オブジェクト指向

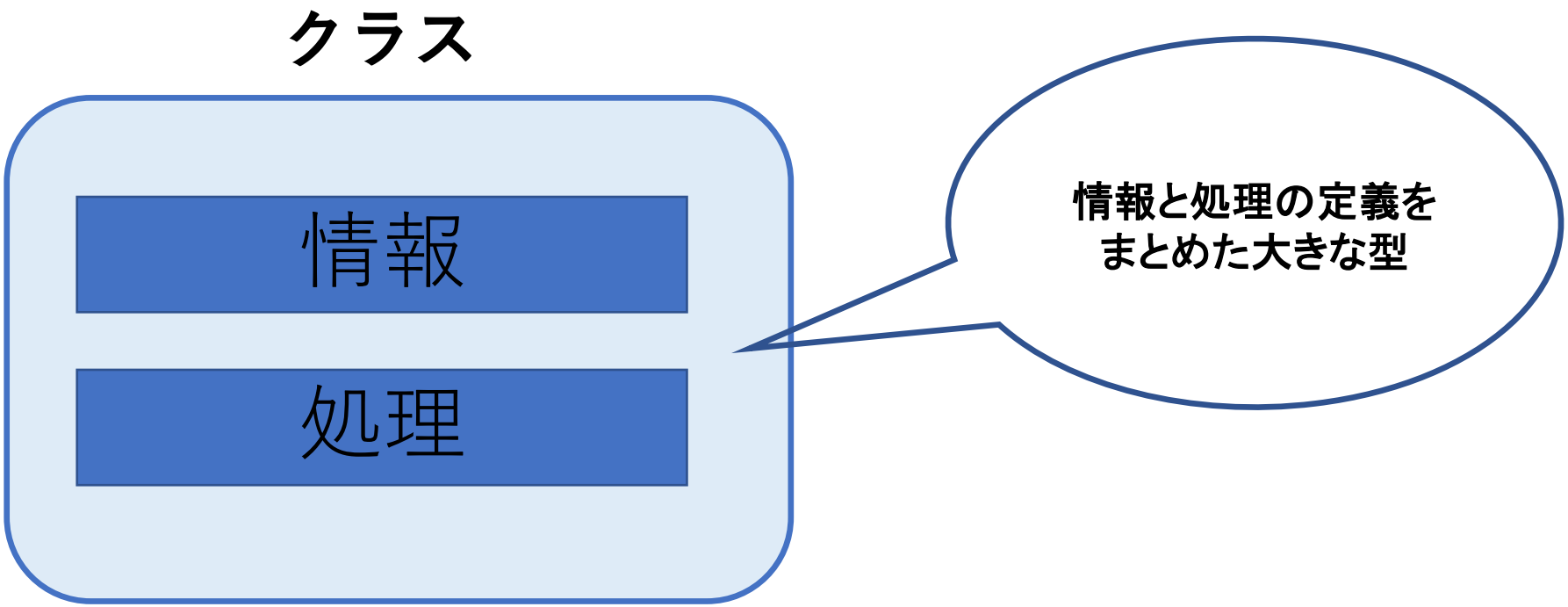
Ver.1.2

クラスとインスタンス

クラスとは

機能やデータの集合の定義のこと

- 1つのデータ型として定義する
- 定義したクラスは他のクラスから利用することができる



インスタンスとは

Goodworks
technology for the enterprise

インスタンスとは、クラスという概念から生成される実体のこと

・Javaではクラスに対して、独立した固有の状態をもつ実体(インスタンス)を作ることができる
・クラスからインスタンスを生成することをインスタンス化という
・インスタンスのことをオブジェクトという

あらかじめクラスで定義された処理やデータ通りにメモリ上に展開して実行できる状態にしたもの

インスタンス化

Goodworks
technology for the enterprise

■ 構文

```
【クラス名】　【変数名】＝　new　【クラス名】　（）；
```

- newキーワードの後ろにクラス名()と記述することで
  インスタンスを生成する

インスタンス化

■ 例

```
public class SampleInstance {
        public int age;
}
```

```
public class MainInstance {
        public static void main(String[] args) {
                SampleInstance si = new SampleInstance();
                System.out.println(si);
        }
}
```

参照型とは

Goodworks
technology for the enterprise

参照型とはインスタンスIDを管理する型

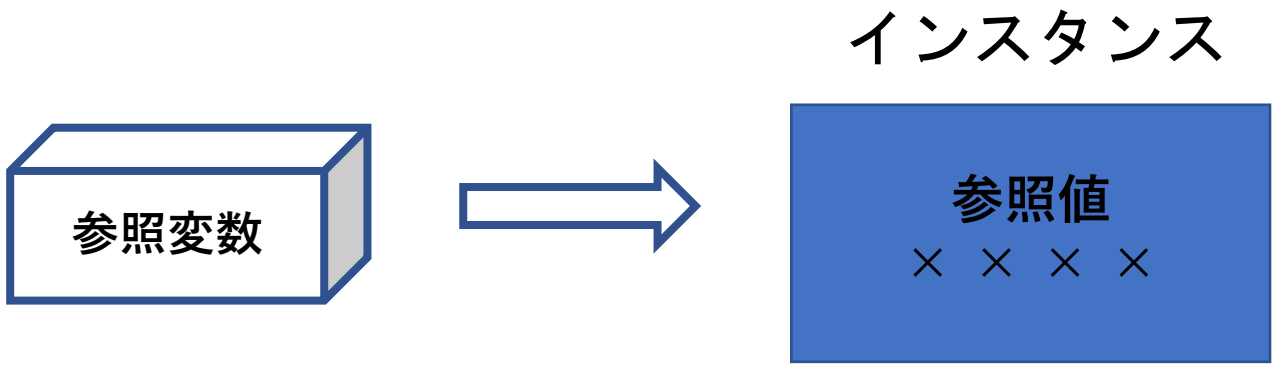
・インスタンスIDとはメモリ上に割り当てられたインスタンスの所在地
・参照型はインスタンスの所在地のみを扱う
・基本データ型以外の方は全て参照型

