オブジェクト指向技術 第15回
― 復習 ―

立命館大学 情報理工学部
丸山 勝久

maru@cs.ritsumei.ac.jp

クラスの記述

クラス宣言

フィールドの宣言

コンストラクタの宣言

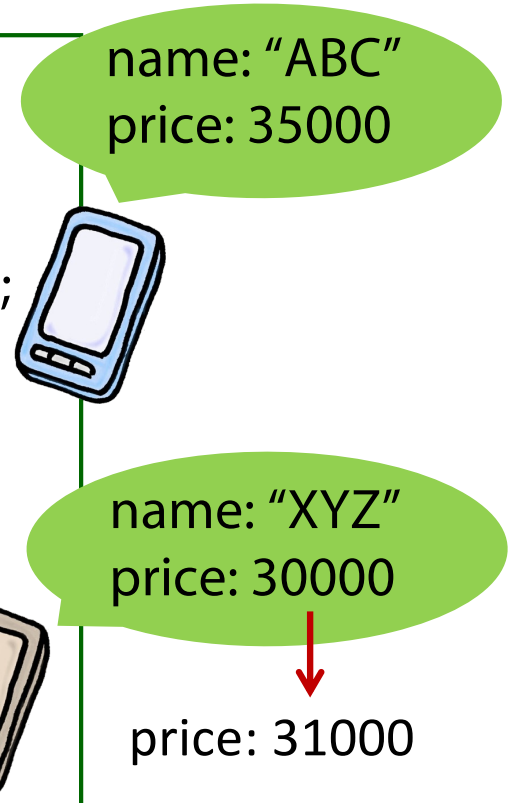
メソッドの宣言

```
public class Smartphone {
   private String name;
   int price;

   public Smartphone(String name, int p) {
      this.name = name;
      price = p;
   }
   public Smartphone(String name) {
      this(name, 30000);
   }
   public String getName() {
      return name;
   }
   public void setPrice(int p) {
      price = p;
   }
   public int getPrice() {
      return price;
   }
   public String getPriceInfo() {
      return getPrice() + "-yen";
   }
   public void print() {
      System.out.println(
        name + ": " + getPriceInfo());
   }
}
```

クラスの利用

new演算子とドット(.)演算子を利用

```
public class Main1 {

    public static void main(String[] args) {
        Smartphone phone1 = new Smartphone(“ABC”, 35000);
        System.out.println("Price = " + phone1.price);
        System.out.println("Price = " + phone1.getPrice());

        Smartphone phone2 = new Smartphone("XYZ");
        System.out.println("Price = " + phone2.price);
        phone2.setPrice(phone2.price + 1000);
        System.out.println("Price = " + phone2.getPrice());
    }
}
```



小テスト1の解答

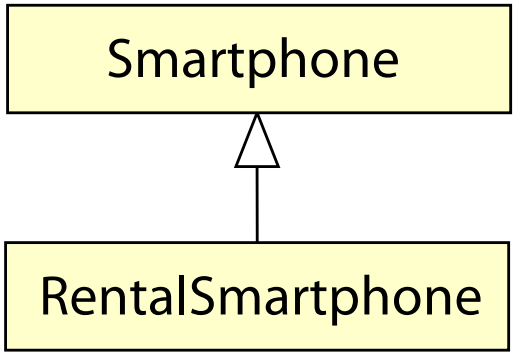
```
class Book {
    private String title;
    private int price;

    Book(String title, int price) {
        this.title = title;
        this.price = price;
    }
    Book(String title) {
        this(title, 1000);
    }
    void setPrice(int p) {
        price = p;
    }
    String getInfo() {
        return title + ":" + price;
    }
}
```

```
public class Quiz1 {
    public static void main(String[] args) {
        Book book1 = new Book("X", 3500);
        System.out.println(book1.getInfo());   X:3500

        Book book2 = new Book("Y");
        book2.setPrice(2400);
        System.out.println(book2.getInfo());   Y:2400

        Book book3 = book1;
        System.out.println(book3.getInfo());   X:3500
        book1.setPrice(3000);
        System.out.println(book3.getInfo());   X:3000

        book3 = book2;
        book3.setPrice(2200);
        System.out.println(book2.getInfo());   Y:2200
    }
}
```

