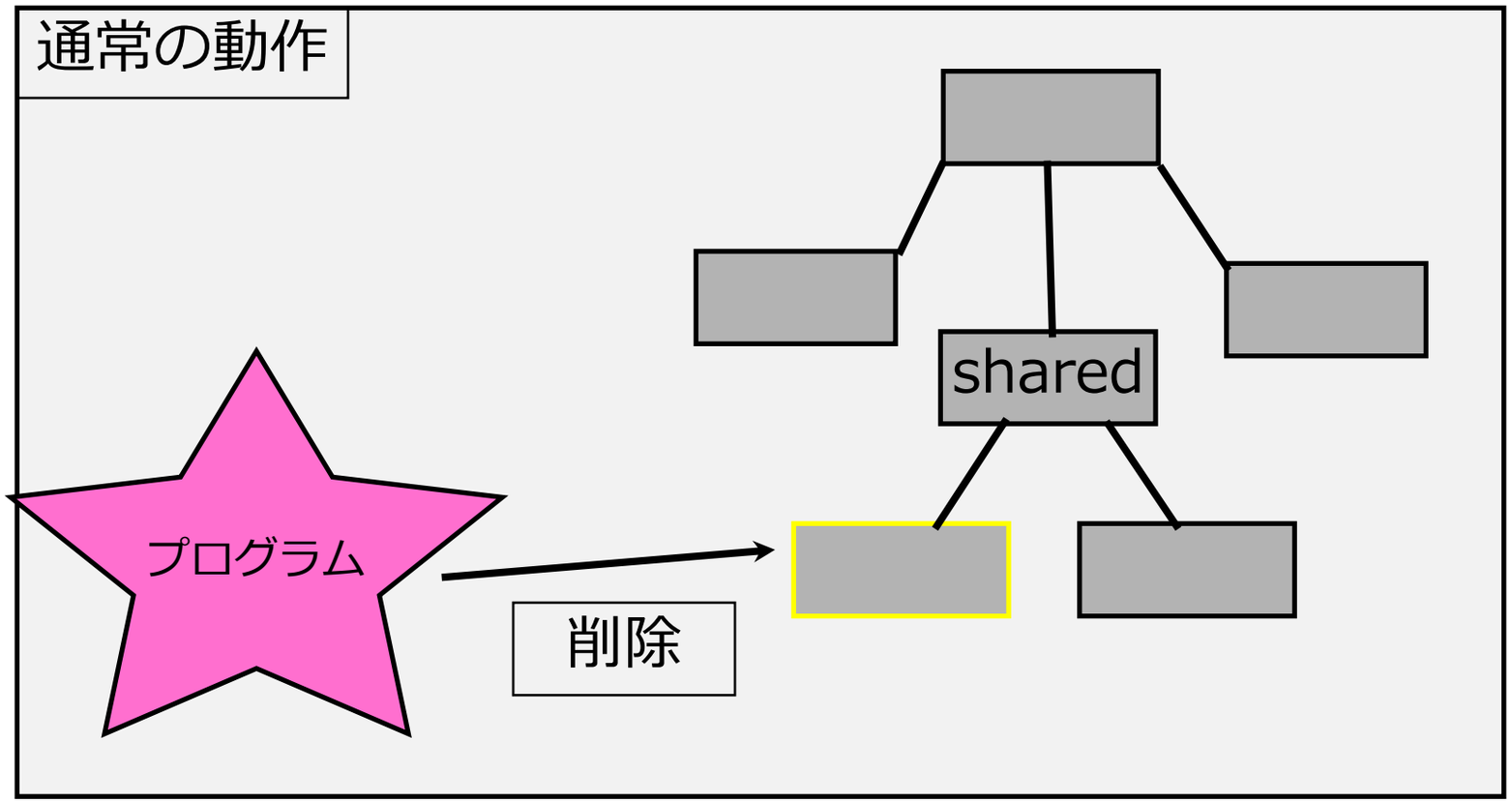
- Java の入出力ストリーム
- ファイルシステムの扱い
- **共有ディレクトリの危険性**

共有ディレクトリの危険性

- **共有ディレクトリ**
  - マルチユーザシステムにおいて, 複数のユーザに対してアクセスが許されているディレクトリのことを「共有ディレクトリ」と呼ぶ.
    - POSIX 系における “/tmp” や Windows における “C:¥Windows¥Temp” など

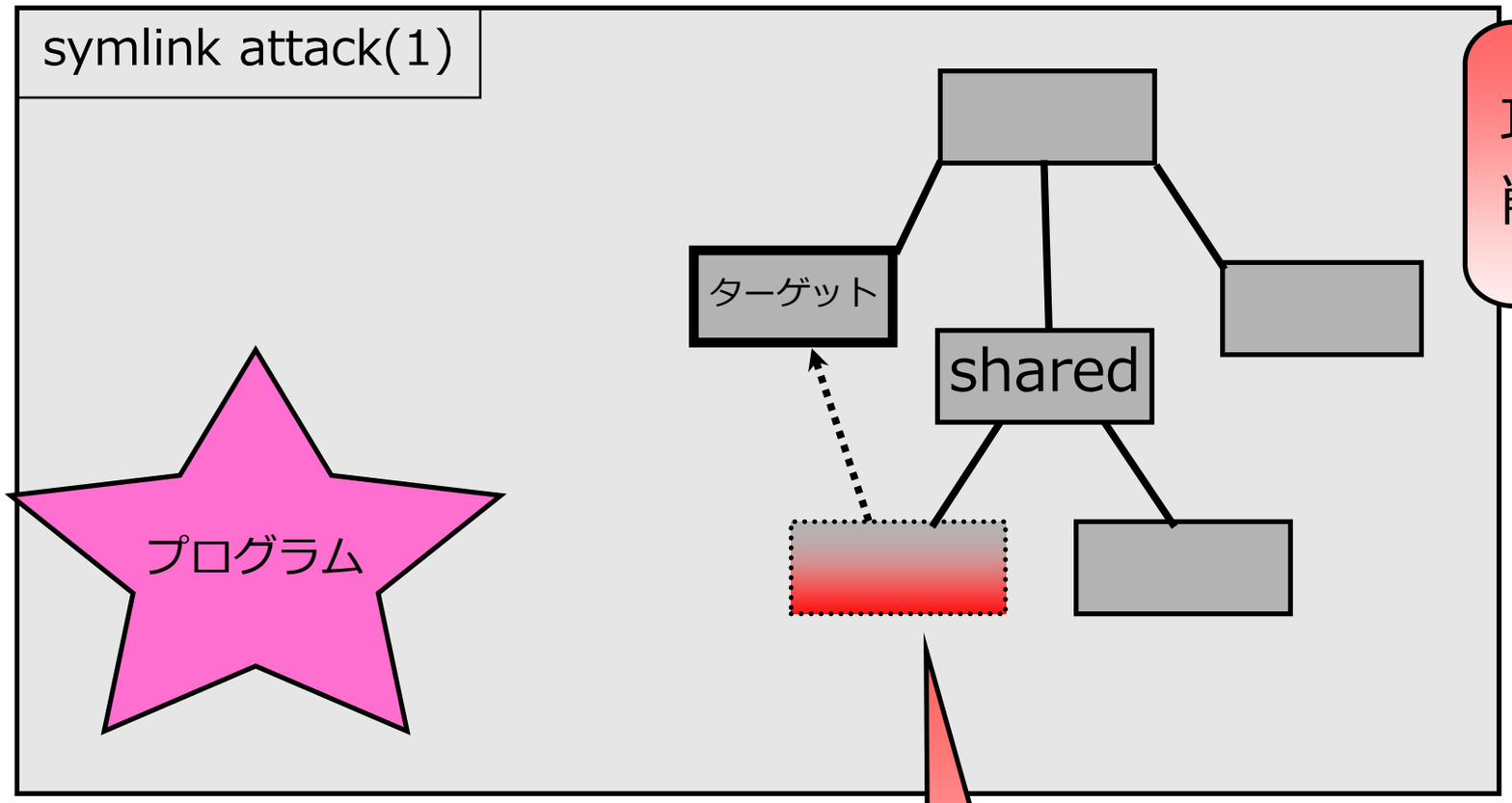
共有ディレクトリは危険!
他のユーザがファイルを改ざん/削除したり, 他のファイルへのリンクを作るかもしれない

