オブジェクト指向技術 第11回
― オブジェクト指向プログラミング ―

立命館大学 情報理工学部
丸山 勝久

maru@cs.ritsumei.ac.jp

講義内容

- オブジェクト指向プログラミング
  - Java
  - クラス，メソッド，フィールド
  - パッケージとアクセス制御
  - インスタンスの生成とアクセス
  - 継承，抽象クラス，インタフェース
  - オーバーライド，オーバーロード
  - 配列とコレクション
  - 例外処理，入出力処理
  - GUIプログラミング
  - イベント駆動モデル

配列

- 同じ型の複数の要素を並べたもの
- 配列もクラス(参照型)
  - new演算子でインスタンスを生成

```
int[] numbers;  // int numbers[] と書いてもよい
numbers = new int[10];
numbers[5] = 1;
System.out.println(“5th = ” + numbers[5]);
```

10個分の領域を確保
(添字0～9)

```
int max = 10;
int[] numbers2 = numbers[max];
```

配列の長さを実行時に指定可能

```
Smartphone[] phones = new Smartphone[2];
phones[0] = new Smartphone("A", 35000);
System.out.println(phones[0].getName());
```

任意のインスタンスの
配列を定義可能

配列の初期化

```
int[] numbers = { 1, 2, 3, 4, 5 };
String[] strings = { "A", "B" };
Smartphone[] phones = {
    new Smartphone("A", 35000),
    new Smartphone("B")
};
```

配列宣言と同時に
初期値を設定可能

```
int[] numbers = { 1, 2, 3, 4, 5 };
int[] alias = numbers;
System.out.println(numbers[0]);
System.out.println(alias[0]);
```

配列は参照型
インスタンスnumbersへの
参照値が複製されてaliasに代入

配列へのアクセス

配列の長さ(大きさ)

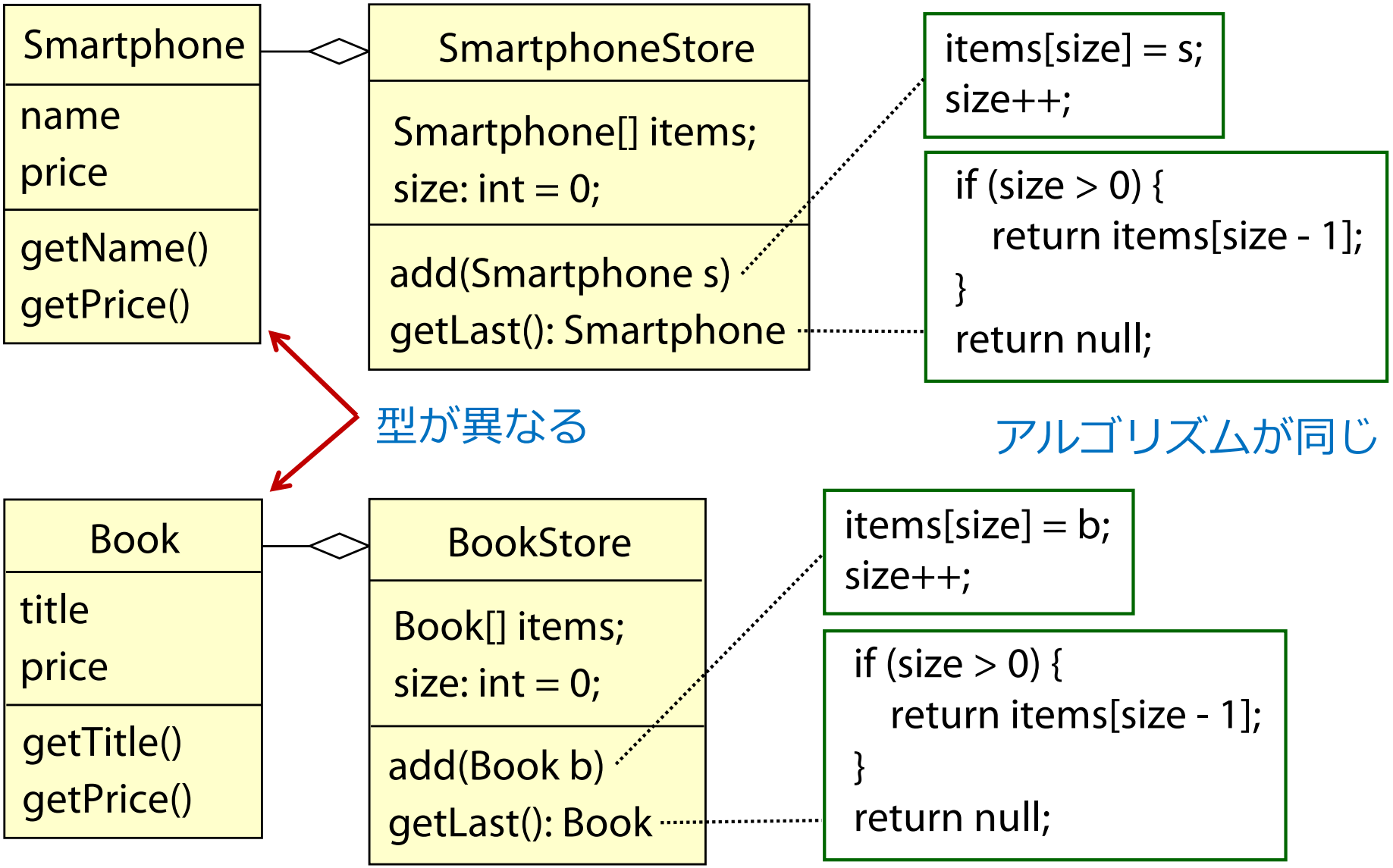
```
int[] numbers = { 1, 2, 3, 4, 5 };
for (int index = 0; index < numbers.length; index++) {
    System.out.println(numbers[index]);
}
```

```
int[] numbers = { 1, 2, 3, 4, 5 };
for (int number : numbers) {
    System.out.println(number);
}
```