第9回のまとめ

- クラスを必ず記述する
- フィールドとメソッドはクラス宣言の中に記述する
- パッケージを使うと，同じ名前のクラスを区別できる
- thisは自インスタンスを指す
- インスタンスを生成する際には，new演算子を使う
- インスタンスにアクセスする際には，ドット演算子を使う
- クラス変数は，そのクラスのすべてのインタンスで共有される
- クラスメソッドはインスタンスなしで呼び出せる
- 参照型の代入や引数渡しでは，インスタンスの参照値が複製される

継承の記述



継承

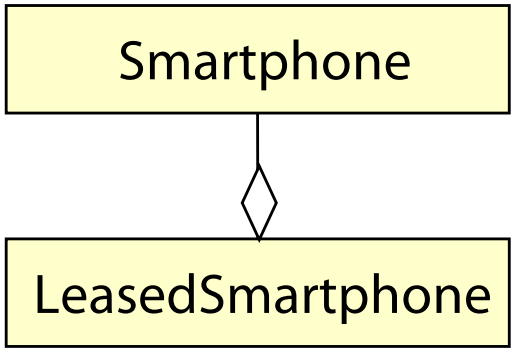
```
public class Smartphone {
    …
    public Smartphone(String name, int p) { … }
    public int getPrice() { return price; }
}
```

```
public class RentalSmartphone extends Smartphone {
    private int period;

    public RentalSmartphone(String name, int price, int period) {
        super(name, price);
        this.period = period;
    }
    public void setPeriod(int period) { this.period = period; }
    public int getPrice() { return price * period; }
}
```

オーバーライド

合成(集約)の記述

- 別のクラスを包含する仕組み




```
public class Smartphone {
   …
   public Smartphone(String name, int p) { … }
   public int getPrice() { return price; }
}
```

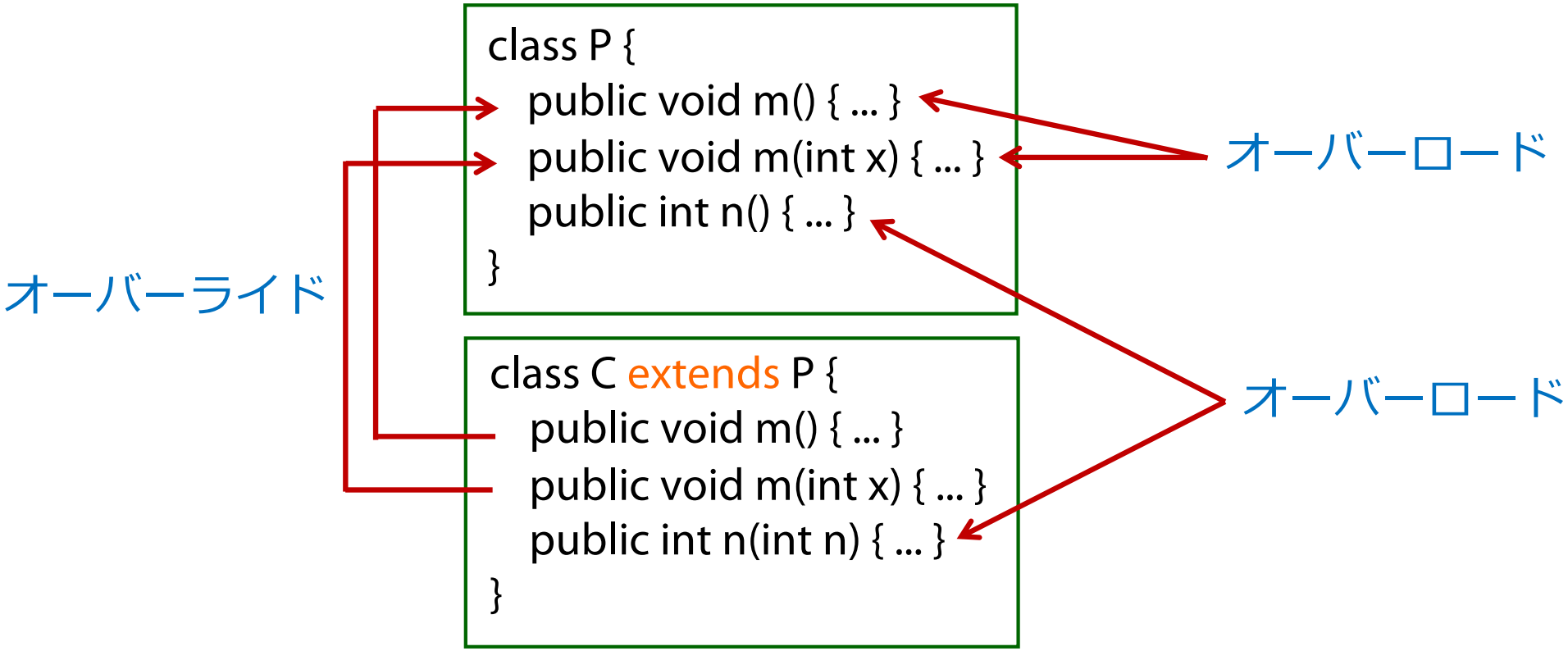
委譲(転送)

```
public class LeasedSmartphone {
   private Smartphone phone;
   private String company;
   public LeasedSmartphone(Smartphone phone, String c) {
      this.phone = phone; company = c;
      int price = (int)(phone.getPrice() * 0.8); phone.setPrice(price);
   }
   public String getCompany() { return company; }
   public int getPrice() { return phone.getPrice(); }
}
```