Symlink 攻撃



他のプログラムがファイルにアクセスしなければ問題は発生しない

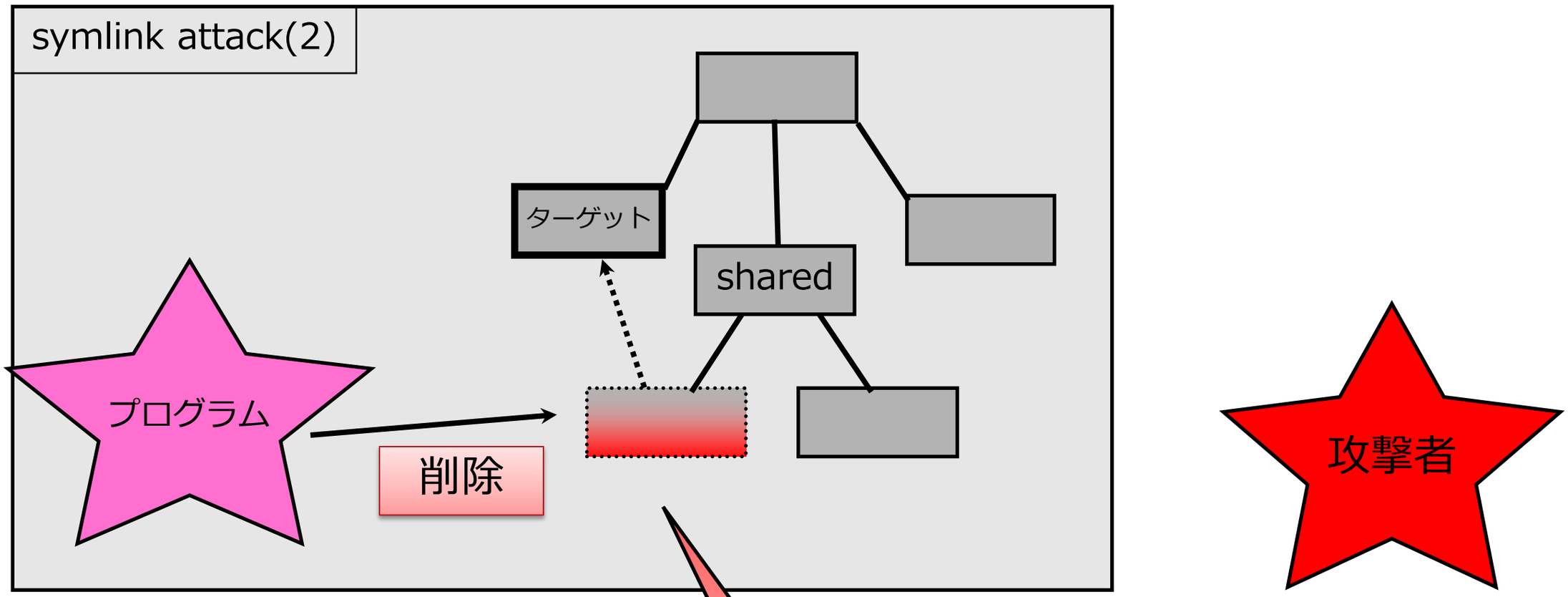
Symlink 攻撃



攻撃者はターゲットファイルを削除する権限を持っていない

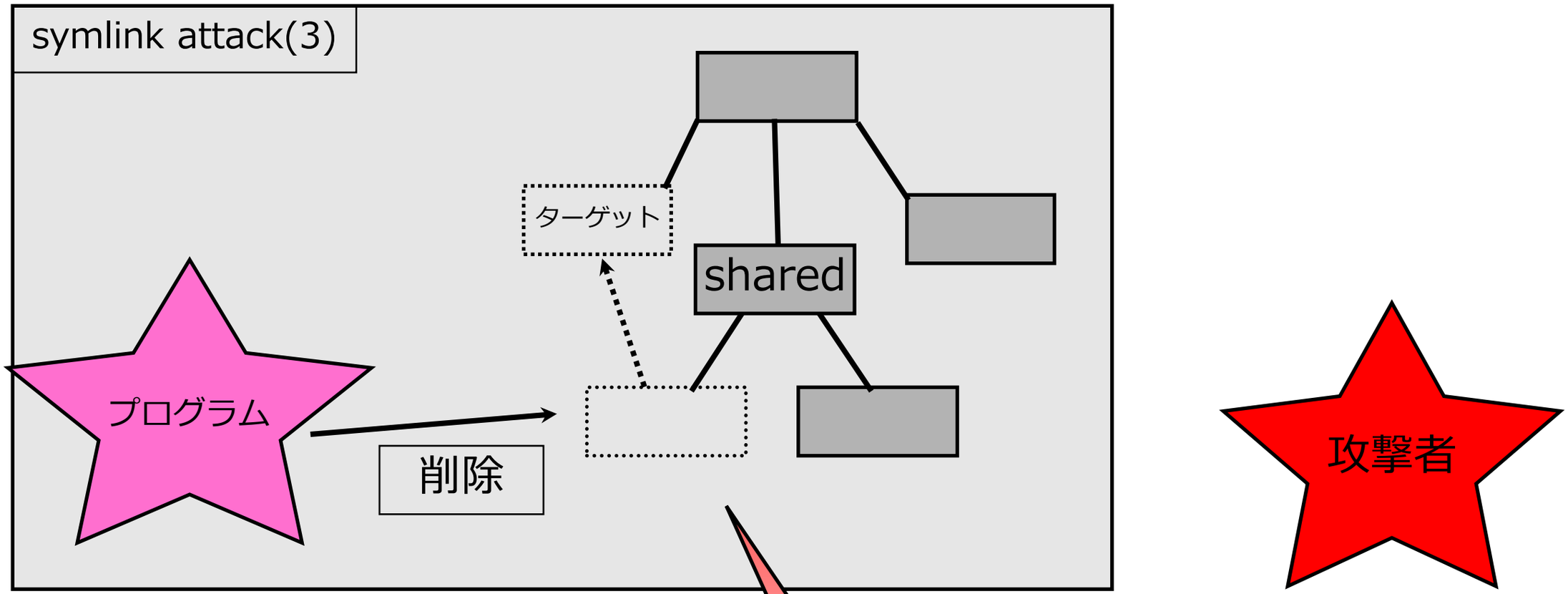
攻撃者はプログラムが操作するファイルをあらかじめ削除し, ターゲットファイルへのシンボリックリンクを作る.

