**Java[文法]**

Last Update： 2025/01/30

京都職業能力開発促進センター

目次

[1. Java言語の概要 3](#_Toc189152850)

[2. Javaの基本文法 7](#_Toc189152851)

[3. 配列 40](#_Toc189152852)

[4. メソッド 51](#_Toc189152853)

[5. クラス 60](#_Toc189152854)

[6. パッケージ 77](#_Toc189152855)

[7. 文字列 83](#_Toc189152856)

[8. オブジェクト指向 90](#_Toc189152857)

[9. 抽象クラス 113](#_Toc189152858)

[10. インターフェイス 117](#_Toc189152859)

[11. 例外 127](#_Toc189152860)

[12. コレクション 134](#_Toc189152861)

[13. GUIアプリケーション 140](#_Toc189152862)

## Java言語の概要

### プログラムとは

プログラムとは、コンピュータに特定の処理を実行させるために記述された一連の命令のことを指します。プログラムは、コンピュータが理解できる形式で書かれた「命令書」のようなもので、さまざまなタスクを自動的に実行させるために作成されます。この「命令書」は人間が理解しやすいように、高水準言語と呼ばれる英語に近い文法や構造で記述します。これがプログラミング言語です。プログラミング言語で記述されたテキストファイルのことをソースコードと呼びます。ソースコードそのものをコンピュータが実行することはできません。コンピュータが実行するにはコンピュータが理解可能な低水準言語である「機械語」へ変換する必要があります。この変換方法には主に2つの方法があります。

#### インタプリタ方式

インタプリタ方式とは実行するタイミングで、プログラムを１行ずつ機械語に変換しながら実行する方式です。これにより、プログラムの実行中にエラーが見つかりやすく、修正も迅速に行えますが、全体的な実行速度は遅くなる傾向があります。PythonやJavaScript、PHPは、インタプリタ方式で実行されることが多いです。

ダイアグラム

自動的に生成された説明

#### コンパイラ方式

コンパイラ方式はソースコード全体を一度に機械語に変換し、実行ファイルを作成する方式です。この変換処理を「コンパイル」と呼びます。コンパイルが成功すると、実行可能なファイルが作成され、その後、実行ファイルを実行します。実行速度は高速ですが、ソースコード作成後、コンパイル作業を挟む必要があり実行までの手間が増えます。また、エラー等があるとその修正にも手間がかかります。C言語やC++などは、主にコンパイラ方式を採用しています。

時計 が含まれている画像

自動的に生成された説明

### Java言語の特徴

#### 徹底したオブジェクト指向

Javaはオブジェクト指向言語が一般的に定着した後に作られた言語です。そのため、初期のオブジェクト指向言語で明らかになった問題点を克服しています。例えば人間が理解することが難しくになる多重継承を禁止し、かわりにインタフェースを提供しています。また、初期に作られたオブジェクト指向言語は、従来の構造化言語のスタイルに合わせて、クラスを作成しなくてもプログラムを動作させれたのに対して、Javaは必ずクラスを作成する必要があります。また、 Javaの豊富な機能のほとんどがオブジェクト(クラス)として提供されます。

#### OSに依存しない

Javaはプログラムを一切変更することなく様々なOSやマシン上で動作するように設計されています。また、コンパイルして実行ファイル(クラスファイル)を作成しますが、この実行ファイルも一切の変更をすることなく様々なOS やマシン上で動作します。

これを実現しているのがJRE(Java Runtime Environment)です。JREは様々なOSやマシン上にJVM(Java Virtual Machine)とよばれるJava実行環境を提供します。この実行環境上でJavaを動かすことで、JavaはOSやマシンの違いを意識せずに開発を行うことができます。

ダイアグラム

自動的に生成された説明Javaでは、プログラムを実行するためにコンパイルをする必要があります。すなわち、プログラマの作業としてはコンパイラ方式と同じです。しかし、コンパイルして作成される実行ファイルはCPUが理解する機械語ではなく、Javaの仕様で定められた仮想の機械語(Javaバイトコード)です。そのため、CPUは直接実行できません。そのかわり、JavaバイトコードはJVMが理解することができ、JVMはそれを一つずつ機械語に変換しながら実行します。すなわち、JVMは実行ファイルをインタプリタ方式で機械語に翻訳しています。このようにコンパイルで共通の形式(Javaバイトコード)に一度変換し、次に各CPUやOSが理解できる機械語に変換するという2段階作業によってプラットフォームによる違いを吸収しています。

#### 豊富なJavaAPI

Javaは、ファイル操作、画像処理、ネットワーク通信、マルチスレッド、データベース操作など、さまざまな機能を利用できる豊富な仕組みを備えています。これらの機能は、JavaAPI (Java Application Programming Interface) と呼ばれ、オブジェクト指向の概念に基づいて設計されています。そのため、簡単に扱えるだけでなく、継承を活用することでJavaAPIを柔軟に拡張し、自分のニーズに合った形で改良することも可能です。JavaAPIは非常に幅広い分野をカバーしており、多くの機能が提供されていますが、それぞれの機能はパッケージという単位で整理されています。これにより、プログラマは自分が必要とする機能に関連するパッケージやクラスだけを学習するだけでよく、全てを把握する必要はありません。また、JavaAPIは定期的に新しい機能が追加されたり、既存の機能が改良されたりしており、最新技術への対応も充実しています。

JavaAPIを使うことで、複雑な処理も簡単に実現できるため、効率的な開発が可能となります。

#### さまざまな開発で利用

上記の豊富なJavaAPIにより、JavaはWebアプリケーション開発や、スマートフォンアプリ開発(Android)、業務システム開発など様々な開発現場で利用されています。また、オブジェクト指向言語の特徴でもある部品化により役割分担が容易となり大規模なシステム開発でも多く利用されています。歴史も古く(約30年)安定しているため、基盤システムや金融システムといった非常に高度なセキュリティが求められるシステムの開発では最有力候補の１つとなります。

#### Javaの開発環境

* JDK(Java Development Kit)

Javaプログラムの開発や実行を行うための各種ツール

* JRE(Java Runtime Environment)

Javaプログラムを実行するためのセット

* Java SE(Java Platform, Standard Edition)

Java言語でプログラミングする上で必要なライブラリ(元からあるプログラム)

* JVM(Java Virtual Machine)

Javaプログラムをコンパイルしてできた中間コードをOSに合わせて実行する

テキスト

自動的に生成された説明<https://docs.oracle.com/javase/jp/8/docs/index.html>

### IDE（統合開発環境）

IDE(Integrated Development Environment:統合開発環境)とは、ソフトウェア開発を行う上で必要なさまざまなツールが１つにまとまったものことです。コーディングのためのエディタやコンパイラ、テスト用のデバッガ、補完機能、ファイル管理機能などが含まれています。Javaの開発でよく利用されているIDEを紹介します。

#### IntelliJ IDEA(インテリジェイ　アイディア)

JetBrains社(チェコ共和国)が開発しているIDEで現在Java開発で最も利用されているIDEです。Java及び、Kotlinの開発に特化しているため非常に快適に開発が行えます。有料版のUltimateと無料版のCommunity Editionがありますが、Javaの基本の内容であれば無料版でも問題なく使用できます。補完機能が優秀です。

<https://www.jetbrains.com/products/compare/?product=idea&product=idea-ce>

#### Eclipse(エクリプス)

10年前くらいまではJavaで一番利用されているIDEでした。IBM社がJavaの開発ツールとして開発し、のちにオープンソース化され非営利組織Eclipse Foundationが開発を引き継いだ。動作が遅いですが、歴史が長いためインターネット上にトラブルシューティングの情報が多く存在します。完全無料で日本語化をしたい場合は「Pleiades All in One」からダウンロードすることをお勧めします。

<https://willbrains.jp/>

#### Visual Studio Code

Microsoftが開発しているIDEでさまざまな言語に対応しています。デフォルトでは必要最小限の機能しか搭載されていないため非常に軽量に動作します。上記2つのIDEとは異なり、Java開発に特化していないためJavaの開発で利用したい場合はVSCodeのダウンロードだけでは利用できず、別途JDKのダウンロードやプラグインの追加などの初期設定が必要です。

Javaに関するアンケート結果等

JetBrains社が実施したJava言語についてのアンケート

<https://www.jetbrains.com/ja-jp/lp/devecosystem-2023/java/#java_ide>

paizaプログラミング言語に関する調査（2023年版）

<https://www.paiza.co.jp/news/20231214/231214_survey_programming/>

## Javaの基本文法

### はじめてのJavaプログラム

【Hello.java】(ファイル名)

|  |
| --- |
| public class Hello {  public static void main(String[] args) {  System.*out*.println("こんにちは!!");  }  } |

#### コンパイル方法

プログラムを記述したファイルのことを一般的にソースファイルと呼びます。一般的な言語ではソースファイルの名前はプログラマが自由に決めることができますが、Javaではソースファイル名に制約があります。ソースファイルを作成したらコンパイルを行います。コンパイルをするには、次のコマンドを実行します。

|  |
| --- |
| >javac␣ファイル名.java |

※文字化けが起きる場合は文字コードを指定し、コンパイルを行います。

|  |
| --- |
| >javac␣-encoding␣"UTF-8"␣ファイル名.java |

#### コンパイルエラー

**テキスト

自動的に生成された説明**コンパイル時、プログラムにエラーがあるとそれが表示されるので、プログラムを修正して再度コンパイルします。例えば、次のようなエラーメッセージが出たとします。

最初にエラーのあるソースファイル名と行番号、エラーの種類が表示されます。上の例ではHello.javaファイルの4行目にエラーが発生してます。そして、エラーの詳細が以降に記述されます。エラー箇所が「^記号」で示されていますが、この位置はあくまで推定位置であり、正確ではありません。エラーを見つけるには、 示された位置の周辺を探す必要があります。

Javaは「型チェックが厳しい言語」であると言われています。Javaではすべての値、すべての変数、すべての式において型が決まっています。そして、型が整合しているかのチェックはコンパイル時に行われます。代入時など、型があっていないとエラーになりますので、注意してください。

コンパイルが終わるとクラスファイルが作成されます。クラスファイルの拡張子は「.class」です。今回のプログラムではHello.classというクラスファイルが作成されます。1行目のclass以降に記述されているHelloがクラス名です。

#### 実行方法

プログラムを実行するには、次のコマンドを実行します。

|  |
| --- |
| >java␣クラス名 |

javaコマンドの後に指定するのはファイル名ではなく、クラス名であることに注意してください。したがって、大文字小文字の区別があり、拡張子はいりません。

#### 実行時エラー

プログラムを実行しているときにもエラーが発生することがあります。例えば、次のようなメッセージを出してプログラムが終了します。

|  |
| --- |
| エラー: メイン・メソッドがクラスHelloで見つかりません。次のようにメイン・メソッドを定義してください。  public static void main(String[] args) |

コンパイルしたときに発見されたエラーをコンパイルエラー、実行したときに発生したエラーを実行時エラーと呼びます。コンパイル時はプログラムの文法上のミスを発見するだけです。したがって、文法的には間違いではなくても、実際に実行するとエラーになるということがよくあります。また、一般的に実行時エラーはミスを修正するのが難しいことが多いです。

上の例は、「mainメソッド」を定義していないクラスを実行したときに発生します。

#### コメント

コメントはコンパイラに無視され、プログラムの動作に影響を与えません。そのため、メモ書きや他のプログラマに読んで欲しい注意書きなどを記述することができます。

Javaのコメントは、次の3種類があります。

|  |  |
| --- | --- |
| 種類 | 意味 |
| /\* コメント \*/ | /\*と\*/で囲まれた部分をコンパイラは無視する。複数行に渡っていてもよい。 |
| // コメント | //以降から行末までの部分をコンパイラは無視する。 |
| /\*\* コメント \*/ | /\*\*と\*/で囲まれた部分をコンパイラは無視する。複数行に渡ってもよい。ただし、コメントはJavadocというツールを使ってドキュメントを作成する際に使用される。 |

【Hello.java】

|  |
| --- |
| */\*  このプログラムは文字を表示するプログラム  クラス名はHello  \*/* public class Hello {  public static void main(String[] args) {  System.*out*.println("こんにちは!!"); *//「こんにちは!!」と表示   //次のコードは「私の名前は○○です」と出力* System.*out*.println("私の名前は○○です");  } } |

#### クラス

プログラムの「public class Hello{ から }まで」は「Helloクラス」を作成しています。クラスとは、オブジェクト指向プログラミングでの基本的概念の一つで、オブジェクトの設計図を意味します。Javaプログラムでは、最低でも1つクラスを作成する必要があります。

クラスについては後ほど詳しく説明しますので、それまではclassの次に作成するクラスの名前を指定し、クラスの内容を{}括弧で囲むことに注意してください。

|  |
| --- |
| 書式：クラス定義 |
| public class Hello {  *//ここにクラスの内容を記述* } |

#### メインメソッド

プログラムの「public static void main(String[] args){ から }まで」がHelloクラスのmain()メソッドです。メソッドには、オブジェクトが処理する内容を記述します。メソッドの詳細については後述しますが、main()メソッドはjavaコマンド実行 時 にJVMが最初に呼び出す特別なメソッドです。

|  |
| --- |
| 書式：mainメソッド定義 |
| public static void main(String[] args) {  *//処理内容①;  //処理内容②;* } |

#### 文字列出力

System.out.println()メソッドは指定された内容を標準出力へ出力し、改行します。標準出力とは、それぞれのOSでの標準的な文字の出力先です。一般的には画面に文字が出力されますが、場合によってはファイルやプリンタ、ネットワークだったりします。

|  |
| --- |
| 書式：System.out.println()メソッド |
| System.*out*.println(出力する内容); *//println()メソッドは表示後改行あり* |

今回は「こんにちは!!」という文字列を指定しています。Javaでは文字列は必ず「””」記号で囲むことになっています。数値を出力する場合「""」は必要ありません。

|  |
| --- |
| 書式：System.out.print()メソッド |
| System.*out*.print(出力する内容); *//print()メソッドは表示後改行なし* |

print()メソッドは、出力内容を標準出力へ出力した後 に改行しません。

【Hello2.java】

|  |
| --- |
| public class Hello2 {  public static void main(String[] args) {  System.*out*.println("こんにちは!!"); *//改行あり* System.*out*.print("私の名前は"); *//改行なし* System.*out*.println("○○です"); *//改行あり*  System.*out*.println(20240101); *//数値は""なし* } } |

#### インデントと「;」記号

Javaではスペースとタブと改行をすべて空白として扱います。すなわち、プログラム中のスペースとタブと改行に違いはありません。唯一の例外は、文字列と//形式のコメントのなかにある改行です。文字列を改行するとコンパイルエラーになりますし、//形式のコメントでは改行はコメントの終了を意味します。

スペースとタブと改行が同じものとして扱われるため、自由な記述方法ができます。これをフリーフォーマットと呼びます。しかし、自由に記述できるからといって好き勝手なスタイルで記述されると他人には非常に読みづらいものとなります。特にグループで作業をする場合には、仕事の効率に影響するためプログラムの書き方のルールを定めるほどです。

そのため幾つかの違いはありますが、一般的なプログラムの書き方があります。そのなかでも最も重要なのがインデントです。インデントとは、プログラム中の範囲をはっきりするためにタブやスペースで行頭を一定間隔あけることです。

例えば、上記にプログラムではHello2クラスの範囲をはっきりさせるために行頭にインデントを入れています。さらにmain()メソッドの範囲をはっきりさせるためにもインデントを入れています。このようにすることで、プログラムは格段に読みやすくなります。

Javaではスペースとタブと改行が同じ物として扱われるので文末がわかりません。そのため、必ず文末を示すための「;」記号を使用します。

### 変数と型

#### 変数と型について

多くのプログラム言語では変数というものを使用します。変数とは値を一時的に入れておく入れ物のことです。変数に値を入れることを代入と呼びます。また、変数には必ず型があります。型は、変数がどんな種類のものかをあらわします。Javaでは、変数の型によってその変数に代入できるものとできないものが決まります。もし、できないものを代入しようとすると、エラーになります。

Javaの型を分類すると、基本データ型と参照型に分かれます。基本データ型と参照型は、同じように記述をしますが、変数に保存された値の扱いが異なります。はじめは基本データ型について学習していきます。参照型については後述します。以下に型の種類を表示します。

赤字で示した型はプログラムで頻繁に使用する型です。初めから全ての型を覚えると大変なので、まずは赤字で示した型を覚えるようにしてください。

【Variable.java】

|  |
| --- |
| class Variable {  public static void main(String[] args) {   *//変数宣言: 変数型 変数名* int height;   *//変数宣言と同時に = で初期値を代入* double weight = 50.5;  boolean bool = true;  char c = '字';   *//変数に値を代入* height = 150;   *//変数の値を出力 +演算子で変数の値と文字列を結合* System.*out*.println("身長は" + height + "cmです");  System.*out*.println("体重は" + weight + "kgです");  System.*out*.println(bool); *//booleanの値* System.*out*.println(c); *//charの値* } } |

#### 変数宣言

変数を使うには、まずその変数がどの型なのか明らかにする必要があります。それが変数宣言です。 変数宣言は次のように記述します。

|  |
| --- |
| 書式：変数宣言 |
| 型名 変数名; *//変数の型によりその変数で扱える値が決まる*  型名 変数名 = 初期値; *//変数宣言と同時に代入することで初期化* |

基本型には下の表の型名を指定します。

|  |  |  |  |
| --- | --- | --- | --- |
| 型の種類 | 型名 | データ長(byte) | 格納できる値の範囲 |
| 整数型 | byte  short  int  long | 1  2  4  8 | -128 ～ 127  -32768 ～ 32767  -21億 ～ 21億  -9京 ～ 9京 |
| 浮動小数点型 | float  double | 4  8 | ±7～8桁(少数点以下も含む)  ±16～17桁(小数点以下も含む) |
| 真理値 | boolean | 1 | true または false |
| 文字 | char | 16 | 1文字 |

#### 識別子

コンパイラが変数・クラス・インタフェース・メソッド・パッケージを区別するための名前であり、識別子には次の規則があります。

* 使える文字はUnicode文字(英字「A-Z」「a-z」 数字「0-9」 記号「\_」)
* 先頭の文字に数字は使用不可
* 大文字・小文字は区別される
* 予約語は使用不可
* データの意味が分かりやすい英単語を使用しましょう

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 予約語の一覧 | | | | | |
| abstract | const | finally | int | public | this |
| boolean | continue | float | interface | return | throw |
| break | default | for | long | short | throws |
| byte | do | goto | native | static | transient |
| case | double | if | new | strictfp | try |
| catch | else | implements | package | super | void |
| char | extends | import | private | switch | volatile |
| class | final | instanceof | protected | synchronized | while |

#### リテラル

リテラルとは、ソースコードにおいて一定の値を表すために用いられる表記法です。各型のリテラルを説明します。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 型の種類 | 型名 | リテラル | | 例 |
| 基本データ型 | char | 1 文字を「'」で囲んだもの | | 'a','あ' |
| int | 10進  8進  16進 | 「0～9」の並び  「0」で始まる「0～7」の並び  「0x」で始まる「0～9、A～F」の並び | 10  012  0xA |
| long | int型リテラルの末尾に「l」or「L」を追加したもの | | 10L |
| float | 小数 | 末尾に「f」または「F」を追加したもの | 0.5F |
| double | 「0～9」と「.」の並び | 0.5 |
| boolean | 「true」または「false」 | | true |
| 参照型 | String | 0字以上の文字列を「"」で囲んだもの | | "ABC" |
|  | 「null」何も代入されていないことを表す | | null |

#### エスケープシーケンス

エスケープシーケンスとは特殊な文字を表現するための記述方法です。「\」記号は文字コードの環境によりWindows環境であれば「¥」円記号、Unix環境であれば「＼」バックスラッシュ記号となります。

|  |  |
| --- | --- |
| エスケープシーケンス | 一般的な動作 |
| \n | 改行する |
| \t | タブスペースをあける |
| \' | 'を表示する |
| \" | "を表示する |
| \\ | \表示する |

【Escape.java】

|  |
| --- |
| public class Escape {  public static void main(String[] args) {   System.*out*.println("\\nは「\n」です");  System.*out*.println("\\tは「\t」です");   System.*out*.println("\\\'は「\'」です");  System.*out*.println("\\\"は「\"」です");  System.*out*.println("\\\\は「\\」です");  } } |

#### 演算子

演算子とは、データに対しての演算処理を行う記号や文字列のことを指します。そして演算の対象となるデータのことをオペランドといいます。

|  |  |
| --- | --- |
| 演算 | a = b + 100 |
| オペランド | a, b, 100 |
| 演算子 | =, + |

Javaの演算子には単項演算子、二項演算子、三項演算子の３種類の演算子があります。上記の例はオペランドが2つあるので「+」演算子は二項演算子です。

演算子によっては、同じ記号でもオペランドの数や種類により、意味が異なるので注意しましょう。

|  |  |
| --- | --- |
| 単項演算子 | 例) -a, a++, !a |
| 二項演算子 | 例) a + b, a = b |

#### 代入演算子

代入演算子は右辺のオペランドを左辺のオペランドに代入する二項演算子です。

|  |  |
| --- | --- |
| 代入演算子 | |
| = | 例) a = b ← bをaに代入する(bをaの値に上書きする) |

※代入の注意点

* 数値型を文字列型の変数に代入するなどの基本型と参照型の変数の代入はできない。
* 大きい範囲の型を小さい範囲の型に代入はできない。
* 小さい範囲の型を大きい範囲の型に代入は可能。

|  |
| --- |
| 例) 〇 double型　=　int型  × int型　　　　＝　double型 |

【Operator.java】

|  |
| --- |
| class Operator {  public static void main(String[] args) {   int intNum = -10; *//-は単項演算子* double doubleNum = 100.1;   System.*out*.println("intNum = " + intNum);  System.*out*.println("doubleNum = " + doubleNum + "\n");   *//=は二項演算子(右辺を左辺に代入) // intNum = doubleNum; //大きい型を小さい型に代入* doubleNum = intNum; *//小さい型を大きい型に代入* System.*out*.println("intNum = " + intNum);  System.*out*.println("doubleNum = " + doubleNum);  } } |

#### 算術演算子

算術演算子は数値型の値の演算を行う二項演算子です。

|  |  |  |
| --- | --- | --- |
| 算術演算子 | 使用例 | 説明 |
| + | x + y | x と y を加える |
| - | x - y | x から y を引く |
| \* | x \* y | x と y を掛ける |
| / | x / y | x を y で割る |
| % | x % y | x を y で割った余り |

※算術演算時の注意点

算術演算時にオペランドの型が異なる場合は大きい型に自動的に統一されてから演算が行われる。ただし、型変換は一時的であり代入時は代入される変数の型に変換されます。

|  |
| --- |
| 例） double型 = double型 \* int型(double型に型変換)  x = 1.1 \* 5  x = 1.1 \* 5.0  x = 5.5(代入) |

#### 複合代入演算子

代入演算子と算術演算子を組み合わせた複合代入演算子も用意されています。

|  |  |  |
| --- | --- | --- |
| 複合代入演算子 | 使用例 | 説明 |
| += | x += y | x = x + y |
| -= | x -= y | x = x - y |
| \*= | x \*= y | x = x \* y |
| /= | x /= y | x = x / y |
| %= | x %= y | x = x % y |

【Operator2.java】

|  |
| --- |
| class Operator2 {  public static void main(String[] args) {   System.*out*.println("3 \* 2.1 = " + (3 \* 2.1)); *//doubleに統一* System.*out*.println("10 % 3 = " + (10 % 3)); *//割ったあまり* System.*out*.println("10 / 3 = " + (10 / 3)); *//割った商* System.*out*.println("10 / 3.0 = " + (10 / 3.0)); *//doubleで演算* int x = 10;  System.*out*.println("x += 10 = " +( x += 10)); *// x=x+10;* } } |

#### 関係演算子

関係演算子は2つのオペランドの値を比較する二項演算子です。比較の結果としてboolean型の値を返します。つまり、その条件を満たす場合はtrue、満たさない場合はfalseとなります。

|  |  |  |
| --- | --- | --- |
| 関係演算子 | 使用例 | 説明 |
| == | x == y | xとyの値が等しい場合true となる |
| != | x != y | x と y の値が等しくない場合 true となる |
| < | x < y | x が y より小さい場合 true となる |
| <= | x <= y | x が y 以下の場合 true となる |
| > | x > y | x が y より大きい場合 true となる |
| ＞= | x >= y | x が y 以上の場合 true となる |

※値が等しいかどうか判断するときは「==」演算子を使います。「=」は代入演算子のため間違え

　　に注意してください。

#### 論理演算子

論理演算子はboolean型の値同士の比較を行う二項演算子です。比較の結果としてboolean型の値を 返します。

|  |  |
| --- | --- |
| 論理演算子 | 名前 |
| &&, & | 論理積、 AND 条件 |
| ||, | | 論理和、 OR 条件 |
| ^ | 排他的論理和、 XOR 条件 |
| ! | 否定、 NOT 条件 |

使用例と結果については、 次の表を参照してください。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | y | x && y | x || y | x ^ y | !x |
| true | true | true | true | false | false |
| true | false | false | true | true |
| false | true | false | true | true | true |
| false | false | false | false | false |

※&&及び||演算子については、左辺のオペランドの結果が明らかな場合、右辺のオペランドは

　　計算されず、&及び|演算子は、必ず右辺のオペランドも実行されます。

【Operator3.java】

|  |
| --- |
| class Operator3 {  public static void main(String[] args) {   System.*out*.println("3 < 4 は " + (3 < 4));  System.*out*.println("3 >= 4 は " + (3 >= 4));  System.*out*.println("3 != 3 は " + (3 != 3));   System.*out*.println( "true || false は " + (true || false));   int age = 30;  System.*out*.println("20代ですか？ :" + (20 <= age && age <= 29));  } } |

#### インクリメント演算子、デクリメント演算子

これらの演算子は、指定した変数に「+1」したり、「-1」したりする単項演算子です。

|  |  |  |
| --- | --- | --- |
| 演算子 | 使用例 | 説明 |
| ++変数 | ++x | 先に x に 1 加算する |
| 変数++ | x++ | 後で x に 1 加算する |
| --変数 | --x | 先に x を 1 減算する |
| 変数-- | x-- | 後で x を 1 減算する |

※前置と後置の違い

インクリメント演算子とデクリメント演算子は変数の前と後のどちらにも置くことができます。

単体で使用する場合は違いがありませんが、他の処理と組み合わせて使用すると結果が変わってきます。

|  |
| --- |
| x = 0;  a = ++x + 10; //加算前にインクリメントが行われるため a = 11;  b = x++ + 10; //加算後にインクリメントが行われるため b = 10; |