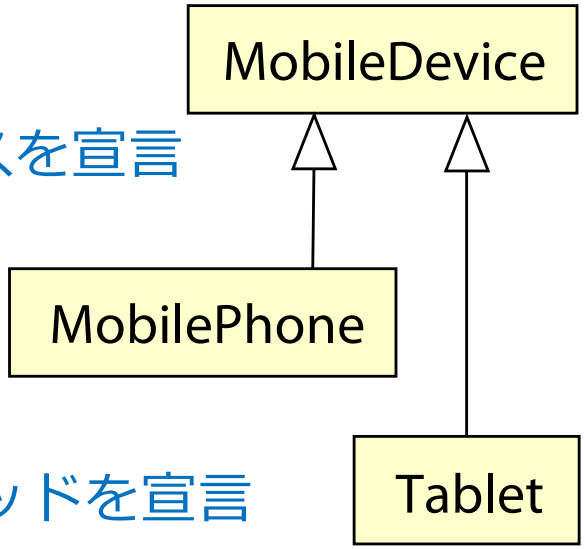
参照を格納

オーバーロードとオーバライド

- オーバーロード(overload)
  - 同じ名前のメソッドを複数宣言すること
    - 引数の数や型が互いに異なる
- オーバーライド(override)
  - 子クラスのメンバが親クラスのメンバを再定義(上書き)すること



抽象クラスと継承の記述




```
abstract public class MobileDevice {
    protected String name;
    protected MobileDevice(String name) {
        this.name = name;
    }
    abstract public void feature();
}
```

抽象クラスを宣言

抽象メソッドを宣言

```
public class MobilePhone extends MobileDevice {
    public MobilePhone(String name) { super(name); }
    public void feature() { System.out.println(name + " is Small"); }
}
```

feature()を実装

```
public class Tablet extends MobileDevice {
    public Tablet(String name) { super(name); }
    public String getName() { return name; }
    public void feature() { System.out.println(name +  " is Big"); }
}
```

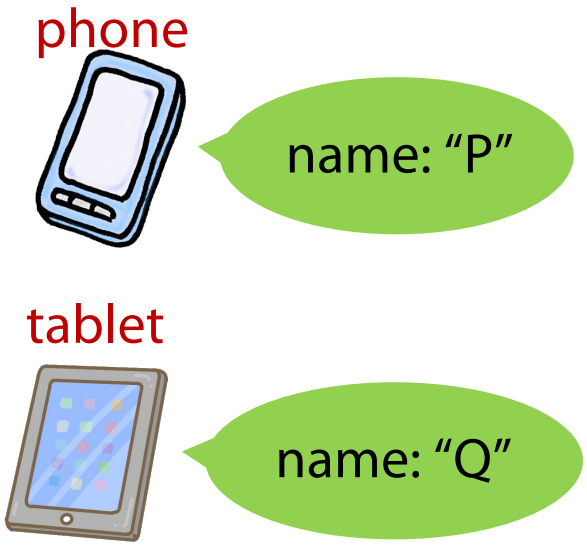
feature()を実装

多態性と動的束縛の記述

```
public class Main4 {
    public static void main(String[] args) {
        MobilePhone phone = new MobilePhone("P");
        phone.feature();

        Tablet tablet = new Tablet("Q");
        tablet.feature();

        MobileDevice device;
        device = phone;
        device.feature();
        device = tablet;
        device.feature();
    }
}
```

