Java 10+a年 振り返り (上巻)

Record
Switch式
Sealed

モダンJavaの
学び直しに！！

上巻はJava 8～17

著　世迷言ラボ

Java 10+a年振り返り（上巻） - 立ち読み版！！

世迷言ラボ

2026-04-11 初版 技術書典20

まえがき

この本は、Java 8からJava 17までの主な新機能をまとめた本です（上巻）。Java 18以降は下巻で扱います。

Java 8は、2014年にリリースされました。その後、Java 9が2017年9月にリリースされ、以降は半年に1度、3月と9月にリリースされ続けています。

半年に1度という短いリリースサイクルに対して、ソフトウェアの開発、保守期間は長く、最新の言語仕様をキャッチアップできていないJavaエンジニアも多くいると思います。

この本を読んで、Javaの新しい機能についてインデックスを作っていきましょう。

対象読者

本書の対象読者は、Java SEの各バージョンの特徴を理解し、Javaがどのように進化してきたか、また今後どのような方向性で進化していくかを理解したい人です。

本書は、Java 8からJava 17までの主な新機能をバージョンごとにまとめているため、現場で使えるバージョンが固定されている方でも「ここまでなら使えるんだ」という読み方ができます。

また、業務での開発環境がJava 8で塩漬けになっており、Javaの進化を追いかけたい人にもおすすめです。

この本を書いたモチベーション

技術書典19にて、Javaの本が少ないという話がありました。それなら自分で書いてみようと思ったのが、最初のきっかけです。

JJUGなどのコミュニティにも顔を出し、Javaコミュニティへの恩返し、貢献という意味合いもあります。

免責事項

目次

第1章
Javaとは

本章では、Javaの基本的な概念と、Java SEの位置づけ、そしてJavaの進化を支えるJEP（JDK Enhancement Proposal）について解説します。

1.1 Javaとは

Javaは、1995年にSun Microsystems（現在はOracle Corporation）によって開発されたプログラミング言語およびプラットフォームです。「Write Once, Run Anywhere（一度書けばどこでも動く）」という理念のもと、プラットフォームに依存しないアプリケーション開発を可能にしています。

1.1.1 Javaの特徴

Javaの主な特徴は以下の通りです。

- オブジェクト指向: クラスとオブジェクトを基本とした設計
- プラットフォーム非依存: JVM（Java Virtual Machine）上で動作するため、OSに依存しない
- メモリ管理: ガベージコレクション（GC）による自動メモリ管理
- 豊富なライブラリ: 標準ライブラリ（Java API）が充実している
- 強い型付け: コンパイル時に型チェックを行い、実行時エラーを減らす
- マルチスレッド対応: 並行処理を言語レベルでサポート

1.1.2 Javaのエディション

Javaには、用途に応じて複数のエディションが存在します。

- Java SE (Standard Edition): デスクトップアプリケーションや基本的なサーバーアプリケーション向け
- Java EE (Enterprise Edition): 大規模エンタープライズアプリケーション向け（現在はJakarta EEとして独立）
- Java ME (Micro Edition): 組み込み機器やモバイルデバイス向け

本書では、Java SEを中心に扱います。

1.2 Java SEとは

Java SE（Java Platform, Standard Edition）は、Javaプラットフォームの基盤となるエディションです。Java言語仕様、JVM仕様、標準ライブラリ（Java API）などが含まれます。

1.2.1 Java SEの構成要素

Java SEは、以下の主要な要素で構成されています。

JDK (Java Development Kit)

Java開発キットで、Javaアプリケーションを開発するために必要なツール一式が含まれています。

- Javac: Javaコンパイラ
- Java: Java実行環境

- jar: JARファイル作成ツール
- Javadoc: APIドキュメント生成ツール
- jshell: 対話型Java実行環境（Java 9以降）

JRE (Java Runtime Environment)

