Java21 エッセンス

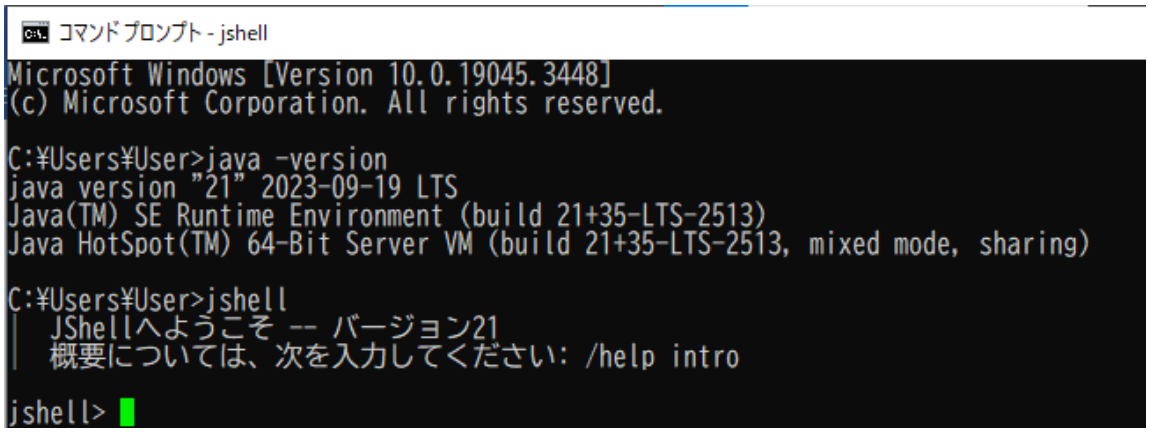
内容

はじめに

Java SE(Standard Edition)は Sun Microsystems 社が 2008 年に Java 6 をオープンソース化(GPL2)して以降、OpenJDK[1]で開発（商標は Oracle 社が保有）されています。Oracle 社が一部有償で提供している OracleJDK の他にも The Adoptium Working Group（旧 AdoptOpenJDK）[2]等複数の組織が無償で JDK のバイナリを配布していますが、どれも OpenJDK のソースを基にしています。

Java のバージョンは LTS(Long Term Support：長期サポート)版として長期（複数年）サポートされるものと、最新機能を取り込み続け一定期間後に次のバージョンで置換される非 LTS バージョン（Java9 以降、半年間隔で公開）があり、Java 8, 11, 17 に続く LTS として 2023 年 9 月に Java 21 が公開されました[3]。システム開発への利用に向けて注意点や新しい機能についても纏めます。

動作確認に使った OS、JDK、Jshell のバージョンやビルド番号は以下のものです。



1. Java 21 の危険なコード

初期（1990 年代）の Java はバイト（中間）・コードを逐次解釈しながら実行したり、メモリに空きがなくなったときに行う使用済オブジェクトの整理（GC:ガーベッジコレクション）の効率が今よりも悪く、性能に問題がありました。また、ブラウザのプラグインとしてネットワークで送る用途もありメモリ・サイズを少なくする必要から、源流である C 言語由来のプリミティブ型を残したり文字列をコンスタントプールで共用する等の効率化が行われました。Java 5 からはオートボクシング[4]というプリミティブ型(int,long...他)とそのラッパークラス(Integer,Long...他)の間を自動変換する機能（コンパイラでコード挿入）が付きましたが、これも便利さと同時に落とし穴にもなっています。

その後 JIT コンパイラや GC 並列化による高速化、大容量メモリを持つサーバでの利用が主体になることで性能問題は大幅に改善していますが、Java 言語仕様と Java 仮想マシン（JVM）仕様の当該部分は従来のままで残っており今後もこれらに関する注意が必要です。

---

[1] OpenJDK https://openjdk.org/　　GitHub https://github.com/openjdk/jdk

[2] The Adoptium Working Group（Eclipse Foundation） https://adoptium.net/

[3] Java SE Support ロードマップ https://www.oracle.com/jp/java/technologies/java-se-support-roadmap.html

[4] オートボクシング https://docs.oracle.com/javase/jp/1.5.0/guide/language/autoboxing.html