MobilePhoneの feature()を呼出し

Tabletの feature()を呼出し



小テスト2の解答

```
class Book {
    private String title;
    private int price;

    Book(String title, int price) {
        this.title = title;
        this.price = price;
    }
    void setPrice(int p) { price = p; }
    String getInfo() {
        return title + ":" + price;
    }
}
```

```
class EBook extends Book {
    EBook(String title, int price) {
        super(title, price);
    }
    String getInfo() {
        return "E#" + super.getInfo();
    }
}
```

```
public class Quiz2 {
    public static void main(String[] args) {
        Book a = new Book("A", 1000);
        System.out.println(a.getInfo());

        EBook b = new EBook("B", 3000);
        System.out.println(b.getInfo());

        b.setPrice(2500);
        System.out.println(b.getInfo());

        Book c = b;
        b.setPrice(2000);
        System.out.println(c.getInfo());

        c = a;
        a.setPrice(1500);
        System.out.println(c.getInfo());
    }
}
```

```
A:1000
E#B:3000
E#B:2500
E#B:2000
A:1500
```

第10回のまとめ

- 継承はextendsを用いて，明示的に定義する
- オーバーロードにより，同じ名前のメソッドを複数宣言することができる
- オーバーライドにより，子クラスのメンバが親クラスのメンバを再定義(上書き)することができる
- superは親クラスのメンバを参照する際に利用する
- 抽象クラスはabstractを用いて，明示的に宣言する
- インタフェースはimplementsを用いて継承する
- 多態性を利用することで，親クラスの型で宣言された変数に子クラスのインスタンスを代入することができる
- 動的束縛を利用することで，呼び出されるメソッドを実行時に切り替えることができる

ジェネリクス

- 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

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

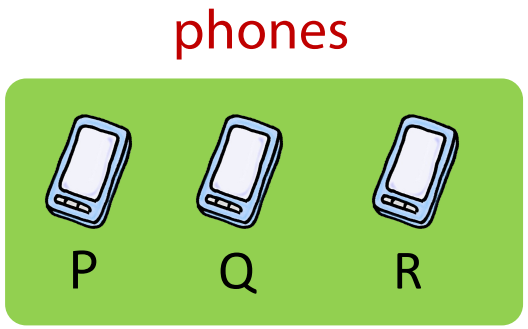
```
Store<Book> bookStore = new Store<>();
bookStore.add(new Book("B", 1000));
Book b = bookStore.getLast();
System.out.println(b.getTitle());
```

getLast()の戻り値は
必ずBookの
インスタンス

配列とコレクション

配列

```
Smartphone[] phones = new Smartphone[2];
phones[0] = new Smartphone("P", 35000));
phones[1] = new Smartphone("Q");
phones[2] = new Smartphone("R", 28000));
```




```
for (int index = 0; index < phones.length; index++) {
    phones[index].print();
}
```

コレクション

```
List<Smartphone> phones = new ArrayList<>();
phones.add(new Smartphone("P", 35000));
phones.add(new Smartphone("Q"));
phones.add(new Smartphone("R", 28000));
```

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

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]);
}
```

小テスト3の解答

```
abstract class Book {
    protected int price;

    Book(int price) {
        this.price = price;
    }
    int getPrice() { return price; }
    abstract void discount();
}
```

```
class Novel extends Book {
    Novel(int price) { super(price); }
    void discount() {
        price = price / 2;
    }
}
```