結果が分かりづらくなるため、インクリメント演算子とデクリメント演算子はできるだけ単体で使用しましょう。

【Increment.java】

|  |
| --- |
| public class Increment {  public static void main(String[] args) {  int x = 0;  int y = 0;  System.*out*.println("++x は " + ++x);  System.*out*.println("y++ は " + y++);   x = 0;  y = 0;  x = ++x + 10;  y = y++ + 10;  System.*out*.println("++x + 10 は " + x);  System.*out*.println("y++ + 10 は " + y);  } } |

#### 文字列連結演算子

文字列と文字列、または文字列と文字列以外の値を連結する二項演算子です。結果はString型の値となります。

|  |  |  |
| --- | --- | --- |
| 演算子 | 使用例 | 説明 |
| + | x + y | x と y を連結する |

「+」演算子はオペランドの型によって加算または文字列の連結という2つの演算を行います。

「＋」演算子は左から順番に処理されるため、加算と演算を同時に行う場合は記述順に注意してください。優先したい処理がある場合は()で囲みます。

【Join.java】

|  |
| --- |
| public class Join {  public static void main(String[] args) {  System.*out*.println(1 + 2 + " = 1 + 2");  System.*out*.println("1 + 2 = " + 1 + 2);  System.*out*.println("1 + 2 = " + (1 + 2));  } } |

#### キャスト演算子

型変換を行う単項演算子です。算術演算子や代入演算子において、オペランドの型が異なっていた場合、自動的に型変換が行われたり、 またコンパイルエラーが起こります。

自動的な型変換やコンパイルエラーを避けるためにはこの型に変換すると明示する必要があります。それがキャスト演算子です。

|  |  |  |
| --- | --- | --- |
| キャスト演算子 | 使用例 | 説明 |
| (型名) | (int)x | xの値をint型に変換(変数xの型に影響はない) |

キャスト演算子を利用すると本来はできない大きい型から小さい型への変換を行うことができます。ただし、範囲が大きい型を小さい型へキャストすると値の精度が落ちます。

(double型の値をint型に変換した場合は小数点以下の値が消える)

【Cast.java】

|  |
| --- |
| public class Cast {  public static void main(String[] args) {   int a = 0;  double b = 1.1;   a = (int) b; //double型の値をint型へキャスト   System.*out*.println("a = " + a);  System.*out*.println("b = " + b);  } } |

#### 演算子の優先順位

一つの式の中で、複数の演算子を利用できます。その場合どの演算子が先に演算されるかは各演算子の優先順位によって決まります。各演算子の優先順位を表にまとめました。

また、可読性を考えると、 括弧を使って優先順位を明らかにする方法をお勧めします。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 演算子 | オペランドの型 | 優先度 | 結合規則 | 演算 |
| ++ | 数値 | 1 | 右 | 前置または後置のインクリメント（単項） |
| -- | 数値 | 1 | 右 | 前置または後置のディクリメント（単項） |
| +,- | 数値 | 1 | 右 | 単項演算子のプラス、マイナス |
| ! | ブール型 | 1 | 右 | 論理的否定（単項） |
| (型) | 任意 | 1 | 右 | キャスト |
| \*,/,% | 数値 | 2 | 左 | 乗算、除算、剰余 |
| +,- | 数値 | 3 | 左 | 加算、減算 |
| + | String型 | 3 | 左 | 文字列の連結 |
| <,<= | 数値 | 5 | 左 | ～より小さい、～以下の判定 |
| >,>= | 数値 | 5 | 左 | ～より大きい、～以上の判定 |
| instanceof | オブジェクト | 5 | 左 | 型の比較 |
| == | 基本型 | 6 | 左 | 等しい(同じ値を持つ)の判定 |
| != | 基本型 | 6 | 左 | 等しくない(異なる値を持つ)の判定 |
| && | ブール型 | 10 | 左 | 条件節のAND |
| || | ブール型 | 11 | 左 | 条件節のOR |
| ?: | ブール型,  任意,任意 | 12 | 右 | 条件式（三項） |
| = | 変数,任意 | 13 | 右 | 代入 |
| \*=,/=,%=,  +=,-=, | 変数,任意 | 13 | 右 | 演算を伴う代入 |

#### 文字列の書式表示

画面表示を行う場合、今までは「System.out.println()」を利用していました。この方法では出力する値によっては表示が揃わないことがあります。表示を揃えたい場合は「System.out.printf()」を利用すると便利です。

「printf()」では第一引数で出力する文字列書式を指定し、第２引数以降で文字列に埋め込む値を指定します。

|  |
| --- |
| 書式:System.out.printf() |
| **int** num = 123;  **double** num2 = 0.456;  *//printfで出力(%dの位置へnumの値が埋め込まれて出力される*  System.***out***.printf(**"numの値は%d, num2の値は%fです。\n"**, num, num2); |

後から値を埋め込む位置に「%d」を設定します。この「%～」のことをプレースホルダといいます。プレースホルダは埋め込む値によって「～」の値が変化することに注意してください。

|  |  |
| --- | --- |
| 埋め込む値の型 | プレースホルダ |
| 整数 | %d |
| 文字列 | ％s |
| 少数 | %f |
| 真偽値 | ％ｂ |

表示桁数を指定したい場合は「％桁d」で桁数を指定します。「%3d」の場合は3桁で表示します。

少数点以下の桁を指定する場合は、「%.桁f」で指定します。

また、左寄せは「％-桁d」と「-」を指定します。

そのほかにもさまざまな書式を指定できます。詳しくはリファレンスを参照してください。

「printf()」は最後に改行されないため、改行したい場合は「\n」を追加してください。

また、全角文字を指定すると揃わない可能性があります。

【PrintfMain.java】

|  |
| --- |
| **public class** PrintfMain {  **public static void** main(String[] args) {   *//変数の宣言* **int** num = 123;  **double** num2 = 0.456;  String str = **"Hello"**;   *//printfメソッドで出力* System.***out***.printf(**"numの値は%dです。\n"**, num);  System.***out***.printf(**"num2の値は%.2fです。\n"**, num2);  System.***out***.printf(**"strの値は%sです。\n"**, str);  } } |

### 制御構文

#### キーボード入力

キーボードからの入力を受け付ける方法としてjavaではScanｎerクラスを使う方法があります。

Ｓｃａｎｎｅｒクラスを使うことで利用者からの入力データを受け取り、その値によってプログラムの処理を変化させることができます。Scannerクラスの使い方を以下に示します。

1. Scannerクラスのインポート(Scannerクラスが書かれている場所を指定)
2. Scannerクラスのインスタンス生成(Scannerクラスを利用することを宣言)
3. キーボードからの入力を読み取り、変数に代入

読み込む値の型によって使用するメソッドが異なる

|  |  |
| --- | --- |
| 読み込む型 | 利用するメソッド |
| 文字列(スペースまで) | next() |
| 文字列(1行分) | nextLine() |
| 整数 | nextInt() |
| 実数 | nextDouble() |

【ScannerMain.java】

|  |
| --- |
| import java.util.Scanner; *//(1)Scannerクラスのインポート* public class ScannerMain {  public static void main(String[] args) {   Scanner scanner = new Scanner(System.*in*); *//(2)Scannerクラスのインスタンス生成* System.*out*.print("名前を入力してください:");  String name = scanner.nextLine(); *//(3)nextLine()で1行分の文字を読み込み* System.*out*.print("年齢を入力してください:");  int age = scanner.nextInt(); *//(3)nextInt()で整数を読み込み* System.*out*.print("身長を入力してください(小数点以下1桁目で):");  double height = scanner.nextDouble(); *//(3)nextDouble()で実数を読み込み* System.*out*.println(name + "さん、こんにちは");  System.*out*.println("年齢は" + age + "才");  System.*out*.println("身長は" + height + "cm");  } } |

#### 乱数値の生成

乱数値とはプログラム上で利用するランダムな数値のことです。乱数はゲームや暗号化、テストデータの生成などさまざまなプログラムで利用されています。

プログラムで一般的に利用されている乱数は疑似乱数と呼ばれ、この値は完全なランダムではなく一定のアルゴリズムに基づいて生成された数値です。疑似乱数はある初期値(シード値)に基づいて計算されるため、同じシード値を使用すれば同じ順序で乱数が生成されます。疑似乱数のメリットは、効率的に生成でき、再現性を持たせることができる点です。真の乱数は生成するのに時間がかかるため、プログラム上ではあまり利用されません。ここでは疑似乱数の生成の仕方を紹介します。

乱数の生成方法は2つあります。

* Randomクラスを使用する方法
* Mathクラスのrandom()メソッドを使用する方法

Randomクラスを使用する方法から説明します。Randomクラスを使用するにはScannerクラスと同様の作業が必要になります。

1. Randomクラスのインポート(Randomクラスが書かれている場所を指定)
2. Randomクラスのインスタンス生成(Randomクラスを利用することを宣言)
3. nextInt()メソッドを使用し、乱数を生成

生成される乱数は0以上、nextInt()メソッドの()内で指定した整数値未満の値になります。

【RandomMain.java】

|  |
| --- |
| **import** java.util.Random;  **public class** RandomMain {  **public static void** main(String[] args) {   Random random = **new** Random();  **int** randomValue = random.nextInt(10); *// 0～9の乱数を生成* System.***out***.println(**"乱数1:"** + randomValue);   **int** randomValue2 = random.nextInt(10) + 1; *// 1～10の乱数を生成* System.***out***.println(**"乱数2:"** + randomValue2);  } } |

nextInt()メソッド以外にも指定範囲内の小数の値を出力するnextDouble()メソッドや、シードを指定するsetSeed()メソッドなどが用意されています。

※乱数生成前にsetSeed(Long値)を指定すると常に同じ乱数を生成できます。

Mathクラスのrandom()メソッドを使用する方法は、乱数を生成したい場所でMath.random()と記述するだけです。手軽な反面、「0.0以上1.0未満」の少数の値しか生成できません。特定の範囲内の乱数を生成したい場合は追加で計算が必要になります。

【MathRandomMain.java】

|  |
| --- |
| **public class** MathRandomMain {  **public static void** main(String[] args) {   **double** randomValue = Math.*random*(); *// 0以上1未満の乱数生成* System.***out***.println(randomValue);   **int** randomValue10 = (**int**)(Math.*random*() \* 10); *// 0以上10未満の乱数生成* System.***out***.println(randomValue10);   **int** randomValue100 = (**int**)(Math.*random*() \* 100) + 1; *// 1以上100未満の乱数生成* System.***out***.println(randomValue100);  } } |

複雑な乱数の生成や範囲指定が必要な場合はRandomクラスの方が適しています。手軽に乱数を使用したい場合はMath.random()が便利です。それそれの利用目的に合わせて使い分けてください。

#### 制御構文とは

Javaのプログラムは通常、順次実行によって上から順番に処理が実行されます。制御構文を利用すると、条件によって実行内容を変えたり、繰り返し実行したり、プログラムの流れを制御することができます。Javaには制御構文として以下のもの用意されています。

|  |  |  |
| --- | --- | --- |
| 制御 | 制御文 | 説明 |
| 条件分岐 | if, if-else, switch | 条件によって処理を変える |
| 繰り返し | for, while, do-while | 条件を満たすまで処理を繰り返す |
| 分岐 | break, continue, return | 無条件に処理の流れを変える |
| 例外処理 | try-catch-finally, throw | 例外処理を行う |

【IfMain.java】

|  |
| --- |
| import java.util.Scanner;  public class IfMain {  public static void main(String[] args) {   if(true){  System.*out*.println("trueです");  }  Scanner scanner = new Scanner(System.*in*);  System.*out*.print("整数を入力してください:");  int number = scanner.nextInt();  System.*out*.println("number = " + number);   if(number > 10){  System.*out*.println("10より大きいです");  }else{  System.*out*.println("10以下です");  }  if(number >= 0 && number < 10) {  System.*out*.println("0から9の間です");  } else if(number >= 10 && number < 20){  System.*out*.println("10から19の間です");  } else if(number >= 20 && number < 30){  System.*out*.println("20から29の間です");  } else {  System.*out*.println("30以上か、0未満です");  }  if(number % 2 == 0){  System.*out*.println("偶数です");  }else{  System.*out*.println("奇数です");  }  } } |

#### if文　/　if-else文

if文は条件を調べ、成立していたら処理を行い、成立していなかったら何も行わないという制御構文です。条件式の部分にはboolean型の結果を返す式を指定します。成立していない時の処理を記述したい場合はelse文を利用します。

比較演算子や関係演算子の結果はboolean型になりますので、これらの演算子を用いて条件式を作ります。

以下にプログラムの処理を分かりやすく表現するための図であるフローチャートと各構文の書式を示します。

|  |  |
| --- | --- |
| if文 | 書式 |
|  | **if**(*/\*条件式\*/*){  *//条件式がtrueの場合の処理①*  *//条件式がtrueの場合の処理②*  } |

|  |  |
| --- | --- |
| if-else文 | 書式 |
| 図形, 矢印  自動的に生成された説明 | **if**(*/\*条件式\*/*){  *//条件式がtrueの場合の処理①*  *//条件式がtrueの場合の処理②*  } **else**{  *//条件式がfalseの場合の処理①*  *//条件式がfalseの場合の処理②*  } |

if文は実行処理が１行のみの場合は{}を省いて記述することができます。ただし、処理の塊が分かりにくくなるため、基本的には{}を省略せずに記述することをお勧めします。

また、if文をより簡潔に記述できる文法として条件演算子(三項演算子)というものもあります。詳しくは個別に調べて利用してみてください。

#### if-else文の組み合わせ

if-else文を組み合わせることで、複数の条件によって処理を分けることができます。else-ifに続けていくつでも条件式を書くことができます。if文やif-else文をネストする方法もありますが、ネストする方法はプログラムの可読性が悪くなる恐れがあるためできるだけ避けた方が良いです。

|  |  |
| --- | --- |
| if-else文のネスト | 書式 |
|  | **if**(*/\*条件式①\*/*){  *//条件式①がtrueの場合の処理*  }**else**{  **if**(*/\*条件式②\*/*) {  *//条件式②がtrueの場合の処理*  }**else**{  **if**(*/\*条件式③\*/*) {  *//条件式③がtrueの場合の処理*  }**else**{  *//条件式①、条件式②、条件式③が*  *//全てfalseの場合の処理*  }  } } |

|  |  |
| --- | --- |
| else-if文 | 書式 |
| 図形, 多角形  自動的に生成された説明 | **if** (*/\*条件式①\*/*) {  *//条件式①がtrueの場合の処理*  } **else if** (*/\*条件式②\*/*) {  *//条件式②がtrueの場合の処理*  } **else if** (*/\*条件式③\*/*) {  *//条件式③がtrueの場合の処理*  } **else** {  *//条件式①、条件式②、条件式③が*  *//全てfalseの場合の処理* } |

else-ifで記述する場合は、先に記述した条件式がtrueになった場合、他の条件式がtrueであってもほかの条件式は処理されません。従って厳しい条件式を先に記述し、緩い条件式を後に記述すると、意図した処理結果を得られます。

#### 文字列の比較

数値の値が等しいかの判定には「==」演算子が利用でき、両方の値が等しい時にtrueとなります。文字列の値が等しいかどうかの判定では「＝＝」演算子の利用が推奨されていません。特定の条件下で同じ文字列であってもfalseと判定される可能性があるためです。詳細は後ほどの章で説明します。文字列(String型)の比較の場合は「equals()」メソッドを利用してください。

|  |
| --- |
| 書式：equals()メソッド |
| "文字列".equals("文字列") *//true or falseになる*  "文字列"の箇所は  → 文字列リテラル  → String型の変数  を指定できる |

【EqualsMain.java】

|  |
| --- |
| public class EqualsMain {  public static void main(String[] args) {   String str1 = "Hello";  String str2 = "Hello";   if (str1.equals("Hello")) {  System.*out*.println("str1と\"Hello\"は等しい");  } else {  System.*out*.println("str1と\"Hello\"は等しくない");  }   if (str1.equals(str2)) {  System.*out*.println("str1とstr2は等しい");  } else {  System.*out*.println("str1とstr2は等しくない");  }   if("Hello".equals(str1)) {  System.*out*.println("\"Hello\"とstr1は等しい");  } else {  System.*out*.println("\"Hello\"とstr1は等しくない");  }  } } |

#### switch文

条件式を複数記述する場合で、変数が特定の値と等しいかの判定を複数回行う場合はelse-if文を分かりやすく記述できる構文としてswitch文が準備されています。switch文は特定の変数がどのような値になるかをもとに、実行する処理を選択する構文です。例えば、else-if文で複数の条件式を記述する場合に、毎回同じ変数の値が特定の値と一致するかを判定するケースがあります。このような場合、switch文を使うことでコードがより簡潔かつ読みやすくなります。

|  |  |
| --- | --- |
| switch文 | 書式 |
|  | switch (*/\*変数値\*/*){  case */\*値1\*/*:  */\*処理1\*/* break;  case */\*値2\*/*:  */\*処理2\*/* break;  case */\*値3\*/*:  */\*処理3\*/* break;  default:  */\*処理4\*/* } |

switch文を利用するにあたって注意点がいくつかあります。

switch文では、各caseラベルの処理の最後にbreak文を記述する必要があります。無くてもコンパイルエラーにはなりませんが、これが無いと次のcaseラベルに処理が移ってしまうため意図しない処理結果になります。break文の書き忘れには十分注意してください。

ただ、この仕組みをわざと利用して処理を記述こともできます。

最後のdefaultは、値の中に該当するものがない時に選択されます。特に何もしないのであれば、省略可能です。

switch文で変数値に利用できる型はバージョンによって変わりますが、Java7以降であれば、「int型」「short型」「char型」「byte型」「enum型(列挙型)」「String型」が利用できます。「double型」などは利用できないことに注意してください。

【SwitchMain.java】

|  |
| --- |
| **public class** SwitchMain {  **public static void** main(String[] args) {   String week = **"水"**;   **switch** (week) {  **case "月"**:  System.***out***.println(**"月曜日"**);  **break**;  **case "火"**:  System.***out***.println(**"火曜日"**);  **break**;  **case "水"**:  System.***out***.println(**"水曜日"**);  **break**;  **case "木"**:  System.***out***.println(**"木曜日"**);  **break**;  **case "金"**:  System.***out***.println(**"金曜日"**);  **break**;  **case "土"**:  System.***out***.println(**"土曜日"**);  **break**;  **case "日"**:  System.***out***.println(**"日曜日"**);  **break**;  **default**:  System.***out***.println(**"曜日ではありません"**);  }  }  } |

String型の変数「week」の値によって表示される文字列が変わるプログラムです。

「week」の値が「水」の場合は、caseラベルが「水」のprintln()メソッドが実行されて「水曜日」という文字列が表示されます。表示後、break文によってswitchブロックを抜けます。「week」の値がどのcaseにも該当しない場合はdefaultの処理が実行されます。

【SwitchMain2.java】

|  |
| --- |
| **import** java.util.Scanner;  **public class** SwitchMain2 {  **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print(**"月を入力してください:"**);  **int** month = scanner.nextInt();   **switch** (month) {  **case** 3:  **case** 4:  **case** 5:  System.***out***.println(**"春です"**);  **break**;  **case** 6:  **case** 7:  **case** 8:  System.***out***.println(**"夏です"**);  **break**;  **case** 9:  **case** 10:  **case** 11:  System.***out***.println(**"秋です"**);  **break**;  **case** 12:  **case** 1:  **case** 2:  System.***out***.println(**"冬です"**);  **break**;  **default**:  System.***out***.println(**"1～12までの月を入力してください"**);  }  } } |

各caseラベルにbreak文を記述しないと、処理が次のcaseに移ります。この機能を利用して、上記のようなプログラムを作成可能です。「month」の値が「3,4,5」の場合は「春です」と表示し、「6,7,8」の場合に「夏です」と表示するようになります。

【SwitchMain3.java】

|  |
| --- |
| **import** java.util.Scanner;  **public class** SwitchMain3 {  **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print(**"月を入力してください:"**);  **int** month = scanner.nextInt();   *//Java 12以降の新しい書き方  /\*  caseラベルにカンマ区切りで複数の値を指定できる  -> はアロー演算子  break文が不要  \*/* **switch** (month){  **case** 3,4,5 -> System.***out***.println(**"春です"**);  **case** 6,7,8 -> System.***out***.println(**"夏です"**);  **case** 9,10,11 -> System.***out***.println(**"秋です"**);  **case** 12,1,2 -> System.***out***.println(**"冬です"**);  }  *//switch式*  */\*  switch文の代わりに式として使える  式の結果を変数に代入できる  switch式はbreak文が不要  defaultラベルは必須  \*/*  String str = **""**;  str = **switch** (month){  **case** 3,4,5 -> **"春です"**;  **case** 6,7,8 -> **"夏です"**;  **case** 9,10,11 -> **"秋です"**;  **case** 12,1,2 -> **"冬です"**;  **default** -> **"1～12までの月を入力してください"**;  };  System.***out***.println(str);  } } |

switch文を記述する方法はJavaのバージョンアップに合わせて色々な文法が追加されてきました。break文の記述ミスを防ぐためにbreak文の不要な文法が出てきたり、switch文を式として扱えるswitch式などが利用できます。これらの文法は利用するJavaのバージョンによっては使えない可能性もあるため、実行環境に注意して利用してください。

#### while文

while文は条件が成立している間処理を繰り返す構文です。同じ処理を複数回実行する場合などに利用します。if文と同じように条件式を指定し、条件式の結果がtrueの間はブロック内の処理を繰り返し実行します。結果がfalseになった場合に繰り返しを終了し、while文の下の処理に進みます。

|  |  |
| --- | --- |
| while文 | 書式 |
| 図形  中程度の精度で自動的に生成された説明 | **while** (*/\*条件式\*/*){  *//繰り返し実行する処理*  } |

【WhileMain.java】

|  |
| --- |
| **import** java.util.Scanner;  **public class** WhileMain {   **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);   String name = **""**;  *//while文の条件式・・・falseになるまで{}内の処理を繰り返す*  **while**(!name.equals(**"exit"**)){  System.***out***.print(**"名前を入力してください:"**);   name = scanner.nextLine();  System.***out***.println(name + **"さん、こんにちは"**);  }  } } |

#### do-while文

while文では繰り返しを開始する前に条件式の判定を行い、ブロック内の処理を実行するかを決めるため、条件によってはブロック内の処理を一度も実行しない可能性があります。do-while文ではブロック内の処理を実行した後、繰り返すかの判定をするため、必ず１度はブロック内の処理を実行するプログラムを記述することができます。

|  |  |
| --- | --- |
| do-while文 | 書式 |
|  | do{  *//繰り返し実行する処理*  } **while** (*/\*条件式\*/*); |

do-while文はwhile()の後ろに「;」を記述する必要があります。付け忘れに気をつけてください。

【DoWhileMain.java】

|  |
| --- |
| **import** java.util.Scanner;  **public class** DoWhileMain{  **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);   System.***out***.println(**"割り算を行います。"**);  System.***out***.print(**"割られる数を入力してください:"**);  **int** num1 = scanner.nextInt();   **int** num2;   **do** {  System.***out***.print(**"割る数を入力してください:"**);  num2 = scanner.nextInt();   } **while** (num2 == 0);   System.***out***.println(num1 + **" / "** + num2 + **" = "** + (**double**)num1 / num2);  } } |

#### for文

while文、do-while文は「条件が成立している間繰り返す」制御構文でした。for文は「指定回数分だけ繰り返す」制御構文です。どちらも繰り返し処理を行うためのループ構文ですが、繰り返す回数が決まっている場合はfor文を、繰り返す回数は決まっておらず、特定の条件まで繰り返す場合はwhile文をというように使い分けます。

for文は()内に「初期化処理」「条件式」「更新処理」を指定する必要があります。

・初期化処理： ループ回数を数えるカウンタ変数を初期化します。

・条件式: 繰り返し実行するかを判定する(trueで繰り返し、falseで終了)

・更新処理: カウンタ変数の値を更新(１周するごとにカウントアップ)

|  |  |
| --- | --- |
| for文 | 書式 |
|  | **for**(*/\*初期化\*/* ; */\*条件式\*/* ; */\*更新\*/*){  *//繰り返し実行する処理*  } |

「初期化」と「条件式」の間と「条件式」と「更新」の間に「;」が必要になるため注意してください。

【ForMain.java】

|  |
| --- |
| **public class** ForMain {  **public static void** main(String[] args) {  **for**(**int** i = 1; i <= 5 ; i++){  System.***out***.println(i + **"回目のループです"**);  }   **for**(**int** i = 5; i >= 1 ; i--){  System.***out***.println((6 - i) + **"回目のループです"**);  }  } } |

for文の入れ子(ネスト)とは、for文の中にさらにfor文を入れる構造のことを指します。この構造を使うことで、二重の繰り返し処理を行うことができ、例えば二次元配列の操作や、表形式のデータ処理などに便利です。

|  |  |
| --- | --- |
| for文の入れ子 | 書式 |
|  | **for**(*/\*初期化\*/* ; */\*条件式\*/* ; */\*更新\*/*){  **for**(*/\*初期化\*/* ; */\*条件式\*/* ; */\*更新\*/*){  *//繰り返し実行する処理*  }} |

【NestedForMain.java】

|  |
| --- |
| **public class** NestedForMain {  **public static void** main(String[] args) {  **for** (**int** i = 1; i <= 9; i++) { *// 1から9までの縦のループ* **for** (**int** j = 1; j <= 9; j++) { *// 1から9までの横のループ* System.***out***.print(i \* j + **"\t"**); *// 九九の結果を出力* }  System.***out***.println(); *// 横のループが終わったら改行* }  } } |

#### continue文

繰り返し構文内で使用する命令で、繰り返し処理をスキップしたいときに利用します。continue文を実行すると繰り返しブロック内の後の処理を実行せずに次の繰り返しへ進みます。通常、if文と同時に使用することが多く、特定の条件に当てはまった場合に処理をスキップさせる等の使い方をします。

|  |  |
| --- | --- |
| continue文 | 書式 |
|  | **for** (*/\*初期化\*/* ; */\*条件式\*/* ; */\*更新\*/*){   **if** (*/\*条件式\*/*) {  **continue**;  }  *//繰り返し実行する処理* } |

【ContinueMain.java】

|  |
| --- |
| **import** java.util.Scanner;  **public class** ContinueMain {  **public static void** main(String[] args) {  Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print(**"1～5までのスキップする数字を入力してください:"**);  **int** number = scanner.nextInt();   **for** (**int** i = 1; i <= 5; i++) {  **if** (i == number) {  **continue**; *// ここで処理をスキップ* }  System.***out***.println(i + **"回目のループです"**);  }  } } |