Java実行環境で、Javaアプリケーションを実行するために必要な最小限の環境です。

- JVM: Java Virtual Machine
- 標準ライブラリ: Java API（クラスライブラリ）

なお、Java 11以降、OracleはJREの単独配布を終了しました。現在は、JDKに含まれるJREを使用するか、`jlink`コマンドを使って必要なモジュールのみを含むカスタムランタイムイメージを作成する方法が推奨されています。

JVM (Java Virtual Machine)

Javaバイトコードを実行する仮想マシンです。プラットフォームごとに異なる実装がありますが、同じバイトコードを実行できるため、Javaの「Write Once, Run Anywhere」を実現しています。

1.2.2 Java SEのバージョン管理

Java SEは、2017年9月のJava 9以降、半年に1度のリリースサイクルを採用しています。

- Feature Release: 3月と9月に定期リリース

- LTS (Long Term Support): Java 17以降は3年に1度、長期サポート版をリリース（Java 17, 21, 25...）
- Update Release: セキュリティアップデートやバグフィックス

1.3 JEP（JDK Enhancement Proposal）とは

JEP（JDK Enhancement Proposal）は、JDKに新機能を追加したり、既存機能を改善したりする提案のプロセスです。OpenJDKコミュニティが管理しており、Javaの進化を支える重要な仕組みです。

1.3.1 JEPのライフサイクル

JEPは以下のようなライフサイクルをたどります。

1. Draft（草案）: 初期の提案段階
2. Submitted（提出済み）: 正式に提出された提案
3. Candidate（候補）: レビュー中の提案
4. Funded（予算確保）: 実装リソースが確保された提案
5. Completed（完了）: 実装が完了し、JDKに統合された提案
6. Closed（クローズ）: 却下または取り下げられた提案

1.3.2 JEPの種類

JEPには、さまざまな種類があります。

- Feature JEP: 新機能の追加
- Process JEP: 開発プロセスの改善
- Informational JEP: 情報提供や設計ドキュメント

1.3.3 プレビュー機能とインキュベータ機能

Java SEでは、新機能を段階的に導入するために、プレビュー機能とインキュベータ機能という仕組みがあります。

プレビュー機能（Preview Features）

言語仕様やJVM仕様に関わる新機能を試験的に提供する仕組みです。プレビュー機能は、以下の特徴があります。

- コンパイル時に`--enable-preview`フラグが必要
- 実行時にも`--enable-preview`フラグが必要
- 将来のバージョンで仕様が変更される可能性
- 複数のバージョンでプレビューを経て、正式機能となる

例: Records（Java 14, 15でプレビュー → Java 16で正式機能）

インキュベータ機能（Incubator Features）

APIに関する新機能を試験的に提供する仕組みです。インキュベータモジュールとして提供され、以下の特徴があります。

- モジュール名に`jdk.incubator.`プレフィックスが付く
- 明示的にモジュールを要求する必要がある
- 将来のバージョンでAPIが変更される可能性

例: HTTP Client API（Java 9, 10でインキュベータ → Java 11で正式機能）

1.3.4 JEPの確認方法

JEPは、OpenJDKの公式サイトで確認できます。

- JEP一覧: https://openjdk.org/jeps/0
- 各バージョンのJEP: リリースノートに記載

本書では、各バージョンで導入された主要なJEPについて解説していきます。

1.4 まとめ

本章では、Javaの基本的な概念、Java SEの位置づけ、そしてJavaの進化を支えるJEPについて解説しました。

- Javaは「Write Once, Run Anywhere」を実現するプラットフォーム
- Java SEは、Javaの基盤となるエディション
- JEPは、JDKの新機能や改善を提案するプロセス
- プレビュー機能とインキュベータ機能により、新機能が段階的に導入される

次章以降では、Java 8以降の各バージョンで導入された具体的な機能について見ていきます。

第2章
OpenJDKの主要プロジェクト

JavaはOpenJDKを中心としたオープンな開発体制で進化を続けています。その中でも言語・ランタイムに大きな影響を与える長期プロジェクトがいくつか走っており、各バージョンのリリースにはこれらのプロジェクトの成果が反映されています。

