Javaセキュアコーディングセミナー東京
第4回
メソッドとセキュリティ

2012年12月16日(日)
JPCERTコーディネーションセンター
脆弱性解析チーム
熊谷 裕志
戸田 洋三

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

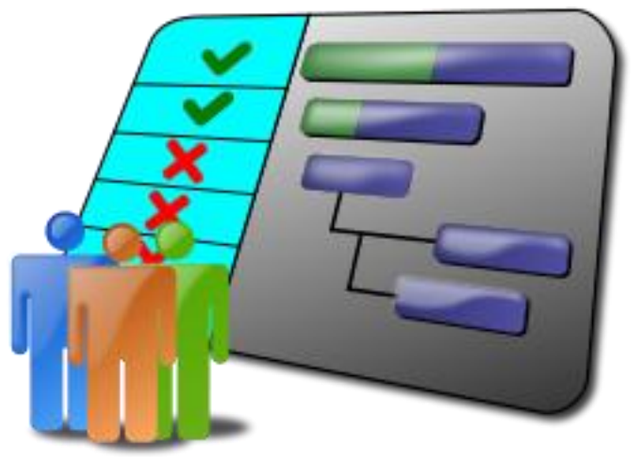
本セミナーについて

- 4回連続セミナーです
  - 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
    - メソッドとセキュリティ

- 開発環境持参

今日の時間割

- 講義 (13:30--15:00)
  - メソッド/セキュリティ (13:30--14:35)
  - ファイナライザ (14:45--15:15)
- ハンズオン (15:15--16:30)
  - 15:25--15:55
  - 15:55--16:30



メソッド

メソッドとは

クラスの操作は、結果を得るためにオブジェクトのデータに操作を指示するメソッド(method)により宣言されます。メソッドは、他のオブジェクトから隠蔽されている内部の実装の詳細にアクセスします。メソッドの中にデータを隠蔽することにより、他のオブジェクトからアクセスできないようにすることは、データのカプセル化(data encapsulation)の基本中の基本です。

プログラミング言語Java第4版、§1.8メソッドとパラメータ

メソッドとは

2つのメソッドが同じ名前と引数の型を保持している場合,
それらのメソッドは**同じシグネチャ**(*same argument types*)を保持することになる

Java言語仕様第3版、§8.4.2 メソッドのシグネチャ

メソッドとは

```
class Person {
  private int id;
  private String name;
  public Person(int i, String s){ id=i; name=s; }
  public int id(){ return id; }
  public String name(){ return name; }

  public static void main(String[] args){
    person p = new person(1,”taro”);
    System.out.println(p.name() + “: “ + p.id());
  }
}
```

- クラスPersonに、メソッドid()とname()とmain(String[])が定義されている。

メソッドオーバーロードを乱用しない

- Java言語ではメソッドのオーバーロードが可能
  - 2つのメソッドは、異なる数のパラメータか異なる型のパラメータを持っていれば、つまり異なるシグネチャであれば、同じ名前を持つことができます。メソッドの1つの名前が複数の意味を持つので、この機能はオーバーロード(*overloading*)と呼ばれます。
  - 1つのメソッドを呼び出す場合に、オーバーロードされているメソッドから最も一致するメソッドを探すために、コンパイラーは引数の型を使用します。

プログラミング言語Java第4版、§2.8メソッドのオーバーロード

メソッドオーバーロードを乱用しない

- オーバーロードを使った単純なサンプルコード

```
class overload {
  public void id(String s){ System.out.println("String"); }
  public void id(Integer i){ System.out.println("Integer"); }

  public static void main(String[] args) {
    overload o = new overload();
    o.id("choichoi");
    o.id(42);
  }
}
```

- 実行例

```
$ java overload
String
Integer
$
```

メソッドオーバーロードを乱用しない

- メソッドのオーバーロード
  - メソッド名が同じでも引数リストが異なれば異なるメソッド
  - シグネチャで区別される
  - 呼び出すメソッドはコンパイル時に決まる

- メソッドのオーバーライド
  - 呼び出すメソッドは実行時に決定される

メソッドオーバーロードを乱用しない

- メソッド id(int i)を追加

```
class Overload {
  public void id(String s){ System.out.println("String"); }
  public void id(Integer i){ System.out.println("Integer"); }
  public void id(int i){ System.out.println("int"); }

  public static void main(String[] args) {
    Overload o = new Overload();
    o.id("choichoi");
    o.id(42);
  }
}
```