#### break文

switch文でも出てきましたが、break文は繰り返し構文内でも利用されます。繰り返し処理を強制終了したいときに使用します。continue文と同様に、if文と同時に使用することが多く、特定の条件に当てはまった場合に繰り返しを終了する等の使い方をします。

|  |  |
| --- | --- |
| break文 | 書式 |
|  | **for** (*/\*初期化\*/* ; */\*条件式\*/* ; */\*更新\*/*){   **if** (*/\*条件式\*/*) {  **break**;  }  *//繰り返し実行する処理* } |

【BreakMain.java】

|  |
| --- |
| **import** java.util.Scanner;  **public class** BreakMain {  **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print(**"1～5までの繰り返しを終える数字を入力してください:"**);  **int** number = scanner.nextInt();   **for**(**int** i = 1; i <= 5 ; i++){  **if**(i == number){  **break**; *// ここで処理を終了* }  System.***out***.println(i + **"回目のループです"**);  }  } } |

## 配列

### 配列とは

同じデータ型の複数の値を一つの変数としてまとめて扱うためのデータ構造です。配列を使うことで、複数のデータを効率的に管理・操作できるようになります。通常の変数は一つのデータしか格納できません。複数のデータ（たとえば、5人の学生の点数）を保存したい場合、それぞれに個別の変数を用意する必要があります。

【NoArray.java】

|  |
| --- |
| *//配列を使わないクラス* **public class** NoArray {  **public static void** main(String[] args) {   **int** score1 = 80;  **int** score2 = 90;  **int** score3 = 100;  **int** score4 = 70;  **int** score5 = 60;   **int** sum = score1 + score2 + score3 + score4 + score5;  **int** avg = sum / 5;   System.***out***.println(**"1番目の点数："** + score1);  System.***out***.println(**"2番目の点数："** + score2);  System.***out***.println(**"3番目の点数："** + score3);  System.***out***.println(**"4番目の点数："** + score4);  System.***out***.println(**"5番目の点数："** + score5);   System.***out***.println();  System.***out***.println(**"5人の合計点："** + sum);  System.***out***.println(**"5人の平均点："** + avg);   } } |

このようにたくさんの変数を用意するのは煩雑で管理も大変になります。また、変数ごとに変数名を付ける必要が出てきます。変数名が異なると繰り返し処理等も利用しにくいです。

配列を利用すると、複数のデータをすべて一つの配列変数で扱えるため、管理が非常に簡単になります。

また、配列の各値に対して、同じような処理を行う場合、for文を利用することで、処理を簡潔に記述することができます。

#### 配列の使い方

配列を利用する場合も変数と同様に配列変数の宣言が必要になります。ここでは生成する配列に格納する値の型と配列変数名を記述します。配列はすべての型で利用することができるため、「int型」や「String型」などの型を指定します。配列であることを示すためにインデックス演算子[]を利用します。

|  |  |
| --- | --- |
| 書式：配列変数宣言 | |
| 配列要素の型名[] 配列変数名;  *例)* int[] scores; | |  |  | | --- | --- | |  |  | | int[]型 |  | |  | scores | |

データの個数(要素数)を指定し、配列を生成します。配列の生成にはnew演算子を利用します。

配列生成時に各要素に要素番号(インデックス、添え字）が割り当てられます。

要素番号は「0　～　要素数-1」までになるので注意してください。

|  |  |
| --- | --- |
| 書式：配列の生成 | |
| 配列変数 = new 配列の型名[要素数];  *例)* scores = new int[5]; |  |

配列要素への代入や値の取得は[]内に要素番号を指定することで行うことができます。

|  |  |
| --- | --- |
| 書式：配列要素への操作 | |
| *//代入*  配列変数[要素番号] = 代入値;  *例)* scores[0] = 80;  *//取得*  配列変数[要素番号]  *例)* int score = scores[0]; |  |

配列は生成時に各要素に対してデフォルト値が入力されています。デフォルト値は型によって異なりますが、int型の場合は「0」が入っています。

配列の要素数を取得するには「length」を利用します。

|  |  |
| --- | --- |
| 書式：配列要素数の取得 | |
| 配列変数.length  *例)* scores.length; | **ダイアグラム  自動的に生成された説明**  要素数(5)  scores.length |

配列とfor文は非常に相性が良い組み合わせです。配列は連続してデータを扱う構文であり、for文はそのデータを順番に繰り返し処理するのに優れています。for文のカウンタ変数を繰り返し処理の中で配列の要素番号として利用すると配列の全ての値に対して処理を行えます。

|  |  |
| --- | --- |
| 書式：配列とfor文 | |
| **for** (**int** i = 0; i < scores.**length**; i++){    sum = sum + scores[i];  } | **ダイアグラム  自動的に生成された説明**  処理 |

【Array.java】

|  |
| --- |
| **public class** Array {  **public static void** main(String[] args) {   **int**[] scores = **new int**[5];  scores[0] = 80;  scores[1] = 90;  scores[2] = 100;  scores[3] = 70;  scores[4] = 60;   **int** sum = 0;   **for** (**int** i = 0; i < scores.**length**; i++) {  System.***out***.println((i + 1) + **"番目の点数："** + scores[i]);  sum = sum + scores[i];  }   **int** avg = sum / scores.**length**;   System.***out***.println();  System.***out***.println( scores.**length** + **"人の合計点："** + sum);  System.***out***.println( scores.**length** + **"人の平均点："** + avg);  } } |

### 配列の応用

#### 配列の応用

配列宣言時に各要素に入る値が決まっている場合は、配列宣言と同時に初期値を代入(初期化)することができます。配列宣言に代入演算子(=)で{}内に値を「,」区切りで列挙すると、指定の初期値が入った配列が作成されます。この時、配列の要素数は{}内の値の数になります。

|  |  |
| --- | --- |
| 書式：配列の初期化 | |
| 配列要素の型名[] 配列変数名 = {値, 値, …};  *例)* int[] scores = {80, 90, 100, 70, 60}; | **ダイアグラム  自動的に生成された説明** |

配列操作用のクラスとしてArrayｓクラスが用意されています。Arraysクラスを利用する場合はインポートが必要です。「java.util.Arrays」をインポートしてください。ArraysクラスのtoString()メソッドを利用すると配列の全要素を文字列形式で出力できます。

|  |  |
| --- | --- |
| 書式：配列の全要素を文字列で出力 | |
| Arrays.toString(配列変数); | **ダイアグラム  自動的に生成された説明**  [80, 90, 100, 70, 60] |

配列とfor文の相性が非常に良いため、専用の構文として拡張for文(for-each文)が用意されています。拡張for文を利用すると要素番号や要素数を意識せずに配列の要素を順番に1つずつ取り出して処理をすることができます。拡張for文では()内に条件式等は記述せず、「：」の右辺に配列変数、左辺に各要素を格納するための変数を記述します。左辺の変数には繰り返しごとに各要素の値が格納されます。

|  |  |
| --- | --- |
| 書式：配列の初期化 | |
| **for** (変数の型 変数名 : 配列変数) {   System.***out***.println(変数);  } | **ダイアグラム  自動的に生成された説明**   |  | | --- | | 80 | | 変数 | |

拡張for文は簡潔な記述で便利ですが、従来のfoｒ文の方が適している場合もあります。例えば、「何番目の値はいくつ」などの要素番号を利用するときや、特定の範囲の要素を処理するなどの場合です。用途によって通常のfor文と拡張for文を使い分けてください。

【Array2.java】

|  |
| --- |
| **import** java.util.Arrays;  **public class** Array2 {  **public static void** main(String[] args) {   *//String型の配列を宣言して初期化* String[] names = {**"田中"**, **"佐藤"**, **"鈴木"**, **"山田"**, **"加藤"**};   *//Arrays.toString()メソッドで配列を文字列に変換* String namesArray = Arrays.*toString*(names);   *//配列の中身を出力* System.***out***.println(**"names = "** + names);  System.***out***.println(**"Arrays.toString() = "** + namesArray);    System.***out***.print(**"拡張for文で配列の中身を出力 = "**);  *//拡張for文で配列の中身を出力* **for** (String name : names) {  System.***out***.print(name + **", "**);  }  } } |

配列を操作する際に頻繁に遭遇する例外(エラー)があります。

**ArrayIndexOutOfBoundsException・・・**

配列の範囲外の要素番号にアクセスしようとしたときに発生します。配列の要素番号は常に0から始まり、配列の長さ-1までが有効な範囲です。要素番号は注意して設定してください。

Array...配列

Index...要素番号

Out-of-bounds...領域外

Exception...例外

#### 配列の参照

**配列**と**変数**には重要な違いがあります。Javaでは、配列はオブジェクトとして扱われ、**参照型**の一種となります。変数と比較しながら、それらの違いを説明します。

変数はデータを格納するための名前付きの領域です。変数には基本型(int型やdouble型など)の値と、参照型(オブジェクトや配列)の値を格納できます。基本型は英語でプリミティブ型とも言います。

基本型の変数は直接その領域に値が格納されます。

|  |  |
| --- | --- |
| 書式：基本型(プリミティブ型) | |
| int num = 10;  int num2 = num; | |  | | --- | | 10 |   num   |  | | --- | | 10 |   num2 |

基本型の変数を他の変数に代入する場合は別々のメモリ領域を用意して値がコピーされます。

参照型の変数は実際の値を別のメモリ領域に格納し、変数の中にはそのメモリのアドレス値(参照値)が格納されます。

|  |  |
| --- | --- |
| 書式：参照型 | |
| int[] num = {10, 20, 30}  int[] num2 = num; |  |

参照型の変数を他の変数に代入する場合はアドレス値のみがコピーされます。従って、numとnum2は同じ配列を操作することになります。

通常の変数は、値そのものを保持しており、その変数を別の変数に代入すると、新しいメモリ領域に値がコピーされます。しかし、配列の場合は、データは**コピーされず**、データの場所（メモリの位置）を指し示すだけなので、複数の変数が同じデータを共有することになります。

(※Excelで説明)

参照型が値そのものを複製してコピーしない理由は、メモリの節約とパフォーマンス向上のためです。

メモリの節約

参照型が値を複製せずにデータの場所だけを参照することで、メモリの無駄な使用を防ぐことができます。もし、参照型がデータをそのまま複製する仕組みであれば、大きなデータ（例えば大量の数値を格納した配列や文字列など）をコピーするたびに大量のメモリを使ってしまいます。

パフォーマンス向上

もし参照型がデータを複製してしまうと、毎回大きなデータをコピーする処理が発生し、プログラムの実行速度が遅くなってしまいます。参照という仕組みを使うことで、データそのものをコピーする処理を行わないため、素早く処理が実行されるようになります。

【Array3.java】

|  |
| --- |
| **import** java.util.Arrays;  **public class** Array3 {  **public static void** main(String[] args) {   *//基本型のコピー* **int** num = 5;  **int** num2 = 10;   System.***out***.println(**"num = "** + num + **" num2 = "** + num2);   num2 = num;   System.***out***.println(**"num = "** + num + **" num2 = "** + num2);   num = 20;   System.***out***.println(**"num = "** + num + **" num2 = "** + num2);   *//参照型のコピー* **int**[] nums = {5, 10};  **int**[] nums2 = {20, 30};   System.***out***.println(**"nums = "** + Arrays.*toString*(nums) + **" nums2 = "** + Arrays.*toString*(nums2));   nums2 = nums;   System.***out***.println(**"nums = "** + Arrays.*toString*(nums) + **" nums2 = "** + Arrays.*toString*(nums2));   nums[0] = 100;   System.***out***.println(**"nums = "** + Arrays.*toString*(nums) + **" nums2 = "** + Arrays.*toString*(nums2));   } } |

#### nullとは

「null」とは参照型の変数が「何もない状態」や「値が存在しない」ことを示すために使用されます。つまり、変数が何も参照していない状態を表すためのものです。これにより、変数が初期化されていないことや、意図的に「空」であることを表現できます。

参照型の変数は初期化されない状態では自動的に「null」という値が割り当てられます。これにより、その変数が何も参照していない状態であることが分かります。String型等のオブジェクト型の配列では初期値として各要素に「null」が代入されています。

|  |  |
| --- | --- |
| 書式：null | |
| String str = **"ABC"**; System.***out***.println(str);  str = **null**; System.***out***.println(str); |  |

nullの注意点

nullを使う場合、注意しなければならないのが、値がnullである変数に対して操作を行うとエラーが発生する可能性があるという点です。このエラーは「NullPointerException」と呼ばれ、nullの状態のオブジェクトやメソッドにアクセスしようとしたときに発生します。

変数にnullが入っている可能性がある場合は、必ず「nullチェック」を行うようにしてください。

変数にnullを代入することができるのは参照型の場合のみです。int型などの基本型の変数にはnullを代入することはできません。

参照型の変数にnullを代入すると参照されていたオブジェクトは誰からもアクセスできないゴミとしてメモリ上に残り続けます。これを掃除する機能としてJavaにはガベージコレクションというしくみが用意されています。ガベージコレクションはメモリ上で、どの変数からも参照されなくなった領域を自動的に探し出して削除してくれます。

【Null.java】

|  |
| --- |
| **public class** Null {  **public static void** main(String[] args) {   *//String配列は初期値がnull* String[] names = **new** String[3];   **for** (**int** i = 0; i < names.**length**; i++) {  System.***out***.println((i + 1) + **"番目の名前："** + names[i]);  }  System.***out***.println();   names[0] = **"田中"**;  names[1] = **"佐藤"**;  names[2] = **"鈴木"**;   **for** (**int** i = 0; i < names.**length**; i++) {  System.***out***.println((i + 1) + **"番目の名前："** + names[i]);  }  System.***out***.println();   *// nullを代入* names[1] = **null**;   **for** (**int** i = 0; i < names.**length**; i++) {  System.***out***.println((i + 1) + **"番目の名前："** + names[i]);  }  System.***out***.println();   **for** (**int** i = 0; i < names.**length**; i++) {  *//nullの場合処理をスキップ* **if** (names[i] == **null**) {  **continue**;  }  *//nullの場合はNullPointerExceptionが発生* System.***out***.println((i + 1) + **"番目の文字数："** + names[i].length());  }  } } |

#### 多次元配列

**多次元配列**とは、配列の中にさらに配列が含まれている構造を持つ配列のことです。これは、**複数の次元**（軸）でデータを表現できるため、2次元以上のデータを扱いたい場合に便利です。特に、**行列（マトリックス）や表形式のデータ**など、複雑なデータ構造を扱う際に使われます。

通常の配列（1次元配列）は、一列にデータを並べて管理しますが、2次元配列で、これは「行」と「列」でデータを管理するものです。

例えば、学校の座席表やチェスの盤面など、縦横の両方向にデータを並べるときに2次元配列が役立ちます。

|  |  |
| --- | --- |
| イメージ | 実際の格納方法 |
| |  |  |  |  | | --- | --- | --- | --- | | scores | [0] | [1] | [2] | | [0] | 80 | 60 | 70 | | [1] | 90 | 70 | 60 | | [2] | 50 | 50 | 70 | |  |

|  |
| --- |
| 書式：多次元配列 |
| *//多次元配列の宣言*  配列の型[][] 配列変数名 = **new** 配列の型[行数][列数];  *//多次元配列に値を代入* 配列変数名[行の値][列の値] = 値;  *//多次元配列の初期化* 配列の型[][] 配列名 = {  {値, 値, 値},   {値, 値, 値} }; |

【Array4.java】

|  |
| --- |
| **import** java.util.Arrays;  **public class** Array4 {  **public static void** main(String[] args) {   *//多次元配列を宣言* **int**[][] scores = **new int**[3][3];   *//多次元配列に値を代入* scores[0][0] = 80;  scores[0][1] = 90;  scores[0][2] = 100;  scores[1][0] = 70;  scores[1][1] = 60;  scores[1][2] = 50;  scores[2][0] = 40;  scores[2][1] = 30;  scores[2][2] = 20;   *//多次元配列の初期化* **int**[][] scores2 = {  {80, 90, 100},  {70, 60, 50},  {40, 30, 20}  };   *//多次元配列の値を出力* **for** (**int** i = 0; i < scores.**length**; i++) {  System.***out***.println((i + 1) + **"番目の数学の点数："** + scores[i][0]);  System.***out***.println((i + 1) + **"番目の国語の点数："** + scores[i][1]);  System.***out***.println((i + 1) + **"番目の英語の点数："** + scores[i][2]);  System.***out***.println();  }   *//多次元配列をfor文の入れ子で出力* **for** (**int** i = 0; i < scores2.**length**; i++) {  **for** (**int** j = 0; j < scores2[i].**length**; j++) {  System.***out***.println((i + 1) + **"番目の科目"** + (j + 1) + **"の点数："** + scores2[i][j]);  }  System.***out***.println();  }   *//多次元配列を拡張for文の入れ子で出力* **for** (**int**[] score : scores2) {  **for** (**int** i : score) {  System.***out***.print(i + **" "**);  }  System.***out***.println();  }  System.***out***.println(Arrays.*deepToString*(scores2));  } } |

## メソッド

### メソッドとは

今までのプログラムはすべてmainメソッド内に実行する処理を記述して実行していました。

「メソッド」機能を利用すると、特定の処理をmainメソッド外に分離することができます。メソッドとして処理を分離すると、任意のタイミングで何度でも呼び出して使うことができるため、同じような処理を何度も書く必要がなくなり、再利用性が高まります。また、処理を特定の役割ごとにメソッドとしてまとめることで、後から修正する際に、影響範囲を限定でき、修正や拡張がしやすくなります。プログラムの可読性(見やすさ,読みやすさ)も向上します。

【NoMethod.java】

|  |
| --- |
| **import** java.util.Random; **import** java.util.Scanner;  **public class** NoMethod {   **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);   System.***out***.print(**"inputNum:"**);  **int** inputNum = scanner.nextInt();   System.***out***.print(**"inputNum2:"**);  **int** inputNum2 = scanner.nextInt();   **int** result = 1;   **for** (**int** i = 0; i < inputNum2; i++) {  result \*= inputNum;  }  System.***out***.println(inputNum + **"の"** + inputNum2 + **"乗 = "** + result);   System.***out***.println(**"ランダムな整数を生成します"**);  Random random = **new** Random();  **int** randomNum = random.nextInt(10);  **int** randomNum2 = random.nextInt(10);   System.***out***.println(**"randomNum = "** + randomNum);  System.***out***.println(**"randomNum2 = "** + randomNum2);   **int** result2 = 1;   **for** (**int** i = 0; i < randomNum2; i++) {  result2 \*= randomNum;  }  System.***out***.println(randomNum + **"の"** + randomNum2 + **"乗 = "** + result2);  } } |

|  |
| --- |
| 書式:メソッド定義（※mainメソッドがあるクラス内で定義する場合） |
| **public static** 戻り値の型 メソッド名(引数の型 引数名, ...) {  *// 処理* } |

|  |
| --- |
| 書式:メソッド呼び出し |
| メソッド名(引数, ...); |

【Method.java】

|  |
| --- |
| **import** java.util.Random; **import** java.util.Scanner;  **public class** Method {  **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print(**"inputNum:"**);  **int** inputNum = scanner.nextInt();  System.***out***.print(**"inputNum2:"**);  **int** inputNum2 = scanner.nextInt();   *//べき乗の計算を行うメソッドを呼び出す  pow*(inputNum, inputNum2);   System.***out***.println(**"ランダムな整数を生成します"**);  Random random = **new** Random();  **int** randomNum = random.nextInt(10);  **int** randomNum2 = random.nextInt(10);   *//べき乗の計算を行うメソッドを呼び出す  pow*(randomNum, randomNum2);  }   *//べき乗の計算を行うメソッドを定義* **public static void** pow(**int** num, **int** num2) {  **int** result = 1;   **for** (**int** i = 0; i < num2; i++) {  result \*= num;  }   System.***out***.println(**"num = "** + num);  System.***out***.println(**"num2 = "** + num2);  System.***out***.println(num + **"の"** + num2 + **"乗 = "** + result);  } } |

#### メソッドの定義

メソッド定義はmainメソッドの外で、classの内に記述する必要があります。メソッドの記述順は実行結果に影響がありません。プログラムが実行されたら、最初にmainメソッドが実行され、mainメソッド内で呼び出されたときに初めて指定されたメソッドが実行されます。メソッドを呼び出す場合、mainメソッドの処理は中断され、呼び出したメソッドの処理が開始し、処理が終了すると再び呼び出した位置からmainメソッドが再開します。

メソッドを定義する際の「public static」については後の章で説明するため今はこの文字列が必要だということだけ覚えておいてください。

戻り値の型

「戻り値」とはメソッドの処理が終了したタイミングで呼び出し元に戻す値のことです。処理結果等を伝える役割があります。定義の初めに、この戻り値の型を指定する必要があります。整数値を返したい場合は「int型」を、文字列を返したい場合は「String型」などを指定します。戻り値がない場合は「空」を意味する「void」を指定します。

|  |  |
| --- | --- |
| 書式:戻り値の型 | |
| *//戻り値がない場合*  **public static void メソッド名(){**  *//処理内容*  **}** | *//戻り値がint型の場合*  **public static int メソッド名(){**  *//処理内容*  **}** |

メソッド名

「メソッド名」は任意の文字列を指定できます。メソッド内でどんな処理をしているのか分かりやすい文字列にしてください。頭文字は小文字にします。

引数

「引数」とはメソッドを呼び出すときに一緒に渡す値のことです。メソッドに何かしら処理をさせる場合に材料となる値も一緒に渡します。その材料を元に処理が実行され結果が戻り値として返ってきます。引数は()の中に渡される値を受け取る変数として用意します。引数の型は渡される値と同じ型の必要があり、型が異なる場合はエラーになります。

引数が複数ある場合は「,」区切りで複数個指定できます。引数はそのメソッド内でのみ利用できる変数になります。引数が無い場合は()の中に何も記述しません。

|  |  |
| --- | --- |
| 書式:引数 | |
| *//引数がない場合*  **public static void メソッド名(){**  *//処理内容*  **}** | *//引数がint型1つ,double型1つの場合*  **public static int メソッド名(int a, double b){**  *//処理内容*  **}** |

メソッドの実際の処理内容は｛｝ブロック内に記述します。

メソッドの呼び出し

定義したメソッドを呼び出すには、mainメソッド内で「メソッド名(引数, ...)」と記述します。引数が無い場合は()の中を空にしてください。引数が複数ある場合は「,」区切りでメソッド定義で指定した数と型と型の順番で記述する必要があります。順番や型が間違っているとエラーになります。

#### return文

メソッドから戻り値を返す場合はreturn文を記述します。returnの後ろに指定した値が戻り値となり、呼び出し元に帰ります。メソッド定義で最初に指定した型と戻り値は一致していなければなりません。return文は複数記述することもできますが、returnが実行されるとすぐに処理が呼び出し元へ返るため、return文の後ろに処理を記述しても実行されません。

呼び出し元で戻り値を受け取るにはメソッド呼び出しを「＝演算子」の右辺に記述し、左辺に受け取るための変数を指定します。

【ReturnMethod.java】

|  |
| --- |
| **import** java.util.Scanner;  **public class** ReturnMethod {  **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);  System.***out***.print(**"inputNum:"**);  **int** inputNum = scanner.nextInt();  System.***out***.print(**"inputNum2:"**);  **int** inputNum2 = scanner.nextInt();  System.***out***.print(**"inputNum3:"**);  **int** inputNum3 = scanner.nextInt();   *//最大値を求めるメソッドを呼び出し、int型のmax変数でも戻り値を受け取る*  **int** max = *max*(inputNum, inputNum2, inputNum3);   System.***out***.println(**"最大値は"** + max + **"です"**);  }   *//３つの整数を受け取り、最大値を返すメソッド* **public static int** max(**int** num, **int** num2, **int** num3) {   **int** max = num;   **if** (num2 > max) {  max = num2;  }  **if** (num3 > max) {  max = num3;  }   **return** max;  } } |

#### スコープ

変数は宣言した場所によって利用できる範囲が変わります。利用可能範囲のことをスコープと呼びます。

フィールド(クラス変数)

クラスの直下、メソッドの外で宣言された変数はそのクラス内であればどこでも利用できる変数になります。このような変数をフィールド（クラス変数）と呼びます。

ローカル変数

メソッドの内側、if文やfor文などの特定の{}ブロック内で宣言された変数をローカル変数と呼びます。ローカル変数は、宣言されたブロックの外では利用できず、処理がブロック外に移ったタイミングで削除されます。

ローカル変数のスコープを分かりやすくするためにも各ブロックごとに正しく字下げを行いましょう。

|  |
| --- |
| 書式:変数のスコープ |
| **public class** クラス名{  *//フィールド* **public static** int a;  class内ならどこでも利用可能  **public static void** main(String[] args) {  if(){  ifブロック内でしか利用できない  *//ローカル変数*  int b;  *//if文の処理*  *//if文の処理*  }  }  メソッド内でしか  利用できない  *//引数*  **public static 戻り値の型 メソッド名**(int c) {  *//メソッドの処理*  *//メソッドの処理*    }  } |

変数はスコープが重なっていない場合は同じ名前の変数を作っても問題ありません。それぞれの変数は別々に用意されお互いに影響を受けずに利用できます。

スコープが重なっている場合は、ローカル変数同士ではエラーになります。フィールドとローカル変数は重ねることができ、ローカル変数が優先されて実行されます。フィールドを利用したいときは「クラス名.フィールド名」で利用するとフィールドを利用できます。

【Scope.java】

|  |
| --- |
| **public class** Scope {   *//フィールド* **public static int** *num* = 10;   **public static void** main(String[] args) {   *//ローカル変数* **int** num = 20;  System.***out***.println(**"num = "** + num);  System.***out***.println(**"Scope.num = "** + Scope.*num*); *// System.out.println("num2 = " + num2);   //メソッドの呼び出し  method*();  }   **public static void** method() {  **int** num2 = 30;  System.***out***.println(**"num = "** + *num*);  System.***out***.println(**"num2 = "** + num2);  } } |

フィールドを宣言する際は、メソッドの時と同じように「public static」を付けて宣言してください。フィールドとして宣言した「num」変数はSopeクラス内であればどこからでも利用できます。main()メソッドではローカル変数としてフィールドと同じ変数名の「num」変数を宣言しています。同じ変数名の変数がある場合は単純に「num」だけで呼び出すとローカル変数の「num」が動作します。フィールドの「num」を利用したいときは、今回はクラス名がScopeとなっているため、「Scope.num」で呼び出します。変数「num2」はmethod()内で宣言した変数のため、method()外では利用できません。

