Javaのセキュリティに関するコーディング規約を、ご指定のカテゴリに沿ってまとめました。各項目で、セキュリティ上のポイント、悪いコード例、そして推奨される良いコード例を示します。これらの規約は、JPCERT/CCが参照しているCERT Java Secure Coding Standardの原則に基づいています。

## 1. 入力値検証（IDS）：すべての外部入力を検証・正規化・無害化する 🛡️

### 解説

外部から受け取るすべての入力値（ユーザー入力、ファイル、ネットワーク経由のデータなど）は信頼できないものとして扱う必要があります。これらの値を検証し、期待される形式や範囲に合致するか確認し、必要に応じて正規化（例：大文字・小文字統一、エンコーディング統一）や無害化（例：HTMLエスケープ、SQLエスケープ）を施すことで、インジェクション攻撃（SQLインジェクション、クロスサイトスクリプティングなど）や予期せぬ動作を防ぎます。

### 悪い例

Java

// SQLインジェクションの脆弱性がある例  
String userName = request.getParameter("userName");  
String query = "SELECT \* FROM users WHERE name = '" + userName + "'"; // userNameを直接結合  
Statement stmt = connection.createStatement();  
ResultSet rs = stmt.executeQuery(query);

### 良い例

Java

// プリペアドステートメントを使用しSQLインジェクションを対策  
String userName = request.getParameter("userName");  
String query = "SELECT \* FROM users WHERE name = ?";  
PreparedStatement pstmt = connection.prepareStatement(query);  
pstmt.setString(1, userName); // プレースホルダに値を設定  
ResultSet rs = pstmt.executeQuery();  
  
// 入力値の形式を正規表現で検証する例  
String email = request.getParameter("email");  
if (!email.matches("^[A-Za-z0-9+\_.-]+@(.+)$")) {  
 throw new IllegalArgumentException("無効なメールアドレス形式です。");  
}

## 2. 宣言と初期化（DCL）：初期化の循環や曖昧な final フィールドを避ける 🧱

### 解説

変数は使用前に必ず初期化し、特にfinalフィールドはコンストラクタ完了までに初期化されることを保証する必要があります。初期化の循環（例：クラスAの初期化がクラスBを必要とし、クラスBの初期化がクラスAを必要とする）はデッドロックや未初期化状態でのアクセスを引き起こす可能性があります。finalフィールドがコンストラクタで初期化されない場合、コンパイラエラーとなりますが、複雑な初期化ロジックでは意図しない動作を招くこともあります。

### 悪い例

Java

// finalフィールドが初期化されない可能性がある (実際にはコンパイルエラー)  
// public class BadDCL {  
// private final int value;  
// public BadDCL(boolean initialize) {  
// if (initialize) {  
// this.value = 10;  
// }  
// // initializeがfalseの場合、valueが初期化されない  
// }  
// }  
  
// 初期化の循環（概念例）  
// class A {  
// private static B instanceB = new B();  
// public A() { System.out.println("A initialized"); }  
// }  
// class B {  
// private static A instanceA = new A();  
// public B() { System.out.println("B initialized"); }  
// }

### 良い例

Java

// finalフィールドは宣言時またはコンストラクタで確実に初期化する  
public class GoodDCL {  
 private final int value;  
 private final String name;  
  
 public GoodDCL(int value, String name) {  
 if (name == null) {  
 throw new IllegalArgumentException("Name cannot be null");  
 }  
 this.value = value;  
 this.name = name; // nullチェックなどを行った上で初期化  
 }  
  
 public GoodDCL(String name) {  
 this(0, name); // デフォルト値を設定して他のコンストラクタを呼び出す  
 }  
}

## 3. 式と数値演算（EXP, NUM）：戻り値のチェック、オーバーフロー／ビット演算ミスを防ぐ 🔢

### 解説

数値演算では、オーバーフローやアンダーフローが発生し、予期しない結果や脆弱性を生む可能性があります。特に符号なしの値を扱う場合や、異なるサイズの数値型間で変換を行う際には注意が必要です。メソッドの戻り値（特にエラーを示す値）は適切にチェックし、無視しないようにします。ビット演算の誤りは、フラグの判定ミスやデータの破損につながります。

### 悪い例

Java

// 整数オーバーフローを考慮しない  
int transactionAmount = 100000;  
int currentBalance = Integer.MAX\_VALUE - 50000;  
int newBalance = currentBalance + transactionAmount; // オーバーフローの可能性  
  
// メソッドの戻り値を無視する (例: File.delete()など)  
File file = new File("temp.txt");  
file.delete(); // 削除が成功したか確認していない

### 良い例

Java

// Java 8以降のMath.addExactでオーバーフローを検知  
int transactionAmount = 100000;  
int currentBalance = Integer.MAX\_VALUE - 50000;  
try {  
 int newBalance = Math.addExact(currentBalance, transactionAmount);  
 System.out.println("New balance: " + newBalance);  
} catch (ArithmeticException e) {  
 System.err.println("Error: Integer overflow detected during balance calculation.");  
 // 適切なエラー処理  
}  
  
// メソッドの戻り値を確認する  
File file = new File("temp.txt");  
if (!file.delete()) {  
 System.err.println("ファイルの削除に失敗しました: " + file.getName());  
}  
  
// long型を使ってオーバーフローのリスクを軽減  
long longBalance = (long)currentBalance + transactionAmount;  
if (longBalance > Integer.MAX\_VALUE || longBalance < Integer.MIN\_VALUE) {  
 System.err.println("Error: Calculation result exceeds integer limits.");  
 // 適切なエラー処理  
} else {  
 int newBalance = (int) longBalance;  
 System.out.println("New balance: " + newBalance);  
}