不具合が起こる原因と対策

Java 言語仕様や JVM 仕様を理解していないと想定外の動作になる場合があります。以下に特に注意が必要な事項を挙げます。
特に明示がないものは、Java 8 以降（またはそれ以前のバージョンから）Java 21 に至るまで同一の仕様になっています。

No.	種類	概要	対策／備考　他
1	String の比較	ソース中で文字リテラルを使うとコンパイラが重複する文字列をコンスタントプールに格納してここから参照するようにします。このため、以下のように動作します。 String a = "あ"; String b = "あ"; String c = new String("あ"); //ヒープに格納される (a==b)　// **true** コンスタントプール同士の比較 (a==c)　// **false** コンスタントプールとヒープの比較 https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-4.html	たまたま比較演算子(==)で期待どおり動作する場合がありますが、同一文字列でも生成の方法で比較結果が変わります。 ［対策］ String.equals()か compareTo()で文字列同士の比較を行う。 〔参考：String.intern()を使うと実行中にコンスタントプールに追加できます〕
2	ラッパークラス（Byte,Short,Integer,Long,...）の比較 ・Autoboxing/Unboxing ・-128~127 Integer.IntegerCache	①ラッパークラスと数値の比較 Integer intCls = 200;//暗黙の裡に Integer.valueOf(200)実行 (intCls == 200)//**true** 暗黙の裡に intCls.intValue()後比較 **※Java9 以降 new Integer()は非推奨になりました⇒valueOf** ②ラッパークラスどうしを比較 Integer intCls_1 = 200; //Integer.valueOf(200) Integer intCls_2 = 200; //Integer.valueOf(200) (intCls_1 == intCls_2); // **false** 数値変換せず参照を比較 ③Integer：-128～127 の比較（既存のオブジェクト使用） Integer intCls_1 = -128; Integer intCls_2 = -128; (intCls_1 == intCls_2);　// **true** キャッシュ同士を比較	プリミティブ型とラッパークラスの比較は自動的に行われる Autoboxing/Unboxing で String と同じ問題を起こします。 ①片方がプリミティブ型の場合、オブジェクトを数値変換(Unboxing)してから比較するため true になります https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.21.1 ②はオブジェクトが異なるので false で、③はキャッシュ済の Integer のオブジェクトを使うので同一の参照になります。 ［対策］ equals()か compareTo()で比較を行う。または、プリミティブ型に明示的に変換〔intValue(), longVlue()〕してから比較する。

No.	種類	概要	対策／備考　他
3	オブジェクトの大量生成（String、ラッパークラスの更新等）	オブジェクトを大量に作ると、ヒープの割り当てと解放（GC）の負荷が増します。Stgirng や Integer 等の数値型のラッパークラスは値を変えられないので文字列の連結や分割、置換を行う都度新しいオブジェクトが作られます。	ヒープに十分な空きがなくなると GC が発生します。GC は幾つかの手法から選択できますが、いずれを選んでもアプリの性能劣化は避けられません。更に大量になり GC でも空きができなくなると java.lang.OutOfMemoryError 例外で JVM が停止します。 ［対策］ 文字列の加工が必要な場合は java.lang.StringBuilder クラスを、更に複数のスレッドから参照する必要がある場合は java.lang. StringBuffer を使う。
4	引数と戻り値の保護 final オブジェクトの更新	引数と戻り値でメソッド間をやり取りされるオブジェクト（参照）は更新が可能です。以下コードを jshell で実行 **//final 付きの引数のオブジェクトの更新メソッドが呼べる** void modifier(final List<String> arg){ 　　arg.add("modifier add val"); } **//更に、final を付けた変数をパラメタで渡しても…** void tester(){ 　　final List<String> para = new ArrayList<String>(); 　　modifier(para); 　　System.out.println(para); } **(実行)パラメタで渡したオブジェクトは更新可能です** jshell> tester(); [modifier add val]	メソッドの宣言の引数に final 修飾子を付けたコード例を見ますが、final の効果は定義されたメソッド内に限定され引数に格納されている参照の書き替えを抑止するだけです。参照するオブジェクトの更新系メソッドの呼び出しが制限されることはありません。 ［対策］ 内容を更新されたくない場合は以下を使う。 ・int, long 等のプリミティブ型 ・Integer 等のラッパークラス、String や BigDecimal 等の書き換えができないオブジェクト ・Collection(Map, List, Set 他)の「変更不可能なビュー」 Java8：Collection coll= 　　Collections.unmodifiableCollection(Arrays.asList("a", "b", "c")); （又は）List<String> list = 　　Collections. unmodifiableList (Arrays.asList("a", "b", "c")); Java9：List<String> list = List.of("a", "b", "c"); ・Java16 の不変クラス record を使う