#### 配列を利用するメソッド

メソッドは配列を引数として利用したり、戻り値として配列を返すことができます。利用方法は通常の変数と同様ですが、配列は参照型であるため受け渡しの際は参照渡しとなることに注意をしてください。

【Sort.java】

|  |
| --- |
| **import** java.util.Arrays;  **public class** Sort {  **public static void** main(String[] args) {   **int**[] array = {40,30,10,50};   System.***out***.println(Arrays.*toString*(array));   *//メソッドの引数として配列を渡す  sort*(array);   System.***out***.println(Arrays.*toString*(array));  }   *//配列を昇順に並び替えるメソッド(選択ソート)  //40と順番に比較して小さい値があれば入れ替える* **public static void** sort(**int**[] array){  **for** (**int** i = 0; i < array.**length** -1; i++) {  **for** (**int** j = i + 1; j < array.**length**; j++) {   **if**(array[i] > array[j]){  **int** temp = array[i];  array[i] = array[j];  array[j] = temp;  }  }  }  } } |

main()メソッドで配列を用意し、sort()メソッドの引数として用意した配列を渡しています。この時、配列は参照渡しとなるため、配列があるメモリ上の場所の情報のみを渡しています。したがって、mainメソッドの配列変数「array」とsortメソッドの引数「array」は同じ配列を扱っていることになります。そのため、sortメソッド内で配列の値を並び替えると、mainメソッドの配列も変わります。

並び替えには「選択ソート」を利用しています。「選択ソート」とはソートするときのアルゴリズムの一種で、順番に大小比較をして小さい値があった場合は入れ替えるという並び替え方法です。

|  |
| --- |
| 選択ソート |
| -------------------------------------------------------------------------------    ------------------------------------------------------------------------------- |

#### オーバーロード

Javaでは引数の数や型を変えることで同じ名前のメソッドを作ることができます。同じような処理を行う場合に引数の型ごとにメソッド名を変えると、メソッドを作るときも、呼び出すときも異なるメソッド名をすべて覚えておく必要があり、コードが煩雑になります。たとえば、整数の加算は addInt、小数の加算は addDouble、文字列の結合は concatString などと分けてしまうと、どれを使うべきか考える手間が増え、呼び出す際にもミスが発生しやすくなります。オーバーロードを使うと、同じような処理を目的とするメソッドに統一した名前を付けることができるので、コードの読みやすさや保守性が向上します。また、呼び出し側がメソッドの名前を意識せず、引数に応じて適切な処理が自動で選択されるため、より直感的にメソッドを使うことができます。

【Overload.java】

|  |
| --- |
| **public class** Overload {   **public static void** main(String[] args) {   **int** a = 10;  **int** b = 20;  **double** c = 10.2;  **double** d = 20.5;   System.***out***.println(*add*(a, b)); *//int + int* System.***out***.println(*add*(c, d)); *//double + double* System.***out***.println(*add*(a, c)); *//int + double* System.***out***.println(*add*(d, b)); *//double + int* System.***out***.println(*add*(a, b, c)); *//int + int + double* }   *//int型の引数を2つ受け取り、その合計を返す* **public static int** add(**int** num1, **int** num2) {  System.***out***.print(num1 + **" + "** + num2 + **" = "**);  **return** num1 + num2;  }  *//double型の引数を2つ受け取り、その合計を返す* **public static double** add(**double** num1, **double** num2) {  System.***out***.print(num1 + **" + "** + num2 + **" = "**);  **return** num1 + num2;  }  *//int型とdouble型の引数を受け取り、その合計を返す* **public static double** add(**int** num1, **double** num2) {  System.***out***.print(num1 + **" + "** + num2 + **" = "**);  **return** num1 + num2;  }  *//double型とint型の引数を受け取り、その合計を返す* **public static double** add(**double** num1, **int** num2) {  System.***out***.print(num1 + **" + "** + num2 + **" = "**);  **return** num1 + num2;  }  *//int型の引数を2つとdouble型の引数を受け取り、その合計を返す* **public static double** add(**int** num1, **int** num2, **double** num3) {  System.***out***.print(num1 + **" + "** + num2 + **" + "** + num3 + **" = "**);  **return** num1 + num2 + num3;  } } |

## クラス

### クラスとは

Javaはオブジェクト指向言語です。 オブジェクト指向とはプログラミングをする際に効率よく開発する方法として編み出された手法です。実際のものや概念を「オブジェクト」という単位でプログラム上に生成することで、開発を効率よく進めることができます。このオブジェクトは特定の性質や動作を持つ「クラス」という設計図をもとに生成されます。

|  |
| --- |
| クラスとオブジェクト |
| 生成 |

【NoＣｌａｓｓ.java】

|  |
| --- |
| **public class** NoClass {  **public static void** main(String[] args) {   String name1 = **"田中太郎"**;  **int** age1 = 20;  String address1 = **"東京"**;  **int** money1 = 1000;   String name2 = **"佐藤五郎"**;  **int** age2 = 30;  String address2 = **"京都"**;  **int** money2 = 2000;   String name3 = **"鈴木三郎"**;  **int** age3 = 40;  String address3 = **"大阪"**;  **int** money3 = 3000;   System.***out***.println(**"名前："** + name1 + **" 年齢："** + age1 + **" 住所："** + address1 + **" 所持金："** + money1);  System.***out***.println(**"名前："** + name2 + **" 年齢："** + age2 + **" 住所："** + address2 + **" 所持金："** + money2);  System.***out***.println(**"名前："** + name3 + **" 年齢："** + age3 + **" 住所："** + address3 + **" 所持金："** + money3);  } } |

このようにクラスを使わずにプログラムを行う場合は、人物が増えるたびにそれぞれの変数を用意し、値を代入する必要があります。各人物ごとにひとまとまりに扱いたいですが、「名前」や「年齢」、「住所」、「所持金」はそれぞれ型が異なるため、配列も利用することができません。

×　String[] tanaka = {“田中太郎”, 20, “東京”, 1000} 　←「20」と「１０００」がint型

#### クラスの定義

こんな時に便利なのがクラスです。クラスを利用することで関連する変数をひとまとまりとして扱うことができます。今回の例でプログラムの中に登場するオブジェクトは「田中太郎さん」や「佐藤次郎さん」です。このオブジェクトを生成するために必要な設計図としてクラスを作ります。クラスには各オブジェクトが持っている値(属性)を保存できるような変数を準備します。

【Person.java】

|  |
| --- |
| **public class** Person {  *//フィールド*  String **name**; *//名前を保存* **int age**; *//年齢を保存* String **address**; *//住所を保存* **int money**; *//所持金を保存*  } |

Personクラスは設計図として利用するため、処理の流れを記述するmain()メソッドは必要ありません。また、クラスブロック直下に記述した変数(nameや、ageなど)のことをフィールドやフィールド変数と呼びます。

#### クラスの利用

クラスはただの設計図のため、そのままでは実行できません。クラスからオブジェクトを生成して初めて利用できます。利用の処理はmain()メソッドに記述します。

|  |
| --- |
| 書式:インスタンス生成(オブジェクト生成) |
| クラス型　変数　= new クラス名(); |

new演算子を利用することでクラスからオブジェクトを生成することができます。

このことをオブジェクト生成やインスタンス生成、インスタンス化とも呼びます。実際の現場ではインスタンスと呼ぶことが多いため今後はオブジェクトを生成することをインスタンス生成と、生成されたオブジェクトのことをインスタンスと呼ぶこととします。

インスタンス(instance)とは、日本語で実体という意味を持つ英単語です。設計図から実際に利用できる実体(インスタンス)を生成することからインスタンス生成と呼ばれます。

生成したインスタンスは変数へ保存しないと操作できません。インスタンスを保存するための変数を用意します。クラスを作成するとそのクラスの型を利用できるようになるため、インスタンスを保存する変数の型にはそのインスタンスを生成するのに利用したクラスの型を使います。

この変数を利用することで生成したインスタンスにアクセスできます。インスタンスの各フィールドへのアクセスは以下の通りです。

|  |
| --- |
| 書式:インスタンスのフィールドへのアクセス |
| //値の代入  インスタンス変数.フィールド名 = 値;  //値の取得  受け取る変数 = インスタンス変数.フィールド名; |

「インスタンス変数.フィールド名」で各フィールドへのアクセスができ、利用方法は通常の変数と同様に扱えます。

【PersonMain.java】

|  |
| --- |
| **public class** PersonMain {  **public static void** main(String[] args) {   Person tanaka = **new** Person(); *//Personクラスのインスタンスを生成* tanaka.**name** = **"田中太郎"**; *//インスタンスのフィールドに値を代入* tanaka.**age** = 20;  tanaka.**address** = **"東京"**;  tanaka.**money** = 1000;   Person sato = **new** Person(); *//Personクラスのインスタンスを生成* sato.**name** = **"佐藤五郎"**;  sato.**age** = 30;  sato.**address** = **"京都"**;  sato.**money** = 2000;   Person suzuki = **new** Person(); *//Personクラスのインスタンスを生成* suzuki.**name** = **"鈴木三郎"**;  suzuki.**age** = 40;  suzuki.**address** = **"大阪"**;  suzuki.**money** = 3000;   *//インスタンスのフィールドを出力* System.***out***.println(**"名前："** + tanaka.**name** + **" 年齢："** + tanaka.**age** +  **" 住所："** + tanaka.**address** + **" 所持金："** + tanaka.**money**);  System.***out***.println(**"名前："** + sato.**name** + **" 年齢："** + sato.**age** +  **" 住所："** + sato.**address** + **" 所持金："** + sato.**money**);  System.***out***.println(**"名前："** + suzuki.**name** + **" 年齢："** + suzuki.**age** +  **" 住所："** + suzuki.**address** + **" 所持金："** + suzuki.**money**);  } } |

Personクラスからnew演算子でインスタンスを生成し、各フィールドへ値を代入します。この時インスタンスはneｗを実行した回数分だけ生成されるため、同じ「name」フィールドが3つできている状態です。フィールド同士は干渉しないため、別々の値が保存されます。

|  |
| --- |
| クラスとインスタンス |
| ―――――――――――――――――――――――――――――――――――――――    田中インスタンス　　　　　佐藤インスタンス　　　　　鈴木インスタンス |

インスタンス変数も配列と同様に参照型のため、変数にはインスタンスが保存されているメモリのアドレス値が格納されています。

コピーする際には配列と同じような動作をするため注意してください。

インスタンスを生成後、各フィールドに値を代入せずに呼び出すとどうなるでしょうか？

各フィールドの型ごとに初期値が決められていて、その値が格納されています。

|  |
| --- |
| 書式:フィールドの初期値 |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | int型 | = 0 |  | String型 | = null | | double型 | = 0.0 |  | 参照型 | = null | | boolean型 | = false |  |  |  | |

#### コンストラクタ

上記で説明したように、各フィールドには型で決められた初期値が自動的に格納されています。しかし、この初期値は型によって固定されているため好きな初期値を指定することはできません。そのため、クラスでは初期値を入力するための機能としてコンストラクタが用意されています。

|  |
| --- |
| 書式:コンストラクタ |
| public class クラス名{  //フィールド  //コンストラクタの宣言  public クラス名(引数, ...){  //各フィールドへの初期値の代入処理  }  } |

コンストラクタはインスタンス生成(new)時に１度だけ自動的に実行されるメソッドです。コンストラクタに戻り値はなく、メソッド名はクラス名と完全一致していないといけません。()内の引数は、インスタンス生成時に特定の初期値を指定したい場合は記述し、不要な場合は記述しなくても問題ありません。

【Person.java】

|  |
| --- |
| **public class** Person {   String **name**; *//名前を保存* **int age**; *//年齢を保存* String **address**; *//住所を保存* **int money**; *//所持金を保存   //コンストラクタ* **public** Person() {  **this**.**name** = **"名無し"**;  **this**.**age** = 0;  **this**.**address** = **"日本"**;  **this**.**money** = 0;  }   *//コンストラクタのオーバーロード* **public** Person(String name, **int** age, String address, **int** money) {  **this**.**name** = name;  **this**.**age** = age;  **this**.**address** = address;  **this**.**money** = money;  } } |

コンストラクタも通常のメソッドと同様にオーバーロードによる宣言ができます。

(同じ名前のメソッドを引数の種類や数を変えると宣言できる仕組み)

インスタンス生成時の「new　クラス名(・・・);」の()内の値によって実行されるコンストラクタが選択されます。ただし、コンストラクタを宣言すると、インスタンス生成をするときにコンストラクタで宣言した引数を必ず渡す必要があります。

「thisについて」

変数の有効範囲については以前説明を行いましたが、スコープが異なる場合は同じ名前の変数を宣言することができます。引数があるコンストラクタの引数にはString型の「name」という変数があり、フィールドにも「name」という変数があります。この二つの変数を区別するためにJavaでは、「this.」を付けるとフィールド変数を指定したことになります。何も指定を行わないとローカル変数が優先されて選択されます。

スコープの説明ではフィールドには「クラス名.フィールド名」を使うと説明しましたが、main()メソッドがあるクラス内ではフィールドにstaticを付けないとmain()メソッドからアクセスできないというルールがあります。そのため、staticがついたフィールドにアクセスする方法を説明しました。今回のPersonクラスにはmainメソッドがないため「this.フィールド名」でアクセスができます。詳しくはstaticの章で説明します。

【PersonMain2.java】

|  |
| --- |
| **public class** PersonMain2 {  **public static void** main(String[] args) {   *//Personクラスのインスタンスを生成  //引数なしのコンストラクタを呼び出し* Person nanashi = **new** Person();   *//引数ありのコンストラクタを呼び出し* Person tanaka = **new** Person(**"田中太郎"**, 20, **"東京"**, 1000);  Person sato = **new** Person(**"佐藤五郎"**, 30, **"京都"**, 2000);  Person suzuki = **new** Person(**"鈴木三郎"**, 30, **"大阪"**);　　　　*//引数が3つでエラー*    *//インスタンスのフィールドを出力* System.***out***.println(**"名前："** + nanashi.**name** + **" 年齢："** + nanashi.**age** +  **" 住所："**+ nanashi.**address** + **" 所持金："** + nanashi.**money**);  System.***out***.println(**"名前："** + tanaka.**name** + **" 年齢："** + tanaka.**age** +  **" 住所："** + tanaka.**address** + **" 所持金："** + tanaka.**money**);  System.***out***.println(**"名前："** + sato.**name** + **" 年齢："** + sato.**age** +  **" 住所："** + sato.**address** + **" 所持金："** + sato.**money**);  } } |

#### デフォルトコンストラクタ

コンストラクタを定義せずにクラスを作成した場合は、コンパイル時に自動的にデフォルトコンストラクタが作成されます。デフォルトコンストラクタは引数なしの処理なしのコンストラクタです。このデフォルトコンストラクタが作成されることによって引数なしでインスタンス生成が可能になります。引数ありのコンストラクタを明示的に定義した場合は、デフォルトコンストラクタは作成されません。したがって、引数なしのコンストラクタを自分で定義しないと引数なしでインスタンス生成ができなくなります。

|  |
| --- |
| 書式:デフォルトコンストラクタ |
| **public class** クラス名{    *//フィールド    //デフォルトコンストラクタが自動生成される* **public** クラス名(){  *//無し* } } |

#### メソッド

クラスにもメソッドを定義することができます。同じクラスから作成したインスタンスに対して行いたい共通の処理があった場合はクラス内にその処理をメソッドとして記述します。

|  |
| --- |
| 書式:メソッド定義 |
| **public class** クラス名{    *//フィールド    //コンストラクタ*  *//メソッドを定義* **public** 戻り値の型 メソッド名(引数, ...){    *//処理* **return** 戻り値; }  } |

|  |
| --- |
| 書式:メソッド呼び出し |
| *//インスタンス生成* クラス名 インスタンス変数 = **new** クラス名();  *//インスタンスのメソッドの呼び出し(戻り値なし)* インスタンス変数.メソッド名(引数, ...);  *//インスタンスのメソッドを呼び出し(戻り値あり)* 戻り値の型 変数名 = インスタンス変数.メソッド名(引数, ...); |

mainメソッドがないクラスにメソッドを定義する場合はstaticは必要ありません。その他の定義は同じです。戻り値が必要な場合は戻り値の型を指定し、return文で値を返します。戻り値が不要な場合はvoidを指定し、return文は必要ありません。オーバーロードの定義もできます。

インスタンスのメソッドを呼び出すときは、「インスタンス変数.メソッド名(引数,　...)」で呼び出します。

【Person.java】

|  |
| --- |
| **public class** Person {   String **name**; *//名前を保存* **int age**; *//年齢を保存* String **address**; *//住所を保存* **int money**; *//所持金を保存   //コンストラクタ* **public** Person() {  **this**.**name** = **"名無し"**;  **this**.**age** = 0;  **this**.**address** = **"日本"**;  **this**.**money** = 0;  }   *//コンストラクタのオーバーロード* **public** Person(String name, **int** age, String address, **int** money) {  **this**.**name** = name;  **this**.**age** = age;  **this**.**address** = address;  **this**.**money** = money;  }   *//メソッドを定義* **public void** printInfo() {  System.***out***.println(**this**.**name** + **"さんの情報"**);  System.***out***.println(**" 名前："** + **this**.**name**);  System.***out***.println(**" 年齢："** + **this**.**age**);  System.***out***.println(**" 住所："** + **this**.**address**);  System.***out***.println(**" 所持金："** + **this**.**money** + **"\n"**);  } } |

【PersonMain3.java】

|  |
| --- |
| **public class** PersonMain3 {  **public static void** main(String[] args) {   *//Personクラスのインスタンスを生成  //引数なしのコンストラクタを呼び出し* Person nanashi = **new** Person();   *//引数ありのコンストラクタを呼び出し* Person tanaka = **new** Person(**"田中太郎"**, 20, **"東京"**, 1000);  Person sato = **new** Person(**"佐藤五郎"**, 30, **"京都"**, 2000);   *//インスタンスのメソッドを呼び出し(戻り値なし)* nanashi.printInfo();  tanaka.printInfo();  sato.printInfo();  } } |

#### 配列とクラス

同じクラスから複数のインスタンスを生成する場合に配列を用いると、簡潔にプログラムを記述することができます。

クラス型の配列を準備し、各要素にインスタンスを代入することで、ループ処理を行えます。

|  |
| --- |
| 配列とクラス |
|  |

【PersonMain4.java】

|  |
| --- |
| **public class** PersonMain4 {  **public static void** main(String[] args) {   *//Personクラス型の配列を宣言* Person[] people = **new** Person[3];   *//配列にPersonクラスのインスタンスを生成* people[0] = **new** Person(**"田中太郎"**, 20, **"東京"**, 1000);  people[1] = **new** Person(**"佐藤五郎"**, 30, **"京都"**, 2000);  people[2] = **new** Person(**"鈴木三郎"**, 40, **"大阪"**, 3000);   *//拡張for文で配列の中身を取り出し、printInfoメソッドを呼び出す* **for** (Person person : people) {  person.printInfo();  }  } } |

### 静的メンバー

変数やメソッドに付ける「staticキーワード」について説明します。main()メソッドに付けたり、フィールドに付けたり付けなかったりした理由を解説します。

#### 静的(static)フィールド

staticキーワードを付けたフィールドは同一のクラスから作成した全てのインスタンスで共有されるフィールドになります。通常のフィールドは各インスタンスごとに変数が用意され、それぞれ別々に値が保存されますが、staticで宣言したフィールドは、クラスごとに1つだけ用意されます。静的フィールドはインスタンスではなく、クラス自体に所属しているため、インスタンスを生成しなくてもアクセスができます。クラス変数とも呼ばれます。

|  |
| --- |
| 静的(static)フィールド |
|  |

|  |
| --- |
| 書式:静的(static)フィールドの宣言 |
| *//静的フィールドの宣言*  **static 型名** 変数名;  *//静的フィールドの利用(同じクラス内であれば)*  変数名;  *//静的フィールドの利用(クラス外からであれば)*  クラス名. 変数名; |

【Person.java】

|  |
| --- |
| **public class** Person {   String **name**; *//名前を保存* **int age**; *//年齢を保存* String **address**; *//住所を保存* **int money**; *//所持金を保存* **static int** *count* = 0; *//静的フィールド(クラス変数)   //コンストラクタ* **public** Person() {  **this**.**name** = **"名無し"**;  **this**.**age** = 0;  **this**.**address** = **"日本"**;  **this**.**money** = 0;  }   *//コンストラクタのオーバーロード* **public** Person(String name, **int** age, String address, **int** money) {  **this**.**name** = name;  **this**.**age** = age;  **this**.**address** = address;  **this**.**money** = money;   *count*++; *//インスタンスが生成されるたびにカウントを増やす* }  *//メソッドを定義* **public void** printInfo() {  System.***out***.println(**this**.**name** + **"さんの情報"**);  System.***out***.println(**" 名前："** + **this**.**name**);  System.***out***.println(**" 年齢："** + **this**.**age**);  System.***out***.println(**" 住所："** + **this**.**address**);  System.***out***.println(**" 所持金："** + **this**.**money** + **"\n"**);  } } |

【PersonMain5.java】

|  |
| --- |
| **public class** PersonMain5 {  **public static void** main(String[] args) {  System.***out***.println(Person.*count* + **"人生成"**); *//静的フィールドcountを出力*  *//Personクラス型の配列を宣言* Person[] people = **new** Person[3];  people[0] = **new** Person(**"田中太郎"**, 20, **"東京"**, 1000);    System.***out***.println(Person.*count* + **"人生成"**); *//静的フィールドcountを出力* people[1] = **new** Person();    System.***out***.println(Person.*count* + **"人生成"**); *//静的フィールドcountを出力* people[2] = **new** Person(**"鈴木三郎"**, 40, **"大阪"**, 3000);    System.***out***.println(Person.*count* + **"人生成\n"**); *//静的フィールドcountを出力   //拡張for文で配列の中身を取り出し、printInfoメソッドを呼び出す* **for** (Person person : people) {  person.printInfo();  }  } } |

#### 定数

変数に付加できるキーワードとして「finalキーワード」があります。これは変数が再代入できない定数になることを表します。finalを使うことで初期化後、変更できず値が固定されます。円周率や、特定の数値の最大値などのプログラム動作中に値が変化しない定数に利用します。

定数を宣言する場合は、慣習的に定数であることが分かりやすいように変数名はすべて大文字で単語間は「 \_ 」を利用します。

|  |
| --- |
| 書式:定数(final)の宣言 |
| *//定数の宣言(変更されない値)*  **final** 型名 *定数名* = 値; |

定数はプログラムのどこからでもアクセスができるように、ローカル変数ではなくフィールドとして宣言する場合が多いです。このときの宣言の仕方として定型の宣言方法を紹介します。

|  |
| --- |
| 書式:静的(static)な定数(final)の宣言 |
| *//静的な定数の宣言(変更されない値)*  **public static final** 型名 *定数名* = 値; |

【Person.java】

|  |
| --- |
| **public class** Person {   String **name**; *//名前を保存* **int age**; *//年齢を保存* String **address**; *//住所を保存* **int money**; *//所持金を保存* **static int** *count* = 0; *//静的フィールド(クラス変数)* **public static final int** *MAX\_COUNT* = 5; *//静的定数フィールド   //コンストラクタ* **public** Person() {  *//省略*  }   *//コンストラクタのオーバーロード* **public** Person(String name, **int** age, String address, **int** money) {  *//省略* }   *//メソッドを定義* **public void** printInfo() {  *//省略*  } } |

【PersonMain6.java】

|  |
| --- |
| **public class** PersonMain6 {  **public static void** main(String[] args) {  *//静的定数フィールドMAX\_COUNTを出力*  System.***out***.println(Person.*MAX\_COUNT* + **"人まで生成可能"**); System.***out***.println();   Person[] people = **new** Person[Person.*MAX\_COUNT*];   **for** (**int** i = 0; i < people.**length**; i++) {  people[i] = **new** Person(**"田中太郎"**, 20, **"東京"**, 1000);  System.***out***.println(Person.*count* + **"人生成"**);  }  System.***out***.println();   **for** (Person person : people) {  person.printInfo();  }  } } |

これらの定数を記号定数と呼ぶこともあります。記号定数とは、数値や文字列などの固定値に名前を使て扱う定数のことです。これによりコードの読みやすさや保守性が向上し、同じ値を複数個所で使う際のミスを防げます。

定数は、Java　SE(API)で用意されているものがあります。MathクラスのPIやEなどです。これらの定数は「クラス名.定数名」で簡単に利用できます。

(Eは自然対数の底 e)

リファレンス:　<https://docs.oracle.com/javase/jp/17/docs/api/constant-values.html>

【CircleArea.java】

|  |
| --- |
| **public class** CircleArea {  **public static void** main(String[] args) {   System.***out***.print(**"円の半径を入力してください : "**);  **double** radius = **new** java.util.Scanner(System.***in***).nextDouble();   *//Math.PIは円周率を表す定数MathクラスのPIフィールド* **int** area = (**int**) (radius \* radius \* Math.***PI***);   System.***out***.println(**"円の面積は"** + area + **"です"**);  } } |

#### 静的(static)メソッド

staticキーワードを付けたメソッドも静的フィールドと同様、クラスに所属するためインスタンスを生成する前から呼び出すことができます。newせずに利用できるメソッドで便利ですが、制約があるため利用には注意が必要です。

制約　:　静的メソッド内ではstaticが付いていないフィールドやメソッドを利用できません。

インスタンスを生成する前から実行できるということは、インスタンスが存在しない状態でも動作する可能性があるということです。そのため、静的メソッドではインスタンスのフィールドにアクセスできません。静的メソッドは、インスタンスが生成される前から利用できる静的なメンバーにのみアクセスできる仕様になっています。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | staticフィールド | staticメソッド | 通常フィールド | 通常メソッド |
| 通常メソッドから | 〇 | 〇 | 〇 | 〇 |
| staticメソッドから | 〇 | 〇 | × | × |

|  |
| --- |
| 書式:静的(static)メソッドの宣言 |
| *//静的メソッドの宣言*  **static 戻り値の型** メソッド名(引数, ...){  }  *//静的メソッドの利用(同じクラス内であれば)*  *メソッド名(引数, ...)*;  *//静的メソッドの利用(クラス外からであれば)*  クラス名. *メソッド名(引数, ...)*; |

【Person.java】

|  |
| --- |
| **public class** Person {   String **name**; *//名前を保存* **int age**; *//年齢を保存* String **address**; *//住所を保存* **int money**; *//所持金を保存* **static int** *count* = 0; *//静的フィールド(クラス変数)* **public static int** *MAX\_COUNT* = 5; *//静的定数フィールド*  *//コンストラクタ* **public** Person() {  *//省略*  }   *//コンストラクタのオーバーロード* **public** Person(String name, **int** age, String address, **int** money) {  *//省略* }   *//メソッドを定義* **public void** printInfo() {  *//省略*  }  *//静的メソッドを定義*  **public static void** printCount() {System.***out***.println(**"合計"** + *count* + **"人です"**);  }  } |

【PersonMain6.java】

|  |
| --- |
| **public class** PersonMain6 {  **public static void** main(String[] args) {   *//静的定数フィールドMAX\_COUNTを出力* System.***out***.println(Person.*MAX\_COUNT*+ **"人まで生成可能"**);  *//静的メソッドprintCountを呼び出し*  Person.*printCount*();  System.***out***.println();   Person[] people = **new** Person[Person.*MAX\_COUNT*];   **for** (**int** i = 0; i < people.**length**; i++) {  people[i] = **new** Person(**"田中太郎"**, 20, **"東京"**, 1000);  System.***out***.println(Person.*count* + **"人生成"**);  }  System.***out***.println();   **for** (Person person : people) {  person.printInfo();  }   *//静的メソッドprintCountを呼び出し* Person.*printCount*();  } } |

このような特徴を持った静的メソッドは、自身で作成する機会はそれほど多くありませんが、Java SEでは多く用意されているため、使い方は覚えておくようにしてください。

例)

乱数生成メソッド: Math.random()

文字列を数値へ変換するメソッド: Integer.parseInt()

配列の中身を出力するメソッド: Arrays.toString() など...