Symlink 攻撃



プログラムは, シンボリックリンクに置き換えられているとは知らずに問題のファイルを削除する...

Symlink 攻撃



... そして target ファイルは削除されてしまった!

セキュアディレクトリ

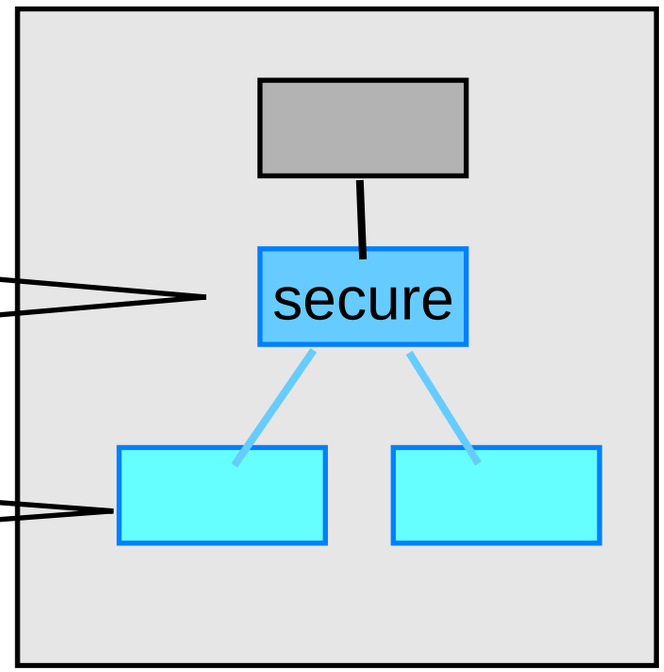
セキュアディレクトリ

- 自分(とシステム管理者)のみにアクセスを制限しているディレクトリ

他のユーザがファイルを改ざんしないことが保証される

セキュアディレクトリ

他のユーザはこれらのファイルを改変できない.



まとめ

- JRE(Java 実行環境)はOS層を抽象化してくれる
- しかし以下のような処理が必要な場合は OSや ファイルシステムの基本的な特徴を理解しておくべき.
  - ファイル操作(作成, 変更, 削除など)
  - ファイルのアクセス制御に関する操作

ファイル入出力に関連するコーディングルール

- FIO00-J. 共有ディレクトリにあるファイルを操作しない
- FIO01-J. 適切なパーミッションを設定してファイルを作成する
- FIO02-J. ファイル関連エラーを検知し、処理する
- FIO03-J. 一時ファイルはプログラムの終了前に削除する