ここでは代表的な5つのプロジェクトを紹介します。

2.1 Project Amber

2017年3月に発足したProject Amberは、Java言語の生産性を高める小〜中規模の機能を継続的に届けることを目的としたプロジェクトです。「大きな言語変更」ではなく、日々のコーディングをより快適にする改善を積み重ねていくのが特徴です。

主な成果として、`var`によるローカル変数型推論（Java 10）、テキストブロック（Java 15）、`record`（Java 16）、`sealed`クラス（Java 17）、パターンマッチング for `instanceof`（Java 16）・for `switch`（Java 21）などが挙げられます。現在も進行中のプロジェクトで、今後のバージョンでもAmber発の機能が追加される予定です。

2.2 Project Loom

2017〜18年頃に立ち上がったProject Loomは、Javaの並行処理モデルを刷新するプロジェクトです。従来のJavaスレッドはOSスレッドと1対1で対応しており、大量の並行タスクを扱う際にメモリやコンテキストスイッチのコストが問題となっていました。

Loomの主な成果は**仮想スレッド（Virtual Threads）**で、Java 21でGAとなりました。仮想スレッドはJVM上で管理される軽量なスレッドで、OSスレッドとは独立して大量に生成できます。既存の`Thread` APIとほぼ互換性があるため、コードをほとんど変えずに高い並行性を実現できます。合わせてStructured Concurrency（構造化並行処理）も提案されており、並行処理の安全なライフサイクル管理が容易になります。

2.3 Project Valhalla

2014年から続くProject Valhallaは、Javaの型システムとジェネリクスを根本から改善することを目指すプロジェクトです。

主なテーマは**値型（Value Types）**の導入です。現在のJavaではプリミティブ型（`int`、`long`など）とオブジェクト型が明確に分かれており、ジェネリクスではプリミティブ型を直接扱えません。Valhallaは Value Classという新しい型を導入することで、ヒープアロケーションやボクシングのコストを削減し、パフォーマンスと表現力を向上させることを目指しています。プロジェクトの規模と複雑さから、実現には長い年月がかかっており、2025年時点でも開発が続いています。

2.4 Project Panama

2018年頃に本格化したProject Panamaは、JavaとネイティブコードやネイティブAPIとの連携を改善することを目的としたプロジェクトです。従来のJNI（Java Native Interface）は記述が煩雑で学習コストも高く、ネイティブライブラリの利用には大きな障壁がありました。

主な成果は2つあります。1つ目は**Foreign Function & Memory API（FFM API）**で、Java 22でGAとなりました。FFM APIを使うと、JNIを使わずにCなどのネイティブライブラリを呼び出したり、Javaヒープ外のメモリを安全に操作したりできます。2つ目は**Vector API**で、SIMD命令を活用したベクトル演算をJavaコードで表現するAPIです。こちらはJava 25時点でもIncubator段階にあります。

2.5 Project Leyden

2020年3月にMark Reinholdが提案したProject Leydenは、Javaアプリケーションの**起動時間・ウォームアップ時間・フットプリントの削減**を目的としたプロジェクトです。

JVMはJITコンパイルにより実行時に高いパフォーマンスを発揮しますが、その反面、起動直後はウォームアップに時間がかかるという特性があります。マイクロサービスやサーバーレス環境ではこの起動コストが課題となるケースがあります。LeydenはClass Data Sharing（CDS）の拡張やAOT（事前）コンパイルの仕組みを整備することで、この問題にJVM標準の手段で対処しようとしています。Java 24以降のリリースにLeydenの成果が順次取り込まれています。