追加されたメソッド

- 実行例

```
$ java Overload
String
int
$
```

呼び出すメソッドが変わってしまった！

メソッドオーバーロードを乱用しない

- メソッドのオーバーロードを乱用すると
  - 動作が分かりにくく誤解を招く
  - デバッグしにくい
  - コードの保守が困難になる

メソッドオーバーロードを乱用しない

違反コード

```
public class Overloader {
  private static String display(ArrayList<Integer> arrlist) {
    return "ArrayList";
  }
  private static String display(LinkedList<String> llist) {
    return "LinkedList";
  }
  private static String display(List<?> list) {
    return "List is not recognized";
  }
  public static void main(String[] args) {
    List<?>[] invokeAll = new List<?>[] {
      new ArrayList<Integer>(),
      new LinkedList<String>(),
      new Vector<Integer>() };
    for (List<?> i : invokeAll) {
      System.out.println(display(i));
    }
  }
}
```

display()メソッドがオーバーロードされている

コンパイル時の型はList

3つ全てについて”List is not recognized”と出力される

メソッドオーバーロードを乱用しない

適合コード

```
public class Overloader {
  private static String display(List<?> l) {
    return (l instanceof ArrayList ? "Arraylist" :
      (l instanceof LinkedList ? "LinkedList" :
      "List is not recognized"));
  }

  public static void main(String[] args) {
    List<?>[] invokeAll = new List<?>[] {
      new ArrayList<Integer>(),
      new LinkedList<String>(),
      new Vector<Integer>() };

    for (List<?> i : invokeAll) {
      System.out.println(display(i));
    }
  }
}
```

実行時に引数の型を識別するには、instanceof()を使う

メソッドオーバーロードを乱用しない

違反コード

```
// Effective Java, 項目 41
public class SetList {
  public static void main(String[] args){
    Set<Integer> set = new TreeSet<Integer>();
    List<Integer> list = new ArrayList<Integer>();

    for (int i=-3; i<3; i++){
      set.add(i);
      list.add(i);
    }
    for (int i=0; i<3; i++){
      set.remove(i);
      list.remove(i);
    }
    System.out.println(set + " " + list);
  }
}
```

```
boolean TreeSet<Integer>.remove(Object o)
Integer ArrayList<Integer>.remove(int index)
boolean ArrayList<Integer>.remove(Object o)
```

ArrayListには動作の異なる2つのremove()メソッドが提供されている

メソッドオーバーロードを乱用しない

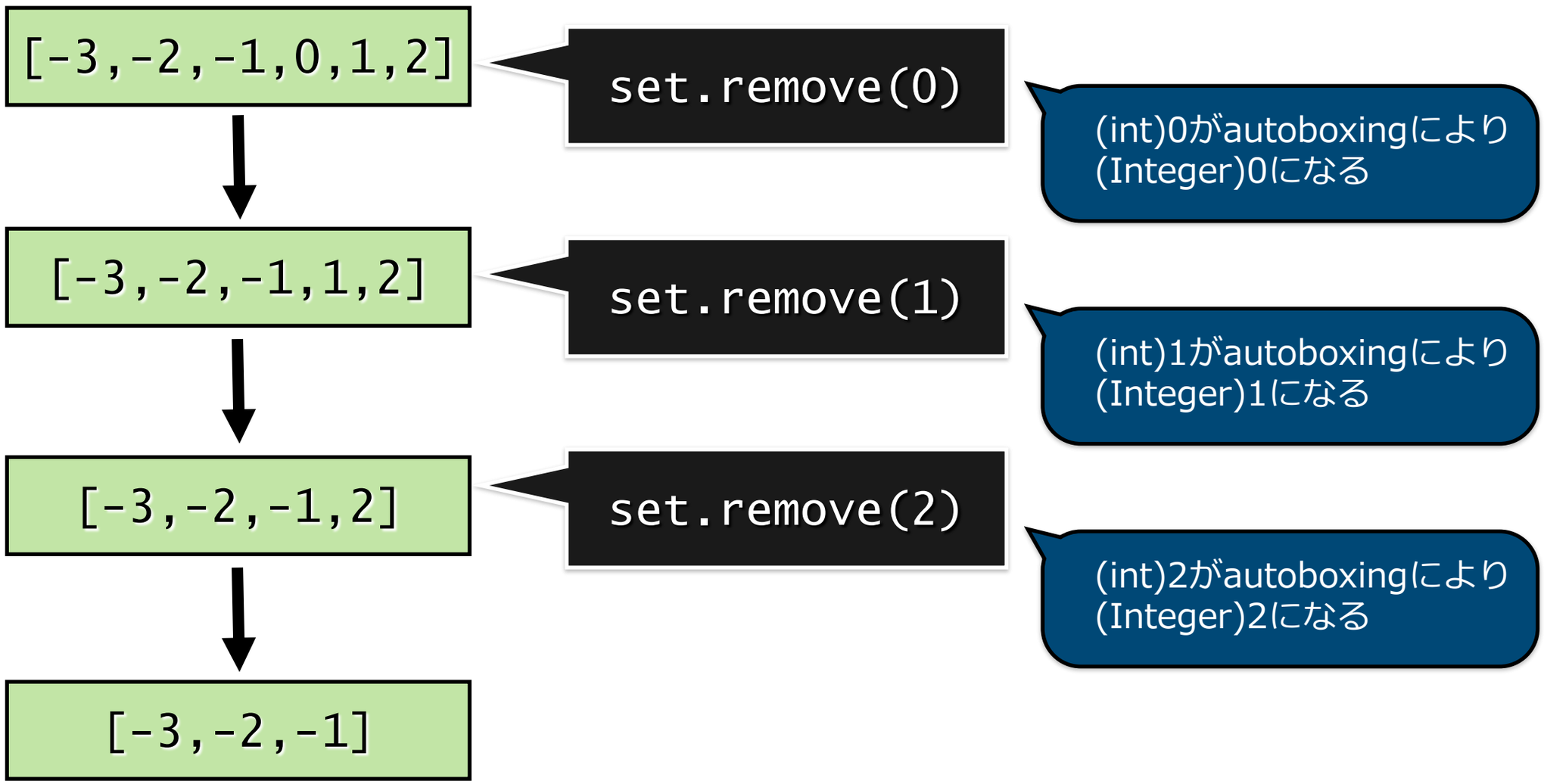
- 実行例

```
$ java SetList
[-3, -2, -1] [-2, 0, 2]
$
```