配列の要素を順番に走査

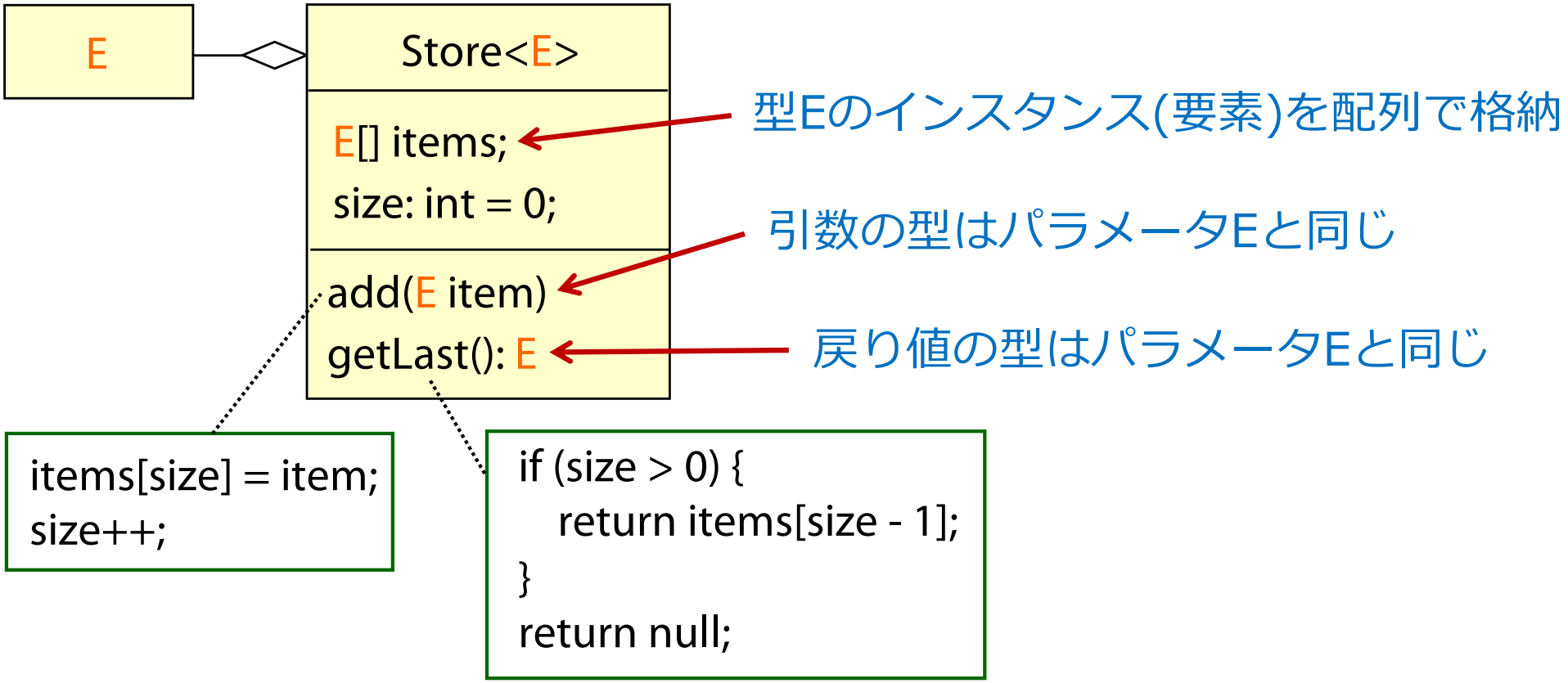
配列の長さを明示的に指定しなくて良い

ジェネリクス(generics)の利用場面



ジェネリクス(generics)

- 異なる型に対して同じアルゴリズムを適用できる仕組み
  - クラスの一部をパラメータ化(parameterized class)



アルゴリズムの記述を1つに統合

ジェネリクスの利用

- コンパイル時に型が区別される

EをSmartphoneに束縛

```
Store<Smartphone> phoneStore = new Store<>();
phoneStore.add(new Smartphone("A", 35000));
// phoneStore.add(new Book("B", 1000));
Smartphone s = phoneStore.getLast();
System.out.println(s.getName());
```