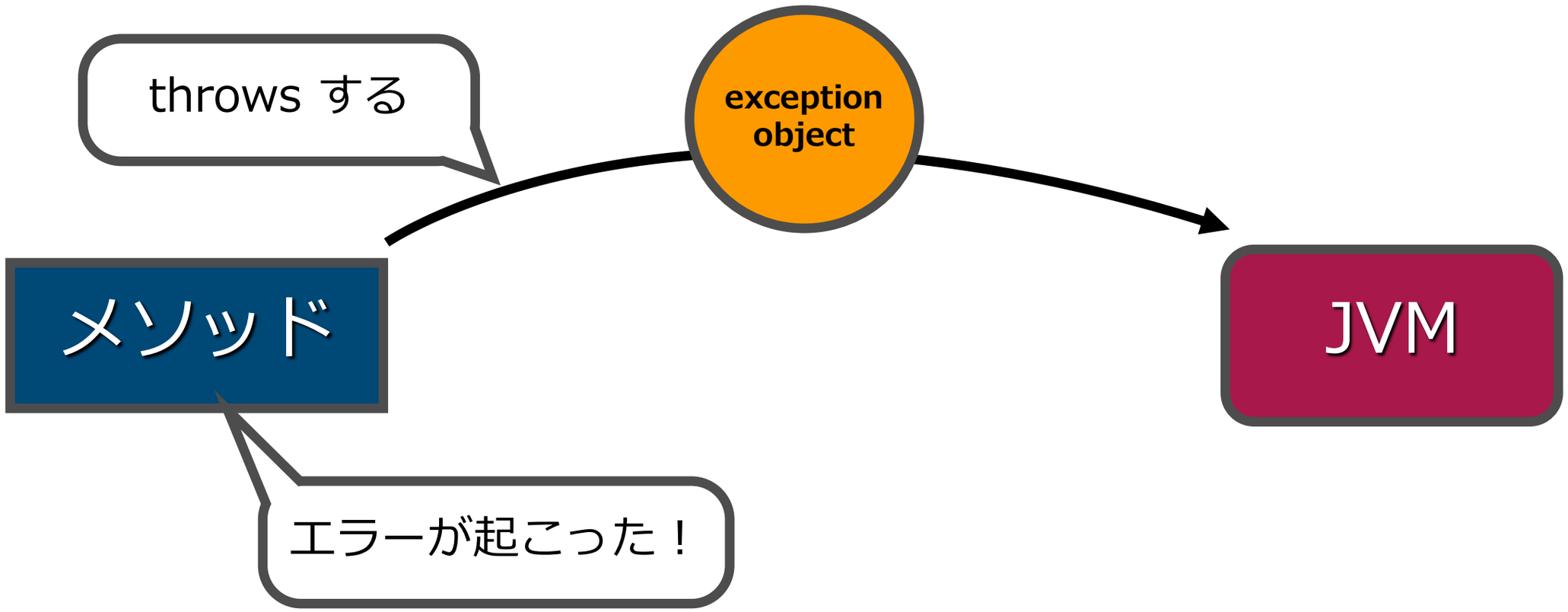
例外時の動作

例外とは

- exception = exceptional event

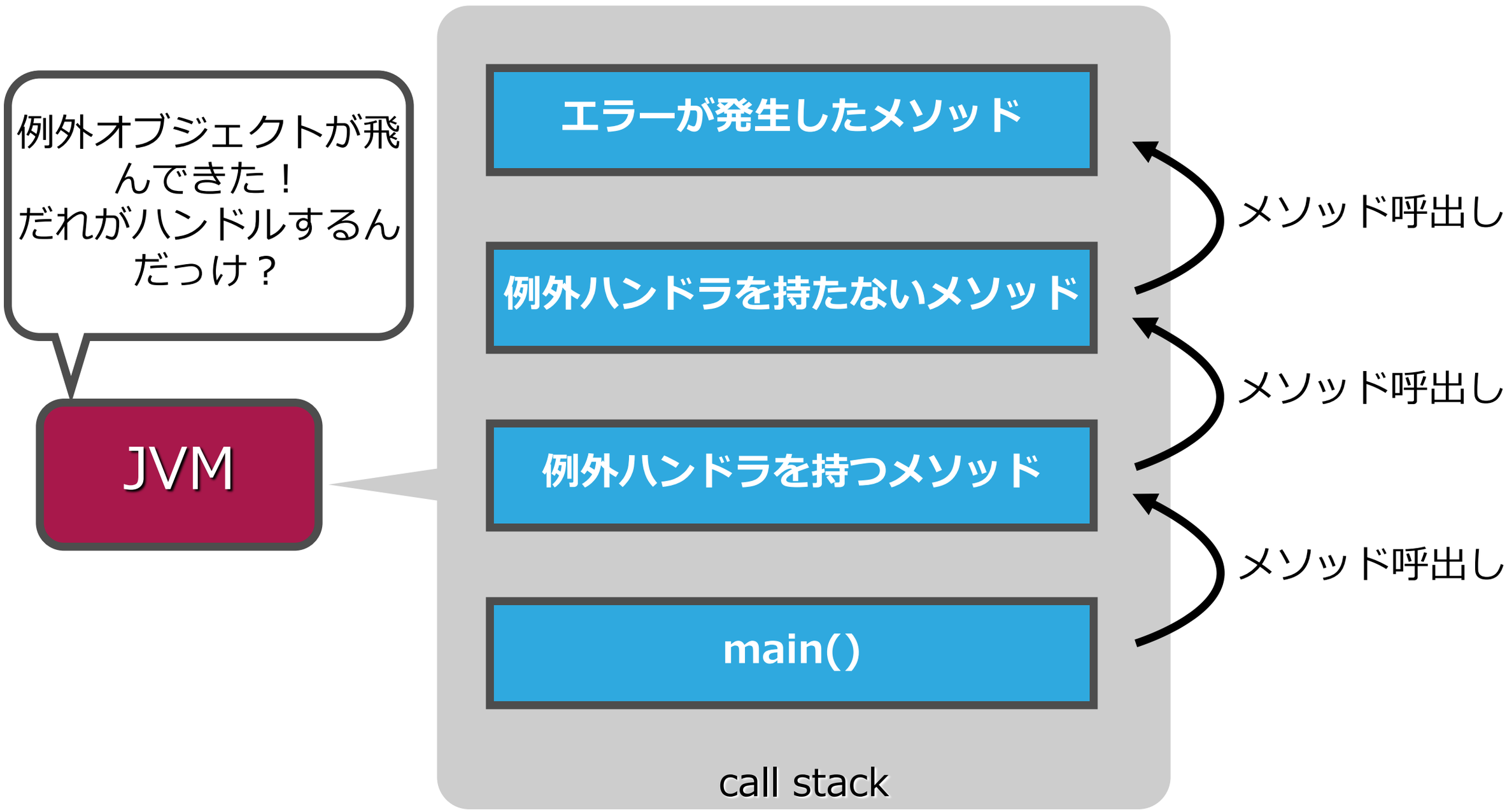
- 定義
  - 例外(exception)とは、プログラムの実行中に発生し、プログラム実行の正常な流れを分断するイベントのこと
    - 例：ゼロ除算、配列の境界外アクセス