[-3,-2,-1] [-3,-2,-1]にならないのはなぜ？

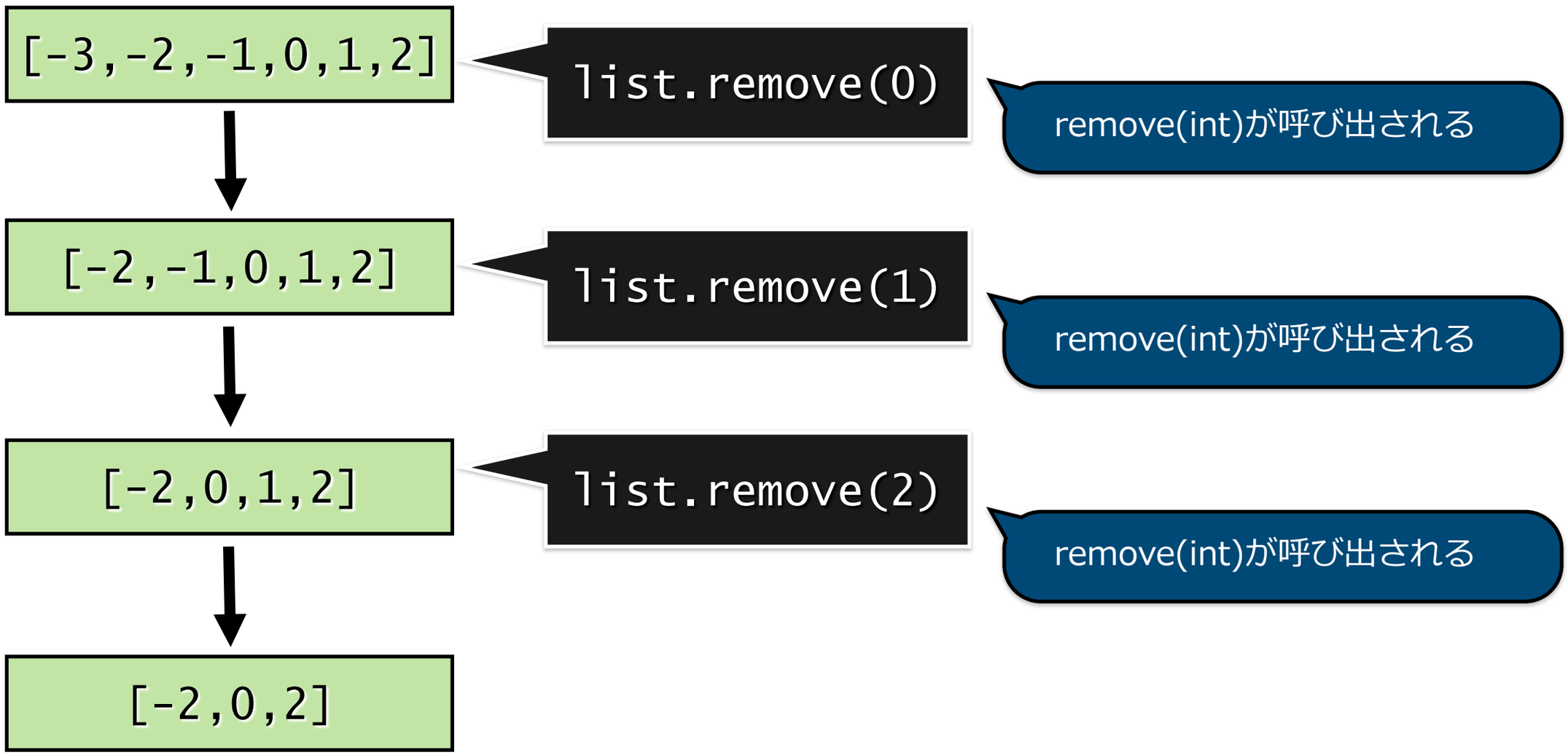
メソッドオーバーロードを乱用しない

- set.remove(i)の動作



メソッドオーバーロードを乱用しない

- list.remove(i)の動作



メソッドオーバーロードを乱用しない

適合コード

```
// Effective Java, 項目 41
public class SetList {
  public static void main(String[] args){
    Set<Integer> set = new TreeSet<Integer>();
    List<Integer> list = new ArrayList<Integer>();

    for (int i=-3; i<3; i++){
      set.add(i);
      list.add(i);
    }
    for (int i=0; i<3; i++){
      set.remove(i);
      list.remove((Integer)i); // あるいは (Integer.valueOf(i))
    }
    System.out.println(set + " " + list);
  }
}
```

まとめ

- オーバーロードを乱用するとコードの可読性が下がり、メソッドを誤用する危険が増す
- なるべくオーバーロードを避け、異なる名前のメソッドを実装するほうが安全

> メソッドをオーバーロードできるからと言って、オーバーロードすべきではありません。一般的には、同じ数のパラメータを持つ複数のシグニチャでメソッドをオーバーロードすることは控えるべきです。

Effective Java，項目41

セキュリティ

privateのフィールドやメソッドにアクセス?

```
public class Example {
  private int i = 3;
  private int j = 4;

  private void zeroI() {
    this.i = 0;
  }
}
```

```
Example e = new Example();
System.out.println("" + e.i);
e.i = 10;
e.zeroI();
```

他のクラスからprivateのiやj、zeroI()にアクセスできる？

リフレクション

クラスとオブジェクトに関するリフレクト情報を取得するクラスおよびインタフェースを提供します。リフレクションを使用すると、ロードされたクラスのフィールド、メソッド、およびコンストラクタに関する情報へのプログラム化されたアクセスを実行し、リフレクトされたフィールド、メソッド、およびコンストラクタを使ってセキュリティーの制約内で基本となる対応部分を操作できます。