phoneStoreには
Smartphoneの
インスタンスしか
add()できない

コンパイルエラー

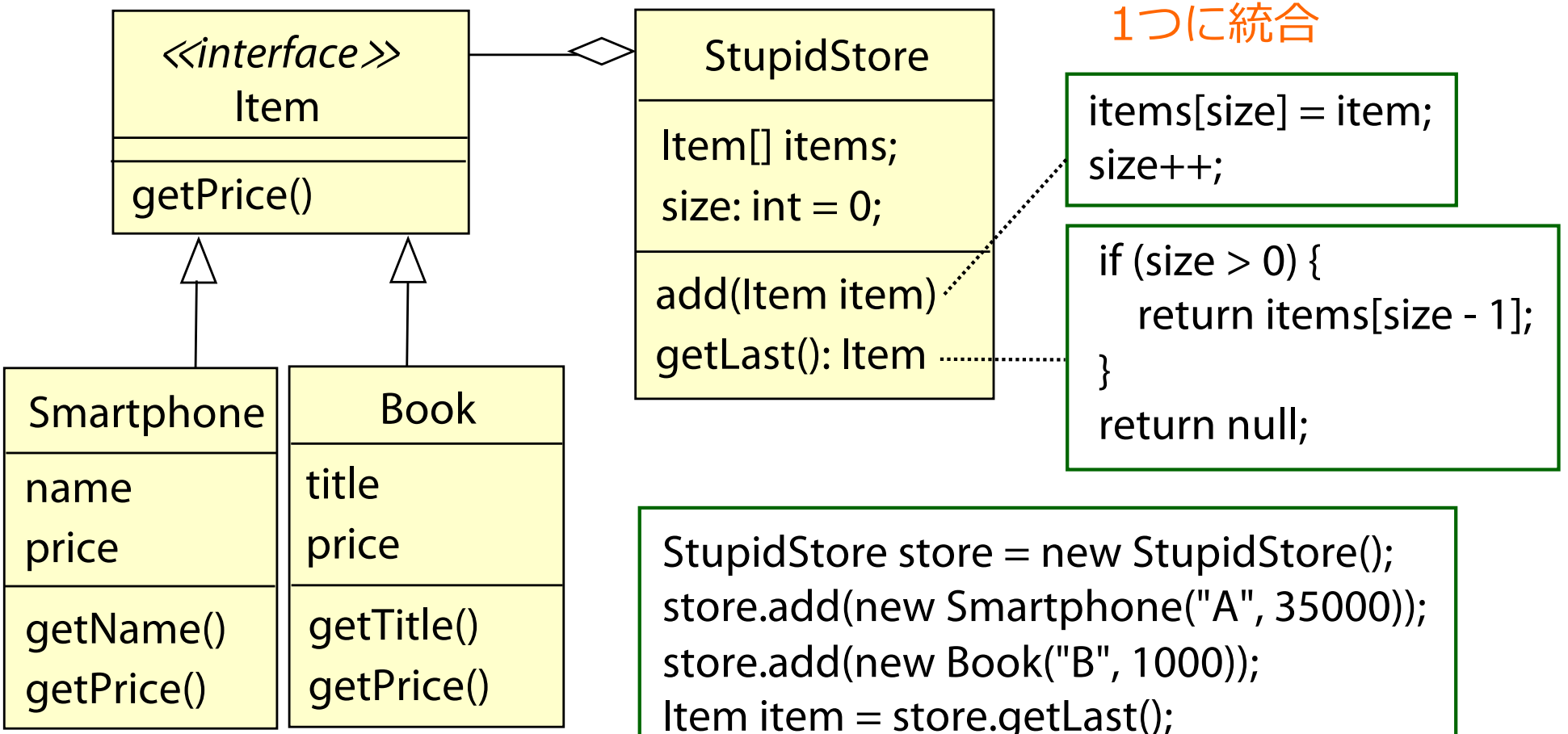
getLast()の戻り値は
必ずSmartphoneの
インスタンス

EをBookに束縛

```
Store<Book> bookStore = new Store<>();
bookStore.add(new Book("B", 1000));
Book b = bookStore.getLast();
System.out.println(b.getTitle());
```

getLast()の戻り値は
必ずBookの
インスタンス

多態性により解決すると



アルゴリズムの記述を
1つに統合

```
items[size] = item;
size++;
```

```
if (size > 0) {
    return items[size - 1];
}
return null;
```

```
StupidStore store = new StupidStore();
store.add(new Smartphone("A", 35000));
store.add(new Book("B", 1000));
Item item = store.getLast();
Book b = (Book)store2.getLast();
System.out.print(b.getTitle());
```

ItemはgetTitle()を持たないため，Bookへのキャストが必要

コレクション

- 任意のインスタンスをまとめて格納する仕組み
  - リスト（*List*, ArrayList）
    - 配列と同様の操作が可能
  - 集合（*Set*, HashSet, TreeSet）
    - 値の重複を許さない
  - マップ（*Map*, HashMap, TreeMap）
    - キーと値の対応付け

*List, Set, Map*はインタフェース

```
List<Smartphone> list = new ArrayList<>();
// ArrayList<Smartphone> list = new ArrayList<>();
Smartphone phone1 = new Smartphone("ABC", 35000);
Smartphone phone2 = new Smartphone("XYZ");
list.add(phone1);
list.add(phone2);
System.out.println("size = " + list.size());
```

より汎用的な型で宣言する方が良い

要素数の取得

コレクションへのアクセス

```
import java.util.*;

public class Main5 {
    public static void main(String[] args) {
        List<Smartphone> list = new ArrayList<>();
        list.add(new Smartphone("P", 35000));
        list.add(new Smartphone("Q"));
        list.add(new Smartphone("R", 28000));
        list.add(new Smartphone("S", 20000));

        for (Smartphone phone : list) {
            phone.print();
        }
    }
}
```

=

```
Iterator<Smartphone> it = list.iterator();
while (it.hasNext()) {
    Smartphone phone = it.next();
    phone.print();
}
```