```
class Comic extends Book {
    Comic(int price) { super(price); }
    void discount() {
    }
}
```

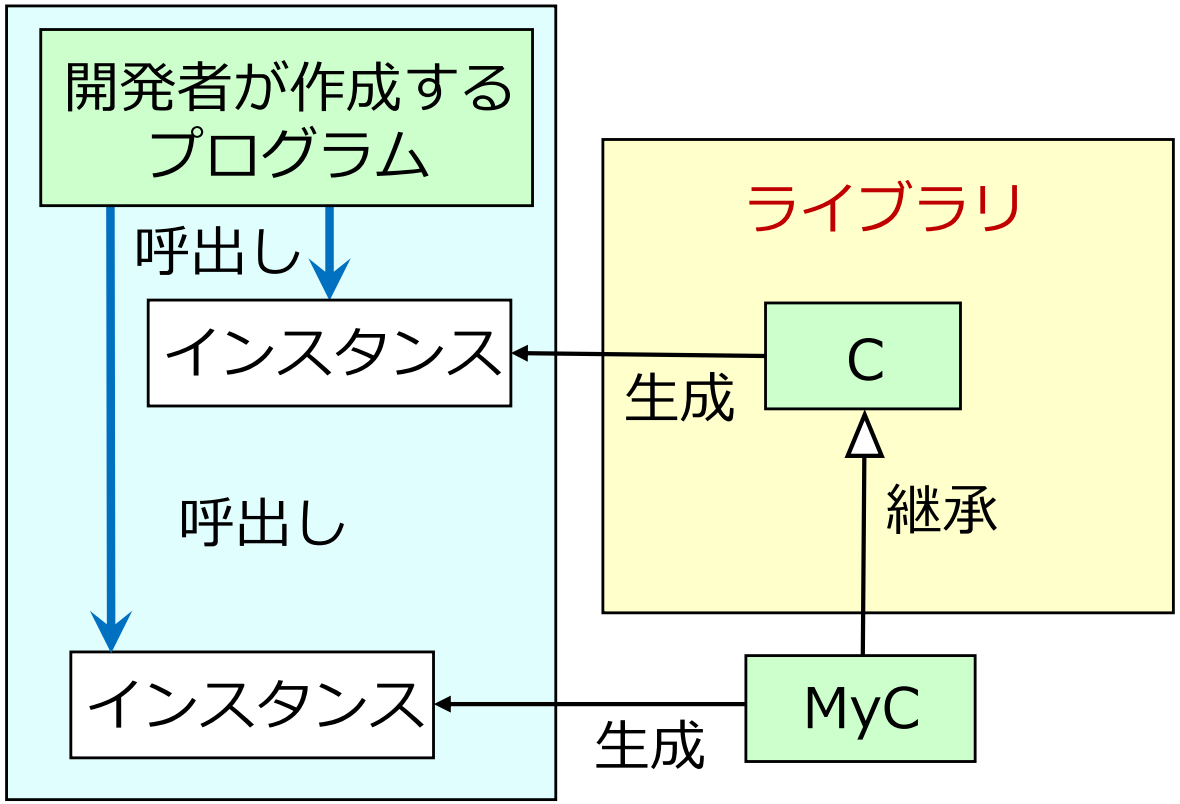
```
import java.util.*;
public class Quiz3 {
    public static void main(String[] args) {
        Book book = new Novel(800);
        List<Book> books = new ArrayList<>();
        books.add(new Novel(600));
        books.add(book);
        books.add(new Comic(500));
                                                    600 800 500
        System.out.println(getOutput(books));
        for (Book b : books) { b.discount(); }
        System.out.println(getOutput(books));
        long c = books.stream()                     300 400 500
            .filter(b -> b.getPrice() > 500).count();
        System.out.println(c);                              0
    }
    static String getOutput(List<Book> books) {
        String s = "";
        for (Book b : books) { s = s + b.getPrice() + " "; }
        return output;
    }
}
```

第11回のまとめ

- 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

クラスライブラリ

- 再利用されることを意図して作成されたモジュールの集まり
  - 既存クラスのインスタンス生成と継承による再利用