第1部 Java SE 8から17

第3章
Java SE 8の紹介

Java SE 8（以降はJava 8と記載）は2014年3月18日にリリースされました。

ラムダ式やStream API、Date Time APIなど「モダンJava」の礎になるような機能が多数リリースされたバージョンです。

3.1 主な新機能

Java 8で導入された主な新機能は次の通りです。

- ラムダ式
- Stream API
- Optional
- Date Time API
- インターフェイスのデフォルトメソッド

関数型のパラダイム、イミュータブルな日付関連API、Null安全に向けたクラス追加など、のちのJava開発環境に大きな変化のある機能が追加されています。

この他にもアノテーション周りの修正がいくつか入っています。

3.2 ラムダ式

ラムダ式は、関数型インターフェース（抽象メソッドをひとつだけ持つインターフェース）を実装する無名関数を簡潔に記述するための構文です。Java 8 より前では、たとえばスレッドの処理を渡す際に無名クラスを使う必要があり、記述が冗長になりがちでした。ラムダ式を使うとそれを大幅に短く書けます。

3.2.1 コード例

Java 8 より前の無名クラスによる書き方と、ラムダ式による書き方を比較します。

```
// Java 8より前：無名クラス
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello, Java!");
    }
};

// Java 8以降：ラムダ式
Runnable r2 = () -> System.out.println("Hello, Java!");
```

引数がある場合や、複数行の処理が必要な場合は次のように書きます。

```
// 引数あり・1行
Comparator<String> c1 = (a, b) -> a.compareTo(b);

// 引数あり・複数行
```

```
Comparator<String> c2 = (a, b) -> {
    System.out.println("comparing: " + a + ", " + b);
    return a.compareTo(b);
};
```

3.2.2 どんなケースで役に立つか

コールバックやイベントハンドラなど、「処理そのものを引数として渡したい」場面で特に有効です。Stream APIと組み合わせることでコレクション処理を宣言的に書けるようになり、コードの意図が伝わりやすくなります。

3.2.3 補足情報・注意点

ラムダ式が代入できるのは**関数型インターフェース**だけです。`@FunctionalInterface`アノテーションを付けると、誤って抽象メソッドを複数定義した際にコンパイルエラーで気づけるため、自作する場合は明示的に付けておくとよいでしょう。

```
@FunctionalInterface
interface Greeter {
    void greet(String name);
}

Greeter g = name -> System.out.println("Hello, " + name +
"!");
g.greet("Java");  // Hello, Java!
```

`java.util.function` パッケージには `Function<T, R>`、`Predicate<T>`、`Consumer<T>`、`Supplier<T>` などの汎用的な関数型インターフェースが標準で用意されているため、多くの場面で自作せずに済みます。

3.3 メソッド参照

メソッド参照は、ラムダ式をさらに簡潔に書くための構文です。ラムダ式の本体が「既存のメソッドを呼び出すだけ」の場合に、そのメソッドを直接参照として渡せます。

3.3.1 コード例

```
List<String> names = List.of("Alice", "Bob", "Charlie");

// ラムダ式
names.forEach(name -> System.out.println(name));

// メソッド参照（同じ意味）
names.forEach(System.out::println);
```

メソッド参照には4種類あります。

```
// 1. 静的メソッド参照
Function<String, Integer> parser = Integer::parseInt;

// 2. 特定インスタンスのメソッド参照
```

```
// 3. 任意インスタンスのメソッド参照
Function<String, String> toUpper = String::toUpperCase;

// 4. コンストラクタ参照
Supplier<ArrayList<String>> listFactory = ArrayList::new;
```

3.3.2 どんなケースで役に立つか

ラムダ式の本体が単純なメソッド呼び出しだけのとき、メソッド参照に置き換えるとコードが読みやすくなります。特にStream APIと組み合わせたパイプライン処理で効果的です。

```
List<String> upper = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

3.3.3 補足情報・注意点

メソッド参照が読みやすくなるかどうかはケースによります。引数に対して何らかの変換を加えるなど、処理の意図を明示したい場合はラムダ式のままにしておくほうがわかりやすいこともあります。無理にメソッド参照に変換しようとせず、読みやすさを優先して使い分けるとよいでしょう。

3.4 Stream API

Stream APIは、コレクションや配列などのデータソースに対してフィルタリング・変換・集計といった処理を宣言的なパイプラインとして記述するためのAPIです。処理の「何をするか」を明確に表現でき、ループを書き連ねる従来のスタイルと比べてコードの意図が伝わりやすくなります。

3.4.1 コード例

商品リストから価格が1000円以上のものを抽出し、価格の高い順に並べて名前だけを取り出す処理を例に示します。

