オブジェクト指向技術 第10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

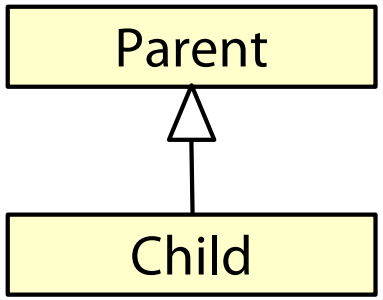
maru@cs.ritsumei.ac.jp

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

継承(inheritance)

- 親クラスのフィールドとメソッドを引き継ぐ仕組み
  - Javaでは親クラスをスーパークラス，子クラスをサブクラスという
  - クラス宣言にextendsキーワードを付加



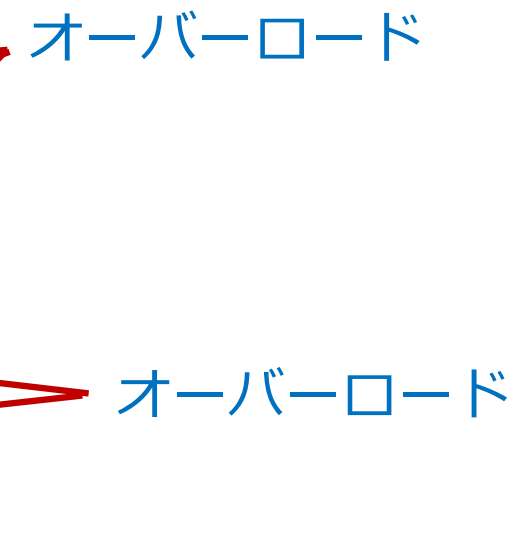

```
class Child extends Parent {
    …
}
```

- 親クラスには継承されていることを書かなくて良い
  - 親クラスのコードを見ても継承されているかは分からない
- メンバの継承は自動的に行われる
  - 子クラスで改めて記述する必要はない
  - コンストラクタは継承されない

オーバーロード(overload)

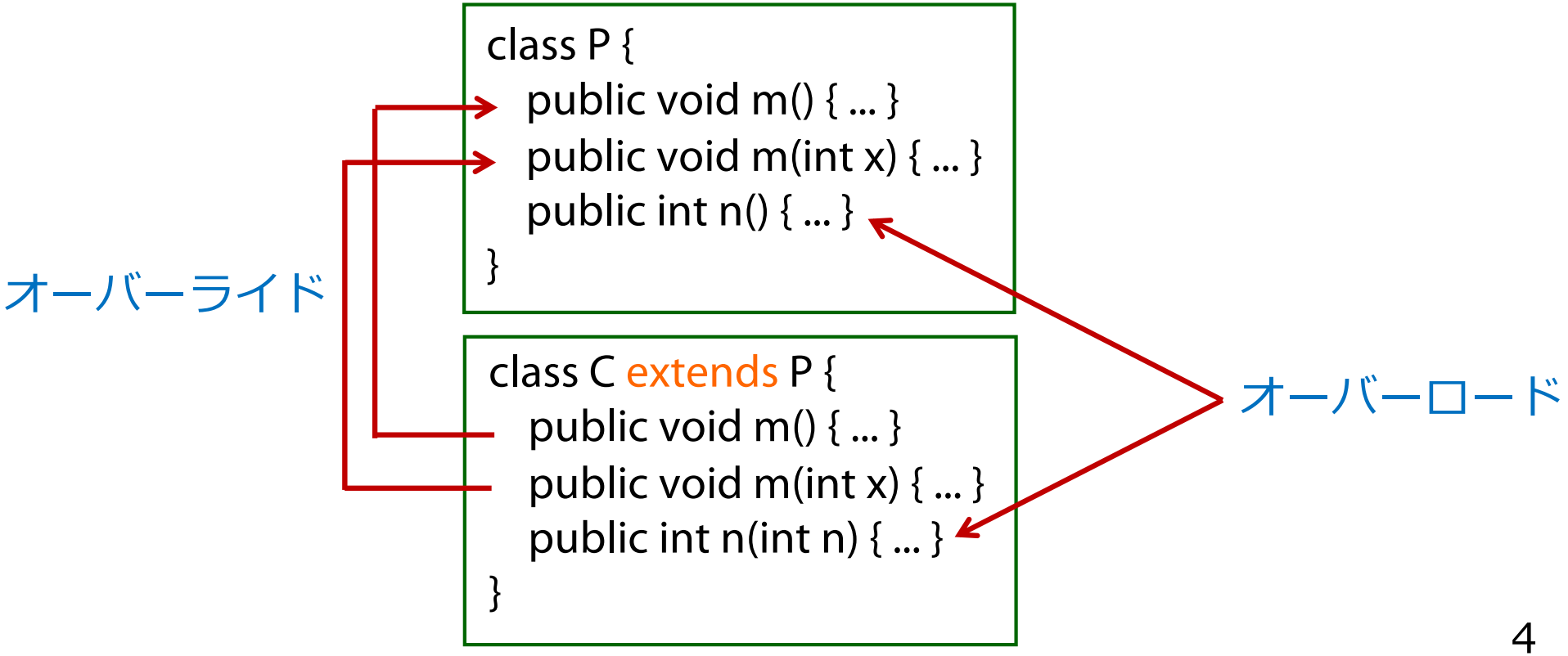
- 同じ名前のメソッドを複数宣言すること
  - 引数の数や型が互いに異なる
  - 名前と引数が同じで戻り値型のみ異なる
    メソッドは複数宣言できない

```
class C {
    public void m() { ... }
    public void m(int x) { ... }
    public void m(int x, int y) { ... }

    public int n(double x) { ... }
    public String n(String s) { ... }
}
```