← 反復子を利用

実行結果

```
P: 35000-yen
Q: 30000-yen
R: 28000-yen
S: 20000-yen
```

参照オブジェクトと値オブジェクト

- 参照オブジェクト(reference object)
  - 識別性により同一性を判定
- 値オブジェクト(value object)
  - 値により等価性を判定

```
phone1 = new Smartphone("090-1111-1111");
phone2 = phone1;
phone3 = new Smartphone("090-1111-1111");
phone4 = new Smartphone("090-8888-8888");
```

phone1とphone2は同一である

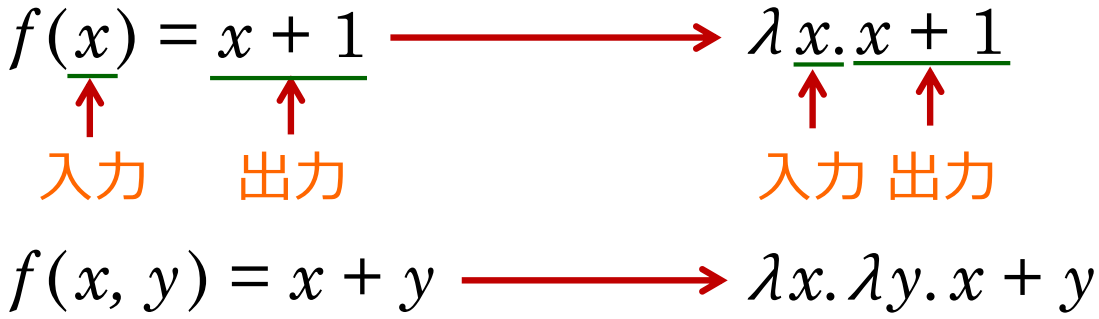
phone1とphone3は同一でない

phone1とphone3は等価である

phone1とphone4は等価でない

ラムダ記法

- ラムダ式
  - 記号ラムダ(λ)を用いて，関数の入力と出力を表現



- 名前つけずに関数を定義可能
- 関数を引数や戻り値に指定できる
- コレクションと一緒に利用されることが多い
  - External iteration(外部的な繰り返し)を
    Internal iteration(内部的な繰り返し)に置換できる

繰り返しの実現

```
List<String> cities = Arrays.asList("Tokyo", "Osaka", "Dalian");
```

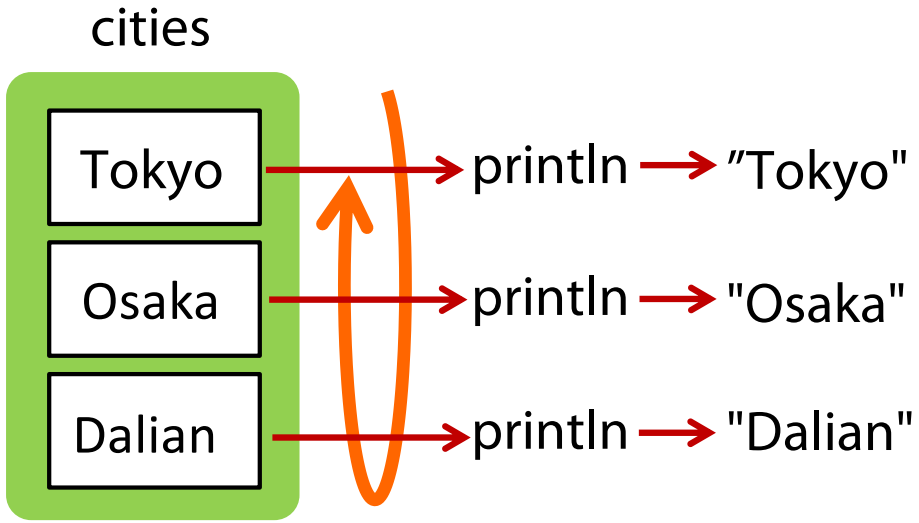
External iteration

```
for (String city : cities) {
    System.out.println(city);
}
```