```
class C {
    void m() { … }
}
```

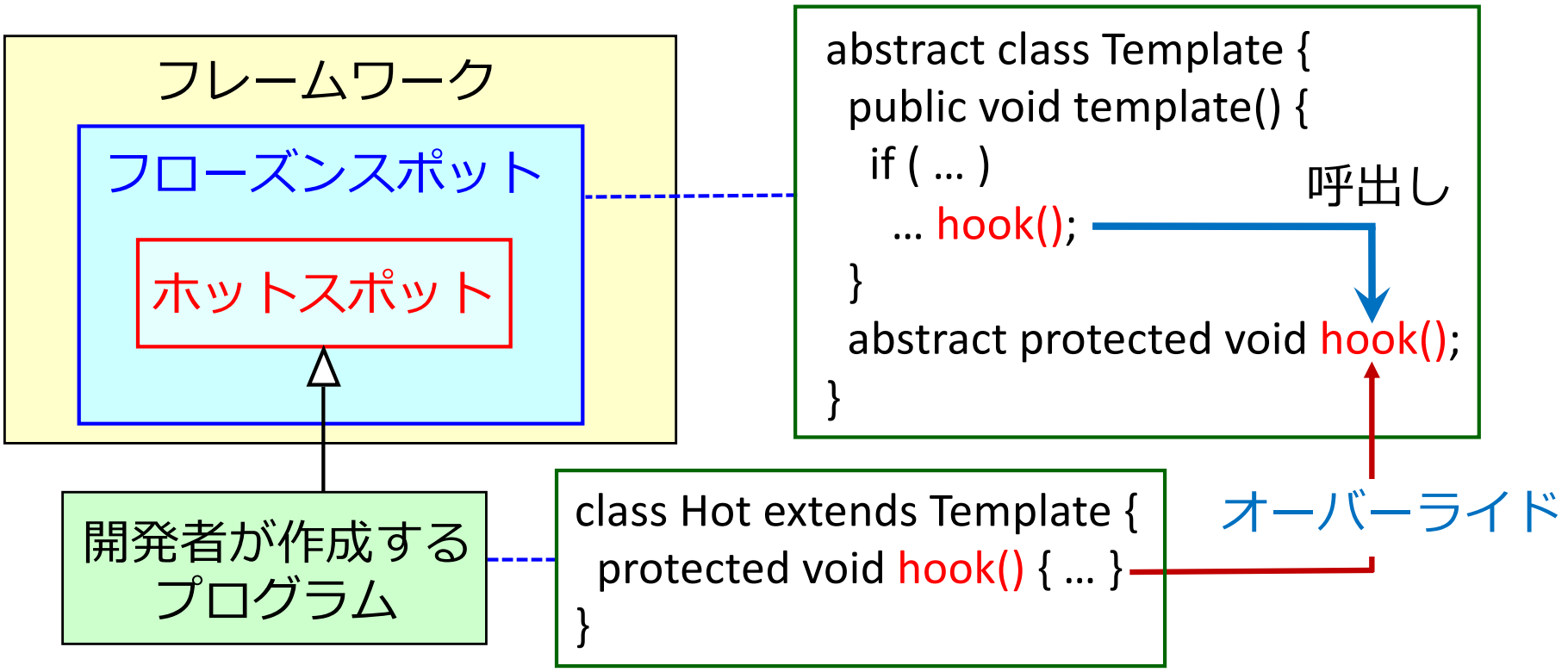
```
C c = new C();
c.m();
```

```
class MyC extends C {
    …
}
```

```
MyC c = new MyC();
c.m();
```

オブジェクト指向フレームワーク

- 変更不可なフローズンスポット(frozen spot)と，可変部分のホットスポット(hot spot)で構成
  - 開発者はホットスポットのみ開発すれば良い



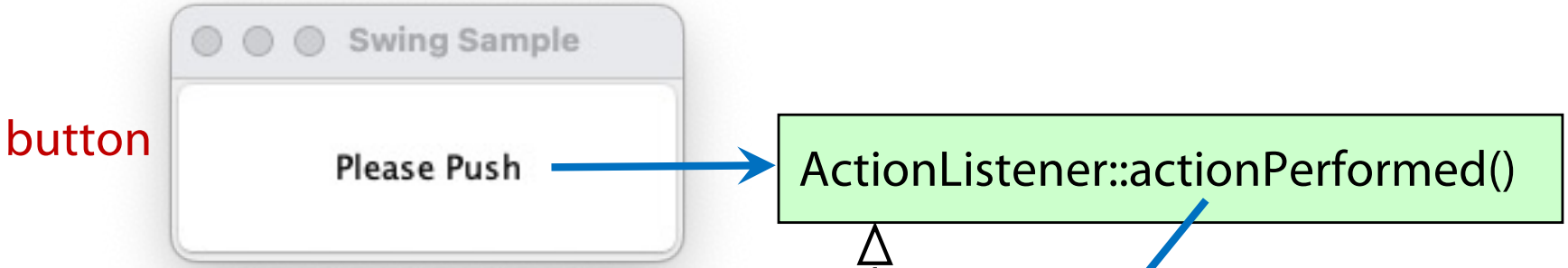
イベント駆動モデル