基本データ型と参照型の違い

Goodworks
technology for the enterprise

基本データ型の変数は独立した格納場所を持っている

参照型の変数はインスタンスの所在地をしめしてるだけ

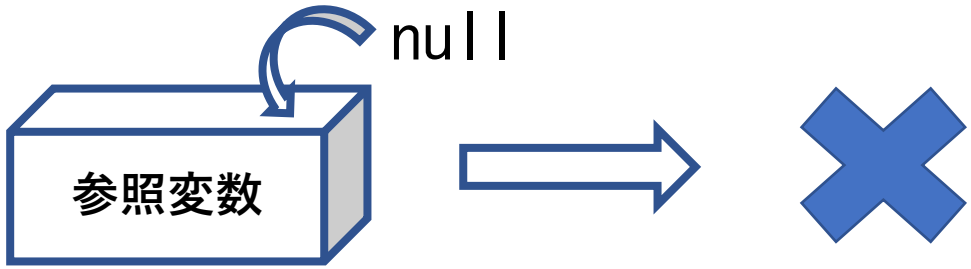


基本データ型と参照型の違い

Goodworks
technology for the enterprise

「null」とは

・参照型の特別な値で、インスタンスがないことを示す
・どのインスタンスも参照していないという意味



nullが代入されている変数に対してメソッドやフィールドにアクセスすることはできない
・メソッド、フィールドは後述

クラスのメンバ

Goodworks
technology for the enterprise

クラス内で定義するクラスの構成要素

メンバの種類

-フィールド（データ）
- インスタンスフィールド
- クラスフィールド

-メソッド（機能）
- インスタンスメソッド
- クラスメソッド

インスタンスフィールドとは

Goodworks
technology for the enterprise

インスタンス毎に生成される変数
インスタンス化しないと利用不可
生成したインスタンスの数分生成される

インスタンスフィールドの定義

Goodworks
technology for the enterprise

■ 構文

```
public class クラス名 {
        public 【データ型】【変数名】;
}
```

・クラスブロックの直下に記述

■ 例

```
public class SampleInsField {
        public int age;
}
```

インスタンスフィールドの代入

Goodworks
technology for the enterprise

■ 構文

```
【参照変数】.【インスタンスフィールド名】＝【値】
```

■ 例

```
public class MainInsField {
	public static void main(String[] args) {
		SampleInsField sif = new SampleInsField();
		sif.age = 22;
	}
}
```

インスタンスフィールドの参照

Goodworks
technology for the enterprise

■ 構文

```
【参照変数】.【インスタンスフィールド名】
```

■ 例

```
public class MainInsField {
	public static void main(String[] args) {
		SampleInsField sif = new SampleInsField();
		sif.age = 22;
		System.out.println(sif.age);
	}
}
```

クラスフィールドとは

Goodworks
technology for the enterprise

クラス毎に生成される変数
生成されたインスタンスに共通した変数になる
インスタンス化せずに利用可能

クラスフィールドの定義

Goodworks
technology for the enterprise

構文

```
public class クラス名 {
        public static【データ型】【変数名】;
}
```

・クラスブロックの直下に記述

・データ型の前にstaticを記述する

例

```
public class SampleClaField {
        public static int age;
}
```

クラスフィールドの代入

構文

```
【クラス名】.【クラスフィールド名】　＝　【値】
```

例

```
public class MainClaField {
	public static void main(String[] args) {
		SampleClaField.age = 22;
	}
}
```

クラスフィールドの参照

Goodworks
technology for the enterprise

構文

```
【クラス名】.【クラスフィールド名】
```

例

```
public class MainClaField {
	public static void main(String[] args) {
		SampleClaField.age = 22;
		System.out.println(SampleClaField.age);
	}
}
```

インスタンスメソッドとは

Goodworks
technology for the enterprise

インスタンス毎に生成されるメソッド
インスタンス化しないと利用不可
生成したインスタンスの数分生成される
インスタンスメソッドから自インスタンスのインスタンスフィールドや
インスタンスメソッドを利用可能

インスタンスメソッドの定義

Goodworks
technology for the enterprise

構文

```
public class クラス名 {
        public 【戻り値の型】【メソッド名】（【引数】）{
                処理;

        }
}
```

例

```
public class SampleInsMethod {
        public int getSum (int a, int b) {
                return a + b;
        }
}
```

インスタンスメソッドの参照

Goodworks
technology for the enterprise

■ 構文

```
【参照変数】.【インスタンスメソッド名】（【引数】）；
```

■ 例

```
public class MainInsMethod {
        public static void main(String[] args) {
                SampleInsMethod sim  = new SampleInsMethod ();
                int sum = sim.getSum(10, 20);
                System.out.println(sum);
        }
}
```

クラスメソッドとは

Goodworks
technology for the enterprise

クラス毎に生成されるメソッド
インスタンス化せずに利用可能
クラスメソッドから自クラスのインスタンスフィールドや
インスタンスメソッドは利用不可

クラスメソッドの定義

Goodworks
technology for the enterprise

構文

```
public class クラス名 {
        public static【戻り値の型】【メソッド名】（【引数】）{
                処理;

        }
}
```

例

```
public class SampleClaMethod {

        public static int getSum (int a, int b) {
                return a + b;
        }
}
```

クラスメソッドの参照

Goodworks
technology for the enterprise

■ 構文

```
【クラス名】.【クラスメソッド名】（【引数】）；
```

■ 例

```
public class MainClaMethod {
        public static void main(String[] args) {
                int sum = SampleClaMethod.getSum(10, 20);
                System.out.println(sum);
        }
}
```

ローカル変数とは

Goodworks
technology for the enterprise

メソッドや制御文のブロック中で宣言する変数のこと
そのメソッド、制御文のブロック内だけで使用できる

・Javaコードにたびたび登場する{・・・}をブロックと呼ぶ
・変数の有効範囲(スコープ)はブロックによって決まる

■ 例

```
public class SampleLocal {
        public int age;    //クラス宣言の直下に宣言した変数はフィールド
        public int getSum (int a, int b) {
                int sum = a + b; //メソッド内で宣言した変数はローカル変数
                return sum;
        }
}
```