必要な ReflectPermission が利用できる場合、AccessibleObject は、アクセスチェックの抑制を可能にします。

http://docs.oracle.com/javase/jp/6/api/java/lang/reflect/package-summary.html

リフレクションを使うと

```
try {
  Class<Example> c = Example.class;
  Example example = new Example();

  Field field = c.getDeclaredField("i");
  field.setAccessible(true);
  System.out.println("" + field.get(example));

} catch (Exception ex) {
  ex.printStackTrace(System.out);
}
```

privateのフィールドに
アクセスできる

setAccessibleを有効にするとリフレクションを使って通常アクセスできないところにアクセスできる

リフレクションを使うと

```
try {
  Class<Example> c = Example.class;
  Example example = new Example();

  Method method = c.getDeclaredMethod("zeroI");
  method.setAccessible(true);
  Object ret = method.invoke(example);
  Field field = c.getDeclaredField("i");
  field.setAccessible(true);

  System.out.println("" + field.get(example));
} catch (Exception ex) {
  ex.printStackTrace(System.out);
}
```

privateのメソッドにもアクセスできる

セキュリティマネージャで制限することができる

セキュリティマネージャ

セキュリティーマネージャーとは、アプリケーションがセキュリティーポリシーを実装できるクラスです。

セキュリティーマネージャーを使えば、セキュリティーを損なう恐れのある操作を実行する前に、操作が何であるかということと、セキュリティーコンテキスト内でその操作の実行が許可されているかどうかがアプリケーションから判断できます。