```
JButton button = new JButton("Please Push");
ButtonListener bl = new ButtonListener();
button.addActionListener(bl);
```

イベントソースを作成

イベントリスナの生成

イベントリスナの登録



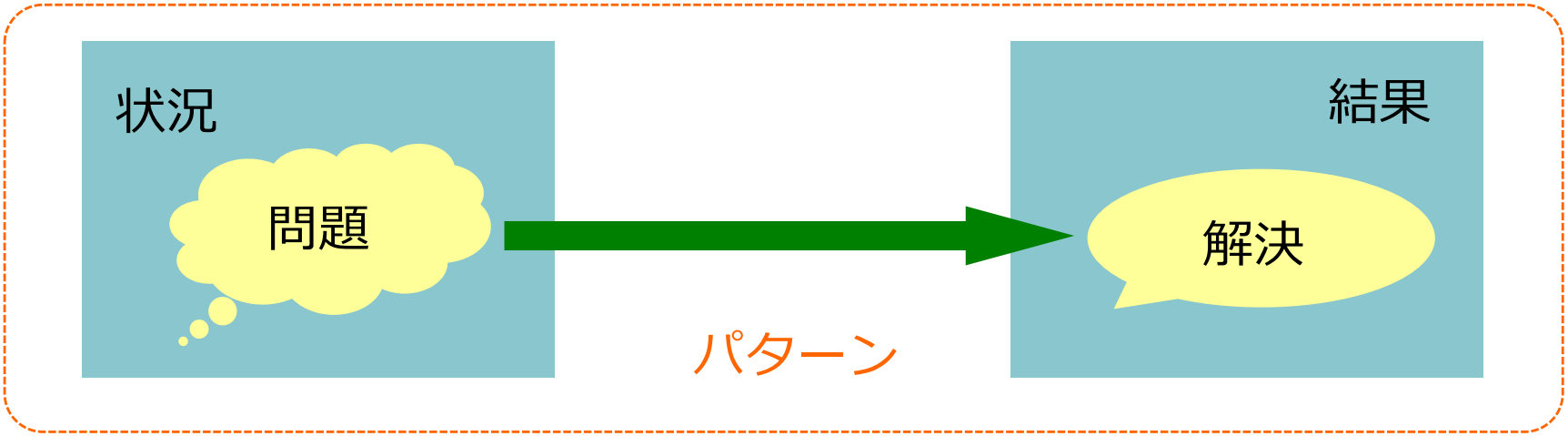