また、main()メソッドにもstaticがついています。これは、main()メソッドが静的メソッドであることを表しています。Javaアプリケーションは最初にmain()メソッドから実行が開始します。そのメソッドはJVMによって呼び出られるため、JVMはプログラムを開始するためにクラスのインスタンスを生成することなく、直接main()メソッドを呼び出す必要があります。そのため、main()メソッドをstaticにすることで、インスタンスを生成せずにクラス名を用いて直接呼び出すことが可能になります。

|  |
| --- |
| main()メソッドの呼び出し |
|  |

## パッケージ

大規模なシステム開発になればなるほど、作成するクラスの数が多くなり、管理が難しくなります。 パッケージはそれらのクラスを効率よく管理するためのJavaのフォルダ階層機能です。コードを整理し、名前の衝突を避けるために利用されます。

### パッケージの利用

#### パッケージの作成

まずは、ＩＤＥ上でパッケージを作成します。「src」を右クリックすると簡単に作成できます。エクスプローラー上でもフォルダが作成されていることを確認してください。

|  |
| --- |
| パッケージの作成 |
|  |

#### パッケージ内のクラス

パッケージ内にクラスを作成する場合は、そのクラスがどのパッケージに所属しているかを明確にするために「パッケージ宣言」をエディターの先頭に記述する必要があります。クラス宣言よりも前です。

記述するパッケージ名は「srcフォルダ」から見て、作成したクラスがどの階層にあるかを記述します。複数のパッケージ階層の中であれば、srcフォルダ内から順番にパッケージ名を「.」区切りで記述します。

|  |
| --- |
| 書式:パッケージ宣言 |
| *//パッケージ宣言* **package** パッケージ名.パッケージ名;  *//クラス宣言* **public** クラス名{  } |

IDEを利用しているとパッケージ宣言は自動的に入力されることが多いですが、利用する上でなぜその記述があるのかを理解するようしてください。

【pack1.Pa1.java】

|  |
| --- |
| *//パッケージ宣言* **package** pack1;  *//クラス宣言* **public class** Pa1 {   *//フィールド* **int pa1**;   *//メソッド* **void** printPa1(){  System.***out***.println(**"Pa1クラスのメソッド"**);    } } |

【pack1.pack1\_1.Pa1**\_**1.java】

|  |
| --- |
| *//パッケージ宣言* **package** pack1.pack1\_1;  **public class** Pa1\_1 {   *//フィールド* **int pa1\_1**;   *//メソッド* **void** printPa1\_1() {  System.***out***.println(**"Pa1\_1クラスのメソッド"**);   } } |

パッケージは名前空間を分ける役割も持っています。同じパッケージ内に同じクラス名のクラスを作成することはできませんが、パッケージを分けることで同じクラス名のクラスを複数作成することができます。

#### パッケージ内のクラスの利用

同じパッケージ内のクラスを利用する場合は、今まで通り、クラス名のみで利用できます。別のパッケージのクラスを利用するには2つの方法があります。

1. パッケージ名を含む「完全限定クラス名(FQCN)」を利用する
2. 利用するクラスの「インポート宣言」を行う

(1)の方法は、どのパッケージのどのクラスを利用するかをフルネームで指定する方法です。

利用するクラスがパッケージ名から明確になる反面、利用するたびにフルネームをすべて指定する必要があるため、記述量が多くなります。

|  |
| --- |
| 書式:完全限定クラス名(FQCN)の利用 |
| *//別パッケージのクラスを利用する場所で* パッケージ名.クラス名; |

(2)の方法は、パッケージ宣言とクラス宣言の間に利用するクラスのインポート宣言を記述します。そうすると、インポートしたクラスを利用する箇所ではクラス名のみの指定で利用できます。

通常はこの(2)の方法で利用することが多いです。

また、IDEの補完機能を利用すると自動的にインポート宣言を記述してくれます。

|  |
| --- |
| 書式:インポート宣言 |
| *//パッケージ宣言(利用するクラス)*  **packageパッケージ名**;  *//インポート宣言(利用されるクラス)*  **import** パッケージ名.クラス名;  *//クラス宣言* **class** クラス名 {  メソッド(){  *//インポートしたクラスを利用*  クラス名;  } } |

【pack2.Pack2Main.java】

|  |
| --- |
| **package** pack2;  *//インポート宣言* **import** pack1.pack1\_1.Pa1\_1;  **public class** Pack2Main {   **public static void** main(String[] args) {   *//完全限定クラス名(FQCN)* pack1.Pa1 pa1 = **new** pack1.Pa1();   *//インポートを利用* Pa1\_1 pa1\_1 = **new** Pa1\_1();  } } |

### アクセス制御

クラス内で定義されているフィールドや、コンストラクタ、メソッドをまとめてメンバーといいます。インスタンスを生成したときにインスタンスが保持している「データ」や「機能」の総称です。

このメンバーに対してどこからどこにアクセスできるかを制御する仕組みが用意されています。アクセス制御を行うことで、他のクラスから不用意にデータを変更されたり、不正に操作されたりするのを防ぐことができます。アクセス制御を設定するためにはアクセス修飾子を利用します。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| アクセス修飾子 | private | 指定なし | protected | public |
| 同じクラス | ○ | ○ | ○ | ○ |
| 同じパッケージ | × | ○ | ○ | ○ |
| 継承したクラス(同じパッケージ) | × | ○ | ○ | ○ |
| 継承したクラス(違うパッケージ) | × | × | ○ | ○ |
| 違うパッケージ | × | × | × | ○ |

継承については後程説明します。ここでは、

「private」を指定したメンバーへは、同じクラス内からしかアクセスできなくなることと、

「public」を指定したメンバーはどこからでもアクセスができること、

「アクセス修飾子」を指定しないと、同じパッケージのクラスからしかアクセスできることを覚えておいてください。

|  |
| --- |
| 書式:アクセス修飾子の指定 |
| *//フィールドをprivateに* **private**型名変数; *//メソッドをpublicに* **public**戻り値の型 メソッド名(引数, ...){  *//処理*  } |

【pack1.pack1\_1.Pa1**\_**1.java】

|  |
| --- |
| *//パッケージ宣言* **package** pack1.pack1\_1;  **public class** Pa1\_1 {   *//フィールドをprivateに* **private int pa1\_1**;   *//メソッドをpublicに* **public void** printPa1\_1() {  System.***out***.println(**"Pa1\_1クラスのメソッド"**);  } } |

【pack2.Pack2Main.java】

|  |
| --- |
| **package** pack2;  *//インポート宣言* **import** pack1.pack1\_1.Pa1\_1;  **public class** Pack2Main {   **public static void** main(String[] args) {   *//完全限定クラス名(FQCN)* pack1.Pa1 pa1 = **new** pack1.Pa1();    *//pa1フィールドは「指定無し」のため別パッケージからアクセスできない* pa1.**pa1**;    *//printPa1メソッドも「指定無し」のため別パッケージからアクセスできない* pa1.printPa1();    *//インポートを利用* Pa1\_1 pa1\_1 = **new** Pa1\_1();    *//pa1\_1フィールドは「private」のため、クラス外からアクセスできない* pa1\_1.**pa1\_1**;    *//printPa1\_1メソッドは「public」のため、どこからでもアクセスできる* pa1\_1.printPa1\_1();  } } |

### Java SEとパッケージ

Javaには「Java SE」と呼ばれる標準ライブラリが用意されており、プログラミングに必要なさまざまな機能をあらかじめ利用できるようになっています。これにより、基礎的な処理や一般的な機能を自分で一から作成せずに、簡単に活用することができます。

これらすべての機能は、パッケージとして整理し、提供されています。したがって、これらの機能を利用するためにはインポート宣言が必要になります。

「java.lang」パッケージ内のクラスのみ、頻繁に利用するため特別に「インポート宣言」をせずに利用できます。

Java SEの主要なパッケージを紹介します。

|  |  |
| --- | --- |
| パッケージ | 提供されているクラスや機能 |
| Java.lang | Javaの基本クラス（StringやMath、Systemなど）を含み、最も頻繁に使われるパッケージです。このパッケージのクラスは自動的にインポートされるため、明示的なインポートが不要です。 |
| Java.util | コレクションフレームワークを提供し、リストやセット、マップなどデータの管理に使うクラスが揃っています。たとえば、ArrayListやHashMapなどのクラスがあります。文字を読み込むScannerクラスもこのパッケージに用意してあります。 |
| Java.io | ファイル操作や入出力に関するクラスを提供し、ファイルの読み書きやデータのやりとりが行えます。たとえば、FileReaderやFileWriterなどがあります。 |
| Java.nio | 非同期I/Oやチャネル・バッファを使った入出力処理をサポートし、効率的なI/O操作を行えます。 |
| Java.net | ネットワーク関連の処理がまとめられており、インターネット通信やソケット通信を行うクラスが含まれています。 |
| Java.time | 日付と時刻を扱うクラスが含まれており、LocalDateやLocalDateTimeなど、日時操作を簡単に行えるクラスがあります。 |

【JavaSEMain.java】

|  |
| --- |
| **public class** JavaSEMain {  **public static void** main(String[] args) {   *//java.utilクラスのScannerクラスの利用* java.util.Scanner scanner = **new** java.util.Scanner(java.lang.System.***in***);   System.***out***.print(**"文字を入力してください : "**);   String str = scanner.next();   System.***out***.println(**"入力値 : "** + str);   System.***out***.println(**"現在の時刻 : "** + java.time.LocalDateTime.*now*());  } } |

## 文字列

Javaでは文字列を扱う場合、その文字列はStringクラスのインスタンスとして扱われます。Stringクラスには、文字列を効率よく扱うために多くのメソッドが用意されています。これにより、文字列の結合や分割、部分的な抽出、検索、置換など、さまざまな操作が簡単に行えます。この章ではStringクラスの利用の仕方を説明します。

### Stringクラス

#### Stringクラスのインスタンス

Javaプログラム内で「””」で囲ったリテラルを記述するとその文字列の値を持つインスタンスが生成されます。この時作成されたインスタンスは内容を変更することができません(不変)。文字列の結合等で文字列が変化する場合は結合後の文字列を持つ新しいインスタンスが生成されます。

また、同じ内容の文字列が複数の変数で使用される場合、メモリの無駄を防ぐために「文字列プール」という仕組みを使います。Stringリテラルが使われると、文字列プールを確認し、同じ内容の文字列がすでに存在する場合、その参照を再利用します。

【StringMain.java】

|  |
| --- |
| **public class** StringMain {  **public static void** main(String[] args) {   String str1 = **"文字列"**; *//リテラルでインスタンスを生成* String str2 = **new** String(**"文字列"**); *//new演算子でインスタンスを生成* String str3 = **"文字列"**; *//str1の参照先と同じ文字列を参照* System.***out***.println(**"str1 == str2 は "** + (str1 == str2));  System.***out***.println(**"str1 == str3 は "** + (str1 == str3));   System.***out***.println(**"str1.equals(str2) は "** + str1.equals(str2));  System.***out***.println(**"str1.equals(str3) は "** + str1.equals(str3));  } } |

#### toString()とequals()

Stringクラスには自分自身の文字列を返す「toString()メソッド」が用意されています。Stringクラスの文字列を表示する場合に利用され、「println()メソッド」の引数として文字列を渡すと暗黙的に呼び出され、文字列を表示します。

「equals()メソッド」は文字列が同じかどうかの判定に利用されます。「equalsメソッド」を呼び出したインスタンスと、引数のインスタンスが同じ文字列を保持している場合は「true」を返し、違う場合は「false」を返します。

#### Stringクラスのメソッド

その他にもさまざまなメソッドが用意されています。使用頻度が高いメソッドを示します。

|  |  |
| --- | --- |
| 操作 | メソッド(戻り値 メソッド名 引数) |
| 文字列比較の判定 | boolean equals(Object o) |
| 文字列を出力 | String toString() |
| 文字列の長さ | int length() |
| 空文字か判定 | boolean isEmpty() |
| 検索 | |
| 文字列sを含むか判定 | boolean contains(String s) |
| 文字列sで始まるか判定 | boolean startsWith(String s) |
| 文字列sの最初からの位置 | int indexOf(String s) |
| 文字列抜き出し | |
| 指定位置から文字列の抜き出し | substring(int index)  substring(int index, endIndex) |
| 文字列の置き換え | |
| 前後のスペースを除去(全角不可) | String trim() |
| 文字列の置き換え | String replace(String before, String after) |
| 文字列の分割 | |
| 文字列sで分割 | String[] split(String s) |

文字列を番号で指定する場合は「0」からスタートすることに注意してください

|  |
| --- |
| 書式:文字列の番号指定 |
| *// ０ １ ２ ３ ４ ５ ６ ７ ８ ９ ←　文字番号*  **" ポ リ テ ク セ ン タ ー 京 都 "** |

このほかにもさまざまな便利なメソッドが用意されているのでリファレンス等で確認してください。

Java 21 ドキュメント(Stringクラス)

<https://docs.oracle.com/javase/jp/21/docs/api/java.base/java/lang/String.html>

【StringMain2.java】

|  |
| --- |
| **public class** StringMain2 {  **public static void** main(String[] args) {   *// ０１２３４５６７８９* String str1 = **"ポリテクセンター京都"**;  String str2 = **"京都府長岡京市友岡１丁目２−１"**;   *//文字列の長さを取得* System.***out***.println(str1.length());  System.***out***.println();   *//空文字かどうかを判定* String str3 = **""**;  System.***out***.println(str1.isEmpty());  System.***out***.println(str3.isEmpty());  System.***out***.println();   *//文字列検索* System.***out***.println(str1.contains(**"京都"**)); *//文字列を含むかどうか* System.***out***.println(str1.startsWith(**"ポリ"**)); *//指定した文字列で始まるかどうか* System.***out***.println(str1.endsWith(**"京都"**)); *//指定した文字列で終わるかどうか* System.***out***.println(str1.indexOf(**"セン"**)); *//指定した文字列が最初に出現する位置* System.***out***.println();   *//文字列の抜き出し* System.***out***.println(str2.substring(3)); *//指定した位置から最後まで* System.***out***.println(str2.substring(7, 12)); *//指定した位置から指定した位置まで   //文字列の置換* String str4 = **" 京都府　"**;  System.***out***.println(str4.trim()); *//前後の空白を削除* System.***out***.println(str1.replace(**"京都"**, **"大阪"**)); *//指定した文字列を置換* String str5 = **"京都府 長岡京市 友岡 １丁目２−１"**;  System.***out***.println(str5.replace(**" "**, **""**)); *//指定した文字列を置換* System.***out***.println();   *//文字列の分割* String str6 = **"東京,大阪,京都"**;  String[] strArray = str6.split(**","**); *//指定した文字列で分割し、配列に格納* **for**(String str : strArray) {  System.***out***.println(str);  }  } } |

#### 文字列結合

通常、文字列を結合する際には「 + 」演算子を利用します。しかし、この方法では文字列を結合するたびに新しいStringクラスのインスタンスが生成されます。Stringは不変（immutable）であるため、+演算子を使うたびに元の文字列に新たな内容が追加された新しいインスタンスが作られ、メモリ効率が悪くなります。したがって、結合処理を何度も実行する場合は専用のクラスを利用することが推奨されています。

「StringBufferクラス」を利用することで新たなインスタンスを生成せずに文字列結合を行えるため、パフォーマンスが向上します。

|  |
| --- |
| 書式: StringBufferクラス |
| *//StringBufferクラスのインスタンスを生成* StringBuffer sb = **new** StringBuffer(**"文字列"**);  *//文字列の追加* sb.append(**"追加"**);  *//文字列の表示* System.***out***.println(sb); |

【StringMain3.java】

|  |
| --- |
| **public class** StringMain3 {  **public static void** main(String[] args) {   *//* *StringBufferクラスのインスタンスを生成* StringBuffer sb = **new** StringBuffer(**"ABC"**);   *//文字列の追加* sb.append(**"DEF"**);  System.***out***.println(sb);   *//実行時間を計測(StringBuffer)* **long** start = System.*currentTimeMillis*();  StringBuffer sb2 = **new** StringBuffer();  **for**(**int** i = 0; i < 100000; i++) {  sb2.append(**"a"**);  }  **long** end = System.*currentTimeMillis*();  System.***out***.println(**"経過時間："** + (end - start) + **"ミリ秒"**);   *//実行時間を計測(String + 演算子)* start = System.*currentTimeMillis*();  String str = **""**;  **for**(**int** i = 0; i < 100000; i++) {  str += **"a"**;  }  end = System.*currentTimeMillis*();  System.***out***.println(**"経過時間："** + (end - start) + **"ミリ秒"**);  } } |

### 文字列変換

Javaでは数値と文字列を相互変換するためのメソッドが用意されています。

#### ラッパークラス

ラッパークラスとは基本型の値をインスタンスとして扱うために用意されたクラスです。Javaでは、基本型(int型,double型など)はクラスではないため、クラス特有の仕組みを利用できません。基本型をクラスとして扱うために、各基本型に対応したラッパークラスが用意されています。

ラッパークラスには値を効率よく利用するためのさまざまなメソッドが用意されています。

|  |  |
| --- | --- |
| 基本型 | ラッパークラス |
| int | Integer |
| double | Double |
| float | Float |
| long | Long |
| short | Short |
| byte | Byte |
| char | Character |
| boolean | Boolean |

【RapperMain.java】

|  |
| --- |
| **public class** RapperMain {  **public static void** main(String[] args) {   **int** i = 100;  Integer integer1 = **new** Integer(i); *//int型をInteger型に変換(非推奨)* Integer integer2 = i; *//自動的に変換される(オートボクシング)* Integer integer3 = 200; *//自動的に変換される(オートボクシング)* System.***out***.println(integer1);  System.***out***.println(integer2);  System.***out***.println(integer3);   **int** i2 = integer1.intValue(); *//Integer型をint型に変換(非推奨)* **int** i3 = integer2; *//自動的に変換される(オートアンボクシング)* System.***out***.println(i2);  System.***out***.println(i3);  } } |

#### 数値を文字列へ変換

数値を文字列へ変換する方法は3種類あります。

1. 「String.valueOf()」を利用する方法
2. 各ラッパークラスの「.toString()」を利用する方法
3. 「“” + 値」を利用する方法

(1)と(2)はどちらでも数値を文字列に変換することはできますが、変換しようとしている値が「null」の場合の挙動が変わります。「valueOf()」は変換する値がnullの場合は「”null”」という文字列を返します。

「toString()」は変換する値が「null」の場合、「NullPointerException」という例外が発生します。

(3)は最もシンプルで分かりやすいですが、実行速度が少し遅いため、何度も文字列変換を行う場合は避けた方が良いです。

【StringCastMain.java】

|  |
| --- |
| **public class** StringCastMain {  **public static void** main(String[] args) {   *//int型をString型に変換* **int** i = 100;  String str1 = String.*valueOf*(i);  String str2 = Integer.*toString*(i);  String str3 = **""** + i;   System.***out***.println(str1);  System.***out***.println(str2);  System.***out***.println(str3);   *//double型をString型に変換* **double** d = 3.14;  String str4 = String.*valueOf*(d);  String str5 = Double.*toString*(d);  String str6 = **""** + d;   System.***out***.println(str4);  System.***out***.println(str5);  System.***out***.println(str6);  } } |

#### 文字列を数値へ変換

文字列を数値へ変換するには変換後の型のラッパークラスに用意されているstaticメソッドの「parse()」を利用します。

int型への変換 →　Integer.parseInt(“数値”)

double型への変換　 →　Double.parseDouble(“数値”)

数値に変換できない文字列が渡された場合は、「NumberFormatException」が発生します。

【StringCastMain2.java】

|  |
| --- |
| **public class** StringCastMain2 {  **public static void** main(String[] args) {   String str = **"123"**;  String str2 = **"123.45"**;  String str3 = **"abc"**;   *//String型をint型に変換* **int** i = Integer.*parseInt*(str);  System.***out***.println(i);   *//String型をdouble型に変換* **double** d = Double.*parseDouble*(str2);  System.***out***.println(d);   **int** i2 = Integer.*parseInt*(str3); *//NumberFormatExceptionが発生* System.***out***.println(i2);  } } |

## オブジェクト指向

クラスを理解したところでここからはより高度なオブジェクト指向の機能を説明します。具体的には、「カプセル化」、「継承」、「多態性(ポリモーフィズム)」についてです。これらの機能ははじめは複雑な仕組みに感じますが、理解を進めることで効率よくプログラミングをする上で非常に便利な仕組みだと感じるようになるはずです。

### カプセル化

#### カプセル化とは

カプセル化とは、前の章で学習した「アクセス制御」を使って実現する考え方です。クラスは、データ（フィールド）と処理（メソッド）を持っていますが、これらが簡単に外部から操作できると、予期しないデータが設定されたり、不適切な操作が行われる可能性があります。Javaでは、大人数のチームで開発することが多く、他のメンバーが作成したクラスを使う場面が頻繁にあります。そのため、アクセス制御を使ってクラスの内部データや処理を適切に管理することで、安全で意図通りに動くプログラムを作りやすくなります。

以下はカプセル化がされていないクラスです。

【BankAccount.java】

|  |
| --- |
| **public class** BankAccount {   *//フィールド* **final int ACCOUNT\_NUMBER**; *//口座番号* String **accountName**; *//口座名義* **int money**; *//残高* **static int** *accountCount* = 0; *//口座数   //コンストラクタ* BankAccount(String name, **int** money) {  **accountNumber** = *accountCount*++;  **this**.**accountName** = name;  **this**.**money** = money;  }   *//残高を確認するメソッド* **void** printAccount() {  System.***out***.println(**"口座番号："** + **accountNumber**);  System.***out***.println(**"口座名義："** + **accountName**);  System.***out***.println(**"残高："** + **money** + **"円\n"**);  System.***out***.println();  } } |

【BankMain.java】

|  |
| --- |
| **public class** BankMain {  **public static void** main(String[] args) {   *// BankAccountクラスのインスタンスを生成* BankAccount[] accountArray = **new** BankAccount[]{  **new** BankAccount( **"田中太郎"**, 100000),  **new** BankAccount( **"鈴木五郎"**, 500000)  };   *// 口座情報一覧を表示* **for** (BankAccount account : accountArray) {  account.printAccount();  }   *// 不正に残高を変更(田中太郎の残高を-100000に変更)* System.***out***.println(**"田中太郎の残高を-100000に変更"**);  accountArray[0].**money** = -100000;   *// 口座情報一覧を表示* **for** (BankAccount account : accountArray) {  account.printAccount();  }  } } |

#### フィールドのカプセル化

独自クラスを作成するとき、そのクラスのフィールドはすべて「private」に設定します。これにより、そのフィールドはクラスの外から直接アクセスできなくなります。外部からフィールドに直接アクセスさせないことで、データを安全に管理しやすくなります。

フィールドは基本的に「private」に設定しますが、定数フィールドは公開しても値が書き換えられることがないため、「public」に設定することが多いです。

|  |
| --- |
| 書式:フィールドのカプセル化 |
| **public** クラス名{   *//フィールド* **private** 型名 フィールド名;  **private** 型名 フィールド名;   *//公開するフィールド(定数フィールド等)* **public static final** 型名 定数フィールド名;   *//メソッド* } |

【BankAccount.java】

|  |
| --- |
| **public class** BankAccount {   *//フィールド*  **public final int ACCOUNT\_NUMBER**; *//口座番号*  **private** String **accountName**; *//口座名義*  **private int money**; *//残高*  **public static int** *accountCount* = 0; *//口座数*  *//コンストラクタ* BankAccount(String name, **int** money) {  **accountNumber** = *accountCount*++;  **accountName** = name;  **money** = money;  }   *//残高を確認するメソッド* **void** printAccount() {  System.***out***.println(**"口座番号："** + **accountNumber**);  System.***out***.println(**"口座名義："** + **accountName**);  System.***out***.println(**"残高："** + **money** + **"円"**);  System.***out***.println();  } } |

#### メソッドのカプセル化

メソッドはそのメソッドの処理内容によって「private」、「public」、「protected」を設定します。

クラス内でのみ使用するメソッドは「private」に設定します。

クラス外から利用する必要があるメソッドには「public」を設定します。クラスの利用者はこの公開されているメソッドを通じて操作してもらい、内部構造を意識しない設計にします。

|  |
| --- |
| 書式:メソッドのカプセル化 |
| **public** クラス名{   *//フィールド*  *//クラス外から利用するメソッド*  **public** 戻り値の型 メソッド名() { } *//クラス内のみ利用するメソッド* **private** 戻り値の型 メソッド名() { }  } |

【BankAccount.java】

|  |
| --- |
| **public class** BankAccount {   *//フィールド*  **public final int ACCOUNT\_NUMBER**; *//口座番号*  **private** String **accountName**; *//口座名義*  **private int money**; *//残高*  **public static int** *accountCount* = 0; *//口座数*  *//コンストラクタ* **public** BankAccount(String name, **int** money) {  **accountNumber** = *accountCount*++;  **accountName** = name;  **money** = money;  }   *//残高を確認するメソッド* **public void** printAccount() {  System.***out***.println(**"口座番号："** + **accountNumber**);  System.***out***.println(**"口座名義："** + **accountName**);  System.***out***.println(**"残高："** + **money** + **"円"**);  System.***out***.println();  } } |