No.	種類	概要	対策／備考　他
5	private なインスタンス変数やクラス(static)変数の外部からの更新	インスタンス変数やクラス変数に private 修飾子が付いていても、参照を渡すと外部から更新が可能になります。 //---参照を受け取る TestClass クラス--- `package exmple;` `import java.util.List;` `class TestClass{` `  TestClass(StringBuilder sta, List<String> ins){` `  sta.append("＋更新");` `  ins.add("TestClass-add");` `  }` `}` //---参照を漏らしてしまった SampleClass クラス--- `package exmple;` `import java.util.*;` `public class SampleClass{` `  static final StringBuilder staVar = new StringBuilder("初期値");` `  private List<String> insVar = new ArrayList<String>();` `  public void callTest(){` `  var test = new exmple.TestClass(staVar, insVar);` `  System.out.println("SampleClass static  var:" + staVar);` `  System.out.println("SampleClass instance var:" + insVar);` `  }` `}` //jshell から SampleClass を実行 jshell> import exmple.SampleClass jshell> new SampleClass().callTest() SampleClass static var:初期値＋更新 SampleClass instance var:[TestClass-add]	final と同様に private も参照を格納している変数のアクセス制限をしているだけなので、参照が渡された先での更新を制御することはできません。 　定数を定義するのに static final を使っても外部のクラスが参照を得るとオブジェクトの種類によっては更新が可能になります。 ［対策］前項「引数と戻り値の保護」と同 内容を更新されたくない場合は以下を使う。 ・int, long 等のプリミティブ型 ・Integer 等のラッパークラス、String や BigDecimal 等の書き換えができないオブジェクト ・Collection(Map, List, Set 他)の「変更不可能なビュー」 Java8：Collections.unmodifiableCollection(Arrays.asList("a", "b", "c")); Java9：List.of("a", "b", "c"); ・Java16 の不変クラス record を使う

No.	種類	概要	対策／備考　他
6	定数とメソッド 　のインライン化 (static final 指定のリスク)	static final の修飾子が付いている定数フィールドは利用する側のクラスに取り込まれ（インライン化)、利用される側の内容が変わっても更新が反映されません。 //定数を宣言する（インライン化される）クラス public class InlinableClass{ 　　public static final String inlinableVar = "**初期値**"; 　　public static int inlinableMethod(int var){ 　　　　return **++var**; 　　} } //定数を使う（インライン化する）クラス public class InlinTakeinClass{ 　　public void tekeinMethod(){ 　　　　System.out.println(InlinableClass.inlinableVar); 　　　　System.out.println(InlinableClass.inlinableMethod(1)); 　　} } //ⅰ.定数を使うクラスの実行 jshell> new InlinTakeinClass().tekeinMethod() 初期値 2 //ⅱ.定数の内容を変更 public class InlinableClass{ 　　public static final String inlinableVar = "**更新値**";*//値変更* 　　public static int inlinableMethod(int var){ 　　　　return **var * 100**; *//処理変更* 　　} } //ⅲ.定数にアクセス…変更後の値が見える jshell> InlinableClass.inlinableVar $3 ==> "更新値" //ⅳ.再度インライン化側クラスを実行 jshell> new InlinTakeinClass().tekeinMethod() **初期値**　　←変数の値に変更の内容が反映されない！ 100　　・・・メソッドの戻り値は変更が反映されている	メソッドも条件を満たせば JIT コンパイラがインライン化の対象にします。メソッドをインライン化するか否かの条件は、コンパイラ制御オプションで指示することができます。 https://docs.oracle.com/en/java/javase/21/vm/writing-directives.html https://docs.oracle.com/javase/jp/17/docs/specs/man/java.html 静的コンパイラー(javac)は static final フィールドの値を参照元にコピーする最適化を行います。これは switch 文の case ラベルは定数のみという制約に関連し Java21 で解消されましたが、Java21 でも定数のインライン化は行われ言語仕様 13.4.9 に以下の記載が残っています。 『問題を回避する最善の方法は、本当に変更される可能性が低い値に対してのみ static 定数変数を使用することです。真の数学定数の場合を除き、ソース コードでは static 定数変数の使用を極力控えることをお勧めします』(原文を和訳) https://docs.oracle.com/javase/specs/jls/se21/html/jls-13.html#jls-13.4.9 ［対策］ ・ソースのリポジトリに対して変更を全検索して影響先を調べるか、依存性を管理できるビルドツールでリコンパイルして入れ替える ・コードで対策する場合は、static final の変数を直接参照するのではなくゲッターメソッドを経由して定数の値を取得するように実装する