オーバーロードとは

Goodworks
technology for the enterprise

引数の定義が異なるならば同じメソッド名を定義できる。

- 引数のデータ型が異なる
- 引数の数が異なる
- 複数の異なるデータ型の定義順が異なる

```
public class SampleOverRoad1 {
        public void msg(){
                System.out.println(“こんにちは”);
        }
        public void msg(String name){
                System.out.println( name +“さん、こんにちは”);
        }
}
```

オーバーロードとは

Goodworks
technology for the enterprise

```
public class SampleOverRoad2 {

        public int sum(int a,int b){
                return a + b;
        }

        public double sum(double a,int b){
                 return a + b;
        }

        public double sum(int a,double b){
                 return a + b;
        }

        public double sum(double a,double b){
                 return a + b;
        }

}
```

コンストラクタとは

インスタンスを生成する際、一度だけ実行される特殊なメソッドのこと

コンストラクタの特徴

- メソッド名がクラス名と同じである
- 戻り値は指定できない（voidの記述は省略）
- 引数の異なるコンストラクタを複数定義可能
- コンストラクタ自体を省略することも可能

コンストラクタの定義

Goodworks
technology for the enterprise

■ 構文

```
public class クラス名 {
        public 【コンストラクタ名】（【引数】）{
                処理;


        }
}
```

・【コンストラクタ名】はクラス名と同じにする

・戻り値は記述不要

コンストラクタの定義

■ 例

```
public class SampleConstructor {
        private int age;

        public SampleConstructor  (int initAge) {
                age = initAge;
        }
}
```

コンストラクタの参照

Goodworks
technology for the enterprise

■ 構文

```
【データ型】【参照変数】＝　new　【コンストラクタ名】（【引数】）；
```

■ 例

```
SampleConstructor sc = new SampleConstructor(10);
```

・コンストラクタ名はクラス名（データ型）と同じ

・呼び出したいコンストラクタの引数の型と数を一致させる

パッケージとは

Goodworks
technology for the enterprise

定義したクラスを分類毎に階層的に整理する仕組み

- パッケージの構造はフォルダと同じ
- クラスは必ずどこかのパッケージに属する必要がある
- パッケージが異なればクラス名の重複が可能

パッケージ

package宣言

Goodworks
technology for the enterprise

構文

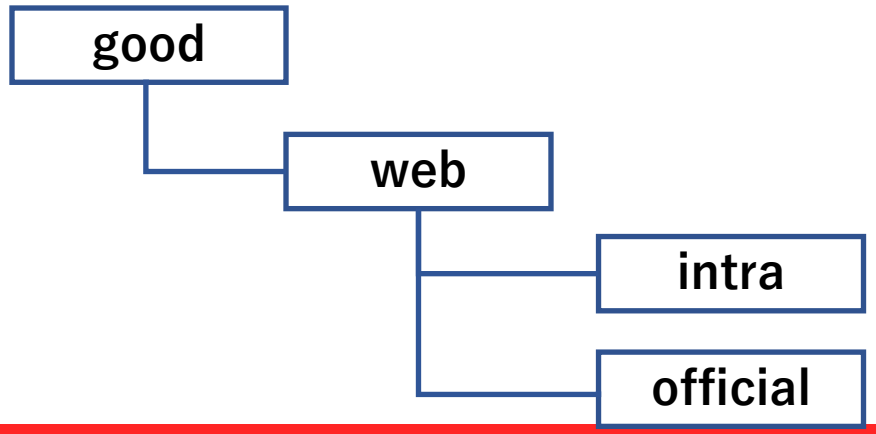
```
package 【パッケージ名】;
```

■ 例

```
package good.web.official;

public  class SamplePackage {

}
```



・「.」は階層構造を表す

import文

別パッケージに格納されているクラスを利用する場合はimport宣言でクラスを指定する

import宣言

■ 構文

```
import　【パッケージ】.【クラス名】;
```

・宣言する場所はパッケージ宣言とクラス定義の間

・【クラス名】に「＊」を記述するとパッケージに属する全てのクラスを使用できる

アクセス制限の修飾子

Goodworks
technology for the enterprise

クラス・メソッド・フィールドの先頭に記述し、
どこからアクセス可能かを指定するキーワード

アクセス修飾子	アクセスできる範囲
public	制限なし（どのクラスからでもアクセス可能）
protected	同一パッケージとサブクラスからアクセス可能
指定なし	同一パッケージのみアクセス可能
private	自クラス内のみアクセス可能（外部からアクセス不可能）

・サブクラスについては後述

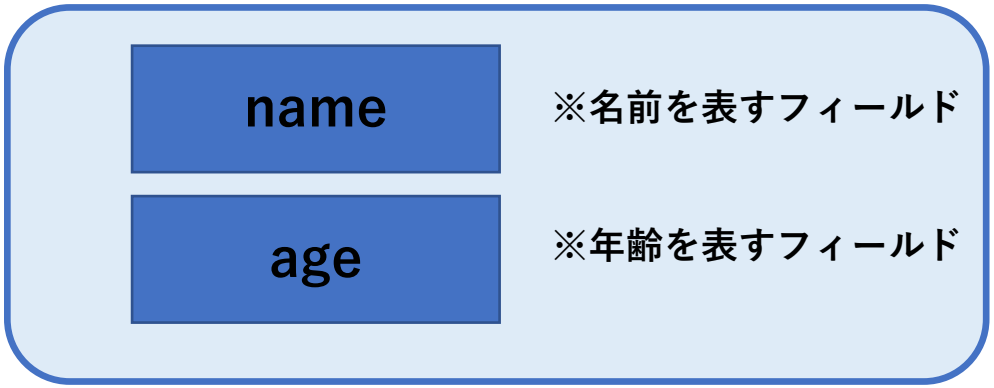
カプセル化とは

Goodworks
technology for the enterprise

クラスが持つフィールドの内容を外部から保護する
ための仕組み

フィールドの内容を意図しない内容に変更されてしまったり、
勝手に内容を参照されることを防ぐ

Personクラス




```
Person person = new Person();
person.age = -1;
```

意図しない-1歳を代入されてしまう！

カプセル化されていないと・・・

カプセル化の例

Goodworks
technology for the enterprise

カプセル化されたクラスの例

```
public class SampleCapsule {
      private int age; // 修飾子をprivateにする

      public int getAge() { // ゲッターメソッド
            return age;
      }

      public void setAge(int newAge) { // セッターメソッド
            age = newAge;
      }
}
```