#### セッター(setter)

フィールドを「private」に設定するとクラスのデータに直接アクセスできなくなります。そのため、データを利用するための専用のメソッドが必要になります。

それが「セッター（setter）」と「ゲッター（getter）」です。

セッターはフィールドの値を設定（変更）するためのメソッドです。外部からフィールドの値を変更できるようにしながらも、不正な値が設定されないようにデータのバリデーションを行うことができます。これにより、フィールドに適切な値のみが設定されるよう管理することが可能です。

セッターには命名規則があり、「setフィールド名」というメソッド名にする必要があります。フィールド名の頭文字は大文字にします。

※バリデーション(validation)・・・データの妥当性を確認すること。バリデーションチェックとも言う

|  |
| --- |
| 書式:セッター(setter) |
| **public** クラス名{  *//フィールド*  *//セッター*  **public** void setフィールド名(引数) {  *//引数の値の確認*  フィールド = 引数の値; }  } |

#### ゲッター(getter)

ゲッターは、フィールドの値を取得するためのメソッドです。通常、フィールドの値を返すだけのシンプルな処理です。ゲッターを通じて値を取得することで、外部コードからフィールドの値を読み取れるようにします。フィールドの値を変更することはできません。

ゲッターにも命名規則があり、「getフィールド名」というメソッド名にする必要があります。フィールド名の頭文字は大文字にします。

|  |
| --- |
| 書式:セッター(setter) |
| **public** クラス名{  *//フィールド*  *//ゲッター*  **public** フィールドの型 getフィールド名 {  return フィールドの値; }  } |

【BankAccount.java】

|  |
| --- |
| **public class** BankAccount {   *//フィールド* **public final int ACCOUNT\_NUMBER**; *//口座番号* **private** String **accountName**; *//口座名義* **private int money**; *//残高* **public static int** *accountCount* = 0; *//口座数   //コンストラクタ* **public** BankAccount(String name, **int** money) {  **ACCOUNT\_NUMBER** = *accountCount*++;  **accountName** = name;  **money** = money;  }  *//セッター* **public void** setAccountName(String accountName) {  **if**(accountName == **null** || accountName.equals(**""**)) {  System.***out***.println(**"口座名義が不正です"**);  }**else**{  **this**.**accountName** = accountName;  }  }  **public void** setMoney (**int** money) {  **if**(money < 0) {  System.***out***.println(**"残高が不正です"**);  }**else** {  **this**.**money** = money;  }  }  *//ゲッター* **public** String getAccountName() {  **return accountName**;  }  **public int** getMoney() {  **return money**;  }  *//残高を確認するメソッド* **public void** printAccount() {  System.***out***.println(**"口座番号："** + **ACCOUNT\_NUMBER**);  System.***out***.println(**"口座名義："** + **accountName**);  System.***out***.println(**"残高："** + **money** + **"円"**);  System.***out***.println();  } } |

【BankMain.java】

|  |
| --- |
| **public class** BankMain {  **public static void** main(String[] args) {   *// BankAccountクラスのインスタンスを生成* BankAccount[] accountList = **new** BankAccount[]{  **new** BankAccount( **"田中太郎"**, 100000),  **new** BankAccount( **"鈴木五郎"**, 500000)  };   *// 口座情報一覧を表示* **for** (BankAccount account : accountList) {  account.printAccount();  }   *// 不正に残高を変更(田中太郎の残高を-100000に変更)* System.***out***.println(**"田中太郎の残高を-100000に変更"**);  accountList[0].setMoney(-100000);  System.***out***.println();   *// 不正に口座名義を変更(鈴木五郎の口座名義を空文字に変更)* System.***out***.println(**"鈴木五郎の口座名義を空文字に変更"**);  accountList[1].setAccountName(**""**);  System.***out***.println();   *// 口座情報一覧を表示* **for** (BankAccount account : accountList) {  System.***out***.println(account.getAccountName() + **"さんの口座情報"**);  account.printAccount();  }  } } |

このようにクラスにカプセル化を施し、外部からのアクセスを制限することで、エラーが起きにくいシステムを開発することができます。

基本的にはフィールドは「private」、メソッドは「public」と覚えておいてください。

### 継承

#### 継承とは

継承とは、新しいクラスを作成する際に、既存のクラスの機能を引き継ぐことができる仕組みです。この機能を使うことで、コードを再利用しやすくなり、同じ処理を何度も記述する必要がなくなるため、効率的なプログラム設計が可能になります。継承元のクラスは「親クラス（スーパークラス）」、その機能を引き継いで新たに作られるクラスは「子クラス（サブクラス）」と呼ばれます。

通常、新しいクラスを作るときには、そのクラスが必要とする機能を全て自分で定義する必要がありますが、継承を利用すると、既に似た機能を持つクラスを親クラスとして引き継ぎ、追加が必要な独自の機能だけを定義するだけで済みます。また、継承を使うことで、関連するクラスを階層的に整理でき、プログラムが見やすく、管理しやすくなります。

継承は複数階層にわたって重ねることが可能で、これにより複雑な機能を整理しやすくなります。例えば、Java SEのクラスライブラリでは、同じような機能を持つクラスが継承によって階層構造で整理されており、クラスの役割や関係性がわかりやすくなっています。

|  |
| --- |
| 継承 |
| ダイアグラム  自動的に生成された説明  親クラス  ↑　継承  子クラス |

<https://docs.oracle.com/javase/jp/21/docs/api/overview-tree.html>

まずは親クラス(スーパークラス)となるItemクラスを作成します。親クラスを作成するときはそのクラスを継承するクラスが共通でもつべき基本的なフィールドやメソッドを定義します。これにより、子クラスが親クラスを継承することで、親クラスで定義した共通の機能を引き継ぐことができ、効率的にコードの再利用が可能になります。

Itemクラスは商品の情報を保存するためのクラスです。商品であれば必ず、「名前」と「価格」があるため、「name」と「price」の2つのフィールドを作成します。【Item.java】

|  |
| --- |
| *//親クラス(スーパークラス)* **public class** Item {  *//フィールド* **private** String **name**;  **private int price**;  *//コンストラクタ* **public** Item(String name, **int** price) {  **this**.**name** = name;  **this**.**price** = price;  }  *//ゲッター* **public** String getName() {  **return name**;  }  **public int** getPrice() {  **return price**;  }  *//セッター* **public void** setName(String name) {  **this**.**name** = name;  }  **public void** setPrice(**int** price) {  **this**.**price** = price;  }  *//画面出力*  **public void** showInfo() {  　 System.***out***.println(**"商品名："** + **name**);  　 System.***out***.println(**"価格："** + **price** + **"円"**); 　　} } |

#### 継承の定義

継承には「extendsキーワード」を利用します。子クラスを定義するときにどのクラスを継承するかを「extends」で指定します。親クラスの変更は必要ありません。

|  |
| --- |
| 書式:継承の定義 |
| *//子クラス(サブクラス)* **public class** サブクラス名 **extends** スーパークラス名 {  *//独自フィールド  //独自メソッド*  } |

Itemクラスを継承するクラスとしてBookクラスを作成します。Bookクラスも商品の情報を保存するためのクラスですが、書籍に特有の情報を追加して管理します。例えば、Bookクラスでは「著者名」や「ISBN番号」など、書籍に関連したデータを扱うことが考えられます。したがって、「author」と「isbn」を独自フィールドとして記述します。書籍にも名前や値段がありますが、これらの機能は親クラスであるItemクラスですでに定義されているため記述する必要はありません。

【Book.java】

|  |
| --- |
| *//Itemクラスを継承した子クラス(サブクラス)* **public class** Book **extends** Item{  *//フィールド* **private** String **author**; *//著者* **private int isbn**; *//ISBN   //ゲッター* **public** String getAuthor() {  **return author**;  }  **public int** getIsbn() {  **return isbn**;  }  *//セッター* **public void** setAuthor(String author) {  **this**.**author** = author;  }  **public void** setIsbn(**int** isbn) {  **this**.**isbn** = isbn;  } } |

#### コンストラクタ

子クラスを作成するとき、親クラスのコンストラクタはそのままでは利用できません。親クラスと子クラスでは、インスタンスを生成するために必要な情報が異なるためです。子クラスには、インスタンス生成時に必要な情報を受け取る独自のコンストラクタを新たに定義する必要があります。

また、子クラスのコンストラクタ内では、必ず「super()キーワード」を用いて親クラスのコンストラクタを呼び出す必要があります。これは、親クラスのフィールドを適切に初期化するために重要です。なお、「super()」は必ず子クラスのコンストラクタの先頭に記述する必要があります。

親クラスのコンストラクタを呼び出すときは、必要に応じて引数を渡してください。

|  |
| --- |
| 書式:子クラスのコンストラクタ |
| **public class** 子クラス名{  *//フィールド*  *//コンストラクタ* **public** 子クラス名(引数, ...){  **super**(引数, ...); *//スーパークラスのコンストラクタを呼び出し(必要に応じて引数を渡す)  //独自フィールドの初期化* }  *//メソッド*  } |

【Book.java】

|  |
| --- |
| *//Itemクラスを継承した子クラス(サブクラス)* **public class** Book **extends** Item{  *//フィールド*  *//コンストラクタ* **public** Book(String name, **int** price, String author, **int** isbn) {  **super**(name, price); *//親クラスのコンストラクタの呼び出し*  **this**.**author** = author;  **this**.**isbn** = isbn;  }  *//ゲッター*  *//セッター*  } |

#### フィールドとメソッド

親クラスのフィールドとメソッドは自動的に継承されるため記述の必要はありません。子クラスの独自フィールドとメソッドを記述します。子クラスのメソッドから親クラスのメソッドを呼び出すときは「メソッド名()」で呼び出せます。明示的に親クラスのメソッドを指定したい場合は、「super.メソッド名()」と記述します。親クラスのフィールドも利用できますが、フィールドが「private」の場合はゲッターかセッター経由でアクセスします。

【Book.java】

|  |
| --- |
| *//Itemクラスを継承した子クラス(サブクラス)* **public class** Book **extends** Item{  *//フィールド*  *//コンストラクタ*  *//ゲッター*  *//セッター*  *//書籍情報を表示するメソッド* **public void** showInfo() {  **super**.showInfo();  *// System.out.println("名前：" + super.getName()); // System.out.println("価格：" + super.getPrice() + "円");*  System.***out***.println(**"著者："** + **author**);  System.***out***.println(**"ISBN："** + **isbn**);  }  } |

#### オーバーライド（Override）

オーバーライドとは、親クラスで定義されたメソッドを子クラスで再定義（上書き）することです。親クラスから継承したメソッドの処理内容をそのまま使うのではなく、子クラスで内容を変更して使いたい場合にオーバーライドを行います。オーバーライドすることで、親クラスのメソッドと同じ名前、同じ引数を持つメソッドを子クラスに定義し親クラスの動作を変更したり拡張することができます。

【オーバーライドの要件】

* メソッド名、引数の数と型、戻り値の型(シグネチャ)が親クラスのメソッドと同じ
* アクセス修飾子は親クラスのメソッドよりも範囲が広ければ変更可能
* メソッドの上に「@Override」アノテーションを付ける

シグネチャが完全に一致していない場合、コンパイラがそのメソッドを新しいメソッドとみなし、オーバーライドが成立せず、意図しない挙動をする可能性があります。それを防ぐために「@Override」アノテーションを追加します。

アノテーションとは、Javaプログラム内で特別なメタ情報（追加情報）を付けるための仕組みです。アノテーションは、コードに対する補足情報として使われ、主にコンパイラやフレームワーク、ツールに対して「このコードをどう扱うべきか」などを指示する役割を持っています。アノテーションは記号「＠」を使って記述し、クラス、メソッド、変数、引数などに付けることができます。

「＠Override」アノテーションは、メソッドに付けることができるアノテーションでこのメソッドがオーバーライドしたメソッドであることをコンパイラに示しています。アノテーションを付けなくてもエラーになりませんが、アノテーションを付けることによって、メソッド名やシグネチャが異なるとコンパイルエラーとなるためオーバーライドミスを伏せぐことができます。

|  |
| --- |
| 書式:オーバーライド(Override) |
| *//オーバーライドメソッド* @Override **アクセス修飾子 戻り値** メソッド名(引数) {　　　*//親クラスのメソッドと完全一致*    *//親クラスのメソッドを上書きする処理*  } |

「オーバーライド」と「オーバーロード」の違い

|  |  |  |
| --- | --- | --- |
|  | オーバーライド(Override) | オーバーロード(Overload) |
| 使用用途 | 親クラスのメソッドを子クラスで上書きする | 引数に応じて複数宣言 |
| 宣言場所 | 親クラスと子クラス | 同じクラス内 |
| メソッド名 | 同じ | 同じ |
| 引数 | 同じ | 異なる |
| 戻り値 | 同じ型 | 異なってもよい |

【Book.java】

|  |
| --- |
| *//Itemクラスを継承した子クラス(サブクラス)* **public class** Book **extends** Item{  *//フィールド*  *//コンストラクタ*  *//ゲッター*  *//セッター*  *//オーバーライドメソッド* @Override **public void** showInfo() { *//親クラスのshowInfo()と同じメソッド名、引数、戻り値*  **super**.showInfo();  System.***out***.println(**"著者："** + **author**);  System.***out***.println(**"ISBN："** + **isbn**); }  } |

#### 継承クラスの利用

Itemクラスを継承したBookクラスを利用するクラスを作成します。

【ItemMain.java】

|  |
| --- |
| **public class** ItemMain {  **public static void** main(String[] args) {   *//Itemクラスのインスタンスを生成* Item[] itemList = **new** Item[]{  **new** Item(**"リンゴ"**, 100),  **new** Item(**"みかん"**, 50),  **new** Item(**"バナナ"**, 80)  };   *//Bookクラスのインスタンスを生成* Book[] bookList = **new** Book[]{  **new** Book(**"Java入門"**, 3000, **"山田太郎"**, 123456789),  **new** Book(**"Python入門"**, 2500, **"佐藤花子"**, 987654321)  };    *//商品情報一覧を表示* System.***out***.println(**"商品情報一覧"**);  **for** (Item item : itemList) {  item.showInfo();  System.***out***.println();  }    *//書籍情報一覧を表示* System.***out***.println(**"書籍情報一覧"**);  **for** (Book book : bookList) {  book.showInfo();  System.***out***.println();  }  } } |

### 多態性 (ポリモーフィズム)

#### 多態性(ポリモーフィズム)とは

多態性とは、「継承」と「オーバーライド」を用いて柔軟で効率的なプログラムを作成できる仕組みのことです。同じ親クラスを持つ子クラスのオブジェクトを親クラスとして扱うことによって同じメソッドを実行しても実際のオブジェクトに応じた処理が実行されます。

|  |
| --- |
| 継承関係 |
| ダイアグラム  自動的に生成された説明 |

※clothes(クロウズ:衣服)

|  |
| --- |
| インスタンスを親クラス型の変数へ代入 |
|  |

前の章で、親クラスのItemクラスとその子クラスのBookクラスを作成しました。ここでは、新しいクラスとしてItemクラスを親クラスとするClothesクラスを作成します。Clothesクラスは衣服の情報を保存するためのクラスです。Clothesクラスの独自フィールドは「type」と「size」です。これらのフィールドは選択肢の中から指定する形式にするため、定数として各値を用意します。

【Clothes.java】

|  |
| --- |
| **public class** Clothes **extends** Item{  *//フィールド* **private** String **type**; *//種類* **private** String **size**; *//サイズ  // サイズを定数で定義*  **public static final** String ***SIZE\_SMALL*** = **"S"**;  **public static final** String ***SIZE\_MEDIUM*** = **"M"**;  **public static final** String ***SIZE\_LARGE*** = **"L"**;  **public static final** String ***SIZE\_XLARGE*** = **"XL"**;  *// 種類を定数で定義*  **public static final** String ***TYPE\_SHIRT*** = **"シャツ"**;  **public static final** String ***TYPE\_PANTS*** = **"パンツ"**;  **public static final** String ***TYPE\_OUTER*** = **"アウター"**;  *//コンストラクタ* **public** Clothes(String name, **int** price, String type, String size) {  **super**(name, price);  **this**.**type** = type;  **this**.**size** = size;  }  *//ゲッター* **public** String getType() {  **return type**;  }  **public** String getSize() {  **return size**;  }  *//セッター* **public void** setType(String type) {  **this**.**type** = type;  }  **public void** setSize(String size) {  **this**.**size** = size;  }  *//オーバーライドメソッド* @Override  **public void** showInfo() {  **super**.showInfo();  System.***out***.println(**"種類　："** + **type**);  System.***out***.println(**"サイズ："** + **size**);  }  *//サイズを確認するClothesクラス独自のメソッド* **public void** sizeCheck() {  System.***out***.println(**super**.getName() + **"のサイズは"** + **size** + **"です"**); }  } |

#### 多態性の機能

継承関係にあるクラスでは、親クラスの型の変数に子クラスのインスタンスを代入できるという特性があります。この特性を利用することで、インスタンスがどのクラスから生成されたものであっても、型を親クラスに統一して扱うことができます。これにより、複数のクラスをひとつの親クラスの型でまとめて処理でき、コードの柔軟性が向上します。

|  |
| --- |
| 書式:親クラス型への代入 |
| *//親クラス型変数への子クラスインスタンスの代入* 親クラス型 インスタンス変数名 = **new** 子クラス名(); |

この仕組みを利用することで便利に処理を記述することができますが、制約もあります。

【親クラス型の変数に子クラスのインスタンスを代入したときの制約】

* + 親クラスが持つフィールドやメソッドのみがアクセス可能
  + オーバーライドしたメソッドを呼び出すと、実際のインスタンスのクラスのメソッドが動作
  + 子クラス独自のメソッドを呼び出したいときは、キャストが必要

親クラス型の変数にインスタンスが代入されている場合、たとえ実際のインスタンスが子クラスであっても、親クラスのインスタンスとしてあつかわれます。そのため、親クラスで定義されているメンバーのみ利用することができます。

ただし、オーバーライドしたメソッドは親クラスと子クラスの両方に定義されているため、例外的に実際のインスタンスのメソッドが実行されます。

子クラス独自のメソッドを実行したい場合、親クラス型の変数のままでは利用できません。この場合、子クラスの型に型変換（キャスト）することで、そのメソッドを実行可能にします。型変換の際には、変換が可能かどうかを確認するために「instanceof」演算子が用意されています。「instanceof」は左辺の変数が右辺のクラスにキャストできる場合に「true」、できない場合は「false」を返します。

型変換を行うときは、「instanceof」演算子で安全に型変換が行えるかを確認してから型変換を行ってください。この確認処理を実行せずに型変換を行い、型変換に失敗したすると

**ClassCastException**

というエラーが発生し、プログラムが強制終了してしまいます。

|  |
| --- |
| 書式:instanceof演算子 |
| *//親クラス型の変数に子クラスのインスタンスを代入* 親クラス型 変数 = **new** 子クラス名();  *//インスタンス変数が子クラスのインスタンスであるかを判定(キャストできる場合はtrue)* **if**(変数 **instanceof** 子クラス型) {    *//子クラス型にキャスト後、子クラスの独自メソッドの実行* ((子クラス型)変数).メソッド();  } |

#### 多態性の利用

多態性の機能を利用したクラスを作成します。このクラスではItemクラスとBookクラス、Clothesクラスを利用します。これらのクラスはItemクラスの子クラスになるため、Itemクラス型の配列に格納することで、繰り返し処理を実行するなど効率よくプログラムを作成することができます。

【Clothes.java】

|  |
| --- |
| **public class** PolymorphismMain {  **public static void** main(String[] args) {   *// Itemクラスを継承した子クラス(サブクラス)のインスタンスを生成* Item[] itemList = **new** Item[]{   *// Itemクラスのインスタンスを生成* **new** Item(**"リンゴ"**, 100),  **new** Item(**"みかん"**, 50),  **new** Item(**"バナナ"**, 80),   *// Bookクラスのインスタンスを生成* **new** Book(**"Java入門"**, 3000, **"山田太郎"**, 123456789),  **new** Book(**"Python入門"**, 2500, **"佐藤花子"**, 987654321),   *// Clothesクラスのインスタンスを生成* **new** Clothes(**"Tシャツ"**, 2000, Clothes.***TYPE\_SHIRT***, Clothes.***SIZE\_MEDIUM***),  **new** Clothes(**"ジーンズ"**, 4000, Clothes.***TYPE\_PANTS***, Clothes.***SIZE\_LARGE***),  **new** Clothes(**"ダウンジャケット"**, 10000, Clothes.***TYPE\_OUTER***, Clothes.***SIZE\_XLARGE***)   };   *// 商品情報一覧を表示* System.***out***.println(**"商品情報一覧"**);  **for** (Item item : itemList) {  item.showInfo(); *// 各クラスでオーバーライドされたshowInfoメソッドが実行される* System.***out***.println();  }   *// Clothesクラスのインスタンスのみサイズを確認* System.***out***.println(**"サイズを確認"**);   *// サイズを確認するClothesクラス独自のメソッドは親クラス型の変数では呼び出せない // itemList[5].sizeCheck(); // コンパイルエラー* **if** (itemList[5] **instanceof** Clothes) { *// instanceof演算子でClothesクラスのインスタンスかどうか判定* ((Clothes) itemList[5]).sizeCheck(); *// サイズを確認するClothesクラス独自のメソッド* }  } } |

他の例として、商品の購入時の支払い方法についてのプログラムを作成します。支払いには複数の方法があり、購入者が選択した支払い方法によって支払い時に実行されるプログラムが変化します。

支払い方法には「現金」「クレジットカード」「電子マネー」があり、それぞれに応じた支払い処理を実行する仕組みを多態性（ポリモーフィズム）を使って実装します。

まずは個々の支払い方法の親クラスとなるPaymentクラスを作成します。このクラスではすべての支払い方法で共通の処理やフィールドを記述します。

親クラス

【Payment.java】

|  |
| --- |
| **package** payment;  **public class** Payment {   **private** String **paymentName**; *//支払い方法* **private int paymentPrice**; *//支払い金額   //コンストラクタ* **public** Payment(String paymentName, **int** paymentPrice) {  **this**.**paymentName** = paymentName;  **this**.**paymentPrice** = paymentPrice;  }  *//ゲッター* **public** String getPaymentName() {  **return paymentName**;  }  **public int** getPaymentPrice() {  **return paymentPrice**;  }  *//セッター* **public void** setPaymentName(String paymentName) {  **this**.**paymentName** = paymentName;  }  **public void** setPaymentPrice(**int** paymentPrice) {  **this**.**paymentPrice** = paymentPrice;  }  *//支払い処理を開始するメソッド* **public void** startPayment() {  System.***out***.println(**paymentName** + **"の支払い処理を開始します"**);  }  *//支払い処理のメソッド* **public void** pay() {  System.***out***.println(**"支払い方法："** + **paymentName**);  System.***out***.println(**"支払い金額："** + **paymentPrice** + **"円"**);  System.***out***.println();  }  *//支払い処理を終了するメソッド* **public void** endPayment() {  System.***out***.println(**"支払い処理が完了しました"**);  System.***out***.println(**"購入ありがとうございました"**);  } } |

次に、各支払い方法に対応する処理を、それぞれ「Paymentクラス」を継承したクラスとして作成します。支払い処理は共通の「pay()メソッド」をオーバーライドして実装することで、支払い方法にかかわらず、同じ呼び出し方で処理を実行できるようにします。

子クラス

【CashPayment.java】

|  |
| --- |
| **package** payment;  **import** java.util.Scanner;  **public class** CashPayment **extends** Payment {   **public static final** String ***CASH*** = **"現金"**;   *//コンストラクタ* **public** CashPayment(**int** paymentPrice) {  **super**(***CASH***, paymentPrice);  }   *////支払い処理のメソッド* @Override  **public void** pay() {  Scanner scanner = **new** Scanner(System.***in***);   *//支払い処理を開始するメソッド* **super**.startPayment();  **super**.pay();   **int** paymentPrice = **super**.getPaymentPrice();  **int** input = 0;   **do**{  System.***out***.println( paymentPrice + **"円を入れてください"**);  System.***out***.println(**"現在の金額："** + input + **"円"**);  System.***out***.print(**">"**);  input += scanner.nextInt();  System.***out***.println();   }**while**(paymentPrice > input);   System.***out***.println(**"現在の金額："** + input + **"円"**);  System.***out***.println();  System.***out***.println(**"金額が足りました"**);  System.***out***.println(**"おつりは"** + (input - paymentPrice) + **"円です"**);  System.***out***.println();   *//支払い処理を終了するメソッド* **super**.endPayment();  } } |

子クラス

【CreditCardPayment.java】

|  |
| --- |
| **package** payment;  **public class** CreditCardPayment **extends** Payment {   **public static final** String ***CREDIT\_CARD*** = **"クレジットカード"**;   *//コンストラクタ* **public** CreditCardPayment(**int** paymentPrice) {  **super**(***CREDIT\_CARD***, paymentPrice);  }   *//支払い処理のメソッド* @Override  **public void** pay() {   *//支払い処理を開始するメソッド* **super**.startPayment();  **super**.pay();   **int** input = 0;   **do** {  System.***out***.println(**"クレジットカードを挿入してください：0"**);  System.***out***.print(**">"**);  java.util.Scanner scanner = **new** java.util.Scanner(System.***in***);  input = scanner.nextInt();  System.***out***.println();   **if**(input == 0) {  System.***out***.println(**"認証中..."**);  System.***out***.println(**"認証完了"**);  System.***out***.println(**"支払い金額："** + **super**.getPaymentPrice() + **"円"**);  System.***out***.println(**"支払い処理終了"**);  System.***out***.println();  **break**;  }  }**while**(**true**);   *//支払い処理を終了するメソッド* **super**.endPayment();  } } |

子クラス

【ElectronicMoneyPayment.java】

|  |
| --- |
| **package** payment;  **import** java.util.Scanner;  **public class** ElectronicMoneyPayment **extends** Payment {   **public static final** String ***ELECTRONIC\_MONEY*** = **"電子マネー"**;   *//コンストラクタ* **public** ElectronicMoneyPayment(**int** paymentPrice) {  **super**(***ELECTRONIC\_MONEY***, paymentPrice);  }   *//支払い処理のメソッド* @Override  **public void** pay() {   *//支払い処理を開始するメソッド* **super**.startPayment();  **super**.pay();   **int** input = 0;   **do** {  System.***out***.println(**"電子マネーをタッチしてください：0"**);  System.***out***.print(**">"**);  Scanner scanner = **new** Scanner(System.***in***);  input = scanner.nextInt();  System.***out***.println();   **if**(input == 0) {  System.***out***.println(**"支払い金額："** + **super**.getPaymentPrice() + **"円"**);  System.***out***.println(**"支払い処理終了"**);  System.***out***.println();  **break**;  }  }**while**(**true**);   *//支払い処理を終了するメソッド* **super**.endPayment();  } } |