アプリケーションは、そのような操作を禁止したり許可したりすることができます。

http://docs.oracle.com/javase/jp/6/api/java/lang/SecurityManager.html

Java：セキュリティモデル

- サンドボックスによって保護されている
  - —セキュリティポリシーにもとづいて操作を許可する
    - 許可されていない操作をすると例外が発生

例えば：Javaアプレットは

- ローカルのリソースにはアクセス出来ない
- ダウンロード元のサーバとのみ通信可

セキュリティマネージャを使う

違反コード

```
import java.util.Hashtable;

class SensitiveHash {
  Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

  public void removeEntry(Object key) {
    ht.remove(key);
  }

  // 中略
}
```

- removeEntry()がpublic
- 悪意ある攻撃者が自由に呼び出せる

セキュリティマネージャを使う

適合コード

```
import java.util.Hashtable;

class SensitiveHash {
  private Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

  public final void removeEntry(Object key) {
    check("removeKeyPermission");
    ht.remove(key);
  }

  private void check(String directive) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
      sm.checkSecurityAccess(directive);
    }
  }

  // 中略
}
```

セキュリティポリシーファイル

指定した署名で署名されているクラスに許可を与える

指定したディレクトリからロードされたクラスに許可を与える

```
grant SignedBy “hogehoge” codeBase "file:${user.dir}/sensitive" {
  permission java.security.SecurityPermission "removeKeyPermission";
};
```

http://docs.oracle.com/javase/jp/6/technotes/guides/security/spec/security-spec.doc3.html#20128

policytool

付属のPolicy Toolユーティリティを使用して、ポリシーファイルを作成することもできる。

http://docs.oracle.com/javase/jp/6/technotes/guides/security/PolicyGuide.html

実行してみる

```
$ java -Djava.security.manager  -Djava.security.policy=my.policy -
classpath “./sensitive:./” UseHash
```

セキュリティマネージャを使う

適合コード

```
import java.util.Hashtable;
import java.security.AccessController;
import java.security.SecurityPermission;

class SensitiveHash {
  private Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

  public final void removeEntry(Object key) {
    check("removeKeyPermission");
    ht.remove(key);
  }

  private void check(String directive) {
    SecurityPermission sp = new SecurityPermission(directive);
    AccessController.checkPermission(sp);
  }

  // 中略
}
```

AccessController#checkPermissionを呼び出すこの方法が推奨されている

ファイナライザ

finalize()メソッド

- クラス Object には, finalize と呼ばれる protected メソッドが用意されており, 他のクラスからこのメソッドをオーバーライドすることができる。あるオブジェクトに対して起動可能な特定の finalize 定義は, そのオブジェクトのファイナライザ(finalizer)と呼ばれる。

Java言語仕様第3版、§12.6 クラス・インスタンスのファイナライズ

finalize()メソッド

```
public class Object {
   ......
   ......
   protected void finalize() throws Throwable
   ......
   ......
}
```

> このオブジェクトへの参照はもうないとガベージコレクションによって判断されたときに、ガベージコレクタによって呼び出されます。サブクラスは `finalize` メソッドをオーバーライドして、システムリソースを破棄したり、その他のクリーンアップを行ったりすることができます。

JavaSE6 API仕様、クラスObject, finalizeメソッドの説明

finalize()メソッドを使わない

- finalizeメソッドの利用に関しては数々の問題が存在するため、その利用は例外的場合に限るべき
- 実行に関して無保証
  - 並行実行の可能性
  - 例外の扱い
  - リソース一般の後処理には使えない

実行されるかどうか無保証

- finalizeメソッドは実行されるとは限らない
  - メモリに余裕があればGCは働かない
    - finalize メソッドも実行されない

- オブジェクトの状態をファイルに保存するなどの終了処理をfinalizeメソッドで実行してはいけない
- 実行タイミングが重要な処理をfinalizeメソッドで実行してはいけない

実行順序や並行実行の可能性, 例外の扱い

- finalizeメソッドの実行順序

> プログラミング言語 Java では, ファイナライズ・メソッドの呼び出し順序を規定していない。このためファイナライザはさまざまな順序で呼び出され, 並列的に呼び出される可能性もある。

Java言語仕様第3版、§12.6.2 ファイナライザの起動は順序付けられていない

- 複数の(オブジェクトの)**finalize**メソッドの実行順序は指定できない
- 複数の(オブジェクトの)**finalize**メソッドが並列に実行されるかも
- **finalize**メソッドのなかからスローされた例外は無視される
- **finalize** メソッドの実行自体は中断