・フィールドをアクセス修飾子をprivate にする

・フィールドアクセス用のアクセサメソッドを定義する

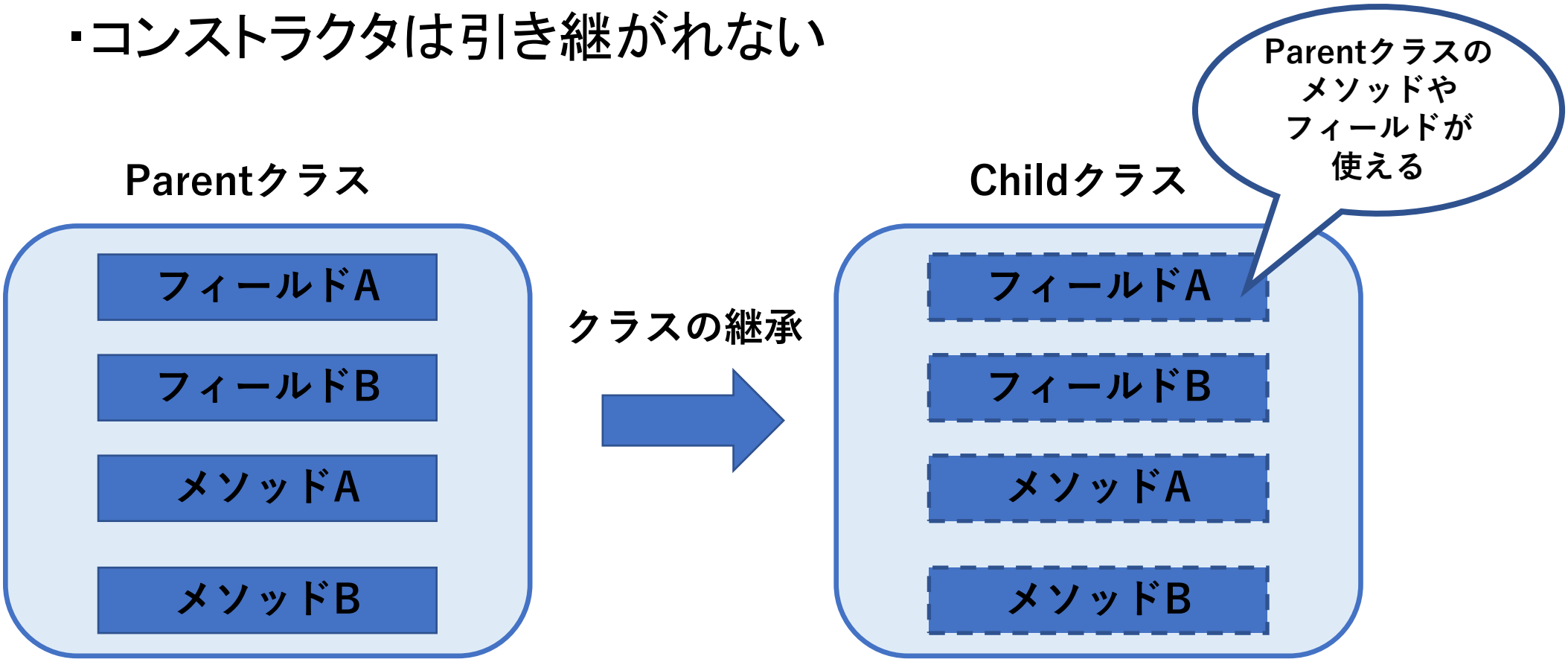
継承

継承とは

Goodworks
technology for the enterprise

既に定義済みのクラスの定義内容を引き継ぐこと

・メソッド・フィールドを定義し直さなくてもそのまま使える
・コンストラクタは引き継がれない



継承の用語

Goodworks
technology for the enterprise

-元のクラス（継承されるクラス）
- スーパークラス
- 親クラス
- 継承元

-元のクラスを継承したクラス
- サブクラス
- 子クラス
- 派生クラス
- 継承先

継承の定義

Goodworks
technology for the enterprise

■ 構文

```
public class 【サブクラス名】 extends 【スーパークラス名】 {

}
```

・extendsキーワードによって継承を表現
・クラス定義にはスーパークラスは一つしか指定できない
（単一継承）

継承の定義

Goodworks
technology for the enterprise

■ 例(スーパークラス)

```
public class  SuperSample1  {
      public String superStr;

      public void superMethod() {
            System.out.println(“スーパークラスのメソッドです。”);
      }
}
```

継承の定義

■ 例(サブクラス)

```
public class  SubSample1 extends SuperSample1 {
      public String subStr;

      public void subMethod() {
            System.out.println(“サブクラスのメソッドです。”);
      }
}
```

継承したクラスの利用

Goodworks
technology for the enterprise

■ 構文

```
【サブクラス名】【参照変数名】 = new【サブクラスのコンストラクタ】();
```

・これまでのインスタンス化と同じ
・サブクラスをインスタンス化するとスーパークラスの
フィールドやメソッドも利用することができる

継承したクラスの利用

Goodworks
technology for the enterprise

■ 例

```
public class  MainPg1  {
      public static void main(String[] args) {
            SubSample1 ss1 = new SubSample1();
            // スーパークラスのフィールド
             ss1.superStr = “スーパークラス”;
            // サブクラスのフィールド
             ss1.subStr = “サブクラス”;

            // スーパークラスのメソッド
             ss1.superMethod();
            // サブクラスのメソッド
             ss1.subMethod();
      }
}
```