```
// Java 8より前
List<Product> filtered = new ArrayList<>();
for (Product p : products) {
    if (p.getPrice() >= 1000) {
        filtered.add(p);
    }
}
filtered.sort(Comparator.comparingInt(Product::getPrice).r
eversed());
List<String> names = new ArrayList<>();
for (Product p : filtered) {
    names.add(p.getName());
}
```

```
// Stream APIを使った書き方
List<String> names = products.stream()
    .filter(p -> p.getPrice() >= 1000)
```

```
    .sorted(Comparator.comparingInt(Product::getPrice).rev
ersed())
    .map(Product::getName)
    .collect(Collectors.toList());
```

よく使う主なメソッドを次に示します。

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);

// filter：条件に合う要素だけ残す
List<Integer> evens = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());  // [2, 4, 6, 8, 10]

// map：各要素を変換する
List<Integer> doubled = numbers.stream()
    .map(n -> n * 2)
    .collect(Collectors.toList());  // [2, 4, 6, ...]

// reduce：要素を集約する
int sum = numbers.stream()
    .reduce(0, Integer::sum);  // 55

// count：要素数を返す
long count = numbers.stream()
    .filter(n -> n > 5)
    .count();  // 5
```

3.4.2 どんなケースで役に立つか

コレクションに対して複数の条件を組み合わせて処理したい場面で力を発揮します。「フィルタして変換して集計する」という一連の処理をひとつの流れとして読めるため、コードレビューでも意図が伝わりやすくなります。

また、`parallelStream()`を使うと処理を複数スレッドで並列化できます。データ量が多く、各要素の処理が独立している場合はパフォーマンス改善が期待できます。

```
long count = products.parallelStream()
    .filter(Product::isAvailable)
    .count();
```

3.4.3 補足情報・注意点

StreamはCollectionとは異なり、**一度しか消費できません**。一度終端操作（`collect`や`count`など）を呼び出したStreamを再利用しようとすると`IllegalStateException`が発生します。再利用が必要な場合は都度Streamを生成し直す必要があります。

また、`parallelStream()`は万能ではありません。スレッドの分割・結合にもコストがかかるため、要素数が少ない場合や処理が軽い場合はむしろ`stream()`より遅くなることがあります。ベンチマークをとったうえで使うことを推奨します。

3.5 Optional

`Optional<T>`は、値が存在するかもしれないし、存在しないかもしれないことを型で表現するコンテナクラスです。メソッドの戻り値として`null`を返す代わりに`Optional`を使うことで、呼び出し側に「結果がない可能性がある」ことを明示的に伝えられます。

3.5.1 コード例

```
// Optional.of：nullでない値をラップする（nullを渡すとNullPointerException）
Optional<String> opt1 = Optional.of("Hello");

// Optional.ofNullable：nullの可能性がある値をラップする
Optional<String> opt2 = Optional.ofNullable(maybeNull);

// Optional.empty：空のOptionalを生成する
Optional<String> opt3 = Optional.empty();

// orElse：値がなければデフォルト値を返す
String value = opt2.orElse("default");

// orElseGet：値がなければSupplierで生成する（遅延評価）
String value2 = opt2.orElseGet(() -> computeDefault());

// orElseThrow：値がなければ例外をスローする
String value3 = opt2.orElseThrow(() -> new RuntimeException("値がありません"));

// map：値があれば変換する（なければ空のOptionalを返す）
Optional<Integer> length = opt1.map(String::length);  // Optional[5]
```

```
// ifPresent：値があれば処理を実行する
opt1.ifPresent(s -> System.out.println("Found: " + s));
```