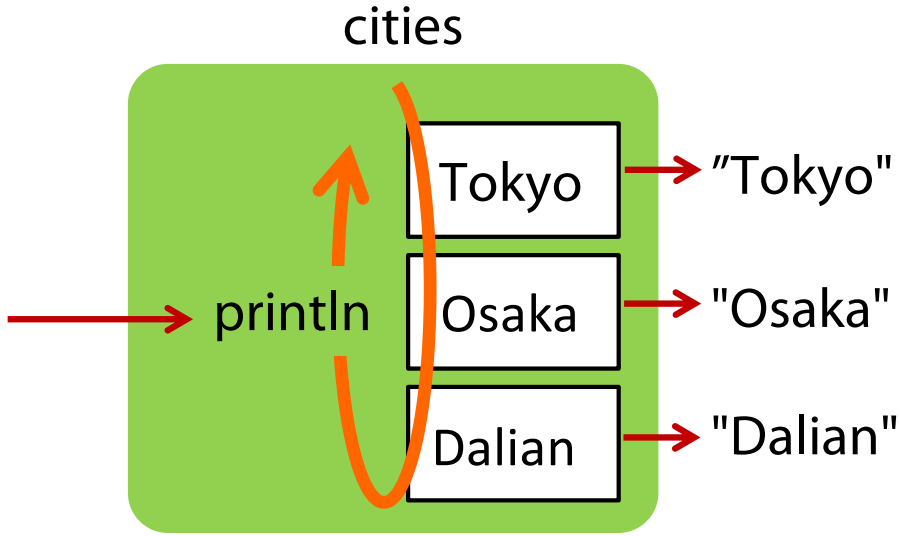
Internal iteration

```
cities.forEach(city -> System.out.println(city));
```

匿名関数(ラムダ式)