暗黙のスーパークラス

Goodworks
technology for the enterprise

Javaプログラミングの継承は1クラスしか継承できない
（単一継承）が段階的に継承することは可能

「java.lang.Object」クラスは全てのクラスのスーパークラス
（暗黙のスーパークラス）
・クラス定義に継承の定義がない場合、自動的に
Objectクラスが継承される
・Objectクラスにもメソッドやフィールドが定義してあり、
すべてのクラスから利用が可能

Objectクラスのメソッド

Goodworks
technology for the enterprise

equalsメソッド

■ 例

```
SamplePg sp1 = new SamplePg();
SamplePg sp2 = new SamplePg();

// 別のインスタンスなので実行結果はfalse
System.out.println(sp1.equals(sp2));
```

・同じインスタンスであればtrue、
別のインスタンスであればfalseを返す

superとthis

superはサブクラスからスーパークラスのフィールドやメソッドを参照する

- superは省略可能
- サブクラスとスーパークラスで名前が重複した際等に使用する

サブクラスのコンストラクタ内でスーパークラスのコンストラクタを呼び出す場合

- super(引数) を使用する

superの使い方

■ 構文

```
super.【スーパークラスのフィールド名またはメソッド名（）】
```

■ 例(スーパークラス)

```
public class SuperSample2 {
        public String superStr;

        public void superMethod() {
                System.out.println(“スーパークラスのメソッド”);
        }
}
```

superの使い方

■ 例(サブクラス)

```
 public class SubSample2 extends SuperSample2 {
        public void subMethod() {
                // スーパークラスのフィールド
                System.out.println(super.superStr);

                // スーパークラスのメソッド
                super.superMethod();
        }
}
```

superとthis

thisは自分自身のクラスを示す

- 自分自身のメソッドやフィールドを指定する場合に使用する
- 簡便のためthisは省略して良い
  -ローカル変数とフィールドの区別するためによく用いられる

コンストラクタ内で自クラスのオーバーロードされた
コンストラクタを呼び出す場合

- this(引数) を使用する

superとthis

Goodworks
technology for the enterprise

■ 構文

```
this.【自クラスのフィールド名またはメソッド名（）】
```

■ 例

```
public class SampleThis {
	private int age;

	public void setAge(int age) {
		// フィールドに引数を代入する
		this.age = age;
	}
}
```

オーバーライドとは

Goodworks
technology for the enterprise

スーパークラスのメソッドをサブクラスで再定義すること

・スーパークラスで定義されているメソッドをサブクラスにて同じメソッドを定義する
・privateメソッドはオーバーライドできない

オーバーライドメソッドの定義

■ 例(スーパークラス)

```
public class SuperSample3 {

        public void method() {
                System.out.println(“スーパークラスのメソッド”);
        }
}
```

オーバーライドメソッドの定義

Goodworks
technology for the enterprise

■ 例(サブクラス)

```
public class SubSample3 extends SuperSample3 {
        @Override
        public void method() {
                // スーパークラスのメソッドを呼び出す
                super.method();
                System.out.println(“オーバーライドしました”);
        }
}
```

・super.メソッド名でスーパークラスのメソッドを呼び出すこともできる。

（サブクラスで再定義しているだけで上書きではない）

オーバーライドメソッドの参照

Goodworks
technology for the enterprise

■ 例

```
public class MainPg3{
	public static void main(String[] args) {
		SubSample3 ss3 = new SubSample3();
		ss3.method();
	}
}
```

・サブクラスのメソッドが実行される
-外部クラスからはオーバーライドされたスーパークラスのメソッドは呼び出せない
-サブクラスからはスーパークラスのメソッドを呼び出せる

インタフェース

インタフェース

Goodworks
technology for the enterprise

ある特定の機能の概要を記述したもの

■ インタフェースのルール

・インタフェースのオブジェクトを生成することは出来ない
・インタフェースを実装したクラスはインタフェース側で
　宣言されている抽象メソッドを実装しなければならない
・インタフェース内で定義した変数は、定数になり、
　変更することができない
・インタフェースは多重継承可能

インタフェースの定義

■ 構文

```
public interface 【インタフェース名】 {

}
```

・インタフェース名はクラス名と同じ命名ルール

■ 例

```
public interface SampleInterface {

}
```

インタフェースの構成

Goodworks
technology for the enterprise

インタフェースは以下の要素のみ定義可能

- 定数
- 抽象メソッド

定数とは

Goodworks
technology for the enterprise

定数とは、値が変らないことが保証された変数
一度代入したら再代入できない変数

■ 構文

```
final 【データ型】【定数名】=【値(固定値)】;
```

・変数の前にfinal修飾子をつける

■ 例

```
final double TAX_RATE = 0.08;
```

・インタフェースで定義する場合はfinalは省略可

抽象メソッドとは

Goodworks
technology for the enterprise

どんなメソッドなのかの記述のみで、処理のないメソッド

- メソッド名の後に直ぐにセミコロン(;)を記述
- 強制的にオーバーライドさせられる
- 抽象メソッドは呼び出し不可能

抽象メソッドの定義

Goodworks
technology for the enterprise

■ 構文

```
public  abstract 【戻り値の型】【メソッド名】（【引数】）;
```

・メソッドの定義の前にabstract修飾子を記述
・引数の後ろにセミコロン(;)

■ 例

```
public abstract void method () ;
```

・インタフェースに定義する場合はabstractは省略可

インタフェース実装クラスの定義

Goodworks
technology for the enterprise

■ 構文

```
public class 【クラス名】 implements 【インタフェース名】 {


}
```

・implementsの後ろに実装したいインタフェース名を記述
・インタフェースに定義されている抽象メソッドを必ずオーバーライドしなければならない

インタフェース実装クラスの定義

Goodworks
technology for the enterprise

■ 例(インタフェース)

```
public interface SampleInterface {
        public double TAX_RATE = 0.08;

        public void method();
}
```

■ 例(実装クラス)

```
public SamplePg4 implements SampleInterface {
        @Override
        public void method() {
                System.out.println(“抽象メソッドを実装”);
                System.out.println(TAX_RATE);
        }
}
```