3.5.2 どんなケースで役に立つか

データベースやキャッシュから「あるかもしれない値」を取得するメソッドの戻り値型として適しています。戻り値がOptionalであれば、呼び出し側はnullチェックを忘れにくくなります。

```
// nullを返す旧スタイル
User user = userRepository.findById(id);  // nullかもしれない
if (user != null) {
    System.out.println(user.getName());
}

// Optionalを返す新スタイル
Optional<User> user = userRepository.findById(id);
user.ifPresent(u -> System.out.println(u.getName()));
```

3.5.3 補足情報・注意点

Optionalはあくまで「戻り値型」としての使用を想定して設計されています。**フィールドやメソッド引数にOptionalを使うのは推奨されません**。フィールドに使うと`Serializable`でなくなる問題もあります。

また、`opt.get()`は値が存在しない場合に`NoSuchElementException`をスローするため、`isPresent()`で確認してから`get()`するのは冗長です。`orElse`や`map`、`ifPresent`などを活用して、`get()`をなるべく使わないスタイルが好ましいでしょう。

3.6 Date Time API

Java 8では`java.time`パッケージとして新しい日時APIが導入されました。それまでの`java.util.Date`や`java.util.Calendar`はミュータブル（変更可能）で設計が複雑でした。新しいAPIはイミュータブルで直感的なメソッド名を持ち、スレッドセーフかつ使いやすいのが特徴です。

3.6.1 コード例

よく使うクラスとその操作例を示します。

```
// LocalDate：日付のみ（時刻なし）
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1990, 3, 15);
LocalDate nextWeek = today.plusWeeks(1);
long daysBetween = ChronoUnit.DAYS.between(birthday, today
);

// LocalTime：時刻のみ（日付なし）
LocalTime now = LocalTime.now();
LocalTime meeting = LocalTime.of(14, 30);

// LocalDateTime：日付と時刻（タイムゾーンなし）
LocalDateTime dt = LocalDateTime.of(2024, 3, 18, 10, 0);
LocalDateTime nextMonth = dt.plusMonths(1);

// ZonedDateTime：タイムゾーンあり
ZonedDateTime tokyo = ZonedDateTime.now(ZoneId.of("Asia/To
kyo"));
ZonedDateTime utc = tokyo.withZoneSameInstant(ZoneOffset.U
TC);

// DateTimeFormatter：フォーマット・パース
```

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern(
"yyyy/MM/dd");
String formatted = today.format(formatter);           // "
2024/03/18"
LocalDate parsed = LocalDate.parse("2024/03/18", formatter
);
```

3.6.2 どんなケースで役に立つか

日付の加算・減算、期間の計算、タイムゾーン変換など、日時に関するほぼすべての処理でDate Time APIが活用できます。従来の`Calendar`との比較で差が特に際立ちます。

「1ヶ月後の日付を求める」という処理を例に比較します。

```
// java.util.Calendar を使った書き方（Java 8より前）
Calendar cal = Calendar.getInstance();
cal.set(2024, 0, 1);  // 月は0始まり（0=1月、11=12月）のため直
感的でない
// cal.set(2024, Calendar.JANUARY, 1);  // 定数を使って月名で
指定することもできる
cal.add(Calendar.MONTH, 1);
Date nextMonth = cal.getTime();
```

```
// LocalDate を使った書き方（Java 8以降）
LocalDate nextMonth = LocalDate.of(2024, 1, 1).plusMonths(
1);
```

`Calendar`は月が0始まりという設計であり、`Calendar.JANUARY`のような定数を使うことである程度カバーできますが、直感的とは言いがたい面がありました。また`Calendar`はミュータブルなため、メソッドに渡した先で`add()`などを呼ばれると呼び出し元の値まで変わってしまうという問題もありました。