No.	種類	概要	対策／備考　他
7	小数点付き数値の計算	Java には浮動小数点付きの数字を扱うための double 型と float 型がありますが、内部表現は二進数（IEEE754 規格）を使っています。小数点以下は $1/2^n$ で表現するので十進数の 0.1 等は扱うことができません。	丸めて近似させるので 0.1 と表示されるのですが、積算すると正確な値になりません。 ［対策］ 十進数の計算を行う場合は、java.math.BigDecimal を使い、文字列で初期値を与える。 正）new java.math.BigDecimal("0.1"); 誤）System.out.println(new java.math.BigDecimal(0.1)); は double(0.1d)として扱われ、以下が表示されます。 0.1000000000000000055511151231257827021181583404541015625
8	内部クラスの外側クラスの変数参照 （内部オブジェクトの外部提供で起こるリスク）	static を付けていない内部クラスには自動的に外側クラスへの「参照」を引数に持つコンストラクタが作られるので、内部クラスの参照を外部のクラスに渡すと外側のクラスオブジェクトを含めて GC の対象にならなくなります(注 1) （詳細は下記 URL-The Java™ Tutorials 参照） https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html	組み込みクラスでは Map インタフェースを実装している HashMap、TreeMap その他のクラスが内部クラス Map.Entry を外部に提供します。 ［対策］ Map.Entry のように、クラスの情報を別インタフェースで開示する用途以外では内部クラスを外部に提供しない
9	Web アプリ、マルチスレッドアプリのインスタンス変数更新	Jakarta サーブレット仕様 ver.6 には次の記述があります「サーブレット コンテナはサーブレット宣言ごとに 1 つのインスタンスのみを使用する必要があります。」 https://jakarta.ee/specifications/servlet/6.0/jakarta-servlet-spec-6.0#number-of-instances	Web アプリでは、インスタンス変数の更新を行っていると並行処理中のリクエストが干渉し、同一のリクエストに対する結果が変わる可能性があります。 マルチスレッドで動作するアプリは Web アプリでなくても同様の障害が発生します。 ［対策］ 作業用は局所変数で宣言。Web アプリの場合リクエストの情報保存が必要ならセッション情報に保存する

（注１）アプリケーション実行中に GC で解放できないオブジェクトとは GC ルート(roots)から辿っていくことができる全てのオブジェクトで、以下のものが GC ルートになります。

・main メソッドと main メソッドから直接・間接的に呼び出されて終了していないメソッドの局所変数、パラメータ
・静的(static)変数＝クラス変数（厳密にはクラスローダがアンロードする迄… https://docs.oracle.com/javase/specs/jls/se21/html/jls-12.html#jls-12.7）
・Java 以外のプログラムの呼出し(JNI)
・その他（JVM 実装により定義されたオブジェクト等）

例えば、Web アプリ等で List 型の static 変数を宣言して何かの情報を蓄積させると何かのオブジェクトは GC の対象にならずに OutOfMemoryError が発生するか、コンテナ（JVM）の再起動が必要になります。

※ Java のバージョンや JVM の実装によりメソッド領域／static 変数を確保する場所（PermGen/Metaspace で検索)が変わったり、適用する方式により GC のタイミングが変わります

※ クラスローダーは Java 9 のモジュール・システムのため構成と役割が変わりました。システムクラスローダーはアプリケーションクラスローダーと名前が変わり（ドキュメント上は混在している)、他はブートストラップクラスローダー、プラットフォームクラスローダーです

JDK 9 への移行：https://docs.oracle.com/javase/jp/9/migrate/toc.htm

Java 21 ClassLoader クラス： https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/ClassLoader.html#getSystemClassLoader()