リソース一般の後処理には使えない

- **finalize**メソッドとリソース管理
  - **finalize**メソッドの実行はメモリの使用状況に依存
  - メモリ以外のリソースの空き状況は関係しない

- 結果：空きメモリが潤沢でも他のリソースが枯渇する可能性
  - GCが実行されない→**finalize**メソッドも実行されない
  - DoS攻撃の危険性

注意!!
finalizeメソッドは C++ の destructor とは違います

リソース一般の後処理には使えない

- リソース一般の後処理には Closeable インタフェースと try-with-resources 構文を活用すべき

  - リソース... 使用開始時にオープン/使用終了時にクローズするもの
    - ストリームのclose メソッド, Timerのcancel メソッド, Graphics の dispose メソッドなど
  - Closeable インタフェースを実装... close() メソッドを持っている

try-with-resources て東京セミナ part3 でちらっと紹介したよね!

finalize()を使う場合の注意点

- 時間のかかる処理を行うべきではない
- 明示的に行うべきクローズ処理の最終チェック手段として使う
- **finalize()**メソッドをオーバライドする場合，親クラスの**finalize()**呼び出しを忘れずに行う

finalizer guardian

- サブクラスで **finalize()** メソッドをオーバライドしている状況で, 親クラスの **finalize()** 呼び出しを保証するための手法

```
public class Foo {

  private final Object finalizerGuardian
    = new Object() {
        @Override protected void finalize()
                                    throws Throwable {
          ..... 外側のオブジェクトをファイナライズする .....
        }
      };

  .....
}
```

ファイナライザに関連するコーディングルール

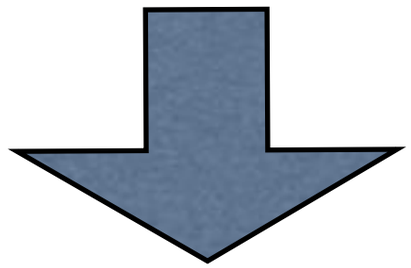
- FIO04-J. 不要になったらリソースを解放する
- FIO14-J. プログラムの終了時には適切なクリーンアップを行う
- MET12-J. ファイナライザは使わない

攻撃手法の紹介：

ファイナライザー攻撃

概要

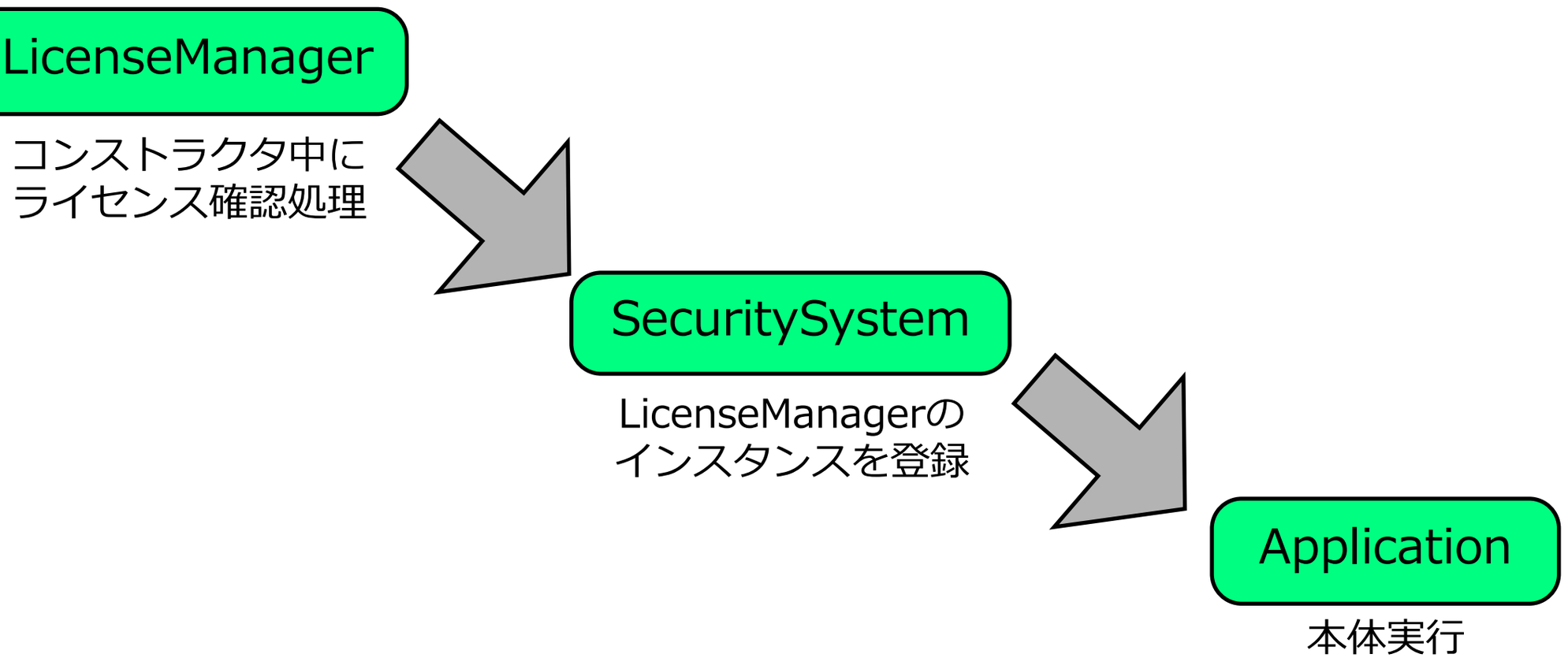
- ライセンス認証を行うサンプルアプリケーションに対し、Java コードを追加するだけで認証回避を行う攻撃手法