例外とは



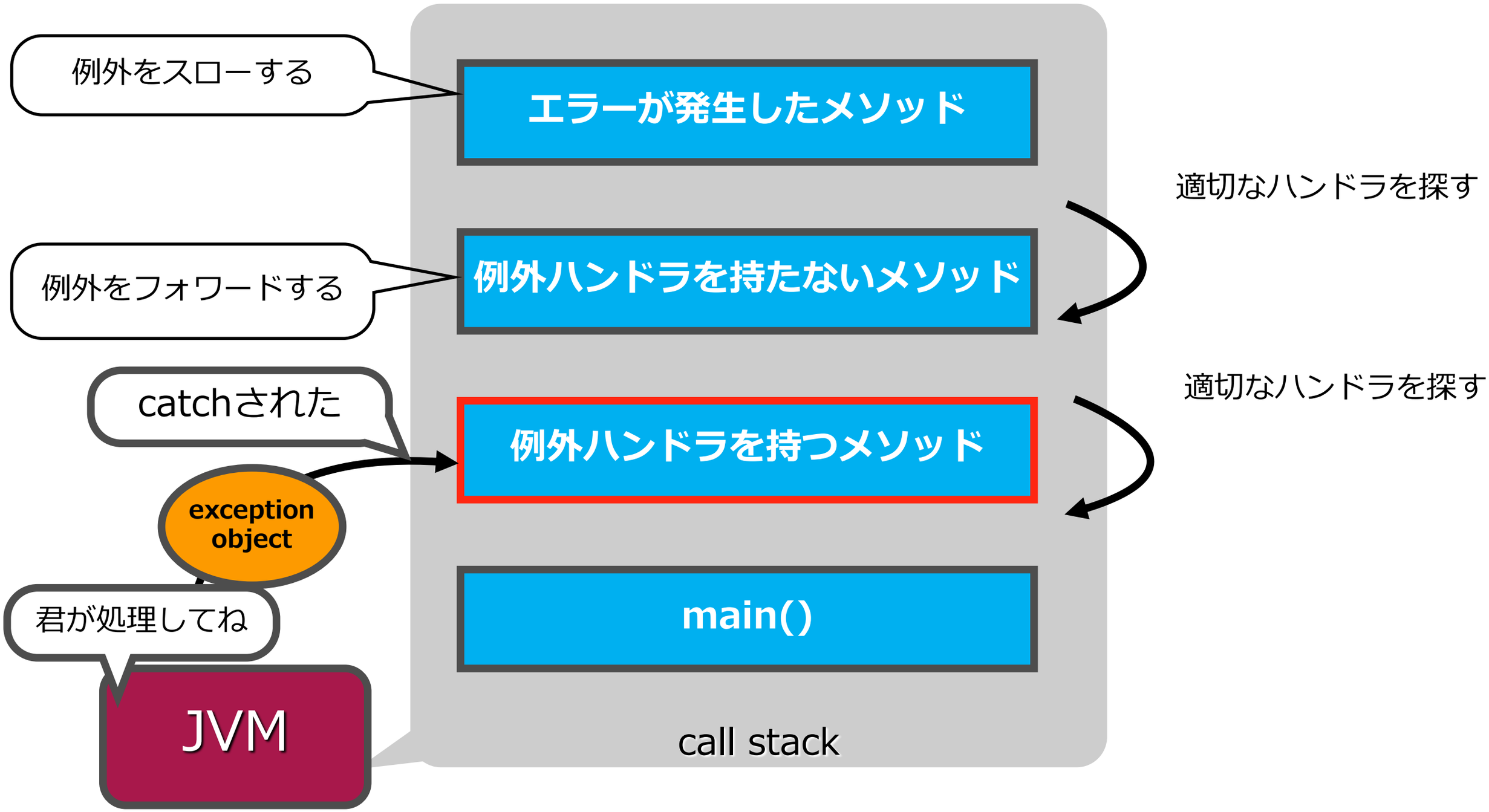
例外とは

- コールスタック(call stack)



例外とは

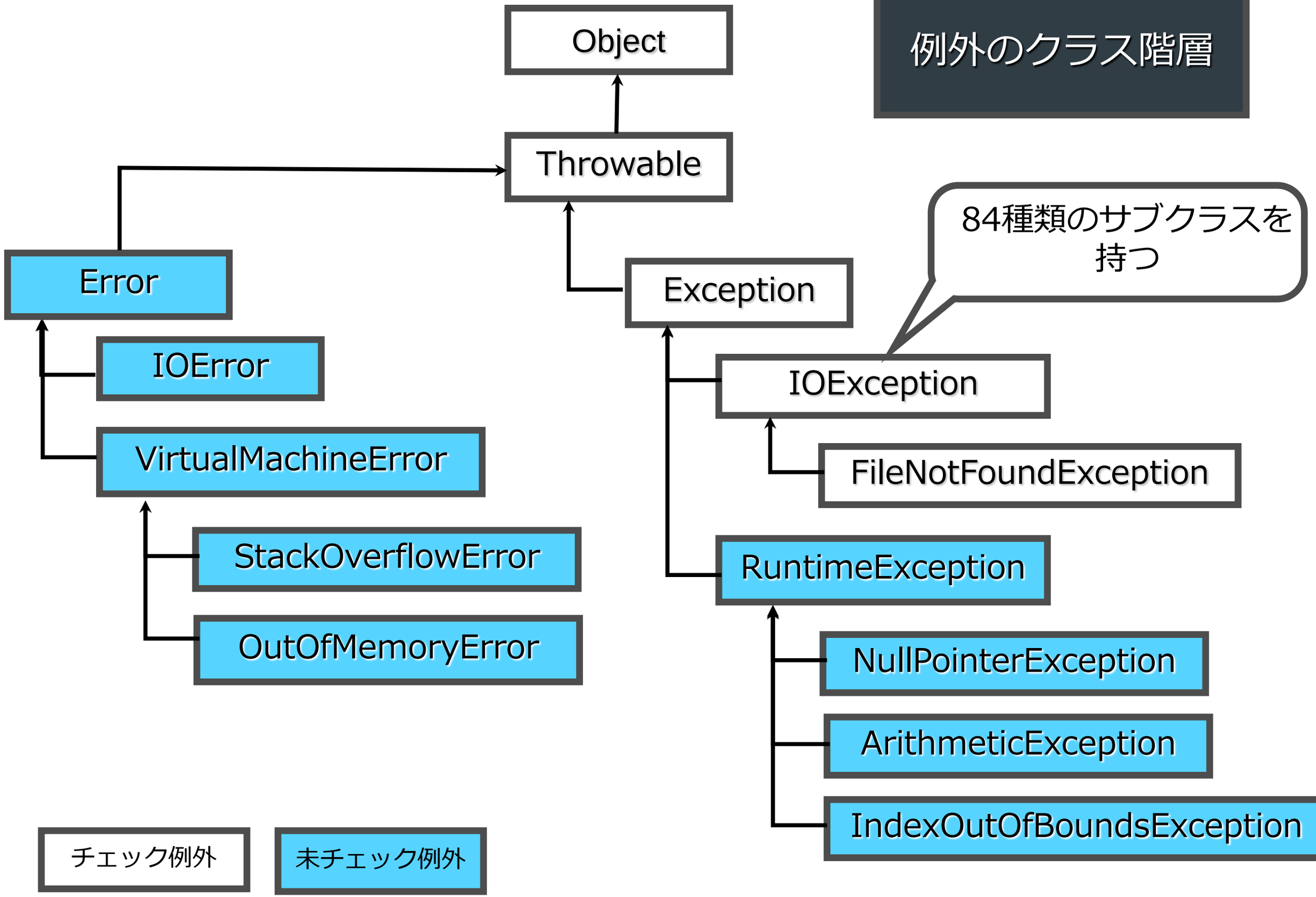
- コールスタックを調べ、例外ハンドラをみつける



例外とは

- 例外 = オブジェクト
  - エラー発生時のプログラムの状態を、コールスタックの上位に伝える
  - エラーを種類によってグループ化/差異化
  - 拡張できる
- 通常の処理を行うコードと、エラー処理のコードの分離

例外のクラス階層



3つの例外

- 未チェック例外 (unchecked exception)
  - **Error**
  - **RuntimeException**
    - コンパイラはチェックしてくれない
    - **throws** 宣言しなくてもよい
    - アプリは例外から戻れないという想定
      - 「実行時例外」という訳語

- **チェック例外** (checked exception)
  - ハンドラを必ず実装
  - コンパイラがチェック

未チェック例外

- **Error** と **RuntimeException**
  - コンパイラによるチェックは行われない
  - メソッドで必ずしもthrowしなくてもよい
  - ランタイム(Javaの実行環境)が出す例外なのでプログラムではどうしようもない、という発想