最後に、これらのクラスを利用するクラスを作成します。このクラスでは支払い方法を指定することで生成するインスタンスを変更しています。ただし、どのインスタンスを生成する場合でも、変数の型は親クラスである「Payment型」とするため、共通の方法で扱うことが可能です。

main処理のクラス

【PaymentMain.java】

|  |
| --- |
| **package** payment;  **import** java.util.Scanner;  **public class** PaymentMain {  **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);  System.***out***.println(**"支払い方法を数字で選択してください"**);  System.***out***.println(**"1:現金 2:クレジットカード 3:電子マネー"**);  System.***out***.print(**">"**);  **int** input = scanner.nextInt();  System.***out***.println();   System.***out***.println(**"支払い金額を入力してください"**);  System.***out***.print(**">"**);  **int** paymentPrice = scanner.nextInt();  System.***out***.println();   *//支払い方法に応じたインスタンスを生成* Payment payment =  **switch** (input){  **case** 1 -> **new** CashPayment(paymentPrice);  **case** 2 -> **new** CreditCardPayment(paymentPrice);  **case** 3 -> **new** ElectronicMoneyPayment(paymentPrice);  **default** -> **null**;  };   *//支払い処理* **if**(payment != **null**) {  payment.pay();  }  } } |

このように「多態性」とは、「継承」と「オーバーライド」の仕組みを活用することで、異なる種類のインスタンスを共通の型で扱いながら、それぞれのインスタンスに応じた動作をさせる仕組みです。これにより、支払い方法を変えたい場合でも、各支払い方法のクラスを個別に操作する必要はなく、共通のメソッドを呼び出すだけで対応できます。

多態性の活用により、プログラムは柔軟で拡張性のある設計になります。新しい支払い方法を追加する場合、そのクラスで既存のメソッドをオーバーライドするだけで簡単に対応でき、他の部分を変更せずにシステムを拡張できます。また、異なるクラス間で共通の操作を呼び出すため、コードの再利用が促進され、メンテナンスも容易になります。

### オブジェクト(Object)クラス

Javaでは全てのクラスの親クラスとなる「Objectクラス」が存在します。このクラスはJavaのクラス階層の最上位に位置しているため、Javaで作成するすべてのクラスは、明示的に指定しなくても自動的にObjectクラスを継承します。クラスを作成するときに「extends」で親クラスを指定しなければ、自動的に「Objectクラス」を継承したことになります。

|  |
| --- |
| 書式:Objectクラス |
|  |

したがって、すべてのクラスでObjectクラスで定義されているメソッドが利用できます。Objectクラスで定義されている代表的なメソッドは以下の通りです。

|  |  |
| --- | --- |
| 使用用途 | メソッド |
| オブジェクト比較 | boolean equals(Object o) |
| オブジェクトの文字列表現 | String toString() |

Stringクラスの説明でも触れたこれらのメソッドは、すべてのクラスで共通に利用できるようObjectクラスに定義されています。しかし、そのまま利用すると、意図と異なる動作や結果が返される場合があります。これは、Objectクラスで定義されたこれらのメソッドが汎用的な形式でのみ提供されており、各クラスに適した詳細な処理が未実装のためです。そのため、これらのメソッドを特定のクラスで正しく機能させたい場合は、必要に応じてそのクラス内でオーバーライドし、目的に沿った動作を定義する必要があります。

まずは、オーバーライドせずに利用するとどんな結果が帰ってくるかを確認します。CashPaymentクラスもObjectクラスを暗黙的に継承しているため、「equals（）メソッド」と「toString()メソッド」が利用できます。

【ObjectMain.java】

|  |
| --- |
| **package** payment;  **public class** ObjectMain {   **public static void** main(String[] args) {   *//CashPaymentクラスのインスタンスを生成  //Objectクラスは全てのクラスの親クラスのため、代入可能* Object obj = **new** CashPayment(1000);   *//toStringメソッドを呼び出し* String str = obj.toString();   *//出力* System.***out***.println(str);   *//printlnメソッドは引数のオブジェクトのtoStringメソッドを呼び出して出力* System.***out***.println(obj);   *//CashPaymentクラスのインスタンスを生成* Object obj2 = **new** CashPayment(1000);   *//equalsメソッドでインスタンスが等しいかどうかを判定* System.***out***.println(obj.equals(obj2));  } } |

このように意図した結果を取得することができません。意図した結果とするにはオーバーライドを行います。CashPaymentクラス以外でも利用できるようにPaymentクラスでオーバーライドを行います。

【Payment.java】

|  |
| --- |
| **public class** Payment{  *//endPayment()の下*  *//toStringメソッドのオーバーライド* @Override **public** String toString() {  **return "支払い方法："** + **paymentName** + **" 支払い金額："** + **paymentPrice** + **"円"**; } *//equalsメソッドのオーバーライド* @Override **public boolean** equals(Object obj) {  *//同じ参照値の場合はtrue* **if**(**this** == obj) {  **return true**;  }  *//参照値が違う場合でも支払い方法と支払い金額が等しい場合はtrue* **if**(obj **instanceof** Payment) {  Payment payment = (Payment)obj;  **if**(**this**.**paymentName**.equals(payment.**paymentName**)  && **this**.**paymentPrice** == payment.**paymentPrice**) {  **return true**;  }  }  *//それ以外はfalse* **return false**; }  } |

## 抽象クラス

親クラスを作成する際には、子クラスが共通で行う処理を親クラスにまとめ、具体的な処理は子クラスでオーバーライドして実装することが一般的です。このような親クラスでは、実際の処理内容を直接書かず、基盤となる部分だけを定義するため、親クラス自体をインスタンス化する必要はありません。そこで、この親クラスはインスタンス生成されて利用されることを意図していないことを明示するために「抽象クラス」として宣言します。抽象クラスにすることで、誤って親クラスをインスタンス化することができなくなり、子クラスが正しくオーバーライドすることを前提とした設計が可能になります。

### 抽象クラスとは

#### 抽象クラスの特徴

クラスを抽象クラスとして宣言した場合の特徴を示します。

* そのクラスをインスタンス化できなくなる(newできない)。継承の親クラスとしてのみ利用される。
* 「抽象メソッド」を記述できる。「抽象メソッド」は具体的な処理を記述せず、メソッド宣言のみを行います。子クラスではこのメソッドをオーバーライドで実装することを強制される。
* 「抽象メソッド」があるクラスは「抽象クラス」としなければならないが、「抽象クラス」は必ずしも「抽象メソッド」を記述する必要はない
* 抽象メソッド以外に具体的なメソッドやフィールドも定義できる。

#### 抽象クラスの定義

実際に抽象クラスを作成します。抽象クラスでは「abstractキーワード」を利用します。処理が記述されている具体的なクラスではなく、処理が未確定のあいまいなクラスという意味で「抽象クラス」と呼ばれます。

※abstract(アブストラクト)・・・抽象的な

|  |
| --- |
| 書式:抽象クラスと抽象メソッド |
| *//抽象クラス* **public abstract** クラス名{  *//通常のフィールド  //通常のメソッド    //抽象メソッド(処理は記述せず、戻り値の型、メソッド名、引数のみ指定)* **public abstract** 戻り値の型 メソッド名(引数);  } |

【Course.java】

|  |
| --- |
| **import** java.util.Arrays;  *//抽象クラス* **public abstract class** Course {   *//フィールド* **private** String **name**; *//コース名* **private int[] startMount**; *//開講月   //コンストラクタ* **public** Course(String name, **int**[] startMonth) {  **this**.**name** = name;  **this**.**startMount** = startMonth;  }  *//ゲッター* **public** String getName() {  **return name**;  }  **public int**[] getStartMonth() {  **return startMount**;  }  *//セッター* **public void** setName(String name) {  **this**.**name** = name;  }  **public void** setStartMonth(**int**[] startMonth) {  **this**.**startMount** = startMonth;  }   *//抽象メソッド(子クラスでオーバーライドを強制)* **public abstract void** showCourse();   *//コースの文字列表現* @Override  **public** String toString() {  **return "Course [name="** + **name** + **", startMount="** + Arrays.*toString*(**startMount**) + **"]"**;  } } |

#### 抽象クラスを継承したクラス

抽象クラスを継承したクラスでは、抽象メソッドをすべてオーバーライドする義務があります。もし抽象メソッドのいずれかをオーバーライドしなかった場合、そのクラス自体も抽象クラスとして宣言しなければなりません。つまり、この場合もインスタンス化はできないということです。継承関係にあるクラスで具体的な実装が完了していない場合は、最終的に全ての抽象メソッドが実装されるまで抽象クラスでなければなりません。

※あいまいな抽象メソッドの具体的な処理を記述することを「実装」といいます。

【ITCourse.java】

|  |
| --- |
| **public class** ITCourse **extends** Course{   **public static final** String ***NAME*** = **"IT生産サポート科"**;   *//コンストラクタ* **public** ITCourse() {  **super**(***NAME***, **new int**[] {5, 11});  }  *//抽象メソッドのオーバーライド* @Override  **public void** showCourse() {  System.***out***.println(**"コース名："** + **super**.getName());  System.***out***.print(**"入所月　："**);  **for**(**int** month : **super**.getStartMonth()) {  System.***out***.print(month + **"月 "**);  }  System.***out***.println();  System.***out***.println(**"ITエンジニアになるための基礎から応用まで学べるコースです"**);  }  @Override **public** String toString() {  **return "IT"** + **super**.toString(); }  } |

【IoTCourse.java】

|  |
| --- |
| **public class** IoTCourse **extends** Course{   **public static final** String ***NAME*** = **"IoT機器開発科"**;   *//コンストラクタ* **public** IoTCourse() {  **super**(***NAME***, **new int**[] {3, 9});  }  *//抽象メソッドのオーバーライド* @Override  **public void** showCourse() {  System.***out***.println(**"コース名："** + **super**.getName());  System.***out***.print(**"入所月　："**);  **for**(**int** month : **super**.getStartMonth()) {  System.***out***.print(month + **"月 "**);  }  System.***out***.println();  System.***out***.println(**"IoT技術者になるための基礎から応用まで学べるコースです"**);  }  @Override **public** String toString() {  **return "IoT"** + **super**.toString(); }  } |

【FACourse.java】

|  |
| --- |
| **public class** FACourse **extends** Course{   **public static final** String ***NAME*** = **"FAシステム技術科"**;   *//コンストラクタ* **public** FACourse() {  **super**(***NAME***, **new int**[] {6, 12});  }  *//抽象メソッドのオーバーライド* @Override  **public void** showCourse() {  System.***out***.println(**"コース名："** + **super**.getName());  System.***out***.print(**"入所月　："**);  **for**(**int** month : **super**.getStartMonth()) {  System.***out***.print(month + **"月 "**);  }  System.***out***.println();  System.***out***.println(**"FA機器の設計から制御技術まで学べるコースです"**);  }  @Override **public** String toString() {  **return "FA"** + **super**.toString(); }  } |

#### 抽象クラスの利用

抽象クラスはインスタンス生成できませんが、クラスの型として利用することはできます。抽象クラスの型を使用することで多態性の機能を活用できます。

【CourseMain.java】

|  |
| --- |
| **public class** CourseMain {  **public static void** main(String[] args) {   *//各コースのインスタンス生成* Course[] courses = {  **new** FACourse(),  **new** ITCourse(),  **new** IoTCourse()  };  *//toString()の呼び出し*  **for**(Course c : courses) {  System.***out***.println(c);  }  System.***out***.println();   *//各コースのオーバーライドされたshowCourseメソッドの呼び出し* **for**(Course c : courses) {  c.showCourse();  System.***out***.println();  }  } } |

## インターフェイス

前の章では「抽象クラス」について説明しました。この章で扱う「インターフェイス」も、考え方としては抽象クラスに似ています。どちらもクラスの設計で共通のメソッドや動作を定義するためのものであり、具体的な実装はサブクラスや実装クラスに任せる仕組みです。違いは、「インターフェイス」には具体的な処理内容を記述できないことです。

### インターフェイスとは

「抽象クラス」には、「抽象メソッド」の他に通常の処理を記述したメソッドを定義することができました。「インターフェイス」のメソッドはすべて「抽象メソッド」となり、フィールドも「定数」のみしか定義できません。「抽象クラス」と「インターフェイス」の違いを表で示します。

|  |  |  |
| --- | --- | --- |
|  | 抽象クラス | インターフェイス |
| インスタンス化 | 不可 | 不可 |
| フィールド | フィールドと定数 | 定数のみ (public static final) |
| メソッド | 通常のメソッドと抽象メソッド | 抽象メソッドのみ (public abstract) |
| 宣言のキーワード | class | interface |
| 継承時のキーワード | extends | implements |
| 継承元の数 | 1つ | 複数可能 |
| 使用例 | 親クラスとして機能を提供 | 子クラスの動作を保証 |

#### インターフェースの特徴

「抽象クラス」は、親クラスとして共通処理をまとめて、子クラスでの処理を統一するために使います。これにより、重複したコードを減らし、効率的にクラスを作成できます。

例えば、動物のクラスを考えた時、「食べる」や「寝る」といった動作はどの動物にも共通するものです。抽象クラスではこれらの共通の動作を定義しておいて、動物ごとに異なる部分（例えば「鳴く」や「走る」など）だけを、子クラスで作ります。

一方で、「インターフェース」は処理内容を持たないため、異なるクラスに共通のメソッドを実装させる目的で使用します。インターフェースには抽象メソッドのみを定義することができ、これを実装したクラスは、インターフェースで定義されたメソッドを必ず実装しなければならないのでその処理を持っていることが保証されます。

例えば、動物の中で「飛ぶ」や「泳ぐ」といった動作は一部の動物のみが行えます。これらの「～できる」といった機能をその動物に付与するとき、「飛べる」という「インターフェース」を実装することで、その動物は「飛ぶ」機能を持っていることが保証されます。

このように「インターフェース」はその「インターフェース」を実装しているクラスが特定の機能を持っていることを保証するときに利用する仕組みです。

自身でインターフェースを作成する機会は少ないですが実装処理は頻繁に行います。特定のインターフェイスを実装すると、指定のメソッドを追加する必要があるため利用方法は覚えておいてください。

#### インターフェースの定義

インターフェースを作成します。インターフェースはクラスではないので「classキーワード」ではなく「interfaceキーワード」を利用します。

|  |
| --- |
| 書式:インターフェース(interface) |
| *//インターフェース* **public interface** インターフェース名 {   *//定数(自動的にpublic static finalが付与される)* 型 ***定数名***;   *//抽象メソッド(自動的にpublic abstractが付与される)* 戻り値の型 メソッド名(引数); } |

まずは、抽象クラスのＡｎｉｍａｌクラスを作成します。このクラスは動物が持っている共通の動作を抽象メソッドで用意します。

【Ａnimａｌ.java】

|  |
| --- |
| *//抽象クラス* **public abstract class** Animal {    *//フィールド* **private** String **name**; *//名前    //コンストラクタ* **public** Animal(String name) {  **this**.**name** = name;  }  *//ゲッター* **public** String getName() {  **return name**;  }  *//セッター* **public void** setName(String name) {  **this**.**name** = name;  }  *//動物が共通で持っている動作  //抽象メソッド* **public abstract void** eat(); *//食べる*  **public abstract void** sleep(); *//眠る* } |

次に、特定の動物が持つ機能としてＦｌｙインターフェースを作成します。

【Ｆｌｙable.java】

|  |
| --- |
| *//インターフェース* **public interface** Flyable {   *//定数(public static finalは省略)* String ***FLY\_MSG*** = **"は飛びます"**;  String ***LAND\_MSG*** = **"は着地します"**;  *//抽象メソッド(public abstractは省略)* **void** fly(); *//飛ぶ* **void** land(); *//着地* } |

#### インターフェースを実装したクラス

クラスを受け継ぐことを「継承(extends)」と呼ばますが、インターフェースを受け継ぐ場合は「実装（implements）」と呼びます。よって、インターフェースを実装するクラスを作成する場合は、class名の後に「implementsキーワード」で実装するインターフェースを指定します。

|  |
| --- |
| 書式:インターフェース(interface)の実装 |
| *//インターフェースを実装するクラス* **public class** クラス名 **implements** インターフェイス{  } |

また、継承は1つのクラスのみでしたが、インターフェースは複数指定することができます。複数指定する場合はインターフェイス名を「,」区切りで記述します。クラスの継承とインターフェイスの実装を同時することも可能です。その場合は、「extends」後に「implements」を指定します。

|  |
| --- |
| 書式:インターフェース(interface)の複数実装 |
| *//インターフェースを実装するクラス* **public class** クラス名 **extends** 親クラス**implements** インターフェイス, インターフェイス{  } |

インターフェイスを継承するインターフェイスを作成することもできます。その場合は「extendsキーワード」を指定します。

【Dog.java】

|  |
| --- |
| **public class** Dog **extends** Animal{   **public static final** String ***NAME*** = **"犬"**;   *//コンストラクタ* **public** Dog() {  **super**(***NAME***);  }   *//抽象メソッドのオーバーライド* @Override  **public void** eat() {  System.***out***.println(**super**.getName() + **"は骨を食べました"**);  }   @Override  **public void** sleep() {  System.***out***.println(**super**.getName() + **"は床で眠りました"**);  } } |

【Crow.java】

|  |
| --- |
| **public class** Crow **extends** Animal **implements** Flyable{   **public static final** String ***NAME*** = **"カラス"**;   *//コンストラクタ* **public** Crow() {  **super**(***NAME***);  }   *//抽象メソッドのオーバーライド* @Override  **public void** eat() {  System.***out***.println(**super**.getName() + **"はゴミを食べました"**);  }   @Override  **public void** sleep() {  System.***out***.println(**super**.getName() + **"は枝で眠りました"**);  }   *//Flyableインターフェースの抽象メソッドのオーバーライド* @Override  **public void** fly() {  System.***out***.println(**super**.getName() + Flyable.***FLY\_MSG***);  }   @Override  **public void** land() {  System.***out***.println(**super**.getName() + Flyable.***LAND\_MSG***);  } } |

【AnimalMain.java】

|  |
| --- |
| **public class** AnimalMain {  **public static void** main(String[] args) {  *// 各インスタンス生成* Animal[] animals = {  **new** Dog(),  **new** Crow()  };   *// 各インスタンスのメソッド呼び出し* **for**(Animal animal : animals) {  animal.eat();  animal.sleep();   *// Flyableインターフェースを実装している場合のみ(飛ぶ機能を持っている場合のみ)* **if**(animal **instanceof** Flyable) {  *//Flyable型にキャスト* Flyable flayAnimal = (Flyable) animal;   *//Flyableインターフェースのメソッド呼び出し* flayAnimal.fly();  flayAnimal.land();  }  System.***out***.println();  }  } } |

このようにクラスに対して特定の機能を追加したい場合などにインターフェイスは利用します。Java　SEではさまざまなインターフェイスが用意されているためそれらの機能を利用する際に使います。

### スレッド

インターフェイスの例としてスレッド処理を紹介します。スレッドとはプログラムの中の「処理の流れ」のことです。Javaプログラムは通常、メインスレッドと呼ばれる単一のスレッドで動作しますが、複数のスレッドを作成することで、並列して処理を実行することができます。ネットワーク通信や、複数アクセスに対応させる場合などに利用されます。

|  |
| --- |
| スレッド |
| シングルスレッド　　　　　　　　　　　　　　マルチスレッド |

#### スレッドの作成

スレッドを作成するには以下の２つの方法があります。

1. 「Theadクラス」を継承する方法

「Threadクラス」を継承したクラスを作成し、その中で「runメソッド」をオーバーライドします。

作成したクラスのインスタンスを生成し、そのインスタンスの「startメソッド」を呼び出すとスレッド　が動作を開始します。

この方法はシンプルにスレッドを定義できますが、他のクラスを継承できないため、汎用性が制限されます。

1. 「Runnableインターフェイス」を実装する方法

「Ｒｕｎｎａｂｌｅインターフェイス」を実装したクラスを生成し、その中で「runメソッド」をオーバーライドします。実装したクラスのインスタンスを作成し、それを「Threadクラス」のコンストラクタに渡してスレッドを生成します。

作成した「Threadクラス」のインスタンスの「startメソッド」を呼び出すとスレッドが動作を開始します。

この方法の方が他クラスを継承できるため、汎用性が高いです。実務ではこちらが推奨されることが多いです。

Threadクラスにはさまざまなメソッドが用意されているため興味がある方はリファレンスを調べてみてください。

【ＳｕｂＴｈｒｅaｄ.java】

|  |
| --- |
| **public class** SubThread **extends** Thread {   @Override  **public void** run() {  *//サブスレッドの処理* System.***out***.println(**"サブスレッドを開始します"**);  **for**(**int** i = 1; i <= 10; i++) {  System.***out***.println(**"サブスレッド："** + i);    *//1秒間スリープ* **try** {  Thread.*sleep*(1000);   } **catch** (InterruptedException e) {  }  }  } } |

【MainＴｈｒｅaｄ.java】

|  |
| --- |
| **public class** MainThread {   **public static void** main(String[] args) {  *//サブスレッドの生成*  SubThread subThread = **new** SubThread();  *//サブスレッドの開始*  subThread.start();   *//メインスレッドの処理* System.***out***.println(**"メインスレッドを開始します"**);  **for**(**int** i = 1; i <= 10; i++) {  System.***out***.println(**"メインスレッド:"** + i);   *//1秒間スリープ* **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  }  }  System.***out***.println(**"メインスレッドを終了します"**);  } } |

【RunnableClass.java】

|  |
| --- |
| **public class** RunnableClass **implements** Runnable{   @Override  **public void** run() {  *//サブスレッドの処理* **for**(**int** i = 1; i <= 10; i++) {   *//通信処理など   //1秒間スリープ* **try** {  Thread.*sleep*(1000);   } **catch** (InterruptedException e) {  }  }  } } |

【RunnableMain.java】

|  |
| --- |
| **public class** RunnableMain {  **public static void** main(String[] args) {   *//サブスレッドの生成* RunnableClass runnableClass = **new** RunnableClass();  Thread thread = **new** Thread(runnableClass);   *//サブスレッドの開始* thread.start();   *//メインスレッドの処理* System.***out***.println(**"ロード中"**);  System.***out***.print(**"["**);   *//サブスレッドが終了するまでループ* **while**(thread.isAlive()) {  System.***out***.print(**"※"**);   *//1秒間スリープ* **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  }  }  System.***out***.println(**"]"**);  System.***out***.println(**"ロード完了"**);  } } |

### 匿名クラスとラムダ式

Javaには簡潔にコードを記述するための「匿名(無名)クラス」と「ラムダ式」という便利な方法があります。これらはクラスの継承や、インターフェイスの実装を簡単に行うための仕組みです。慣れるまでは扱いづらいですが、慣れれば非常に簡潔にプログラムを記述できるようになります。

#### 匿名(無名)クラス

匿名クラスとはその名の通り「名前を持たないクラス」のことです。一度限りの特定の処理を簡潔に記述するために使われます。インターフェースや抽象クラスを実装したり、既存のクラスを継承して使用します。

通常、子クラスを作成する際は、新しいクラス名を宣言して独立したクラスを定義しますが、匿名クラスでは、これを省略しインスタンス生成と同時に処理内容を記述します。

ただし、そうするとインスタンス生成をする際に元となるクラスがないため、生成するクラスの親クラスやインターフェイスを元として生成します。つまり、継承や実装の機能を利用して、クラス生成時にその場で定義されたクラスが匿名クラスとなります。

匿名クラスは独立したクラスとして宣言していないため、再利用ができません。そのクラスから複数のインスタンスを生成することはできず、１度限りの使い切りのクラスとなります。

特定の処理をその場限りで記述する際に便利ですが、大規模な処理には適しません。

|  |
| --- |
| 書式:匿名クラス |
| *//匿名クラスのインスタンス生成＆処理内容* 親クラス インスタンス変数 = **new** 親クラス() {  *//フィールド  //メソッド* }; |

【RunnableClass.java】

|  |
| --- |
| **public class** AnonymousClass {  **public static void** main(String[] args) {   *//匿名クラス(継承:Threadクラス)* Thread thread1 = **new** Thread() {  @Override  **public void** run() {  *//サブスレッドの処理* System.***out***.println(**"Thread1: 開始"**);  **for**(**int** i = 1; i <= 10; i++) {   System.***out***.println(**"Thread1: "** + i);   *//1秒間スリープ* **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  }  }  System.***out***.println(**"Thread1: 終了"**);  }  };   *//匿名クラス(実装:Runnableインターフェース)* Runnable runnable = **new** Runnable() {  @Override  **public void** run() {  *//サブスレッドの処理* System.***out***.println(**"Thread2: 開始"**);  **for**(**int** i = 1; i <= 10; i++) {   System.***out***.println(**"Thread2: "** + i);   *//1秒間スリープ* **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  }  }  System.***out***.println(**"Thread2: 終了"**);  }  };   Thread thread2 = **new** Thread(runnable);   *//サブスレッドの開始* thread1.start();  thread2.start();   } } |

#### ラムダ式

ラムダ式は、匿名クラスの記述をより簡略化するための構文で、関数型インターフェース(抽象メソッド

１つだけのインターフェース)を実装する場面で利用されます。

ラムダ式は、メソッドが1つのインターフェースを実装する場面でのみ利用できます。また、複数メソッドを記述するなども行えません。１度限りの使い切りなのは匿名クラスと同じです。

大幅なコード量の削減ができ、エディターの補完機能等で簡単に入力することができます。

ラムダ式は必要最低限の記述でインスタンス生成ができる構文です。推測できる内容は排除します。

* new演算子はラムダ式自体がインスタンス生成の意味合いを持つため排除
* new演算子の後のクラス名も型で推測できるため排除
* メソッド名も１つのメソッドしか持たないため排除

処理内容を記述するブロックに対してswitch式のときにも利用した「->(アロー演算子)」を使います。

また、処理内容が1行の場合は「{}」も省略することができます。

|  |
| --- |
| 書式:ラムダ式 |
| *//匿名クラスのインスタンス生成＆処理内容* インターフェース インスタンス変数 = (引数) -> {  *//オーバーライドするメソッドの処理内容*  }; |

【ＬａｍｄａＭａｉｎ.java】

|  |
| --- |
| **public class** LamdaMain {  **public static void** main(String[] args) {   *//ラムダ式(実装:Runnableインターフェース)  //関数型インターフェースのメソッドのみラムダ式で記述可能  //Runnableインターフェースは、runメソッドのみを持つ関数型インターフェース* Runnable runnable = () -> {  *//サブスレッドの処理* **for**(**int** i = 1; i <= 10; i++) {   System.***out***.println(**"Thread: "** + i); **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  }  }  };   *//スレッドの実行* Thread thread = **new** Thread(runnable);thread.start();  } } |