  - **finalize()**メソッドを悪用するため、「ファイナライザー攻撃」と呼ばれる手法

サンプルアプリケーションの構成

- アプリ本体: Application クラス
- アプリ本体の実行に先立って、ライセンス認証を行う
  - LicenseManagerクラス: ライセンス情報の確認
  - SecuritySystemクラス: 認証完了したことを記録



サンプルアプリケーション(1/2)

```
public class LicenseManager {
  public LicenseManager() {
    if (!licenseValidation()) {
      throw new SecurityException("License Invalid!");
    }
  }
  private boolean licenseValidation() {
    // ライセンスファイルをリードしてチェックし、ライセンスが正当ならtrueを返す
    return false;
  }
}

     public class SecuritySystem {
  private static LicenseManager licenseManager = null;
  public static void register(LicenseManager lm) {
    // licenseManagerが初期化されていない場合のみ登録
    if (licenseManager == null) {
      if (lm == null) {
        System.out.println("License Manager invalid!");
        System.exit(1);
      }
      licenseManager = lm;
    }
  }
}
```

ここでは必ず認証失敗するようなコードにしている

Heinz M. Kabutz. *Exceptional Constructors - Ressurecting the dead.* Java Specialists' Newsletter. 2001

サンプルアプリケーション(2/2)

```
public class Application {
  public static void main(String[] args) {
    LicenseManager lm;
    try {
      lm = new LicenseManager();
    } catch(SecurityException ex) { lm = null; }

    SecuritySystem.register(lm);
    System.out.println("Now let's get things started");
  }
}
```

- 正しいライセンス情報を持っていないと……
  - ⇒LicenseManagerのインスタンス生成時に例外発生
- LicenseManager のインスタンスを登録しないと……
  - ⇒SecuritySystem.register() で例外発生

サンプルアプリケーション実行例

```
% ls *.java
Application.java    LicenseManager.java
SecuritySystem.java
% javac *.java
% java Application
License Manager invalid!
%
```

たしかに認証失敗している

サンプルアプリケーションを攻撃する

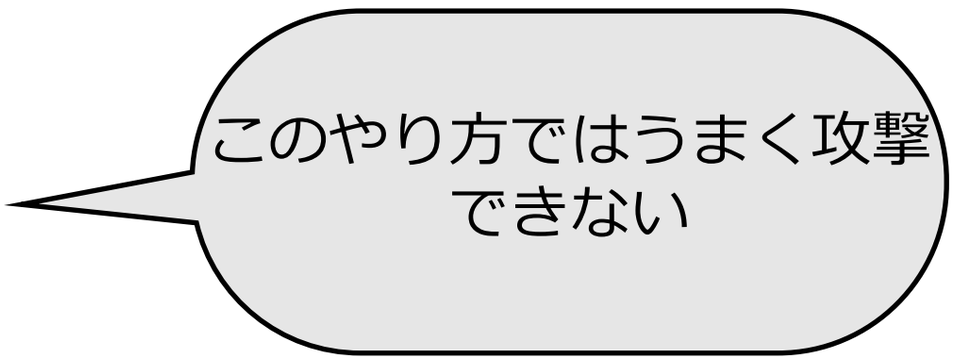
- 攻撃目的
  - **LicenseManager** のセキュリティチェックを回避し、**Application.main()** を実行する
- 前提条件
  - これらのクラスはすべて変更できないものとする