未チェック例外

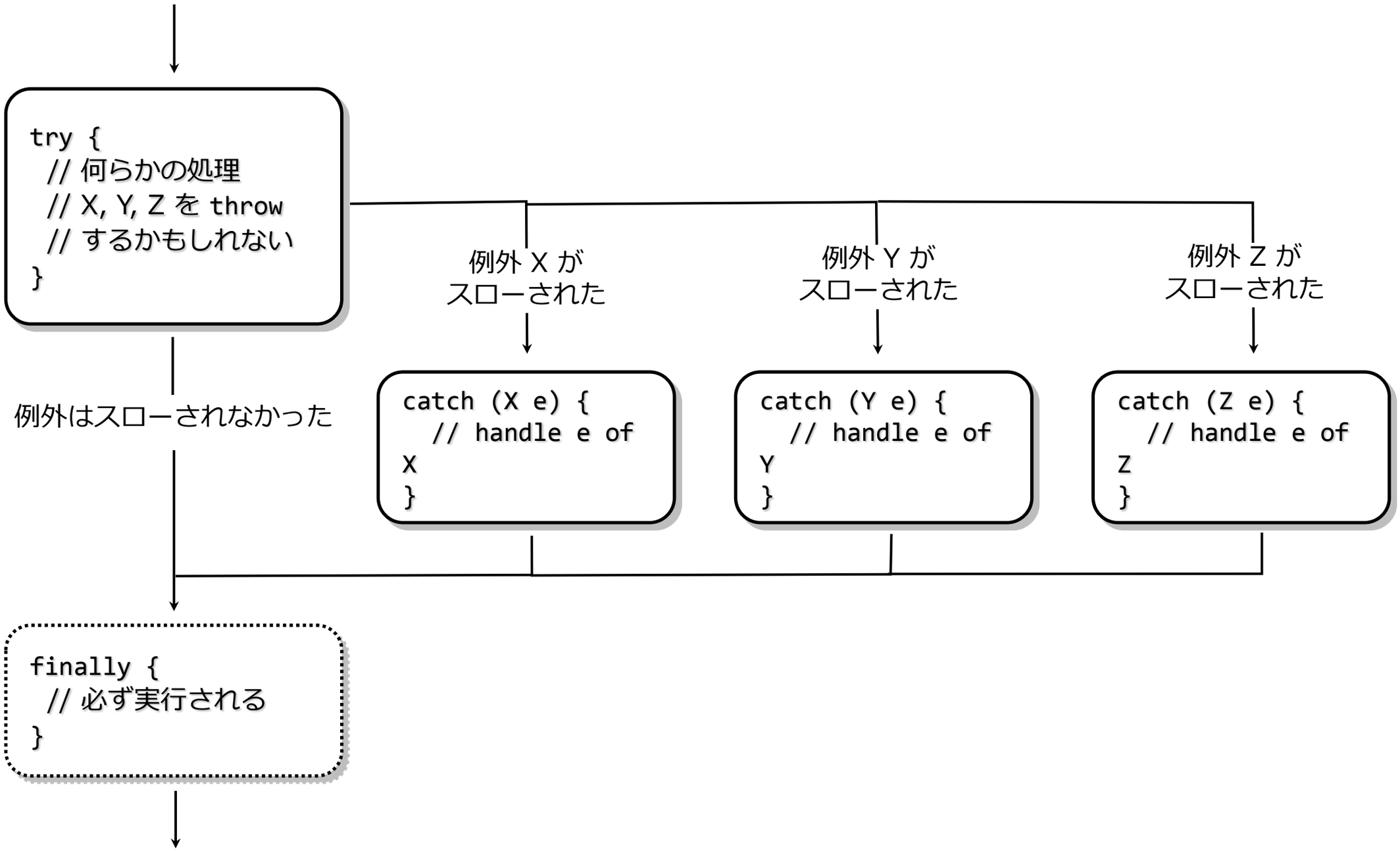
- 未チェック例外クラスであるエラークラス(Errorとそのサブクラス)は、プログラム中の様々な場所で発生する可能性があり、そういったものの回復は難しく、あるいは不可能であるため、コンパイル時にはチェックされない
- 実行時例外クラス(RuntimeExceptionとそのサブクラス)は、こういった例外の宣言を強制してもJavaプログラムの正当性を確立するための著しい助けにはならないだろう、というプログラミング言語の設計者達による判断によって、コンパイル時にはチェックされない。

Java 言語仕様
§11.2.4 コンパイル時の例外のチェックされない理由
§11.2.5 実行時例外がチェックされない理由

チェック例外

- ハンドラを必ず実装しないとだめ
- コンパイラがチェックする

try-catch-finallyの制御構造



例外処理のアンチパターンとセキュリティ

- 例外をキャッチしない (CWE-248)
- チェック例外を無視/抑制する (ERR00-J, CWE-390, CWE-391)
- プログラム制御に例外を使用する (EffectiveJava 57)
- エラーメッセージを通じたセンシティブな情報を漏洩する (CWE-209)
- NullPointerExceptionをキャッチする (ERR08-J, CWE-395)
- 例外スロー時にリソースのクリーンアップを適切に行わない (CWE-460)
- スローする例外が一般的すぎる (CWE-397)
- プログラムの異常状態や例外条件を適切にチェックしない (CWE-754)

アンチパターン1

例外をキャッチしない

例外をキャッチしない

- あとでオブジェクトを使うときにエラーが発生する
- 現実にはこのアプローチをとるコードが非常に多い（らしい）

```
protected void doPost (
HttpServletRequest req, HttpServletResponse res) throws IOException {
    String ip = req.getRemoteAddr();
    InetAddress addr = InetAddress.getByName(ip);
    ...
    out.println("hello " + addr.getHostName());
}
```

- DNS lookupに失敗するとServletは例外をスローするにもかかわらず、キャッチしていない
- アプリがクラッシュ、DoS攻撃に悪用される

アンチパターン2

チェック例外を無視／抑制する

チェック例外を無視/抑制する

```
public String readFile(String filename) {
   String retString = null;
   try {
      // File および FileReader オブジェクトを初期化
      File file = new File(filename);
      FileReader fr = new FileReader(file);

      // 文字バッファを初期化
      long fLen = file.length();
      char[] cBuf = new char[(int) fLen];

      // ファイルからデータを読込む
      int iRead = fr.read(cBuf, 0, (int) fLen);

      // クローズ処理
      fr.close();

      retString = new String(cBuf);
   } catch (Exception ex) { }
   return retString;
}
```