## 例外

プログラムの実行中に発生するエラーのことを例外(Exception)と呼びます。Javaでは、これらの発生した例外の種類に応じた「例外クラス」が用意されています。

例外が発生すると、自動的にこれらのクラスのインスタンスが生成され、例外発生箇所に投げられます(throw)。

### 例外の種類

どんなエラーに対してどのような「Exceptionクラス」が用意されているかを説明します。

|  |
| --- |
| 書式:例外の継承関係 |
|  |

Javaの例外クラスは「Throwableクラス」を継承し、大きく２種類に分類できます。

* Ｅｒｒｏｒクラス

ＯＳや端末自体のエラー。プログラムではどうしようもない致命的なエラーが多いです。

メモリー不足や、フリーズなど。

* Exceptionクラス

プログラムで対処可能なエラー。プログラム内で適切に処理する必要があります。

Exceptionクラスもさらに非検査例外と検査例外の2種類に分類できます。

* + - 非検査例外

実行時に発生する例外で正しくプログラミングすることで発生を防ぐことができる例外です。

null確認を行うことや、配列の要素番号を間違えないようにするなど。

* + - 検査例外

コンパイル時にチェックする例外で、この例外が発生する可能性があるメソッドを利用する場合は、必ず例外に対処する処理を記述する必要があります。

ファイルが見つからないときや、ネットワーク接続がうまくいかない時など。

頻繁に発生する例外について示します。

|  |  |
| --- | --- |
| 例外(Exception) | 発生原因 |
| 非検査例外 | |
| NullPointerException | nullアクセス |
| ArrayIndexOutOfBoundsException | 配列の範囲外の要素番号アクセス |
| IndexOutOfBoundsExｃeption | リストや文字列の範囲外の要素番号アクセス |
| ArithmeticException | 0で割る処理 |
| ClassCastException | 指定の型変換ができない(キャスト時) |
| NumberFormatException | 不正文字列の数値変換 |
| 検査例外 | |
| IOException | 入出力エラー(読み込みエラーなど) |
| FileNotFoundException | 指定ファイルが見つからない |
| SQLException | データベースエラー |
| InterruptedException | スレッドが中断された場合 |
| ClassNotFoundException | 指定クラスが見つからない |

このほかにもさまざまな例外がJava SEには用意されています。詳しくはリファレンスを参照してください。

### 例外の対処

Javaの例外への対処方法は、例外の種類（検査例外と非検査例外）によってアプローチが異なります。それぞれの違いを整理し、対処方法を以下で説明します。

#### 非検査例外の対処

非検査例外は、コンパイル時に処理が強制されない例外です。これにより、開発者は発生する可能性を予測して、これらの例外が発生しないように注意しながらプログラムを記述しなければなりません。

主な対処方法は、例外が発生する可能性がある処理を実行するときに事前チェックを行う方法です。

|  |
| --- |
| 書式:事前チェック |
| *//nullの可能性がある変数* **if**(変数 != **null**) {  *//nullPointerExceptionを発生させるメソッド* }  *//0の可能性がある変数* **if**(変数 != 0) {  *//ArithmeticExceptionを発生させるメソッド* 値 / 変数; } |

【RuntimeExceptionMain.java】

|  |
| --- |
| **public class** RuntimeExceptionMain {  **public static void** main(String[] args) {   *//NullPointerExceptionを発生させるメソッド* String str = **null**;  System.***out***.println(str.length());   *//ArithmeticExceptionを発生させるメソッド* **int** a = 10;  **int** b = 0;  System.***out***.println(a / b);   *//ArrayIndexOutOfBoundsExceptionを発生させるメソッド* **int**[] array = **new int**[3];  System.***out***.println(array[3]);   *//ClassCastExceptionを発生させるメソッド* Object obj = **new** Object();  String str2 = (String)obj;   *//NumberFormatExceptionを発生させるメソッド* String str3 = **"abc"**;  **int** num = Integer.*parseInt*(str3);  } } |

#### 検査例外の対処

検査例外は、コンパイル時に処理が強制される例外です。これにより、プログラムで適切にエラー処理を行うことが求められます。

対処方法は、例外が発生した場合、その例外をキャッチして発生した例外に応じた適切な処理を実行することです。

例外をキャッチするには「try-catch構文」を利用します。

|  |
| --- |
| 書式:try-catch |
| *//例外を検知するtryブロック* **try** {   *//例外が発生するメソッドの呼び出し    //発生した例外をキャッチするcatchブロック* }**catch** (例外のクラス 変数){  *//例外が発生したときの処理* } |

catchブロックには発生する例外に合わせた例外クラスの変数を用意します。例外が発生すると、この変数に例外クラスのインスタンスが代入されます。

複数の例外が発生する可能性がある場合は、catchブロックを複数用意することもできます。

|  |
| --- |
| 書式:try-catch （catchブロック複数） |
| **try** {}**catch** (例外のクラス1 変数){  *//例外1が発生したときの処理* } **catch** (例外のクラス2変数){  *//例外2が発生したときの処理* } **catch** (例外のクラス3変数){  *//例外3が発生したときの処理* } |

例外クラスの型はその例外クラスの親クラスの型でもキャッチすることできます。また、catchブロックは上から順に判定されるため、if-elseブロックのように、緩い条件を先に記述すると、下のブロックは判定されません。したがって、より子クラスの例外を先に記述し、親クラスの例外は後に記述します。

【FileReaderMain.java】

|  |
| --- |
| **import** java.io.BufferedReader; **import** java.io.FileNotFoundException; **import** java.io.FileReader; **import** java.io.IOException;  **public class** FileReaderMain{  **public static void** main(String[] args) {   *//例外が発生する可能性がある処理* **try** {   *//読み込むファイルを指定(srcフォルダ内にtext.txtファイルを用意)* FileReader fr = **new** FileReader(**"src\\text.txt"**);  *//ファイル読み込み用のバッファを用意(1文字ずつ読み込むのは非効率なため)* BufferedReader br = **new** BufferedReader(fr);   *//ファイルの内容を1行ずつ読み込んで表示* **while** (br.ready()) {  System.***out***.println(br.readLine());  br.close(); *//IOExceptionを発生させる*  }   *//ファイルが見つからなかった場合の例外処理* }**catch** (FileNotFoundException e) {  System.***out***.println(**"ファイルが見つかりませんでした"**);  System.***out***.println(**"例外メッセージ: "** + e.getMessage());   *//ファイルの読み込みに失敗した場合の例外処理* } **catch** (IOException e) {  System.***out***.println(**"ファイルの読み込みに失敗しました"**);  e.printStackTrace(System.***out***);  }   System.***out***.println(**"プログラム終了"**);  } } |

#### finallyブロック

Javaでは、例外処理の一環として「finallyブロック」が提供されています。

finallyブロックは、例外の発生の有無にかかわらず必ず実行される処理を記述するために使用されます。

主にリソースの解放や後処理に利用されます。

プログラムで外部のリソース(ファイル操作,ネットワーク接続)を利用する処理を実行する際は、不要になったタイミングで適切に使用の終了(リソースの解放) をする必要があります。

適切にリソースの解放をしない場合、ファイルがロックされたりや、ネットワークの切断が行われない可能性があります。

このリソースの解放処理は例外の発生の有無にかかわらず必ず実行する必要があるため、finallyブロック内に記述します。

|  |
| --- |
| 書式:finally |
| **try** {}**catch** (例外のクラス1 変数){  *//例外1が発生したときの処理* } **catch** (例外のクラス2変数){  *//例外2が発生したときの処理* } **finally** {  *//必ず実行される処理* } |

また、Java 7から「try-with-resource構文」を利用することでリソースを自動的に開放できるようになりました。

tryの後の()内で開放する必要があるリソースを宣言することでtry-catch処理が終了したタイミングで自動的にリソースが解放されます。

|  |
| --- |
| 書式:try-with-resource |
| **try (**  *//解放する必要があるリソースの宣言*  **)**{  *//例外が発生する可能性がある処理* }**catch** (例外のクラス1 変数){  *//例外が発生したときの処理* } |

現在はtry-with-resource構文での記述が推奨されています。

ここでは紹介しませんが例外は自分で発生されたり、例外を別のメソッドに投げたりすることもできます。

【FileWriterMain.java】

|  |
| --- |
| **import** java.io.\*; **import** java.util.Scanner;  **public class** FileWriterMain {  **public static void** main(String[] args) {   *//tryブロック外で利用するため、tryブロック外で宣言* FileWriter fr = **null**;  BufferedWriter bw = **null**;  *//例外が発生する可能性がある処理* **try** {   *//書き込むファイルを指定* fr = **new** FileWriter(**"src\\text.txt"**);  bw = **new** BufferedWriter(fr);   *//ファイルに書き込む* System.***out***.println(**"ファイルに書き込みます"**);  System.***out***.println(**"文字列を入力してください(終了する場合は0を入力)"**);  Scanner scanner = **new** Scanner(System.***in***);    *//無限ループ* **for** (**int** i = 1; ; i++) {  System.***out***.print(i + **"行目: "**);  String str = scanner.nextLine();    *//入力された文字列が"0"の場合はループを抜ける* **if** (str.equals(**"0"**)) {  **break**;  }**else** {  bw.write(str); *//１行分をファイルに書き込む* bw.newLine(); *//改行を入れる* }  }  System.***out***.println(**"ファイルに書き込みが完了しました"**);   }**catch** (FileNotFoundException e) {  System.***out***.println(**"ファイルが見つかりませんでした"**);  System.***out***.println(**"例外メッセージ: "** + e.getMessage());   } **catch** (IOException e) {  System.***out***.println(**"ファイルの読み込みに失敗しました"**);  e.printStackTrace(System.***out***);   *//必ず実行する処理* }**finally** {  System.***out***.println(**"finallyブロック処理"**);  **try** {  **if** (bw != **null**) {  bw.close();  }  **if** (fr != **null**) {  fr.close();  }  } **catch** (IOException e) {  e.printStackTrace(System.***out***);  }  }  System.***out***.println(**"プログラム終了"**);  } } |

【FileWriterMain2.java】

|  |
| --- |
| **iimport** java.io.BufferedWriter; **import** java.io.FileNotFoundException; **import** java.io.FileWriter; **import** java.io.IOException; **import** java.util.Scanner;  **public class** FileWriterMain2 {  **public static void** main(String[] args) {   **try** (  *//解放が必要なリソースの宣言* FileWriter fr = **new** FileWriter(**"src\\text.txt"**, **true**); *//true: 追記モード* BufferedWriter bw = **new** BufferedWriter(fr);  ){   *//ファイルに書き込む* System.***out***.println(**"ファイルに書き込みます"**);  System.***out***.println(**"文字列を入力してください(終了する場合は0を入力)"**);  Scanner scanner = **new** Scanner(System.***in***);   *//無限ループ* **for** (**int** i = 1; ; i++) {  System.***out***.print(i + **"行目: "**);  String str = scanner.nextLine();   *//入力された文字列が"0"の場合はループを抜ける* **if** (str.equals(**"0"**)) {  **break**;  }**else** {  bw.write(str); *//１行分をファイルに書き込む* bw.newLine(); *//改行を入れる* }  }  System.***out***.println(**"ファイルに書き込みが完了しました"**);   }**catch** (FileNotFoundException e) {  System.***out***.println(**"ファイルが見つかりませんでした"**);  System.***out***.println(**"例外メッセージ: "** + e.getMessage());   } **catch** (IOException e) {  System.***out***.println(**"ファイルの読み込みに失敗しました"**);  e.printStackTrace(System.***out***);  }  System.***out***.println(**"プログラム終了"**);  } } |

## コレクション

コレクションとはデータを効率よく格納、管理、操作するためのフレームワークでコレクションフレームワークとも呼ばれます。考え方は配列と似ていますが、配列では利用できない便利な機能が豊富に用意されています。

コレクションフレームワークは「リスト」、「セット」、「マップ」の3つの種類に分けられます。

### リスト(List)

リストは配列同様、順序を保持し重複したデータを格納できる仕組みです。要素番号によって操作できます。ArrayListクラスやLinkedListクラスが用意されています。利用できるメソッドなどはそれぞれほぼ同じですが、内部のデータ構造が違います。今回は使用頻度の高いArrayListクラスについて使い方を説明します。

ArrayListは配列と同じように利用できますが、一番の違いは後から要素数を変更できる点です。

|  |
| --- |
| ArrayList |
|  |

|  |
| --- |
| 書式:ArrayList |
| *//ArrayListを生成(Listインターフェース型に保存)* List<保存する型> 変数名 = **new** ArrayList<保存する型>(); *//2つ目の保存する型は省略可能  //要素を追加* 変数名.add(要素); 変数名.add(要素);  *//要素を取得* 変数名.get(要素番号); |

クラスであれば、どんな型でも保存できるこの仕組みをジェネリクスと呼び、型を指定する<>はダイヤモンド演算子と呼びます。

しかし、基本型(int型,double型など)はクラスではないため利用できません。数値等を保存したい場合は、基本型のラッパークラスを利用してください。(Integerクラス,Doubleクラスなど)

|  |  |
| --- | --- |
| メソッド名 | 処理内容 |
| add(値) | リストの最後に値を追加する |
| get(要素番号) | 要素番号の値を取得する |
| set(要素番号, 値) | 要素番号の値を書き換える |
| remove(要素番号) | 要素を削除し、前へ詰める |
| remove(値) | 値を削除(複数ある場合は最初のもの) |
| size() | 要素数を返す |
| contains(値) | 値が含まれるかを判定 |
| indexOf(値) | 値の要素番号を返す |
| clear() | リストの要素をすべて削除 |
| isEmpty() | リストが空かを判定 |

【ArrayListMain.java】

|  |
| --- |
| **import** java.util.ArrayList;  **public class** ArrayListMain {  **public static void** main(String[] args) {   *//ArrayListを生成* List<String> list = **new** ArrayList<String>();   *//要素を追加* list.add(**"北海道"**); *//0番目* list.add(**"青森県"**); *//1番目* list.add(**"秋田県"**); *//2番目   //要素を取得* System.***out***.println(list.get(0));  System.***out***.println(list.get(1));  System.***out***.println(list.get(2));   *//要素数を取得* System.***out***.println(list.size());   *//要素を削除* list.remove(1);  *//要素を表示* System.***out***.println(list);  *//要素を書き換え* list.set(0, **"岩手県"**);   *//要素を拡張for文で取得* **for**(String str : list) {  System.***out***.println(str);  }  *//要素を全てクリア* list.clear();  *//要素が空か判定* System.***out***.println(list.isEmpty());  } } |

### セット(Set)

セットは順序を持たない、重複不可のデータを格納するために利用します。Ｌｉｓｔと比べると使いどころは限定されますが、同じ値が複数格納されてほしくない場合に利用されます。ＨａｓｈＳｅｔクラス、ＬｉｎｋｅｄＳｅｔクラス、ＴｒｅｅＳｅｔクラスなどが用意されています。

利用できるメソッド名等はListとほぼ同じですが、Setには要素番号がないため、get()やｓet（）は利用できません。

HasｈＳｅｔクラスについて説明します。

|  |
| --- |
| HashSet |
|  |

|  |
| --- |
| 書式:HashSet |
| *//HashSetを生成(Setインターフェース型に保存)* Set<保存する型> 変数名 = **new** HashSet<保存する型>(); *//2つ目の保存する型は省略可能  //要素を追加* 変数名.add(要素); 変数名.add(要素);  *//要素を取得(拡張for文でのみ取得ができる)*  **for**(型 変数 : 変数名) {  System.***out***.println(変数); }  *//要素が存在するか判定* **if**(変数名.contains(値)) {  *//存在する場合の処理* } |

値を取得するには拡張for文を使います。出力される順番は保証されません。特定の値のみを取得することもできません。「contains()メソッド」を利用することで、その値がSetに存在するかの確認を行うことができます。

|  |  |
| --- | --- |
| メソッド名 | 処理内容 |
| add(値) | セットに一致する値がない場合は値を追加する |
| remove(値) | 値を削除(複数ある場合は最初のもの) |
| size() | 要素数を返す |
| contains(値) | 値が含まれるかを判定 |
| clear() | セットの要素をすべて削除 |
| isEmpty() | セットが空かを判定 |

【HashSetMain.java】

|  |
| --- |
| **import** java.util.HashSet; **import** java.util.Set;  **public class** HashSetMain {   **public static void** main(String[] args) {  *//HashSetは要素の順番を保証しない*  *//LinkedHashSetは要素の順番を保証する*  *//TreeSetは要素を辞書順にソートして保持する*  *//HashSetを生成* Set<Integer> set = **new** HashSet<Integer>();   *//要素を追加* set.add(1990);  set.add(2000);  set.add(1992);  set.add(1990);   *//要素を表示* System.***out***.println(set);   **for**(Integer i : set) {  System.***out***.println(i);  }  *//要素が含まれるか判定* **if**(set.contains(1990)){  System.***out***.println(**"1990年は含まれています"**);  }**else**{  System.***out***.println(**"1990年は含まれていません"**);  }  } } |

### マップ(Map)

Mapも頻繁に利用するコレクションです。「キー」と「値」のペアで値を管理します。「キー」は重複不可のデータで、「値」は重複可能なデータとなります。Listのように番号によって値を管理するのではなく、「キー」という名前によって「値」を管理したい場合に利用されます。Setと同様に順序は持ちません。

HashMapクラス、LinkedHashMapクラス、TreeMapクラスが用意されています。

|  |
| --- |
| HashMap |
|  |

|  |
| --- |
| 書式:HashMap |
| *//HashMapを生成(Mapインターフェース型に保存)* Map<キーの型, 値の型> 変数名 = **new** HashMap<>();  *//要素を追加* 変数名.put(キー, 値);  *//要素を取得* 変数名.get(キー);  *//全て要素を取得 //MapはkeySet()メソッドでキーのSetを取得 //KeySetから拡張for文でキーを取得* **for**(String key : 変数名.keySet()) {  *//getメソッドでキーに対応する値を取得* System.***out***.println(変数名.get(key)); } |

Mapは「キー」と「値」の二つのデータ型があるため、「<>」内に「,」区切りで二つ指定します。

また、要素の追加は「put()メソッド」で「キー」と「値」をペアで格納します。「add()メソッド」ではない点に注意してください。

値をすべて取得する方法が少し特殊です。「keySet()メソッド」を利用することで「キー」のＳｅｔを取得できます。そのＳｅｔから、各キーを取り出し、取り出したキーを「get()メソッド」の引数として渡すことで値を取得できます。

|  |  |
| --- | --- |
| メソッド名 | 処理内容 |
| put(キー, 値) | セットにキーと値を追加する |
| get(キー) | キーに対応する値を取得する |
| replace(キー, 値) | キーに対応する値を書き換える |
| remove(キー) | キーに対応したペアを削除 |
| size() | 要素数を返す |
| containsKey(キー) | キーが含まれるかを判定 |
| containsValue(値) | 値が含まれるかを判定 |
| clear() | マップの要素をすべて削除 |
| keySet() | キーのセットを取得 |
| entrySet() | Map.Entryのセットを取得 |
| isEmpty() | マップが空かを判定 |

【HashMapMain.java】

|  |
| --- |
| **import** java.util.HashMap; **import** java.util.Map;  **public class** HashMapMain {  **public static void** main(String[] args) {   *//HashMapを生成(Mapインターフェース型に保存)  //< キーの型, 値の型 >* Map<String, Double> map = **new** HashMap<>();   *//要素を追加  //都道府県ごとの平均年齢* map.put(**"東京都"**, 45.25);  map.put(**"大阪府"**, 47.10);  map.put(**"京都府"**, 47.74);  map.put(**"高知県"**, 51.25);  map.put(**"沖縄県"**, 43.46);   *//要素を取得* System.***out***.println(**"東京都の平均年齢は"** + map.get(**"東京都"**));   System.***out***.println();   *//全ての要素を取得①  //キーの一覧を取得* **for**(String key : map.keySet()) {  System.***out***.println(key + **"の平均年齢は"** + map.get(key));  }   System.***out***.println();   *//全ての要素を取得②  //Map.EntryとはMapの中の1つの要素を表すインターフェース* **for**(Map.Entry<String, Double> entry : map.entrySet()) {  System.***out***.println(entry.getKey() + **"の平均年齢は"** + entry.getValue());  }  } } |

## GUIアプリケーション

### GUIフレームワーク

これまで、学習した内容を利用することで簡単なCUIアプリケーションは作成できるようになりました。ただし、実際に普段利用するアプリケーションはGUIアプリケーションになります。したがって、この章では、簡単なGUIアプリケーションの作成方法について学習します。

実際の開発ではJava言語でGUIアプリケーションを作成する機会はあまり多くありません。Windows アプリケーションを開発する場合は主にC#やVisual Basicなどが利用されることが多いです。Androidアプリケーションを開発する場合は、Java言語かKotlinを利用します。

GUI開発を行う上で重要な用語としてフレームワークという言葉があります。フレームワークとはシステム開発を行う上で便利な機能をまとめた「部品集」のようなものです。GUI開発に必要な基本要素(ボタン、テキストボックス、ウィンドウなど）は、どのGUIでも共通して使われます。そこで、これらの共通部分を開発のたびに一から作成する手間を省くために用意されたものが、GUIフレームワーク です。

さまざまな開発者がさまざまなフレームワークを用意していますが、Java言語における代表的なGUIフレームワークは以下の二つです。

1. Swing(スイング)

Javaの標準から利用できるフレームワークで追加インストールが不要で利用できます。古くから使用されてきたフレームワークのため、インターネット上に多くの情報やサンプルコードが用意されているます。構造も比較的シンプルで初学者が理解しやすくなっています。ただし、UIが少し古く、アニメーションやリッチな表現は得意ではありません。

1. JavaFX

より新しいフレームワークでデザインの自由度も高く、アニメーション等も得意です。FXMLというxmlファイルにレイアウトを記述し、CSSでデザインを調整できます。高機能で便利ですが、Swingと比較すると学習コストが高いです。

この章では比較的理解がしやすいSwingの使い方を解説します。

SwingはJavaの標準で利用できるため、準備することはそれほど多くありません。

Ｉｎｔｅｌｌｉｊ　ＩDEA上でSwingの開発を行いやすい「UI Designer」というプラグインを追加します。

「右上の歯車マーク」-「プラグイン」-「Marketplace」から、「UI Designer」を選択して有効化します。Ｉｎｔｅｌｌｉｊ　ＩDEAを再起動する必要がありますので、再起動してください。

### Swingの利用方法

プロジェクトを作成し、「src」を右クリックし、「新規」から「Swing UI デザイナー」を選択し、「ダイアログクラスの作成」をクリックしてください。

クラス名は「UiTest」とし、全てチェックをつけたまま、「ＯＫ」をクリックします。そうすると自動的にＵｉＴｅｓｔクラスが作成され、実行することができます。

【UiTest.java】

|  |
| --- |
| **import** javax.swing.\*; **import** java.awt.event.\*;  **public class** UiTest **extends** JDialog {  **private** JPanel **contentPane**;  **private** JButton **buttonOK**;  **private** JButton **buttonCancel**;   **public** UiTest() {  setContentPane(**contentPane**);  setModal(**true**);  getRootPane().setDefaultButton(**buttonOK**);   **buttonOK**.addActionListener(**new** ActionListener() {  **public void** actionPerformed(ActionEvent e) {  onOK();  }  });  **buttonCancel**.addActionListener(**new** ActionListener() {  **public void** actionPerformed(ActionEvent e) {  onCancel();  }  });  *// X をクリックしたとき、 delete() を呼ぶ* setDefaultCloseOperation(***DO\_NOTHING\_ON\_CLOSE***);  addWindowListener(**new** WindowAdapter() {  **public void** windowClosing(WindowEvent e) {  onDelete(); *//onDelete()を呼ぶ* }  });  *// ESCAPE で onCancel() を呼ぶ* **contentPane**.registerKeyboardAction(**new** ActionListener() {  **public void** actionPerformed(ActionEvent e) {  onDelete(); *//onDelete()を呼ぶ* }  }, KeyStroke.*getKeyStroke*(KeyEvent.***VK\_ESCAPE***, 0), JComponent.***WHEN\_ANCESTOR\_OF\_FOCUSED\_COMPONENT***);  }  **private void** onOK() {  System.***out***.println(**"OKボタンが押されました"**); *// dispose(); //ダイアログの破棄 コメントアウト* }   **private void** onCancel() {  System.***out***.println(**"キャンセルボタンが押されました"**); *// dispose(); //ダイアログの破棄 コメントアウト* }   **private void** onDelete() { *//onDelete()の実装* System.***out***.println(**"ばつボタンが押されました"**);  dispose();  }   **public static void** main(String[] args) {  UiTest dialog = **new** UiTest();   dialog.setTitle(**"UIテストダイアログ"**); *//タイトルの指定* dialog.setSize(400, 300); 　　　　　　 　*//サイズの指定 // dialog.pack(); 　　　//Windowsサイズの自動調整 コメントアウト* dialog.setVisible(**true**); *//ダイアログの表示* System.*exit*(0); *//プログラムの終了* } } |

「UiTest.form」を開き、「JLabel」と「JTextField」を2つずつ作成します。

グラフィカル ユーザー インターフェイス, テキスト, アプリケーション

自動的に生成された説明「JTextField」の「field name」をそれぞれ、「textName」と「textMail」にしてください。

UiTestクラスの「onOK()メソッド」を次のように書き換えてください。

【UiTest.java】

|  |
| --- |
| **private void** onOK() {  System.***out***.println(**"OKボタンが押されました"**);   **if**(**textName**.getText().equals(**""**) || **textMail**.getText().equals(**""**)) {  JOptionPane.*showMessageDialog*(**contentPane**,  **"名前とメールアドレスを入力してください。"**, **"エラー"**, JOptionPane.***ERROR\_MESSAGE***);    }**else**{  String message = **"登録しました。\n"** +  **"名前："** + **textName**.getText() + **"\n"** +  **"メールアドレス："** + **textMail**.getText();   JOptionPane.*showMessageDialog*(**contentPane**, message, **"確認ダイアログ"**,JOptionPane.***INFORMATION\_MESSAGE***);   }} |

「JOptionPane(オプションペイン)クラス」は簡単なメッセージを表示するためのクラスです。

「showMessageDialog()メソッド」を利用することで、簡単にメッセージダイアログを作成することができます。

このほかにもさまざまな部品が用意されています。興味がある人は試してみてください。

Timerクラスを利用することで、アニメーション等も作成できます。