```
// Calendarのミュータブルな問題
Calendar cal = Calendar.getInstance();
Calendar anotherCal = modifyDate(cal);  // 内部でadd()されると呼び出し元のcalインスタンスも変わる

// LocalDateはイミュータブルなので安全
LocalDate date = LocalDate.now();
LocalDate anotherDate = modifyDate(date);  // 内部でplusMonths()しても元のdateインスタンスは変わらない
```

期間の計算も、`Calendar`では差分を自分で計算する必要がありましたが、`Period`や`ChronoUnit`を使うと宣言的に書けます。

```
// java.util.Calendar で日数差を求める（Java 8より前）
Calendar start = Calendar.getInstance();
Calendar end = Calendar.getInstance();
start.set(2024, 0, 1);
end.set(2024, 11, 31);
long days = (end.getTimeInMillis() - start.getTimeInMillis())
    / (1000 * 60 * 60 * 24);

// ChronoUnit を使った書き方（Java 8以降）
```

```
LocalDate start = LocalDate.of(2024, 1, 1);
LocalDate end = LocalDate.of(2024, 12, 31);
long days = ChronoUnit.DAYS.between(start, end);  // 365
```

3.6.3 補足情報・注意点

既存のシステムとの連携などで`java.util.Date`や`java.sql.Date`との変換が必要になることがあります。`Date.toInstant()`や`Date.from(Instant)`を使って変換できます。

```
// java.util.Date → LocalDateTime
Date oldDate = new Date();
LocalDateTime ldt = oldDate.toInstant()
    .atZone(ZoneId.systemDefault())
    .toLocalDateTime();

// LocalDateTime → java.util.Date
Date newDate = Date.from(ldt.atZone(ZoneId.systemDefault()
).toInstant());
```

3.7 インターフェイスのデフォルトメソッド

Java 8より前、インターフェースはメソッドの宣言だけを持ち、実装を含めることができませんでした。デフォルトメソッドは`default`キーワードを付けることでインターフェースにメソッドの実装を持たせる機能です。これにより、既存のインターフェースを実装しているすべてのクラスを修正することなく、インターフェースに新しいメソッドを追加できるようになりました。

3.7.1 コード例

```
interface Greeter {
    void greet(String name);

    // デフォルトメソッド
    default void greetLoudly(String name) {
        greet(name.toUpperCase());
    }
}

class FormalGreeter implements Greeter {
    @Override
    public void greet(String name) {
        System.out.println("Good day, " + name + ".");
    }
    // greetLoudlyはオーバーライド不要。そのまま使えます。
}

FormalGreeter g = new FormalGreeter();
g.greet("Alice");        // Good day, Alice.
g.greetLoudly("Alice"); // Good day, ALICE.
```

デフォルトから処理を変更したい場合はオーバーライドにより、実装クラス側で処理を変更できます。

```
class CasualGreeter implements Greeter {
    @Override
    public void greet(String name) {
        System.out.println("Hey, " + name + "!");
    }
```

```
    @Override
    public void greetLoudly(String name) {
        System.out.println("HEY, " + name.toUpperCase() +
"!!!");
    }
}
```

3.7.2 どんなケースで役に立つか

ライブラリが提供するインターフェースに後から新機能を追加したい場合に役立ちます。Java 8 で Stream API を導入する際、`Collection`インターフェースに`stream()`や`forEach()`メソッドをデフォルトメソッドとして追加することで、既存のすべての実装クラス（`ArrayList`、`LinkedList`など）を修正せずに済んでいます。

3.7.3 補足情報・注意点

複数のインターフェースが同名のデフォルトメソッドを持つ場合、実装クラスは必ずオーバーライドして解決しなければなりません。さもなければコンパイルエラーになります。

```
interface A {
    default void hello() { System.out.println("Hello from
A"); }
}

interface B {
    default void hello() { System.out.println("Hello from
B"); }
}
```