抽象クラス

抽象クラス

Goodworks
technology for the enterprise

■ 抽象クラスのルール
・抽象クラスのオブジェクトを生成することはできない
・抽象メソッドが存在するクラスは必ず抽象クラスとして宣言しなければならない
・抽象クラスには通常のメソッドを記述することもできる
・抽象メソッドがない抽象クラスを作ることもできる
・サブクラスでは抽象メソッドをオーバーライドしなければならない
・抽象クラスやクラスを多重継承することは出来ない

抽象クラスの定義

Goodworks
technology for the enterprise

■ 例(抽象クラス)

```
public abstract class SampleAbstract {
        private int age;

        // 通常のメソッド
        public void setAge(int age) {
                this.age = age;
        }

        // 抽象メソッド
        public abstract void abstMethod();
}
```

抽象クラスの定義

Goodworks
technology for the enterprise

■ 例(サブクラス)

```
 public class SamplePg5 extends SampleAbstract {
        @Overrride
        public void abstMethod() {
                System.out.println(“抽象メソッドを実装”);
        }
}
```

API

API

Goodworks
technology for the enterprise

APIとは？

・Application Programming Interfaceの略
・プログラミングするうえで頻繁に使われる
機能がクラスファイルとして提供されている

JavaDocのURL
https://docs.oracle.com/javase/jp/8/docs/api/

APIの活用

Goodworks
technology for the enterprise

APIは多くのパッケージ構成で配置されている
- -java.langパッケージ
- -java.utilパッケージ
- -java.ioパッケージ

基本データ型のラッパークラス

Goodworks
technology for the enterprise

int、doubleなどの基本データ型の値をインスタンスとして扱うためのクラス

文字列を基本型に変換する等に使用

基本データ型	対応するラッパークラス
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
boolean	java.lang.Boolean
char	java.lang.Character

ラッパークラスの使用例

Goodworks
technology for the enterprise

```
// 文字列をboolean型に変換
boolean flag = Boolean.parseBoolean( “true” );

// 文字列をint型に変換
int num = Integer.parseInt( “10” );

// 文字列をlong型に変換
long lNum = Long.parseLong( “10” );

// 文字列をdouble型に変換
double dNum = Double.parseDouble( “10.5” );

// 文字列をfloat型に変換
float fNum = Float.parseFloat( “10.5” );
```

※その他のメソッドを調べて実装してみましょう！

文字入力を受け付けるAPI

java.util.Scanner

```
Scanner sc = new Scanner(System.in);

// コンソールに入力した文字列を取得する
String inStr = sc.nextLine();
```

日付を扱うAPI

Goodworks
technology for the enterprise

java.util.Calender

```
// 現在日時を取得
Calendar cal = Calendar.getInstance();
// 西暦を取得
int year = cal.get(Calendar.YEAR);
// 月を取得
int month = cal.get(Calendar.MONTH)+1;
// 日を取得
int day = cal.get(Calendar.DATE);
// 曜日を取得
int week = cal.get(Calendar.DAY_OF_WEEK);

// 10日後を指定
cal.add(Calendar.DATE,10);
```

数学的な演算を行うAPI

Goodworks
technology for the enterprise

java.lang.Math

```
int a = 10;
int b = 20;

// 大きい数を取得
int max = Math.max(a,b);
// 小さい数を取得
int min = Math.min(a,b);

double c = 4d;
double d = 5d;

// 累乗した値を取得
doublc num = Math.pow(c,d);

// 10までのランダムな値を取得
int run = (int)(Math.randam()*10+1);
```

※Math.randam()は「0以上1未満」のランダムな数値をdouble型で取得する

コレクションとは

複数の値をまとめて扱うあらかじめ用意されたAPI
クラスとインタフェースから構成されている

- List
- Map
- Set

リストとは

Goodworks
technology for the enterprise

複数の要素の順番を保持するコレクション
インデックスを使用して要素にアクセスする
配列との違いは要素数が可変であること

- Listインタフェース
- ArrayListクラス
- LinkedListクラス

リストの定義

■ 構文

```
List＜【型】＞【参照変数】＝　new ArrayList＜【型】＞（）；
```

・要素数は配列と同様で0から始まる

ジェネリクス（総称型）とは

・コレクションに格納する要素のデータ型を指定する
・ジェネリクスは参照型のみ指定できる
・基本データ型を指定したい場合はラッパークラスを指定する

リストの定義

■ 例

— 要素がString型のリスト

```
List<Sting> slist = new ArrayList<String> ();
```

— 要素がDouble型のリスト

```
List<Double> dlist = new ArrayList<Double> ();
```

リストのメソッド

Goodworks
technology for the enterprise

主なリストのメソッド

戻り値	メソッドと説明
boolean	add(E e) リストの最後に、指定された要素を追加する
void	add(int index, E element) リスト内の指定された位置に指定された要素を挿入する
E	get(int index) リスト内の指定された位置にある要素を返却する
int	size() リスト内にある要素の数を返却する
E	remove(int index) このリストの指定された位置にある要素を削除する
boolean	isEmpty() リストに要素がない場合に true を返却する

リストのメソッド

Goodworks
technology for the enterprise

■ 例

```
// リストの定義
List<String> slist = new ArrayList<String> ();

// 要素の追加
slist.add(“東京都”);
slist.add(“神奈川県”);
slist.add(“千葉県”);

// 要素の取り出し
String prefecture1 = slist.get(0);
String prefecture2 = slist.get(1);
String prefecture3 = slist.get(2);

// 要素数の取得
int size = slist.size();
```

リストのメソッド

■ for文を使用したリストの要素の取得

```
// インデックスを使用
for (int i = 0; i < slist.size(); i++) {
        System.out.println(slist.get(i));
}

// 拡張for文を使用
for (String prefecture : slist) {
        System.out.println(prefecture);
}
```