## 4. 文字列操作（STR）：大量連結を避け、正規表現や文字列操作時のリスクを回避する 📜

### 解説

Stringオブジェクトは不変（イミュータブル）であるため、ループ内で+演算子を使って文字列を繰り返し連結すると、その都度新しいStringオブジェクトが生成され、パフォーマンスが著しく低下します。このような場合はStringBuilder（スレッドセーフ不要時）やStringBuffer（スレッドセーフ必要時）を使用します。また、複雑すぎる正規表現や、特定の入力に対して極端に処理時間がかかるような正規表現（ReDoS: Regular Expression Denial of Service）の使用は避けるべきです。文字列の比較はequals()メソッドを使用し、==演算子は参照の比較であることに注意します。

### 悪い例

Java

// ループ内でのString連結  
String[] words = {"Hello", " ", "World", "!"};  
String message = "";  
for (String word : words) {  
 message += word; // 効率が悪い  
}  
  
// 複雑でバックトラックが多い可能性のある正規表現 (例)  
// String regex = "(a+)+b"; // 入力によっては処理時間が指数関数的に増加する可能性

### 良い例

Java

// StringBuilderを使用した効率的な文字列連結  
String[] words = {"Hello", " ", "World", "!"};  
StringBuilder sb = new StringBuilder();  
for (String word : words) {  
 sb.append(word);  
}  
String message = sb.toString();  
  
// 正規表現はシンプルで効率的なものを心がける  
// ReDoSのリスクが低いパターンを使用する。必要であればライブラリやツールで検証する。  
String regex = "^[a-zA-Z0-9]+$"; // シンプルな英数字チェック  
  
// 文字列比較は equals() を使用  
String s1 = new String("test");  
String s2 = new String("test");  
// if (s1 == s2) { /\* これはfalseになる \*/ }  
if (s1.equals(s2)) { // 正しい比較  
 System.out.println("Strings are equal.");  
}

## 5. オブジェクト指向（OBJ）：コンストラクタ内のオーバーライド呼び出しを避け、イミュータブル設計を推奨 🧩

### 解説

スーパークラスのコンストラクタ内でオーバーライド可能なメソッドを呼び出すと、サブクラスの初期化が完了する前にそのメソッドが実行される可能性があります。このとき、サブクラスのフィールドが未初期化状態であるため、NullPointerExceptionや意図しない動作を引き起こすことがあります。これを避けるためには、コンストラクタからはfinalメソッド、privateメソッド、またはstaticメソッドのみを呼び出すようにします。

また、状態を変更できないイミュータブル（不変）オブジェクトは、スレッドセーフティの確保や状態管理の簡素化に寄与するため、可能な限り採用することが推奨されます。

### 悪い例

Java

// スーパークラスのコンストラクタでオーバーライド可能なメソッドを呼び出す  
class SuperClass {  
 SuperClass() {  
 doSomething(); // 悪い例: サブクラスでオーバーライドされている場合、サブクラスの初期化前に呼び出される  
 }  
 void doSomething() {  
 System.out.println("SuperClass doSomething");  
 }  
}  
  
class SubClass extends SuperClass {  
 private String name = "Sub";  
 SubClass() {  
 super();  
 System.out.println("SubClass constructor: name = " + name);  
 }  
 @Override  
 void doSomething() {  
 // このメソッドがSuperClassのコンストラクタから呼び出される時、'name'はまだnull  
 System.out.println("SubClass doSomething: name length = " + (name != null ? name.length() : "null"));  
 }  
}  
// new SubClass(); を実行すると、nameがnullの状態でSubClassのdoSomethingが呼ばれる

### 良い例

Java

// コンストラクタ内ではfinalメソッドかprivateメソッドを呼び出す  
class GoodSuperClass {  
 GoodSuperClass() {  
 // doSomething(); // オーバーライド可能なメソッド呼び出しを避ける  
 performInitialization(); // privateメソッドならOK  
 }  
  
 // オーバーライドさせたくない場合はfinalにする  
 public final void performAction() {  
 System.out.println("GoodSuperClass performAction (final)");  
 }  
  
 private void performInitialization() {  
 System.out.println("GoodSuperClass performInitialization (private)");  
 }  
  
 // サブクラスでカスタマイズさせたい場合は、初期化後に呼び出されるメソッドを用意する  
 protected void postConstructAction() {}  
}  
  
class GoodSubClass extends GoodSuperClass {  
 private String name = "GoodSub";  
 GoodSubClass() {  
 super();  
 // ここでnameは初期化済み  
 System.out.println("GoodSubClass constructor: name = " + name);  
 postConstructAction();  
 }  
  
 @Override  
 protected void postConstructAction() {  
 System.out.println("GoodSubClass postConstructAction: name = " + name);  
 }  
}  
  
// イミュータブルクラスの例  
final class ImmutablePoint { // クラスをfinalにする  
 private final double x; // フィールドをfinalにする  
 private final double y;  
  
 public ImmutablePoint(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public double getX() { return x; } // getterのみ提供  
 public double getY() { return y; }  
  
 // 必要に応じて、状態を変更した新しいインスタンスを返すメソッドを用意  
 public ImmutablePoint move(double dx, double dy) {  
 return new ImmutablePoint(this.x + dx, this.y + dy);  
 }  
}

## 6. メソッド（MET）：リフレクションやセキュリティマネージャの乱用を避け、責務ごとに明確に設計 🛠️

### 解説

リフレクションAPI (java.lang.reflect) は強力な機能ですが、カプセル化を破壊し、アクセス制御（privateなど）をバイパスする可能性があります。セキュリティ上の制約を回避するためにリフレクションを乱用すると、コードの安全性が低下し、保守も困難になります。メソッドは単一責任の原則に従い、一つの明確な責務を持つように設計することで、可読性、保守性、テスト容易性が向上します。セキュリティマネージャはJavaのセキュリティ機構の重要な部分ですが、その設定やポリシーの変更は慎重に行い、不必要に権限を緩めないようにします。