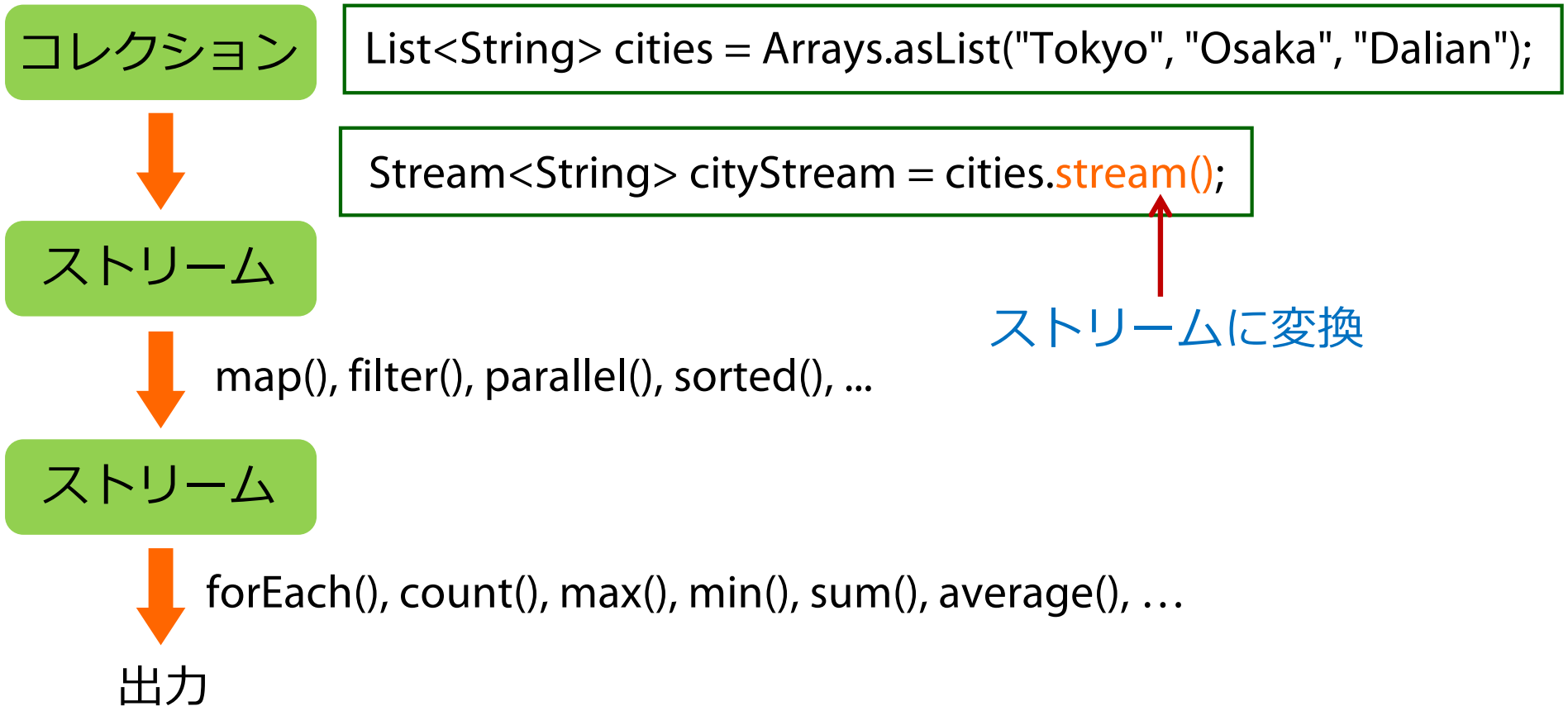
繰り返し処理は外部にある



繰り返し処理を内部に入れる

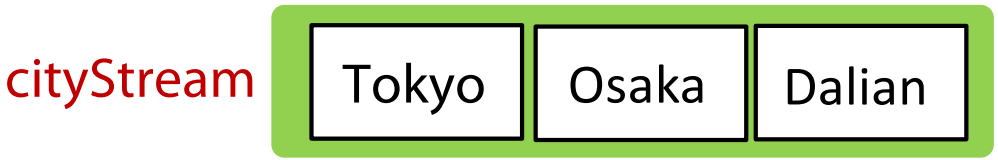
ストリーム(stream)

- java.util.stream.Stream
  - 複数のオブジェクトに対して，
    何らかの処理を一括して実行できるクラス



ストリームの利用例

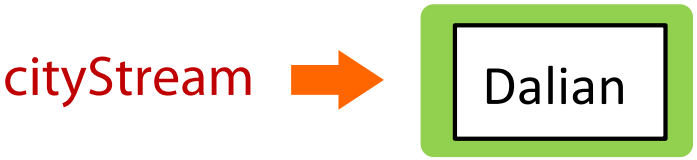
```
List<String> cities = Arrays.asList("Tokyo", "Osaka", "Dalian");
Stream<String> cityStream = cities.stream();
```




```
cityStream
.filter(city -> city.length() > 5)
.count();
```

長さが5を超える文字列だけ残す

文字列の数を返す




```
cityStream
.map(city -> city.toUpperCase())
.forEach(city -> System.out.println(city));
```

文字列を大文字に変換する

文字列を画面に出力する

例外処理

- 正常な処理と例外処理を分離する仕組み

```
int[] numbers = { 1, 2, 3 };
try {
    print(numbers, 0);
    print(numbers, 1);
    print(numbers, 10);
    System.out.println("Success");
} catch (Exception e) {
    System.out.println("Fail: " + e);
} finally {
    System.out.println("Finish!");
}
```

例外捕捉区間

例外処理

必ず実行される

例外処理を呼び出し側に転送

```
void print(int[] numbers, int index) throws Exception {
    System.out.println(numbers[index]);
}
```

例外の種類

- Error系
  - 回復不可能な重大エラーを伝達
  - プログラムを停止
  - クラスErrorを祖先に持つクラス群
    - OutOfMemoryError，StackOverflowError など
- Exception系
  - 回復可能なエラーや特別な状況を伝達
  - プログラムの実行を続行
  - クラスExceptionを祖先に持つクラス群
    - IOException, FileNotFoundException など
- チェック例外(checked exception)
  - コンパイル時に例外処理を検査(try-catch節が必須)
- 非チェック例外(unchecked exception)
  - コンパイル時に例外処理を未検査(try-catch節は任意)