■ 実行結果はいずれも同じ

```
東京都
神奈川県
千葉県
```

マップとは

Goodworks
technology for the enterprise

キーと値の組み合わせでを保持するコレクション
要素の順番は保持されない
キーが重複すると上書きされる

- Mapインタフェース
- HashMapクラス
- LinkedHashMapクラス

マップの定義

Goodworks
technology for the enterprise

■ 構文

```
Map＜【キーの型】, 【値の型】＞【参照変数】
                ＝  new HashMap＜ 【キーの型】, 【値の型】 ＞（） ;
```

・キーと値の型を「,」区切りで記述する

マップのメソッド

主なマップのメソッド

戻り値	メソッドと説明
V	put(K key, V value) 指定された値と指定されたキーをマップに追加する
V	get(Object key) 指定されたキーに対応する値を返却する そのキーがマップに含まれていない場合は null を返却する

マップのメソッド

Goodworks
technology for the enterprise

■ 例

```
// マップの定義
Map<String, String> map = new HashMap<String, String> ();

// 要素の追加
map.put(“東京都”, “23区”);
map.put(“神奈川県”, “横浜市”);
map.put(“千葉県”, “千葉市”);

// キーに対応する値の取得（横浜市を取得）
String city = map.get(“神奈川県”);

// キーが存在しない場合はnull
String city2 = map.get(“埼玉県”);
```