### 悪い例

Java

import java.lang.reflect.Field;  
  
class SensitiveData {  
 private String secret = " খুবই গোপন (Very Secret)"; // 機密データ  
 public SensitiveData() {}  
}  
  
public class BadMethodAccess {  
 public static void main(String[] args) throws Exception {  
 SensitiveData data = new SensitiveData();  
 Field secretField = SensitiveData.class.getDeclaredField("secret");  
 secretField.setAccessible(true); // privateフィールドへのアクセスを強制  
 System.out.println("Stolen secret: " + secretField.get(data));  
 }  
}

### 良い例

Java

// リフレクションの不適切な使用を避ける  
class SecureData {  
 private String secret = " খুবই গোপন (Very Secret)";  
 private final String accessKey;  
  
 public SecureData(String key) {  
 this.accessKey = key; // アクセス制御のためのキー  
 }  
  
 public String getSecret(String providedKey) {  
 if (this.accessKey.equals(providedKey)) {  
 return secret;  
 }  
 throw new SecurityException("アクセス権がありません。");  
 }  
}  
  
public class GoodMethodAccess {  
 public static void main(String[] args) {  
 SecureData data = new SecureData("mySecretKey123");  
 try {  
 // 正しいキーでアクセス  
 System.out.println("Secret: " + data.getSecret("mySecretKey123"));  
 // 間違ったキーでアクセス  
 System.out.println("Secret: " + data.getSecret("wrongKey"));  
 } catch (SecurityException e) {  
 System.err.println(e.getMessage());  
 }  
 // リフレクションに頼らず、明確なインターフェースを通じてデータにアクセスする  
 }  
}

## 7. 例外処理（ERR）：機密情報を露出しない、安全な例外ログと適切な再スローを行う ⚠️

### 解説

例外処理はプログラムの堅牢性を高めるために不可欠ですが、不適切な処理はセキュリティリスクを生む可能性があります。特に、例外メッセージやスタックトレースにシステムの内部情報やユーザーの機密情報（パスワード、個人情報など）が含まれる場合、これらがログやユーザーへのエラー表示を通じて漏洩しないように注意が必要です。例外をキャッチした際は、空のcatchブロックで握りつぶさず、適切にログを記録し、必要であればより抽象的な例外として再スローするか、適切に処理を継続または終了させます。

### 悪い例

Java

// 機密情報を含む可能性のある例外情報をそのままユーザーに表示／ログ出力  
public class BadExceptionHandling {  
 public void processUserData(String username, String password) {  
 try {  
 // 何らかの処理... DB接続エラーなどが発生したとする  
 if (password.length() < 8) { // 単純な例  
 throw new IllegalArgumentException("パスワード '" + password + "' は短すぎます。ユーザー: " + username);  
 }  
 // ...  
 } catch (Exception e) {  
 // 悪い例1: 詳細なエラーメッセージをそのまま外部に出力  
 System.err.println("処理エラー: " + e.getMessage()); // passwordが含まれる可能性  
 // 悪い例2: スタックトレースを無条件に出力 (内部情報が漏れる可能性)  
 e.printStackTrace();  
 }  
 }  
}  
  
// 空のcatchブロック  
public class EmptyCatch {  
 public void readFile(String fileName) {  
 try {  
 // ファイル読み込み処理  
 } catch (java.io.IOException e) {  
 // 何もしない。エラーが握りつぶされ、問題が検知できない。  
 }  
 }  
}

### 良い例

Java

import org.slf4j.Logger; // 例としてSLF4Jを使用  
import org.slf4j.LoggerFactory;  
  
public class GoodExceptionHandling {  
 private static final Logger logger = LoggerFactory.getLogger(GoodExceptionHandling.class);  
  
 public void processUserData(String username, String password) {  
 try {  
 // 何らかの処理...  
 if (password == null || password.length() < 8) { // password自体はログに出さない  
 throw new IllegalArgumentException("無効なパスワード形式です。");  
 }  
 // ... DB処理などで SQLException が発生したとする  
 // pretendDbOperation(username, password);  
 } catch (IllegalArgumentException e) {  
 // ユーザーには一般的なメッセージを表示  
 System.err.println("入力値エラー: " + e.getMessage());  
 // ログにはユーザー名を記録するが、パスワードなどの機密情報は記録しない  
 logger.warn("ユーザー '{}' の処理中に引数エラーが発生しました。", username, e);  
 // 必要に応じてカスタム例外をスロー  
 // throw new UserProcessingException("ユーザー処理に失敗しました。", e);  
 } catch (Exception e) { // より一般的な例外  
 // ユーザーには一般的なエラーメッセージを表示  
 System.err.println("システムエラーが発生しました。管理者に連絡してください。");  
 // ログには詳細（ただし機密情報はマスク）を記録  
 logger.error("ユーザー '{}' の処理中に予期せぬエラーが発生しました。", username, e);  
 // throw new UserProcessingException("予期せぬシステムエラー。", e);  
 }  
 }  
  
 // 例外を適切に処理するか、より適切な例外でラップして再スロー  
 public void readFileAndProcess(String fileName) throws ProcessingException {  
 try {  
 // ファイル読み込み処理  
 // String content = Files.readString(Paths.get(fileName));  
 // processContent(content);  
 if (fileName.equals("nonexistent.txt")) { // ダミーの失敗条件  
 throw new java.io.IOException("ファイルが見つかりません: " + fileName);  
 }  
 } catch (java.io.IOException e) {  
 logger.error("ファイル '{}' の読み込みに失敗しました。", fileName, e);  
 throw new ProcessingException("ファイル処理エラー: " + fileName, e);  
 }  
 }  
}  
  
class ProcessingException extends Exception {  
 public ProcessingException(String message, Throwable cause) {  
 super(message, cause);  
 }  
}