入出力処理

- デバイスへの入出力
  - ストリーム(stream)として扱う
- 標準入出力
  - 標準入力ストリーム (通常はキーボード)
  - 標準出力ストリーム (通常はディスプレイ)
  - 標準エラーストリーム (通常はディスプレイ)
- 入出力ストリームの種類
  - バイナリストリーム：1バイト単位で読み書き
  - テキストストリーム：1文字単位で読み書き
- バッファリング機能
  - 入出力内容をメモリ(バッファ)に一時的に蓄積することで、その都度メモリアクセスするより高速に
  - Bufferedで始まる名前のクラスで提供

コンソール入力

- System.in
- バイト単位の入力ストリームを提供
  - 通常はそのまま使わない
- java.util.Scannerクラスが提供するメソッドを利用
  - String next()
    - 1トークン読み取った結果を文字列として返す
  - String nextLine()
    - 1行読み取った結果を文字列として返す
  - int nextInt()
    - 読み取った結果をint型として返す

```
Scanner scanner = new Scanner(System.in);
String token = scanner.next();
```

コンソール出力

- System.out
  - void print()
    - 引数に指定されたものを標準出力に出力
  - void println()
    - 引数に指定されたものを標準出力に出力し、改行する
  - void flush()
    - ストリームをフラッシュ(バッファの内容を出力)
  - print(), println()は，オーバーロードにより
    さまざまな型の引数を受け付け可能
- System.errも基本的には同じ
  - エラーメッセージを出力する際に利用

```
System.out.println("Output");
System.err.println("Error");
```

ファイル入力

- java.io.FileReader / java.io.BufferedReader
  - String readLine()
    - 1行ごとに読み込み
  - void close()
    - ストリームを閉じる
    - try-with-resources文を利用すれば記述不要

```
try (BufferedReader br = new BufferedReader(new FileReader(filepath))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (FileNotFoundException e) {
    System.out.println("File Not Found: " + filepath);
} catch (IOException e) {
    System.out.println("Cannot read: " + filepath);
}
```

ファイルが存在しない/開けない場合に例外が発生

読み込みに失敗する場合に例外が発生

ファイル出力

- java.io.FileWriter / java.io.BufferedWriter
  - void write(String)
    - 指定した文字列を書き込む
  - void close()
    - ストリームをフラッシュして閉じる
- java.io.PrintWriter
  - void print(String)
    - 指定した文字列を書き込む

```
try (BufferedWritwer br = new BufferedWriter(new FileWriter(filepath))) {
    br.write(data);
} catch (IOException e) {
    System.out.println("Cannot write: " + filepath);
}
```

書き込みに失敗する場合に例外が発生

練習問題(Exec4)

```
class PC {
    int price;
    PC(int price) {
        this.price = price;
    }
    String getInfo() {
        return "$" + price;
    }
}

class DiscountedPC extends PC {
    DiscountedPC(int price) {
        super((int)(price * 0.8));
    }
    String getInfo() {
        return super.getInfo() + "!";
    }
}
```

```
public class Exec4 {
    public static void main(String[] args) {
        PC pc1 = new PC(250);
        DiscountedPC pc2 = new DiscountedPC(250);

        List<PC> pcs = new ArrayList<>();
        pcs.add(pc1);
        pcs.add(pc2);
        pcs.add(new PC(100));
        for (PC pc : pcs) {
            System.out.println(pc.getInfo());  // (1)
        }

        pcs.remove(0);
        for (PC pc : pcs) {
            System.out.println(pc.getInfo());  // (2)
        }
    }
}
```

```
$250
$200!
$100
```

```
$200!
$100
```

まとめ

- 配列は参照型のクラスであるので，
  new演算子でインスタンスを生成する必要がある
- ジェネリクスを利用することで，コレクションに対して安全に要素を格納したり，要素を取り出したりできる
- ラムダ式を利用することで，名前なしの関数を定義できる
- ラムダ式とストリームと組み合わせることで，
  コレクションを簡潔に処理する記述が書ける
- try-catch文により，正常な処理と例外処理を
  分離して記述できる
- コンソール(画面)やファイルの入出力処理には，
  ストリームを利用する

次回の講義の最初に小テストを行います