- 攻撃方針
  - **LicenseManager**のサブクラスを作成し、攻撃者のアプリ(後述の**AttackerApp**)に脆弱なアプリ**Application**を読み込む
  - 問題
    - **LicenseManager**のサブクラスを作っても、サブクラスでは、親クラスでスローされる例外をキャッチできない…

親クラスでスローされる例外を悪用できれば…

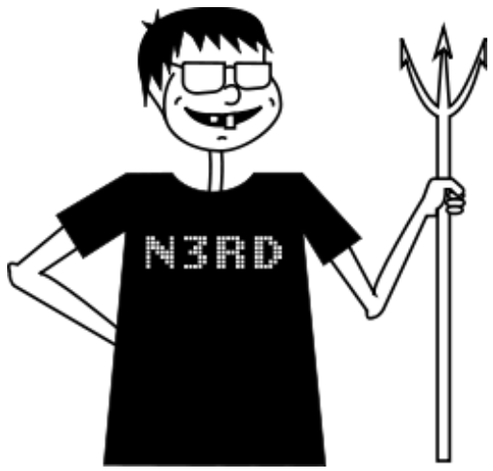
```
public class MyApplication {
  public static void main(String[] args) {
    MyLicenseManager lm;
    try {
      lm = new MyLicenseManager();
    } catch(SecurityException ex) {
      lm = null;
    }
    SecuritySystem.register(lm);
    // Applicationのメインメソッドを呼ぶ
    Application.main(args);
  }
}

public class MyLicenseManager extends LicenseManager {
  public MyLicenseManager() {
    System.out.println("Created MyLicenseManager");
  }
}
```

- MyApplication を実行すると…
- License Manager invalid!



finalize()メソッドの悪用

- **SecuritySystem** に **LicenseManager** のインスタンスを登録できれば勝ち
  - しかもサンプルアプリケーションでは**LicenseManager**のインスタンスの中身はチェックしていない
- **LicenseManager**のインスタンス欲しい
  - ⇒ライセンス情報を持っていない場合、コンストラクタ実行中に例外が発生するので、生成途中で捨てられている
- **finalize()**を使えば、GCされる直前に拾うことができる!



ファイナライザー攻撃を行うコード

攻撃コード

```
public class LicenseManagerInterceptor extends LicenseManager {
  private static LicenseManagerInterceptor instance = null;
  public static LicenseManagerInterceptor make() {
    try {
      new LicenseManagerInterceptor();
    } catch(Exception ex) {} // 例外を無視
    try {
      synchronized(LicenseManagerInterceptor.class) {
        while (instance == null) {
          System.gc();
          LicenseManagerInterceptor.class.wait(100);
        }
      }
    } catch(InterruptedException ex) {
      return null;
    }
    return instance;
  }
  public void finalize() {
    System.out.println("In finalize of " + this);
    synchronized(LicenseManagerInterceptor.class) {
      instance = this;
      LicenseManagerInterceptor.class.notify();
    }
  }
  public LicenseManagerInterceptor() {
    System.out.println("Created LicenseManagerInterceptor");
  }
}
```

finalize()メソッドを追加

ファイナライザー攻撃を行うコード

攻撃コード

```
public class AttackerApp {
  public static void main(String[] args) {
    LicenseManagerInterceptor lm = LicenseManagerInterceptor.make();
    SecuritySystem.register(lm);
    // now we call the other application
    Application.main(args);
  }
}
```

LicenseManagerInterceptor.make()の返り値は、
GC直前に拾い上げたLicenseManagerInterceptor
のインスタンス

ファイナライザ攻撃の流れ

LicenseManagerInterceptor

親クラス**LicenseManager**の
ライセンス確認処理で例外発生

例外を捕捉し、GCが
finalize()メソッドを起動するのを待つ

やがて、GCによって
finalize()起動

初期化途中だった**LicenseManager**
のインスタンスを取得

SecuritySystem

LicenseManagerの
インスタンスを登録

Application

本体実行

SecuritySystemにはすでに
LicenseManagerのインスタンスが
登録されているため、ライセンス認
証は完了したと思わされる

攻撃コード実行例

```
% ls
Application.class    LicenseManager.class    SecuritySystem.class
AttackerApp.java    LicenseManagerInterceptor.java
% javac *.java
% java AttakerApp
In finalize of LicenseManagerInterceptor@7dcb3cd
Now let’s get things started
%
```

ファイナライザ攻撃対策

- **finalize()**メソッドを上書きされないように定義
- 重要なインスタンスは、初期化の完了を必ず確認
- サブクラス化による悪用を防ぐため、クラスを**final**宣言