## 8. 同期・スレッド（VNA, LCK, THI, TPS, TSM）：共有変数の同期、デッドロック回避、不変オブジェクト活用、協調的キャンセルを徹底する 🚦

### 解説

マルチスレッド環境では、複数のスレッドが共有リソース（変数、オブジェクトなど）に同時にアクセスすることで、競合状態やデータの不整合が発生する可能性があります。これを防ぐためには、synchronizedキーワードやjava.util.concurrent.locks.Lockインターフェースを用いた適切な同期処理が必要です。しかし、同期の範囲が広すぎるとパフォーマンスが低下し、ロックの順序を誤るとデッドロックが発生する可能性があります。不変（イミュータブル）オブジェクトは状態が変わらないため、本質的にスレッドセーフであり、同期の必要性を減らします。スレッドのキャンセルは、Thread.interrupt()を利用した協調的な方法で行い、スレッドが安全に終了処理を行えるようにします。

### 悪い例

Java

// 共有変数への非同期アクセス  
class Counter {  
 private int count = 0;  
 public void increment() {  
 count++; // 複数のスレッドから同時に呼び出されると競合状態になる  
 }  
 public int getCount() {  
 return count;  
 }  
}  
  
// デッドロックの可能性 (ロック順序の不一致)  
// Object lock1 = new Object();  
// Object lock2 = new Object();  
  
// Thread t1 = new Thread(() -> {  
// synchronized (lock1) {  
// System.out.println("Thread 1: Holding lock 1...");  
// try { Thread.sleep(100); } catch (InterruptedException e) {}  
// System.out.println("Thread 1: Waiting for lock 2...");  
// synchronized (lock2) {  
// System.out.println("Thread 1: Holding lock 1 & 2...");  
// }  
// }  
// });  
  
// Thread t2 = new Thread(() -> {  
// synchronized (lock2) { // lock1と逆順でロックを取得  
// System.out.println("Thread 2: Holding lock 2...");  
// try { Thread.sleep(100); } catch (InterruptedException e) {}  
// System.out.println("Thread 2: Waiting for lock 1...");  
// synchronized (lock1) {  
// System.out.println("Thread 2: Holding lock 1 & 2...");  
// }  
// }  
// });  
// t1.start(); t2.start();

### 良い例

Java

import java.util.concurrent.locks.ReentrantLock;  
import java.util.concurrent.atomic.AtomicInteger;  
  
// synchronizedメソッドによる同期  
class SynchronizedCounter {  
 private int count = 0;  
 public synchronized void increment() { // メソッド全体を同期  
 count++;  
 }  
 public synchronized int getCount() {  
 return count;  
 }  
}  
  
// java.util.concurrent.atomicパッケージの利用  
class AtomicCounter {  
 private AtomicInteger count = new AtomicInteger(0);  
 public void increment() {  
 count.incrementAndGet(); // アトミック操作  
 }  
 public int getCount() {  
 return count.get();  
 }  
}  
  
// Lockオブジェクトによる柔軟な同期とデッドロック回避のためのロック順序規約  
class FineGrainedLock {  
 private final ReentrantLock lock1 = new ReentrantLock();  
 private final ReentrantLock lock2 = new ReentrantLock();  
  
 public void operation1() {  
 // 常に同じ順序でロックを取得する (例: lock1 -> lock2)  
 lock1.lock();  
 try {  
 System.out.println("Operation 1: Holding lock 1...");  
 lock2.lock();  
 try {  
 System.out.println("Operation 1: Holding lock 1 & 2...");  
 // 共有リソースへの操作  
 } finally {  
 lock2.unlock();  
 }  
 } finally {  
 lock1.unlock();  
 }  
 }  
 // 他の操作も同じロック順序に従う  
}  
  
// 協調的スレッドキャンセル  
class MyRunnable implements Runnable {  
 @Override  
 public void run() {  
 while (!Thread.currentThread().isInterrupted()) {  
 // 作業を行う  
 System.out.println("Working...");  
 try {  
 Thread.sleep(1000); // InterruptedExceptionを発生させる可能性のある処理  
 } catch (InterruptedException e) {  
 System.out.println("Thread interrupted, cleaning up...");  
 Thread.currentThread().interrupt(); // 割り込み状態を再設定  
 break; // ループを抜けて終了  
 }  
 }  
 System.out.println("Thread finished.");  
 }  
}  
// Thread t = new Thread(new MyRunnable());  
// t.start();  
// try { Thread.sleep(3000); } catch (InterruptedException e) {}  
// t.interrupt(); // スレッドに割り込みを要求

## 9. I/O（FIO）：Try-with-resources でリソースリークを防ぎ、パス検証やタイムアウトを必須にする 📄

### 解説

ファイルやネットワーク接続などのI/Oリソースは、使用後に必ずクローズする必要があります。クローズ漏れはリソースリークを引き起こし、システムのパフォーマンス低下や枯渇につながります。Java 7以降で導入されたtry-with-resources文は、リソースの自動クローズを保証し、コードを簡潔にします。

また、外部から与えられたファイルパスをそのまま使用すると、ディレクトリトラバーサル攻撃（例：../../../../../etc/passwdのようなパスで不正なファイルにアクセスする）の危険性があります。ファイルパスは正規化し、意図したディレクトリ内にあることを検証する必要があります。ネットワークI/Oでは、応答がない場合に無期限に待機することを避けるため、適切なタイムアウトを設定することが重要です。

### 悪い例

Java

import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
  
// リソースクローズ漏れの可能性  
public class BadIO {  
 public void copyFile(String srcPath, String destPath) {  
 FileInputStream fis = null;  
 FileOutputStream fos = null;  
 try {  
 fis = new FileInputStream(srcPath); // srcPathの検証なし  
 fos = new FileOutputStream(destPath); // destPathの検証なし  
 byte[] buffer = new byte[1024];  
 int length;  
 while ((length = fis.read(buffer)) > 0) {  
 fos.write(buffer, 0, length);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 // fis.close() や fos.close() が例外をスローする可能性があり、  
 // その場合、後続のcloseが実行されない可能性がある  
 try {  
 if (fis != null) fis.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 try {  
 if (fos != null) fos.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