例外が発生するのは
どこ？

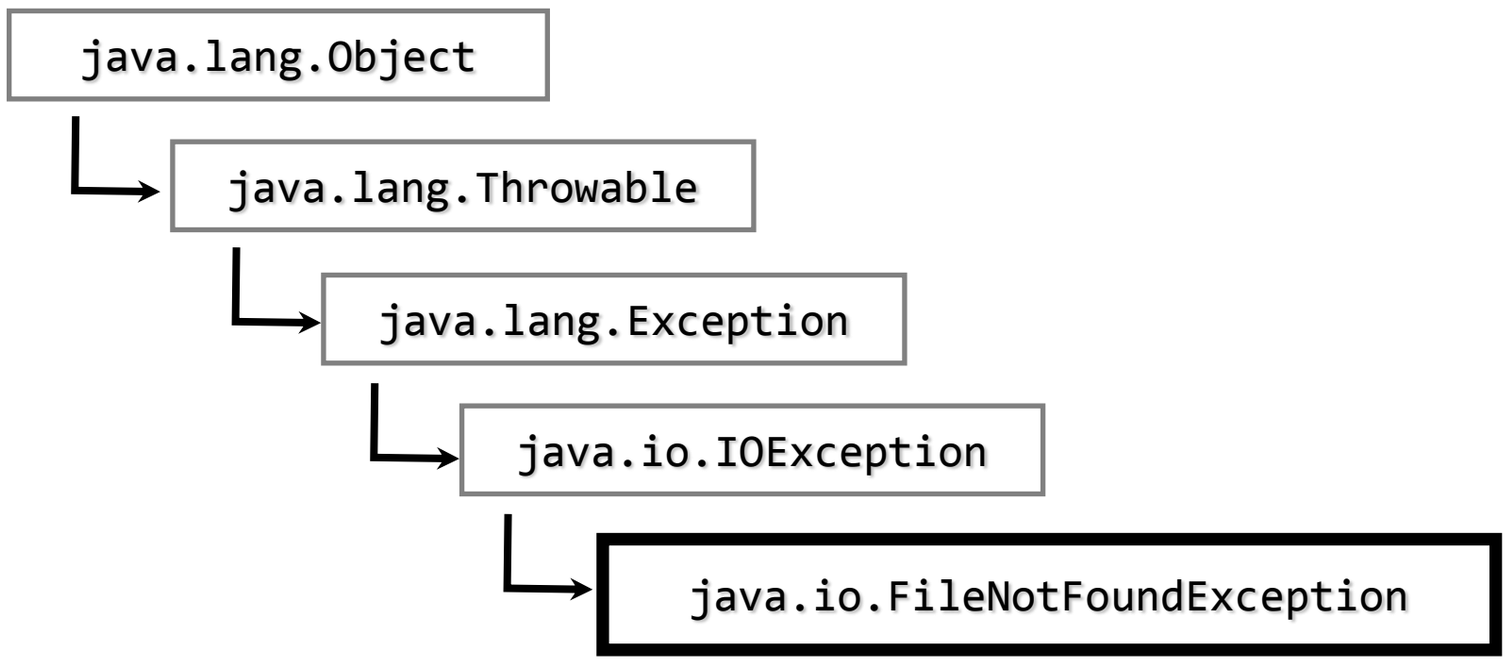
```
public String readFile(String filename) throws FileNotFoundException, IOException, Exception {
   String retString = null;
   try {
      // initialize File and FileReader objects
      File file = new File(filename);
      FileReader fr = new FileReader(file);

      // initialize character buffer
      long fLen = file.length();
      char [] cBuf = new char[(int) fLen];

      // read data from file
      int iRead = fr.read(cBuf, 0, (int) fLen);

      // close file
      fr.close();

      retString = new String(cBuf);
   } catch (FileNotFoundException ex) {
      System.err.println ("Error: FileNotFoundException opening the input file: " +
filename );
      System.err.println ("" + ex.getMessage() );
      throw new FileNotFoundException(ex.getMessage());
   } catch (IOException ex) {
      System.err.println("Error: IOException reading the input file.¥n" + ex.getMessage() );
      throw new IOException(ex);
   } catch (Exception ex) {
      System.err.println("Error: Exception reading the input file.¥n" + ex.getMessage() );
      throw new Exception(ex);
   }
   return retString;
}
```

各例外に対するハンドラを実装（この場合、例外に関する情報をユーザに出力してから、例外を再スローしている）

FileNotFoundException

- FileNotFoundExceptionはチェック例外
- 指定されたファイルが開けない場合、APIは、呼出し元がなんらかのアクションをとることを期待する
  - 例：ユーザに別のファイルを指定させる

- 前述のコードでは、呼出し元に処理を委譲するため、再度例外をスローしていた



アンチパターン3

制御フローに例外を使う

例外の誤用

```
public class Antipattern {
  public static void main(String[] args) {
    try {
      int i = 0;
      while (true) {
        System.out.println(args[i++]);
      }
    } catch (ArrayIndexOutOfBoundsException e) {
    }
  }
}
```

例外の誤用

```
public class Antipattern {
  public static void main(String[] args) {
    int i = 0;
    for (i = 0; i < args.length(); i++) {
      System.out.println(args[i]);
    }
  }
}
```

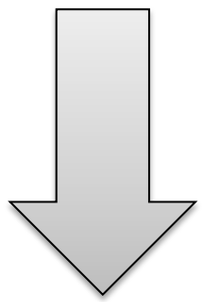
- 例外はコードの最適化に役立たない
- JVMによる最適化を排除する可能性がある

アンチパターン4

例外を通じてセンシティブな情報を漏洩する

例外に含まれるセンシティブな情報

- 実行スタックのスナップショット
- 例外メッセージ
- 例外の種類



例外オブジェクトに含まれるこれらの情報は、
攻撃の手がかりを与えてしまう

センシティブな情報を含む例外

例外名	漏洩する情報/脅威
`java.io.FileNotFoundException`	ファイルシステムの構成、ユーザ名の列挙
`java.sql.SQLException`	データベースの構成、ユーザ名の列挙
`java.net.BindException`	信頼できないクライアントがサーバのポートを選択できる場合、開いているポートを列挙
`java.util.ConcurrentModificationException`	スレッドセーフでないコードに関する情報
`javax.naming.InsufficientResourcesException`	不十分なサーバーリソース (DoSにつながる可能性)
`java.util.MissingResourceException`	リソースの列挙
`java.util.jar.JarException`	ファイルシステムの構成
`java.security.acl.NotOwnerException`	所有者の列挙
`java.lang.OutOfMemoryError`	DoS
`java.lang.StackOverflowError`	DoS

問題のあるコード

```
class FileOpen {
  public static void main(String[] args)
            throws FileNotFoundException, IOException {
    FileReader fr = null;
    try {
      fr = new FileReader("fff.txt");
    } finally {
      if (fr != null){
        fr.close();
      }
    }
  }
}
```

fff.txtが見つからないと、FileNotFoundExceptionがスローされる

```
Exception in thread "main" java.io.FileNotFoundException: fff.txt (No such
file or directory)at java.io.FileInputStream.open(Native Method)
     at java.io.FileInputStream.<init>(FileInputStream.java:120)
     at java.io.FileInputStream.<init>(FileInputStream.java:79)
     at java.io.FileReader.<init>(FileReader.java:41)
     at FileOpen0.main(FileOpen0.java:9)
```

問題のあるコード

```
class FileOpen {
  public static void main(String[] args)
            throws FileNotFoundException, IOException {
    FileReader fr = null;
    try {
      fr = new FileReader(args[0]);
    } catch (FileNotFoundException ex) {
      String logMessage = “Unable to find file:” + arg[1];
      Logger.getLogger(FileOpen.class.getName()).log(Level.SEVERE, logMessage, ex);
    } finally {
      if (fr != null){
        fr.close();
      }
    }
  }
}
```

指定されたファイルが見つからないと、エラーメッセージが作られ、ログファイルに出力される

- ファイルシステムに「あるファイル」が存在するかどうかがログを見れば分かってしまう

センシティブな情報をフィルターする

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
class FileOpen {
  public static void main(String[] args)
            throws FileNotFoundException, IOException {
    FileReader fr = null;
    try {
        fr = new FileReader("fff.txt");
    } catch (FileNotFoundException f){
        throw new FileNotFoundException("not found");
    } finally {
        if (fr != null){
              fr.close();
        }
    }
  }
}
```

ファイル名を例外から削る