2. Java 8 以後の新機能

2014 年に Java 8 がリリースされて以降、多くの機能が追加されたり強化されています。内容は GC の性能向上に関する実装や例外が発生した場合の情報収集・提供、OS スレッドに依存しない仮想スレッドの実装等の JVM の機能と、Java 言語仕様の拡張（既存メソッドの非推奨化含む）があります。

言語仕様拡張の方向性は、Java 8 のラムダ式や Stream の宣言型プログラミングと Java 10 の変数の型推論のような冗長性の軽減に向かっています。各種のライブラリと宣言型のプログラミングで先行している Python や JavaScript、Ruby 等のオブジェクト指向言語を参考に機能追加・拡張が進むと考えられます。

〔Python 3 の例〕次の 3 行で CSV ファイルを全件読み辞書型（キー：バリュー）のリストを作成できます

```
with open(self.conf_file, newline='') as conf:
    reader = csv.DictReader(conf)
    self.confdic = [row for row in reader]
```

※row for row in reader でファイルの全レコードを処理し、[ ]で囲むだけでリスト型になります。Read や Close 等の制御用のコードが必要ありません

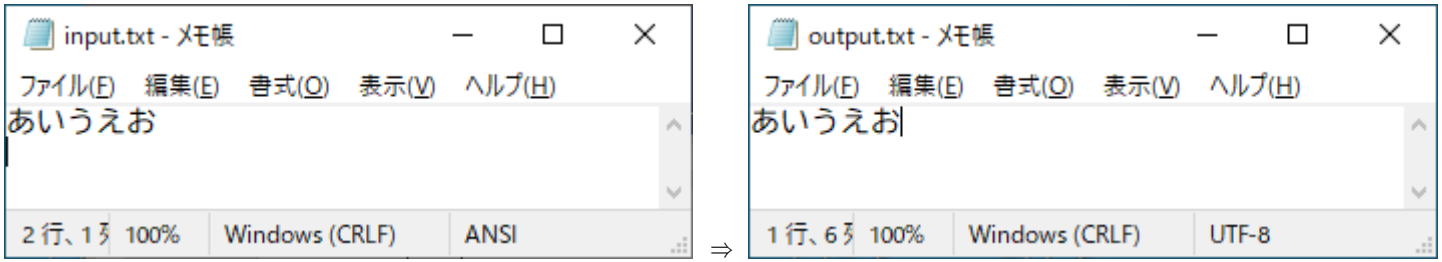
その他、新しく追加された record 型は設定した値を変更できない不変データとすることで一見不便ですが並列処理を前提とした DOP（データ指向プログラミング）に必要な機能だと解説する向きがあります。やはり新しく追加された仮想スレッドと併せて深層学習等の大量計算に対応することを考えているのかもしれません（知らんけど...）。

Java9～Java21 新機能（主に言語仕様）

Java の機能拡張は JDK Enhancement Proposal (JEP)として管理され、通番が振られています。(JEP インデックス：https://openjdk.org/jeps/0 ）

JEP の Preview, Experimental, Incubator という注釈が付いたものは実験や開発者のお試し用で、コンパイル時にフラグを立てないと機能が有効になりません。ここでは正式リリースになった機能のうち、主に言語仕様に関するものを纏めます。

No.	種類	概要
1	Java9 JEP 213:ミリングプロジェクトコイン （続く）	Java プログラミング言語に対して 5 つの小さな修正が提案されています。 ●ダイアモンド構文:左辺で要素型は明らかなため、右辺の要素型（<...>の中身）を省略できる List<String> list = new ArrayList<>(); ●不変のリスト、セットおよびマップを簡単に作成できるようになりました。 https://docs.oracle.com/javase/jp/9/core/creating-immutable-lists-sets-and-maps.htm 〔～Java 9〕List List<String> stringList = Arrays.asList("a", "b", "c"); stringList = Collections.unmodifiableList(stringList); 〔Java 9〕 List stringList = **List.of**("a", "b", "c"); 〔～Java 9〕Set Set<String> stringSet = new HashSet<>(Arrays.asList("a", "b", "c")); stringSet = Collections.unmodifiableSet(stringSet); 〔Java 9〕 Set<String> stringSet = **Set.of**("a", "b", "c"); 〔～Java 9〕Map Map<String, Integer> stringMap = new HashMap<String, Integer>(); stringMap.put("a", 1); stringMap = Collections.unmodifiableMap(stringMap); 〔Java 9〕 Map stringMap = **Map.of**("a", 1, "b", 2, "c", 3); //stringMap ==> {a=1, b=2, c=3}