### 良い例

Java

import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.OutputStream;  
import java.nio.file.Files;  
import java.nio.file.Path;  
import java.nio.file.Paths;  
import java.net.Socket;  
import java.net.InetSocketAddress;  
  
public class GoodIO {  
 // try-with-resources による確実なリソースクローズ  
 public void copyFileSafe(String srcPathStr, String destPathStr) throws IOException {  
 // パス検証 (基本的な例)  
 Path srcPath = Paths.get(srcPathStr).toAbsolutePath();  
 Path destPath = Paths.get(destPathStr).toAbsolutePath();  
  
 // 例えば、特定のベースディレクトリ内に限定する  
 Path allowedBaseDir = Paths.get("/safe/app/data").toAbsolutePath();  
 if (!srcPath.startsWith(allowedBaseDir) || !destPath.startsWith(allowedBaseDir)) {  
 throw new SecurityException("許可されていないファイルパスへのアクセスです。");  
 }  
 // さらに正規化や詳細なチェックを行うべき  
  
 try (InputStream fis = Files.newInputStream(srcPath); // Filesクラスの使用を推奨  
 OutputStream fos = Files.newOutputStream(destPath)) {  
 byte[] buffer = new byte[4096];  
 int length;  
 while ((length = fis.read(buffer)) > 0) {  
 fos.write(buffer, 0, length);  
 }  
 } catch (IOException e) {  
 System.err.println("ファイルコピー中にエラーが発生しました: " + e.getMessage());  
 throw e; // 必要に応じて再スロー  
 }  
 }  
  
 // ネットワークI/Oでのタイムアウト設定  
 public void connectWithTimeout(String host, int port) throws IOException {  
 try (Socket socket = new Socket()) {  
 // 接続タイムアウト (ミリ秒)  
 socket.connect(new InetSocketAddress(host, port), 5000); // 5秒  
 // 通信タイムアウト (ミリ秒)  
 socket.setSoTimeout(10000); // 10秒  
 System.out.println("サーバーに接続しました: " + host + ":" + port);  
 // ...通信処理...  
 } catch (IOException e) {  
 System.err.println("接続または通信エラー (" + host + ":" + port + "): " + e.getMessage());  
 throw e;  
 }  
 }  
}

## 10. シリアライズ（SER）：デシリアライズ時にホワイトリストを使い、許可クラスのみを読み込む 📜➡️🧩

### 解説

Javaのシリアライズ機構はオブジェクトの状態をバイトストリームに変換し、それを復元（デシリアライズ）する機能を提供します。しかし、信頼できないソースからのデータをデシリアライズする場合、攻撃者が巧妙に細工したバイトストリームを送り込むことで、任意のコード実行やサービス不能攻撃（DoS）を引き起こす可能性があります（デシリアライズ脆弱性）。これを防ぐためには、デシリアライズするクラスを厳密に制限するホワイトリスト方式のフィルタリングを導入することが推奨されます。Java 9以降ではObjectInputFilterが標準で提供されています。

### 悪い例

Java

import java.io.\*;  
  
// 任意のオブジェクトを無制限にデシリアライズする  
public class BadDeserialization {  
 public Object deserialize(byte[] data) throws IOException, ClassNotFoundException {  
 ByteArrayInputStream bais = new ByteArrayInputStream(data);  
 ObjectInputStream ois = new ObjectInputStream(bais); // フィルターなし  
 return ois.readObject(); // 危険: 任意のクラスがデシリアライズされる可能性  
 }  
}

### 良い例

Java

import java.io.\*;  
import java.util.Arrays;  
import java.util.HashSet;  
import java.util.Set;  
// Java 9以降の場合  
// import java.io.ObjectInputFilter;  
  
// 許可するクラスのホワイトリスト  
class SafeClassFilter // Java 9未満でObjectInputStreamをサブクラス化する場合の例  
 // またはJava 9以降のObjectInputFilter.Config.createFilterの引数に使用するロジック  
{  
 private static final Set<String> ALLOWED\_CLASSES = new HashSet<>(Arrays.asList(  
 "com.example.myapp.UserData",  
 "java.util.ArrayList", // UserDataが内部で使用する可能性のあるクラスも考慮  
 "int", // プリミティブ型は問題ないが、明示は通常不要  
 "[B" // byte[] など配列型  
 ));  
  
 public static boolean isAllowed(String className) {  
 return ALLOWED\_CLASSES.contains(className);  
 }  
}  
  
// Java 9以降のObjectInputFilterを使用  
public class GoodDeserializationJ9 {  
 public Object deserialize(byte[] data) throws IOException, ClassNotFoundException {  
 ByteArrayInputStream bais = new ByteArrayInputStream(data);  
 try (ObjectInputStream ois = new ObjectInputStream(bais)) {  
 // ホワイトリストフィルターを設定  
 java.io.ObjectInputFilter filter = java.io.ObjectInputFilter.Config.createFilter(  
 "com.example.myapp.UserData;java.util.ArrayList;!\*" // セミコロン区切り、!\*で他を拒否  
 );  
 ois.setObjectInputFilter(filter);  
 return ois.readObject();  
 }  
 }  
}  
  
// Java 8以前でObjectInputStreamをオーバーライドしてフィルタリングする例  
class FilteredObjectInputStream extends ObjectInputStream {  
 public FilteredObjectInputStream(InputStream in) throws IOException {  
 super(in);  
 }  
  
 @Override  
 protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException {  
 String className = desc.getName();  
 if (!SafeClassFilter.isAllowed(className)) {  
 throw new InvalidClassException("許可されていないクラスのデシリアライズ: " + className);  
 }  
 return super.resolveClass(desc);  
 }  
}  
  
public class GoodDeserializationJ8 {  
 public Object deserialize(byte[] data) throws IOException, ClassNotFoundException {  
 ByteArrayInputStream bais = new ByteArrayInputStream(data);  
 // カスタムのFilteredObjectInputStreamを使用  
 try (FilteredObjectInputStream ois = new FilteredObjectInputStream(bais)) {  
 return ois.readObject();  
 }  
 }  
}  
  