```
// コンパイルエラー：どちらのhelloを使うか曖昧
// class C implements A, B {}

// 解決策：明示的にオーバーライドする
class C implements A, B {
    @Override
    public void hello() {
        A.super.hello(); // どちらのデフォルトメソッドを呼ぶか明示できます
    }
}
```

3.8 アノテーション

Java 8ではアノテーションの適用範囲が適用方法についても修正が入り、より広範に利用できるようになりました。

3.8.1 型アノテーション

Java 8より前、アノテーションはクラス・メソッド・フィールド・パラメータなどの宣言にのみ付けられました。Java 8から、型が使われるあらゆる場所にアノテーションを付けられる「型アノテーション」が導入されました。

コード例

```
// ジェネリクスの型引数
List<@NonNull String> names;

// キャスト
```

```
String s = (@NonNull String) obj;

// new
Object obj = new @Initialized MyObject();

// throws句
void process() throws @Critical IOException;
```

どんなケースで役に立つか

`@NonNull`や`@Nullable`といったアノテーションを型が使われる場所に付けることで、静的解析ツール（SpotBugs や Checker Framework など）がより細かく nullability を検査できるようになります。

補足情報・注意点

型アノテーション自体はJava言語の構文上の機能であり、ランタイムに何かをするわけではありません。静的解析ツールと組み合わせることで初めて実用的な効果を発揮します。

3.8.2 反復アノテーション

Java 8 より前、同じアノテーションを同じ要素に複数回付けることはできませんでした。反復アノテーション（Repeating Annotations）は、同一のアノテーションを同じ宣言や型に複数回適用できる機能です。

コード例

反復アノテーションを定義するには、アノテーション自体に`@Repeatable`を付け、コンテナアノテーションを用意します。

```
// コンテナアノテーション
@interface Schedules {
    Schedule[] value();
}

// 反復可能なアノテーション
@Repeatable(Schedules.class)
@interface Schedule {
    String dayOfWeek();
    String time();
}

// 使用例：同じアノテーションを複数回付けられる
@Schedule(dayOfWeek = "Monday", time = "10:00")
@Schedule(dayOfWeek = "Friday", time = "15:00")
void weeklyMeeting() {
    // ...
}
```

Java 8より前は配列形式でまとめて書く必要があり、冗長でした。

```
// Java 8より前
@Schedules({
    @Schedule(dayOfWeek = "Monday", time = "10:00"),
    @Schedule(dayOfWeek = "Friday", time = "15:00")
})
void weeklyMeeting() {}
```

どんなケースで役に立つか

同じ種類の設定を複数指定したいケース、たとえばバリデーションルールの複数適用や、テストのパラメータ指定などで読みやすさが向上します。

補足情報・注意点

反復アノテーションは構文上のシンタックスシュガーです。コンパイラは複数の`@Schedule`をコンテナアノテーション`@Schedules`にまとめて処理します。リフレクションで取得する際は`getAnnotationsByType(Schedule.class)`を使うと反復アノテーションをまとめて取得できます。`getAnnotation(Schedule.class)`では取得できないので注意が必要です。

```
Schedule[] schedules = method.getAnnotationsByType(Schedule.class);
```

あとがき

立ち読み版はここまでです！

技術書典オンラインマーケット等で公開しているものは、立ち読み版の内容に加えてJava 9〜17の情報と付録のバージョン一覧が付いています。

ぜひ、ご確認ください。

著者紹介

kouno（こうの） X@hk_it7
サークル名：世迷言ラボ https://hk-it.hatenablog.com/

サブカル系のECサービスの開発を主にJavaで行なっています。主体的に働きかけて、越境しながら業務に取り組むタイプのプログラマです。

Java 10+a年振り返り （上巻） - 立ち読み版！！

2026年4月11日 初版発行 技術書典20

発行	世迷言ラボ
著者	kouno
連絡先	h.kono.it@gmail.com
印刷所	日光企画


世迷言ラボ
Yomaigoto Lab.