```
Exception in thread "main" java.io.FileNotFoundException: not found
     at FileOpen.main(FileOpen.java:11)
```

センシティブな情報をフィルターする

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
class FileOpen {
  public static void main(String[] args)
            throws FileNotFoundException, IOException {
    FileReader fr = null;
    try {
        fr = new FileReader("fff.txt");
    } catch (FileNotFoundException f){
        throw new FileNotFoundException("not found");
    } finally {
        if (fr != null){
              fr.close();
        }
    }
  }
}
```

IOExceptionをスローする

```
Exception in thread "main" java.io.IOException: something wrong!
     at FileOpen.main(FileOpen.java:11)
```

アンチパターン5

NullPointerException()をキャッチする

java.lang.NullPointerException

- **RuntimeException**を継承している
  - つまり未チェック例外
- オブジェクトが要求されるコードでnullが使われるとスローされる：
  - nullオブジェクトのインスタンスメソッド呼出し
  - nullオブジェクトのフィールドのアクセス/変更
  - 配列だと思ってnullのlengthを取得する
  - 配列だと思ってnullのスロットへアクセス/変更
  - Throwableの値だと思ってnullをスローする

なぜNullPointerExceptionをキャッチしてはいけないのか

- 理由1. NullPointerExceptionをキャッチする方が、値がnullかどうかチェックするコードを追加するより、オーバーヘッドがはるかに大きい

- 理由2. tryブロック中の複数の式がNullPointerExceptionをスローする可能性がある場合、どの式が例外を発生させたのかを判断することは困難／不可能

- 理由3. NullPointerExceptionがスローされた後、プログラムが想定通りに動作する状態であることはまれ

結論：キャッチするんじゃなくて、例外が発生する根本原因を修正すべきである

問題のあるコード

```
boolean isName(String s) {
  try {
    String names[] = s.split(" ");

    if (names.length != 2) {
      return false;
    }
    return (isCapitalized(names[0]) &&
isCapitalized(names[1]));
  } catch (NullPointerException e) {
    return false;
  }
}
```

このコードの何が問題？

問題のあるコード

```
boolean isName(String s) {
  if (s == null) {
    return false;
  }
  String names[] = s.split(" ");
  if (names.length != 2) {
    return false;
  }
  return (isCapitalized(names[0]) &&
isCapitalized(names[1]));
}
```

sを使う前にnullチェックを行う
(メソッドの引数を検証)

- NullPointerExceptionをキャッチすると、null参照を隠蔽したり、アプリのパフォーマンスが低下したり、保守性の低いコードを生み出すことになりかねません。気をつけましょう。

脆弱性事例

Apache ActiveMQ の認証不備

Apache ActiveMQ (AMQ-1272)

- Apache ActiveMQ: メッセージングサーバ
  - JMS (Java Message Service)を実装したミドルウェア
  - システムコンポーネント間のメッセージ通信を実現する

- 脆弱性の概要
  - ユーザ認証に失敗すると認証モジュールは例外を発生.
  - しかし, その例外は無視されて認証に成功したものとされてしまう.
  - その結果, どんなユーザ名/パスワードでもサービスを利用できた.
  - https://issues.apache.org/jira/browse/AMQ-1272

Apache ActiveMQ (AMQ-1272)

認証失敗
認証モジュールは
例外をスロー

例外は無視され,
そのままメッセージ受付

無効なユーザ名/パスコード

メッセージ送信

Apache ActiveMQ (AMQ-1272)

```
public void addConnection(ConnectionContext context, ConnectionInfo info) throws Exception
{

    if (context.getSecurityContext() == null) {
        // Check the username and password.
        String pw = (String)userPasswords.get(info.getUserName());
        if (pw == null || !pw.equals(info.getPassword())) {
            throw new SecurityException("User name or password is invalid.");
        }

        final Set groups = (Set)userGroups.get(info.getUserName());
        SecurityContext s = new SecurityContext(info.getUserName()) {
            public Set<?> getPrincipals() {
                return groups;
            }
        };

        context.setSecurityContext(s);
        securityContexts.add(s);
    }
    super.addConnection(context, info);
}
```

パスワード不一致により認証失敗
SecurityException をスロー

ActiveMQ-4.1.1
SimpleAuthenticationBroker.java

Apache ActiveMQ (AMQ-1272)

```
.....
connectionInfo.setResponseRequired(true);
connectionInfo.setUserName(login);
connectionInfo.setPassword(passcode);

sendToActiveMQ(connectionInfo, new ResponseHandler() {
    public void onResponse(ProtocolConverter converter, Response response)
        throws IOException {



        final SessionInfo sessionInfo = new SessionInfo(sessionId);
        sendToActiveMQ(sessionInfo, null);

        final ProducerInfo producerInfo = new ProducerInfo(producerId);
        sendToActiveMQ(producerInfo, new ResponseHandler() {
            public void onResponse(ProtocolConverter converter, Response response) throws
IOException {
                .....
                .....
```

認証失敗のとき例外がスローされ,
response にその情報がはいってくる.
しかし, なにもチェックしていない.

ActiveMQ-4.1.1
ProtocolConverter.java

Apache ActiveMQ (AMQ-1272)

ActiveMQ-5.0.0
ProtocolConverter.java

```
.....
connectionInfo.setResponseRequired(true);
connectionInfo.setUserName(login);
connectionInfo.setPassword(passcode);

sendToActiveMQ(connectionInfo, new ResponseHandler() {
    public void onResponse(ProtocolConverter converter, Response response) throws IOException {

        if (response.isException()) {
            // If the connection attempt fails we close the socket.
            Throwable exception = ((ExceptionResponse)response).getException();
            handleException(exception, command);
            getTransportFilter().onException(IOExceptionSupport.create(exception));
            return;
        }

        final SessionInfo sessionInfo = new SessionInfo(sessionId);
        sendToActiveMQ(sessionInfo, null);

        final ProducerInfo producerInfo = new ProducerInfo(producerId);
        sendToActiveMQ(producerInfo, new ResponseHandler() {
            public void onResponse(ProtocolConverter converter, Response response) throws IOException {
                ......
                ......
```

response に例外がはいっていた場合は接続中断の処理

まとめ

- 「最適に使用された場合、例外はプログラムの読みやすさ、信頼性、保守性を改善します。不適切に使用された場合、逆効果となります」

Effective Java 第2版, 第9章 例外

参考情報

- Java Tutorials (http://docs.oracle.com/javase/tutorial/index.html)
  - Essential Classes
    - Exceptions
    - Basic I/O

- Java I/O, 2nd Edition
  - Elliotte Rusty Harold, O'reilly