// 想定するデータクラスの例  
// package com.example.myapp;  
// import java.io.Serializable;  
// public class UserData implements Serializable {  
// private static final long serialVersionUID = 1L;  
// String name;  
// int age;  
// // constructor, getters, setters  
// }

**注意**: シリアライズの代替として、JSONやProtocol Buffersなど、より安全でスキーマ定義が明確なデータフォーマットの利用も検討すべきです。

## 11. プラットフォーム（SEC）：安全な暗号アルゴリズムを使用し、外部ライブラリの脆弱性を監視する 🔑

### 解説

アプリケーションのセキュリティは、使用する暗号アルゴリズムの強度や、依存する外部ライブラリの安全性に大きく左右されます。MD5やSHA-1のような古いハッシュアルゴリズムや、DESのような古い暗号化アルゴリズムは既知の脆弱性があり、使用を避けるべきです。パスワードハッシュにはArgon2, scrypt, bcrypt, PBKDF2 with HMAC-SHA256などを、共通鍵暗号にはAES-GCMやAES-CBC（適切なパディングとMACを併用）を、公開鍵暗号にはRSA（適切な鍵長とパディング）やECCを使用します。乱数生成にはjava.security.SecureRandomを使用し、予測可能なjava.util.Randomをセキュリティ用途で使用してはいけません。

また、多くのJavaプロジェクトは外部ライブラリに依存しています。これらのライブラリに脆弱性が発見されることがあるため、OWASP Dependency-Checkのようなツールを使用して定期的に脆弱性をスキャンし、必要に応じてライブラリをアップデートすることが重要です。

### 悪い例

Java

import java.security.MessageDigest;  
import java.util.Random;  
import javax.crypto.Cipher;  
import javax.crypto.spec.SecretKeySpec;  
import java.nio.charset.StandardCharsets;  
import java.util.Base64;  
  
public class BadCrypto {  
 // 古いハッシュアルゴリズムの使用  
 public String hashPasswordMD5(String password) throws Exception {  
 MessageDigest md = MessageDigest.getInstance("MD5");  
 byte[] hashedBytes = md.digest(password.getBytes(StandardCharsets.UTF\_8));  
 return Base64.getEncoder().encodeToString(hashedBytes);  
 }  
  
 // 安全でない乱数生成器のセキュリティ用途での使用  
 public int generateInsecureOtp() {  
 Random random = new Random(); // 予測可能な乱数  
 return random.nextInt(900000) + 100000; // 6桁のOTP  
 }  
  
 // ECBモードのような安全でない暗号モードの使用 (鍵は例として固定)  
 public String encryptWithDES\_ECB(String data) throws Exception {  
 byte[] keyBytes = "an8charKey".getBytes(StandardCharsets.UTF\_8); // DESは8バイト鍵  
 SecretKeySpec key = new SecretKeySpec(keyBytes, "DES");  
 Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding"); // ECBモードは危険  
 cipher.init(Cipher.ENCRYPT\_MODE, key);  
 byte[] encryptedBytes = cipher.doFinal(data.getBytes(StandardCharsets.UTF\_8));  
 return Base64.getEncoder().encodeToString(encryptedBytes);  
 }  
}

### 良い例

Java

import java.security.MessageDigest;  
import java.security.NoSuchAlgorithmException;  
import java.security.SecureRandom;  
import javax.crypto.Cipher;  
import javax.crypto.KeyGenerator;  
import javax.crypto.SecretKey;  
import javax.crypto.spec.GCMParameterSpec;  
import java.nio.charset.StandardCharsets;  
import java.util.Base64;  
  
public class GoodCrypto {  
 // 強力なハッシュアルゴリズム (SHA-256またはそれ以上) とソルトの使用を推奨  
 // パスワードハッシュにはPBKDF2, bcrypt, scrypt, Argon2を使用するのがベストプラクティス  
 public String hashPasswordSHA256(String password, byte[] salt) throws NoSuchAlgorithmException {  
 MessageDigest md = MessageDigest.getInstance("SHA-256");  
 md.update(salt);  
 byte[] hashedBytes = md.digest(password.getBytes(StandardCharsets.UTF\_8));  
 return Base64.getEncoder().encodeToString(hashedBytes);  
 // 実際にはPBKDF2等でストレッチングも行う  
 }  
  
 // SecureRandomの使用  
 public int generateSecureOtp() {  
 SecureRandom random = new SecureRandom();  
 return random.nextInt(900000) + 100000; // 6桁のOTP  
 }  
  
 // AES-GCMモードのような認証付き暗号の使用  
 public String encryptWithAES\_GCM(String data, SecretKey key) throws Exception {  
 Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");  
 // GCMモードでは通常96ビット(12バイト)のIVが推奨される  
 byte[] iv = new byte[12];  
 SecureRandom random = new SecureRandom();  
 random.nextBytes(iv);  
 GCMParameterSpec gcmParameterSpec = new GCMParameterSpec(128, iv); // 128ビット認証タグ  
 cipher.init(Cipher.ENCRYPT\_MODE, key, gcmParameterSpec);  
 byte[] encryptedBytes = cipher.doFinal(data.getBytes(StandardCharsets.UTF\_8));  
 // IVと暗号文を結合して保存・送信することが一般的  
 // 例: Base64.getEncoder().encodeToString(iv) + ":" + Base64.getEncoder().encodeToString(encryptedBytes)  
 return Base64.getEncoder().encodeToString(encryptedBytes); // ここでは暗号文のみ  
 }  
  
 public static SecretKey generateAESKey() throws NoSuchAlgorithmException {  
 KeyGenerator keyGen = KeyGenerator.getInstance("AES");  
 keyGen.init(256); // AES-256  
 return keyGen.generateKey();  
 }  
}  
// 外部ライブラリの脆弱性監視は、MavenやGradleのプラグイン (例: OWASP Dependency-Check)  
// や、GitHubのDependabotのようなサービスを利用して自動化する。