<table>
<tr><th>No.</th><th>種類</th><th>概要</th></tr>
<tr><td></td><td>（JEP213 続き）</td><td>●try-with-resources ステートメント（Java9 で try 文の中で新しい変数を宣言する必要がなくなった）
import java.nio.charset.*
void twr() throws java.io.IOException {
BufferedReader br = Files.newBufferedReader(Paths.get("C:¥¥Users¥¥User¥¥Desktop¥¥input.txt"), Charset.forName("MS932"));
BufferedWriter bw = Files.newBufferedWriter(Paths.get("C:¥¥Users¥¥User¥¥Desktop¥¥output.txt"));//JEP 400: デフォルトで UTF-8
try (br;bw) { //br と bw は try 文と同期して close されます
bw.write(br.readLine()); //ループ処理省略
}
}
jshell>twr()


※（補足）input.txt と output.txt の両方に改行が Widnows(CRLF)と表示されていますが、BufferedReader は改行コードを読込まず BufferedWriter は改行コードを書き込みません（必要なら改行コードを書き込む処理が必要です…念の為）</td></tr>
<tr><td>2</td><td>JavaSE 10 JEP 286：
ローカル変数の型推論</td><td>var list = new ArrayList<String>(); //推測⇒List<String>
var stream = list.stream(); //推測⇒Stream<String>
//--構造がそれほど厳密ではない API でも同様（下例）--
var path = Paths.get("C:¥¥Users¥¥User¥¥Desktop¥¥input.txt");
var bytes = Files.readAllBytes(path);</td></tr>
</table>

No.	種類	概要
3	JavaSE 14 JEP 361: スイッチ式	●Java7 で switch 文が String 型の文字列に対応し、Java8 では int、short、char、byte、enum、String が使える String day = "水曜"; switch (day) { case "月曜": System.out.println("今日は一週間の始まりです。"); break; case "水曜": System.out.println("今日は定退日です。"); break; default: System.out.println("今日は" + day + "です。"); } 〔Java14〕case L ->"ラベルのいずれかの値に一致すると、ラベルの右側のコードのみが実行される switch (day) { case "火曜", "木曜", "金曜" -> System.out.println("今日は" + day + "です。"); case "月曜" -> System.out.println("今日は一週間の始まりです。"); case "水曜" -> System.out.println("今日は定退日です。"); } **今日は定退日です。** *<= String day = "水曜";と↑のコードを実行した結果* //switch ステートメント文を拡張して、式として使用できるようになりました //ブロックが必要な場合は yield ステートメントが、囲んでいる switch 式の値になります int j = switch (day) { case "月曜" -> 0; case "火曜" -> 1; default -> { int result = day.length(); yield result;} }; j ==> 2

No.	種類	概要
4	Java 15 JEP 378: テキスト ブロック	〔～Java15〕 "line 1¥nline 2¥nline 3¥n" または文字列リテラルの連結: "line 1¥n" + "line 2¥n" + "line 3¥n" 〔Java15〕「”」３つで囲むと、空白や改行コードを含んだままの文字列が作れます """ line 1 line 2 line 3 """ <応用例> 変数を埋め込むテンプレートとしても使うことができます String den_no = "10001"; String SQL = """ SELECT * FROM denpyo WHERE denpyono = '%s' ORDER BY "EMP_ID", "LAST_NAME"; """.formatted(den_no); jshell> System.out.println(SQL) SELECT * FROM denpyo WHERE denpyono = '10001' ORDER BY "EMP_ID", "LAST_NAME";