```
class ButtonListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      System.out.println("Pushed Button");
   }
}
```

第12回のまとめ

- ライブラリ内のクラスは，インスタンスを直接生成することで
  再利用するか，継承により再利用する
- フレームワークを利用すると，その内部のクラスだけでなく，
  そのクラスのインスタンスに関する相互作用も再利用できる
- スレッドは，メモリ空間を共有した上で並行に実行できる
- Java Swingでは，メイン・スレッドの他に
  イベントディスパッチ・スレッドが自動的に実行される
- イベント駆動モデルでは，プログラム実行中に発生した
  イベントに従って受動的に処理を行う
- イベントディスパッチ・スレッドがコンポーネントを監視し，
  イベントが発生した場合には，
  あらかじめ登録したイベントリスナにイベントを通知する
- イベント駆動モデルの標準的な実現方法にObserverパターンがある

ソフトウェアパターン

- 開発者の経験や知見を再利用できるように体系化したもの
  - どのような状況(context)で利用するか
  - どのような問題(problem)に適用できるか
  - どのように問題を解決(solution)するか
  - どのような結果(consequence)や効果(利点や欠点)を与えるか
- 熟練開発者たちの過去の成功例
- コミュニケーションを効率的に行うための標準的な語彙



リファクタリング

観測可能な振る舞い
を変えない
＝
外部的挙動を保存

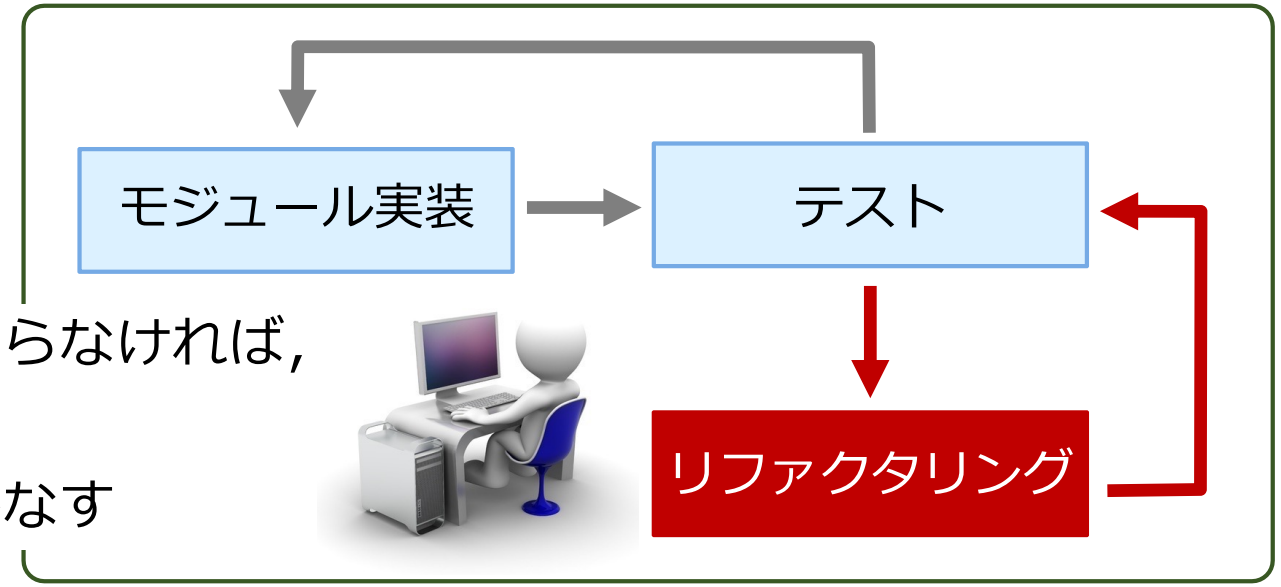
開発直後の
ソース
コード

設計や実装の
劣化

劣化した
ソース
コード

リファクタリング

改善後の
ソース
コード



テスト結果が変わらなければ,
外部的振る舞いを
保存しているとみなす

第13回のまとめ(1/2)

- ソフトウェアパターンは，開発者の経験や知見を再利用できるように体系化したものである
- ソフトウェアパターンは名前を持ち，コミュニケーションを効率的に行うための標準的な語彙となる
- ソフトウェアパターンは，どのような状況で利用するか，どのような問題に適用できるか，どのように問題を解決するか，どのような結果や効果(利点や欠点)を与えるかという観点でカタログ化されている
- デザインパターンは，変更の繰り返しによる良い設計を集めたものである

第13回のまとめ(2/2)

- Observerパターンでは，あるインスタンスの状態が変化したときに変化を自動的に通知する
- Iteratorパターンでは，配列やコレクション内部の要素に対して内部のデータ構造を隠蔽した上で特定の順番で処理を行う
- Template Methodパターンでは，処理の枠組みを先に定め，具体的な処理を後から作成する
- Façadeパターンでは，互いに関係を持つ複数のクラスを利用する窓口を用意する
- Builderパターンでは，複雑な構造を持つインスタンスをいくつかの手順に分割する
- Bridgeパターンでは，機能のクラス階層(機能拡張が目的の継承)と実装のクラス階層(抽象クラスの実装が目的の継承)を分離する
- Strategyパターンでは，アルゴリズムの切り替えをインスタンスの切り替えにより実現する

第14回のまとめ(1/2)

- Compositeパターンでは，葉要素とそれ以外を同一視することで，階層構造を直感的に表現する
- Visitorパターンでは，複雑なデータ構造の中を走査しながら各要素に応じた処理を行う
- Decoratorパターンでは，様々な責任を動的に付与する
- Chain of Responsibility(CoR)パターンでは，要求を処理するためのオブジェクトを鎖状に連結し、
  複数のオブジェクトに処理の機会を与える

第14回まとめ(2/2)

- リファクタリングとは，既存のソフトウェアの外部的な振る舞いを変えることなく，設計や実装を改善する作業である
- リファクタリングは，リストラクチャリングの一種である
- リファクタリングでは，大きな設計変更を小さなコード変換の繰り返しで実現することで外部的振る舞いの保存を保証する
- リファクタリングにおいては，テストの実行結果で外部的振る舞いが保存されていることを確認するのが一般的である
- リファクタリングでは繰り返しテストを実施するため，テストの自動化が必須である
- 不吉な臭いとは，既存のコードにおいて問題を発見するための手がかりや潜在的な問題や欠陥に関する兆候である
- リファクタリングをするのではなく，最初から作り直した方が良い場合がある
- パターン指向リファクタリングでは，パターンがリファクタリングの到達点を提供する