## 12. 環境設定（ENV）：環境変数／プロパティを信用せず、入力フォーマットを検証する ⚙️

### 解説

環境変数やJavaのシステムプロパティは、アプリケーションの動作を設定するために広く利用されますが、これらは外部から設定可能である場合があり、必ずしも信頼できるとは限りません。例えば、データベースの接続文字列、ファイルパス、外部サービスのURLなどを環境変数から取得する場合、その値が期待するフォーマットであるか、不正な値が設定されていないかを検証する必要があります。検証を怠ると、設定ミスによる動作不良だけでなく、インジェクション攻撃や情報漏洩につながる可能性があります。

### 悪い例

Java

// 環境変数から取得した値を検証せずに使用  
public class BadEnvironmentConfig {  
 public String getTempDirectory() {  
 // "TEMP\_DIR" 環境変数が "/tmp" のような値を期待  
 // しかし、悪意のある値 "; rm -rf /" などが設定される可能性を考慮していない  
 String tempDir = System.getenv("TEMP\_DIR");  
 if (tempDir == null || tempDir.isEmpty()) {  
 return "/default/temp";  
 }  
 return tempDir; // 検証なしで使用  
 }  
  
 public int getTimeout() {  
 // "TIMEOUT\_SECONDS" 環境変数が数値を期待  
 String timeoutStr = System.getenv("TIMEOUT\_SECONDS");  
 if (timeoutStr == null || timeoutStr.isEmpty()) {  
 return 30; // デフォルト値  
 }  
 return Integer.parseInt(timeoutStr); // 数値変換エラーや負の値を考慮していない  
 }  
}

### 良い例

Java

import java.nio.file.InvalidPathException;  
import java.nio.file.Path;  
import java.nio.file.Paths;  
import java.util.regex.Pattern;  
  
public class GoodEnvironmentConfig {  
 private static final Pattern ALLOWED\_PATH\_PATTERN = Pattern.compile("^[a-zA-Z0-9/.\_-]+$");  
  
 public Path getTempDirectory() {  
 String tempDirStr = System.getenv("TEMP\_DIR");  
 Path defaultTempDir = Paths.get("/default/app/temp");  
  
 if (tempDirStr == null || tempDirStr.trim().isEmpty()) {  
 System.out.println("TEMP\_DIR環境変数が未設定のため、デフォルト値を使用します: " + defaultTempDir);  
 return defaultTempDir;  
 }  
  
 // 基本的なフォーマット検証 (より厳密な検証が望ましい)  
 if (!ALLOWED\_PATH\_PATTERN.matcher(tempDirStr).matches()) {  
 System.err.println("TEMP\_DIR環境変数の値が不正です: " + tempDirStr + ". デフォルト値を使用します。");  
 return defaultTempDir;  
 }  
  
 try {  
 Path tempDir = Paths.get(tempDirStr).toAbsolutePath();  
 // さらに、パスが期待されるベースディレクトリ内にあるかなどを検証することも可能  
 // if (!tempDir.startsWith("/app/safe\_area")) { ... }  
 System.out.println("TEMP\_DIRとして " + tempDir + " を使用します。");  
 return tempDir;  
 } catch (InvalidPathException e) {  
 System.err.println("TEMP\_DIR環境変数のパスが無効です: " + tempDirStr + ". デフォルト値を使用します。エラー: " + e.getMessage());  
 return defaultTempDir;  
 }  
 }  
  
 public int getTimeout() {  
 String timeoutStr = System.getenv("TIMEOUT\_SECONDS");  
 int defaultTimeout = 30; // 秒  
  
 if (timeoutStr == null || timeoutStr.trim().isEmpty()) {  
 System.out.println("TIMEOUT\_SECONDS環境変数が未設定のため、デフォルト値を使用します: " + defaultTimeout + "秒");  
 return defaultTimeout;  
 }  
  
 try {  
 int timeout = Integer.parseInt(timeoutStr);  
 if (timeout <= 0 || timeout > 300) { // 妥当な範囲内かチェック (例: 1秒以上300秒以下)  
 System.err.println("TIMEOUT\_SECONDSの値 (" + timeout + ") が不正な範囲です。デフォルト値を使用します: " + defaultTimeout + "秒");  
 return defaultTimeout;  
 }  
 System.out.println("TIMEOUT\_SECONDSとして " + timeout + "秒 を使用します。");  
 return timeout;  
 } catch (NumberFormatException e) {  
 System.err.println("TIMEOUT\_SECONDS環境変数の値が数値ではありません: " + timeoutStr + ". デフォルト値を使用します。");  
 return defaultTimeout;  
 }  
 }  
}

## 13. 雑則（MSC）：ログは構造化し、監査に耐えうる形で出力するなど、運用時の注意も含める 📋

### 解説

セキュリティはコードだけでなく、運用面も重要です。ログはインシデント発生時の原因究明や監査証跡として不可欠な情報源となります。ログには、いつ（タイムスタンプ）、誰が（ユーザーID、IPアドレスなど関連情報）、何を（イベント種別）、どのように（処理内容の要約）、結果どうなったか（成功／失敗、エラーコードなど）といった情報を含めるべきです。ただし、パスワード、セッショントークン、クレジットカード番号などの機密情報は絶対にログに平文で記録してはいけません。必要に応じてマスキングや暗号化を施します。ログはJSONやLogfmtのような構造化された形式で出力することで、機械的な収集や分析が容易になります。

また、不要なデバッグコードやコメントアウトされた実験的コードは本番環境に残さない、エラーメッセージはユーザーに過度な内部情報を与えないようにする、といった運用全般に関わる注意点も守る必要があります。

### 悪い例

Java

// ログに機密情報を含めてしまう  
// ログが非構造的で分析しにくい  
public class BadLogging {  
 public void login(String username, String password) {  
 System.out.println("Attempting login for user: " + username + " with password: " + password); // パスワードを平文でログ出力  
 if ("admin".equals(username) && "password123".equals(password)) {  
 System.out.println("Login successful for user: " + username);  
 } else {  
 System.out.println("Login failed for user: " + username);  
 }  
 }  
  
 public void processPayment(String creditCardNumber, double amount) {  
 // ... 支払い処理 ...  
 // ログにクレジットカード番号全体を記録  
 System.out.println("Payment processed for card: " + creditCardNumber + ", amount: " + amount);  
 }  
}