No.	種類	概要
5	Java 16 JEP 395: Records	不変データを集めるための Record クラスは、コンストラクター、データへのアクセサー、equals、hashCode、toString 等の標準的なメソッドを提供します。 record Point(int x, int y) { } //←これを書くだけで Record を継承したクラスが生成されます //--以下、生成されるコード-- class Point extends java.lang.Record { 　private final int x; 　private final int y; 　Point(int x, int y) { 　　this.x = x; 　　this.y = y; 　} 　int x() { return x; } 　int y() { return y; } 　public boolean equals(Object o) { 　　if (!(o instanceof Point)) return false; 　　Point other = (Point) o; 　　return other.x == x && other.y == y; 　} 　public int hashCode() { 　　return Objects.hash(x, y); 　} 　public String toString() { 　　return String.format("Point[x=%d, y=%d]", x, y); 　} } 定義される変数／メソッドは、Jshell から以下の手順で確認できます record Point(int x, int y) {}　//record の Point クラス作成 var p = new Point(3, 5);　　//Point クラスのインスタンス作成 p.getClass().getSuperclass() //Point の親クラス取得 **$*nn* ==> class java.lang.Record** p.getClass().getDeclaredMethods() //宣言メソッド一覧 **$*nn* ==> Method[5] {** **public final boolean Point.equals(java.lang.Object)** **, public final java.lang.String Point.toString()** **, public final int Point.hashCode()** **, public int Point.x()** **, public int Point.y() }** p.getClass().getDeclaredFields() //宣言フィールド一覧 **$*nn* ==> Field[2] { private final int Point.x, private final int Point.y }**

No.	種類	概要
6	Java 18 JEP 400: デフォルト UTF-8	Windows 環境はデフォルトで windows-1252/Cp1252（Windows ラテン文字-1）や windows-31j/MS932（Windows 日本語）が使われていましたが、Windows を含む全環境のデフォルトが UTF-8 に変わります。
7	Java 21 JEP 441: スイッチのパターンマッチング （続く）	●switch 文、式を強化します。 〔JEP 406(JDK 17)、JEP 420(JDK 18)、427(JDK 19)、433(JDK 20)で改良。Record パターン(JEP 405, 432, 440)と同期〕 //switch の条件（セレクタ）に「数値、文字列または enum 定数」以外の型が、case ラベルには定数以外のパターン //が書ける（パターンマッチング）ようになりました static void patternSwitchTest(Object obj) { 　String formatted = switch (obj) { 　　case Integer i -> String.format("int %d", i); //型がマッチしたら同時に引数を宣言して利用することができる 　　case Long l　　-> String.format("long %d", l); 　　case Double d　-> String.format("double %f", d); 　　case String s　-> String.format("String %s", s); 　　default　　-> obj.toString(); 　}; } //when オプションが使えるようになり、パターンが一致した結果をさらにテストできます static void testNew(Object obj) { 　switch (obj) { 　　case String s **when** s.length() == 1 -> ... 　　case String s　　-> ... 　　... 　} }

No.	種類	概要
	(JEP441 続き)	//条件の組合せ例。null 定数が使えるようになりました static void testStringEnhanced(String response) { switch (response) { case **null** -> { } case "y", "Y" -> { System.out.println("You got it"); } case "n", "N" -> { System.out.println("Shame"); } case **String** s **when** s.equalsIgnoreCase("YES") -> { System.out.println("You got it"); } case **String** s **when** s.equalsIgnoreCase("NO") -> { System.out.println("Shame"); } case **String** s -> { System.out.println("Sorry?"); } } }

No.	種類	概要
8	Java 21 JEP 440: Record パターン	// instatnceof で Record 型とマッチすることが確認できたら、同時に引数（int x, int y）を宣言して使用できます record Point(int x, int y) {} static void printSum(Object obj) { if (obj **instanceof Point(int x, int y)**) { System.out.println(**x**+**y**); } }
9	Java 21 JEP 431: シーケンス コレクション（SequencedCollection インターフェイス）	List(例: ArrayList、LinkedList)、SortedSet と NavigableSet の実装(例: TreeSet)、LinkedHashSet には要素の順序を使って操作するための次のメソッドが追加されました。 void addFirst(E)– コレクションの先頭に要素を挿入します void addLast(E)– 要素をコレクションの末尾に追加します E getFirst(); E getLast(); E removeFirst()– 最初の要素を削除して返します E removeLast()– 最後の要素を削除して返します //**使用例** var linkedHashSet = new LinkedHashSet<String>(List.of("aaa","bbb","ccc")); var list = new ArrayList<String>(List.of("xxxx","yyyy","zzzzz")); //〔～Java 21〕 var first = linkedHashSet.iterator().next(); var last = list.get(list.size() - 1); //〔Java 21〕 var first = linkedHashSet.getFirst(); var last = list.getLast();

以上