例外処理

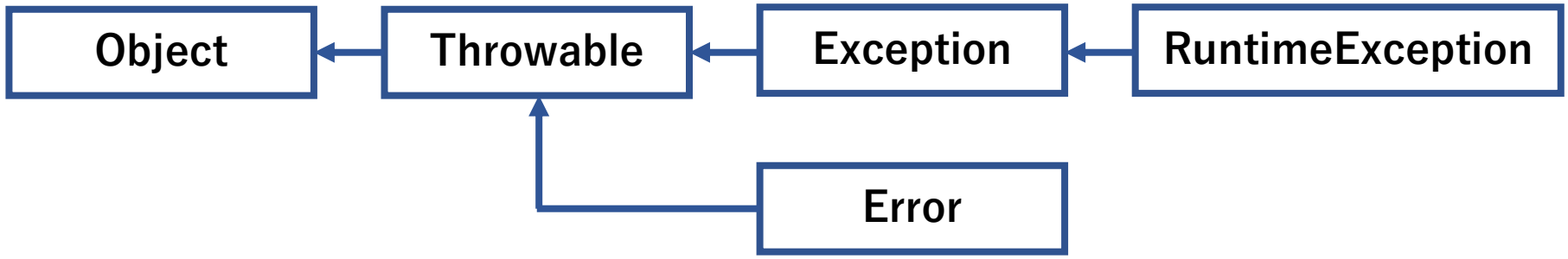
例外とは

プログラム実行時に発生するエラー

- 実行時エラー
- 業務エラー
- システムエラー

例外の種類

- Exception
- RuntimeException
- Error

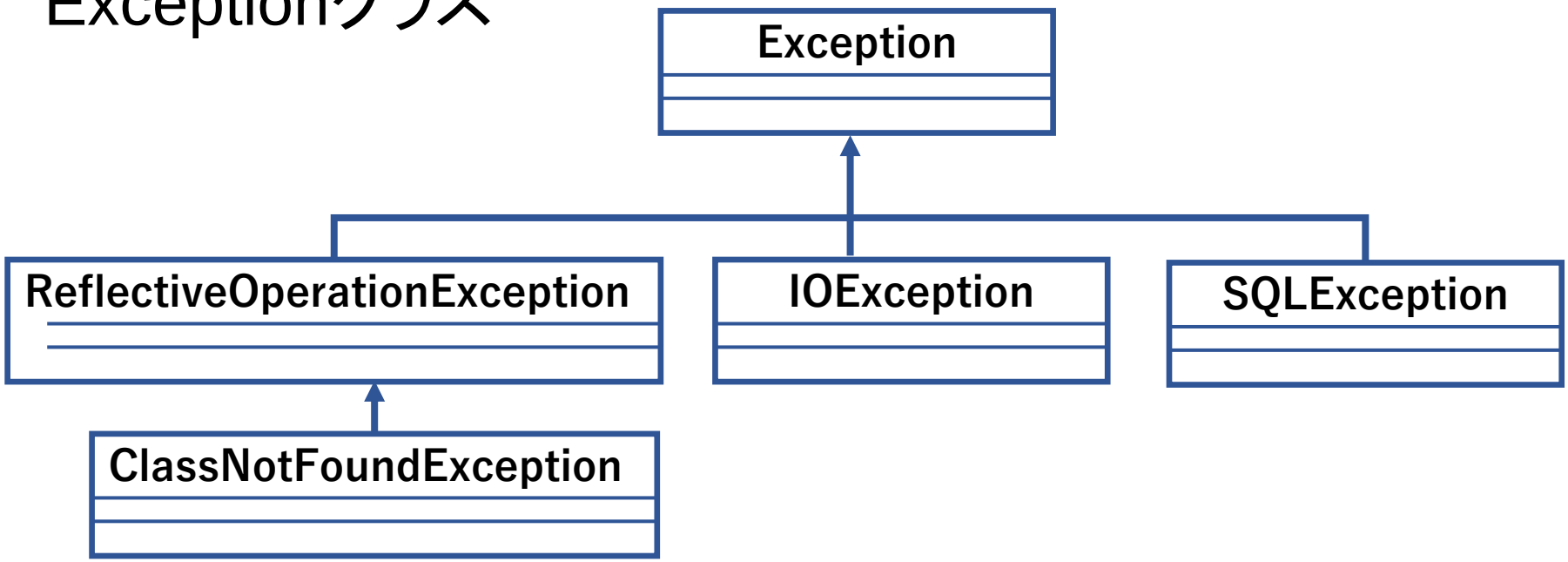


例外クラスの継承関係

例外クラスの継承関係

Goodworks
technology for the enterprise

Exceptionクラス

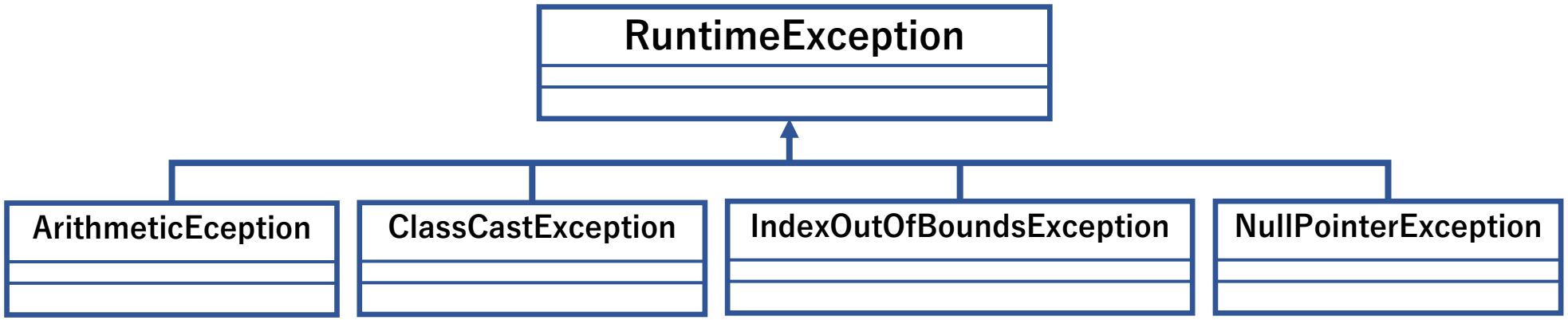


例外クラス名	説明
ClassNotFoundException	指定されたクラスが見つからない場合
IOException	入出力に関する不具合が生じた場合
SQLException	データベース操作に関する不具合が生じた場合

例外クラスの継承関係

Goodworks
technology for the enterprise

RuntimeExceptionクラス



例外クラス名	説明
ArithmeticException	算術演算に関する処理で例外的条件が発生した場合
ClassCastException	クラスを他のクラスにキャスト出来なかった場合
IndexOutOfBoundsException	配列などのインデックスの指定が範囲外だった場合
NullPointerException	nullが代入された変数にアクセスが行われた場合

例外処理

例外が発生した場合に指定した動作をさせる仕組み

- try ～ catch ～ finally
- throw
- throws

try ～ catch文

■ 構文

```
try {
      例外が発生する可能性のある処理
} catch（【例外クラス】　【変数名】）{
      例外発生時の処理
}
```

・try {・・・}のブロック内に例外が発生する可能性のある処理を記述
・catch(例外クラス 変数名)に、対処したい例外クラスの型名と例外インスタンスの参照値が代入される引数名を記述
・catch(){・・・}のブロック内に例外発生時の処理を記述する

try ～ catch文

■ 例

```
try {
      String strNum = “abc”;
      // int型に変換
      int num = Integer.parseInt(strNum);

} catch (NumberFormatException e) {
       System.out.println(“数値に変換できない文字列です。”);
}
```

複数のtry ～ catch文

Goodworks
technology for the enterprise

■ 構文

```
try {
      例外が発生する可能性のある処理
} catch（【例外クラス1】　【変数名1】）{
      例外発生時の処理
} catch（【例外クラス2】　【変数名2】）{
      例外発生時の処理
}
```

・複数の例外が発生する可能性がある場合、
　catchブロックを複数にする
・スーパークラスのcatchより、サブクラスのcatchを上に
　書かないとコンパイルエラーとなる

複数のtry ～ catch文

Goodworks
technology for the enterprise

■ 例

```
try {
      // 配列の0番目の値を取得
      String strNum = args[0];
      //  int型に変換
      int num = Integer.parseInt(strNum);

} catch (IndexOutOfBoundsException e) {
      System.out.println(“配列の要素数を超えています。”);
} catch (NumberFormatException e) {
      System.out.println(“数値に変換できない文字列です。”);
}
```

finally文

Goodworks
technology for the enterprise

■ 構文

```
try {
        例外が発生する可能性のある処理
} catch（【例外クラス】　【変数名】）{
        例外発生時の処理
} finally {
        必ず実行させたい処理
}
```

・例外発生の有無に関わらずに、必ず実行させたい処理を記述する

finally文

Goodworks
technology for the enterprise

■ 例

```
try {
      String strNum = “abc”;
      // int型に変換
      int num = Integer.parseInt(strNum);

} catch (NumberFormatException e) {
       System.out.println(“数値に変換できない文字列です。”);
} finally{
       System.out.println(“処理が終了しました”);
}
```

throw文

■ 構文

```
throw new 【例外クラスのコンストラクタ】(【引数】);
```

・明示的に例外を発生させることができる

■ 例

```
try {
      int age = -10;

      // 例外を発生させる
      if (age < 0) {
            throw new Exception();
      }
} catch (Exception e) {
      System.out.println(“年齢は0歳以上です。”);
}
```

自作例外

Goodworks
technology for the enterprise

Exceptionクラスを継承すると例外クラスを
自作することができる
Exceptionクラスのコンストラクタを呼び出す
コンストラクタを定義する

自作例外の定義

■ 例

```
public class MyException extends Exception {

	// スーパークラスのコンストラクタにメッセージを渡す
	public MyException(String errMsg) {
		super(errMsg);
	}
}
```

自作例外の利用

Goodworks
technology for the enterprise

■ 例

```
try {
      int age = -10;

      // 例外を発生させる
      if (age < 0) {
            throw new MyException(“年齢は0歳以上です。”);
      }
} catch (MyException e) {
      System.out.println(e.getMessage());
}
```

throws宣言

■ 構文

```
メソッド定義　throws　【例外クラス】，・・・{ }
```

・メソッド内で発生する可能性のある例外クラスを列挙して宣言する
・例外処理は呼び出し元に記述する

throws宣言

Goodworks
technology for the enterprise

■ 例

```
public class MainThrows {
	public static void main(String[] args) {
		// 呼び出し元で例外処理
		try {
			setAge(-1);
		} catch (Exception e) {
			System.out.println(“年齢は0歳以上です。”);
		}
	}

	public static void setAge(int age) throws Exception {
		if (age < 0) {
			throw new Exception();
		}
	}
}
```