### 良い例

Java

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import com.fasterxml.jackson.databind.ObjectMapper; // JSON構造化ログの例 (Jacksonライブラリ)  
import com.fasterxml.jackson.databind.node.ObjectNode;  
  
public class GoodLogging {  
 private static final Logger logger = LoggerFactory.getLogger(GoodLogging.class);  
 private static final ObjectMapper jsonMapper = new ObjectMapper();  
  
 public void login(String username, String password) { // passwordはここでは使用しないが、実際はハッシュ化比較  
 ObjectNode logEntry = jsonMapper.createObjectNode();  
 logEntry.put("event\_type", "user\_login\_attempt");  
 logEntry.put("timestamp", System.currentTimeMillis());  
 logEntry.put("username", username);  
 // パスワード自体はログに出力しない  
  
 // ログイン処理 (実際の認証はセキュアな方法で行う)  
 boolean loginSuccess = ("admin".equals(username) /\* && verifyPassword(password, storedHash) \*/);  
  
 if (loginSuccess) {  
 logEntry.put("status", "success");  
 logger.info(logEntry.toString()); // 構造化ログ  
 System.out.println("ユーザー '" + username + "' のログインに成功しました。");  
 } else {  
 logEntry.put("status", "failure");  
 logger.warn(logEntry.toString()); // 構造化ログ  
 System.err.println("ユーザー '" + username + "' のログインに失敗しました。");  
 }  
 }  
  
 public void processPayment(String creditCardNumber, double amount, String userId) {  
 // クレジットカード番号をマスクしてログ記録 (例: 下4桁のみ)  
 String maskedCardNumber = "\*\*\*\*\*\*\*\*\*\*\*\*" + (creditCardNumber.length() > 4 ?  
 creditCardNumber.substring(creditCardNumber.length() - 4) : "\*\*\*\*");  
  
 ObjectNode logEntry = jsonMapper.createObjectNode();  
 logEntry.put("event\_type", "payment\_processing");  
 logEntry.put("timestamp", System.currentTimeMillis());  
 logEntry.put("user\_id", userId);  
 logEntry.put("masked\_card\_number", maskedCardNumber);  
 logEntry.put("amount", amount);  
  
 try {  
 // ... 支払い処理 ...  
 // paymentGateway.charge(creditCardNumber, amount);  
 logEntry.put("status", "success");  
 logger.info(logEntry.toString());  
 } catch (Exception e) {  
 logEntry.put("status", "failure");  
 logEntry.put("error\_message", e.getMessage()); // エラーメッセージは慎重に  
 logger.error(logEntry.toString(), e);  
 }  
 }  
}  
// ログライブラリ (Logback, Log4j2など) の設定で、出力フォーマットをJSONなどに指定することも一般的。  
// 例 (logback.xml):  
// <appender name="STDOUT\_JSON" class="ch.qos.logback.core.ConsoleAppender">  
// <encoder class="net.logstash.logback.encoder.LogstashEncoder"/>  
// </appender>

これらの規約は、セキュアなJavaアプリケーションを開発するための基本的な指針です。実際のプロジェクトでは、これらの規約をベースに、より詳細で具体的なルールを定義し、コードレビューや静的解析ツールを通じて遵守を徹底することが重要です。

**ソース**

1. <https://developer.aliyun.com/article/1474052>

2. <https://github.com/IHTSDO/snomed-release-service> ライセンス（Apache - 2.0）が必要

3. <https://github.com/MentosL/montos-framework-validation>

4. <https://www.kodnest.com/core-java/>

5. <https://javanexus.com/blog/java-deadlocks-solving>

6. <https://github.com/Amanastel/myJavaCode>

7. <https://github.com/Benjit75/SLR-201>

8. <https://www.oschina.net/informat/java+%E6%96%87%E4%BB%B6%E5%86%99%E6%B3%95>

9. <https://developer.aliyun.com/article/1549990>

10. <https://blog.csdn.net/qq_43598138/article/details/105718417>

11. <https://github.com/iroot900/Distributed-File-System-Implementation>

12. <https://github.com/hawkore/ignite-hk> ライセンス（Apache - 2.0）が必要

13. <https://github.com/mohammadrezaaliabadi/is>

14. <https://github.com/blockchain-jd-com/utils>

15. <http://blog.ippon.fr/2008/07/26/qui-a-dit-que-serialversionuid-etait-obligatoire/>

16. <http://blog.csdn.net/luozhi3527/article/details/8202261>

17. <https://github.com/HaleyWang/SpringRemote>

18. <https://juejin.cn/post/7284878428817883196>

19. <https://github.com/leonx04/ThePoloManShop>

20. <https://github.com/AliHaider0343/Java-Servlet-based-Social-Media-Application->

21. <https://stackoverflow.com/questions/22919265/javax-crypto-badpaddingexception-in-decrypt-method-using-base64>

22. <https://blog.csdn.net/2401_84265571/article/details/138263616>

23. <https://bugs.openjdk.org/browse/JDK-8062828?page=com.atlassian.jira.plugin.system.issuetabpanels:comment-tabpanel>

24. <https://github.com/BilalAbbas78/Email-Client-with-Encryption>

25. <https://github.com/AxelitoHyuga/TodoShonen_Project_EDD>

26. <https://github.com/dshafranskiy-r7/vulnerable-project>

27. <https://github.com/NimeshikaSedara/Java-MANET-Application>

28. <https://github.com/Flank/flank> ライセンス（Apache - 2.0）が必要

29. <https://stackoverflow.com/questions/60504197/adding-a-dynamic-value-date-in-customfields-logstashencoder>