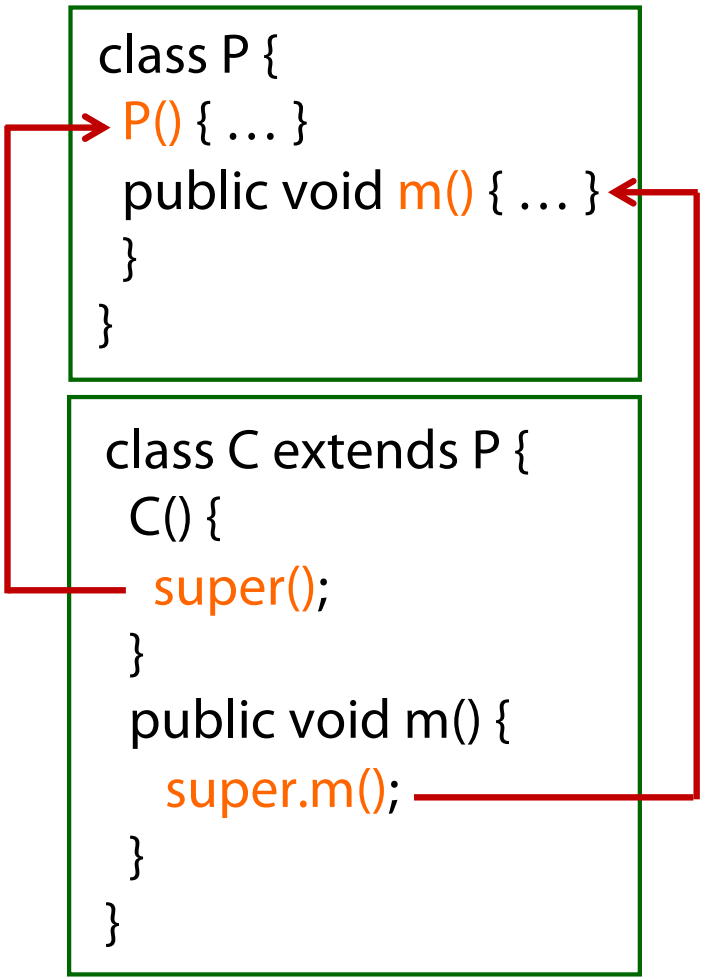
オーバーライド(override)

- 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
  - 子クラスのメソッドは親クラスのメソッドと同じかより広いアクセスを許可しなければならない

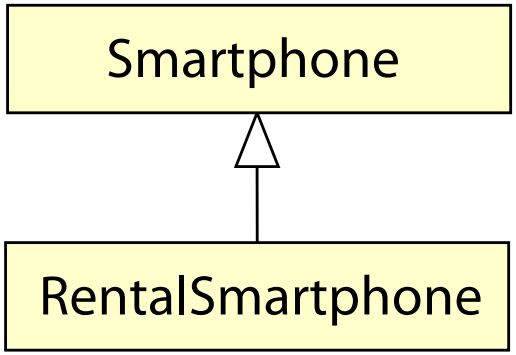


this と super

- 特定のオブジェクトやコンストラクタを指し示す
- this ※前回説明済み
  - 自クラスのオブジェクトを指す
- this() ※前回説明済み
  - 自クラスのコンストラクタを呼び出す
  - コンストラクタ先頭でのみ記述可能
- super
  - 親クラスのメンバを参照する際に利用
- super()
  - 親クラスのコンストラクタを呼び出す
  - コンストラクタ先頭でのみ記述可能



継承の記述例



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

インスタンスの生成とアクセス

```
public class Main2 {
    public static void main(String[] args) {
        RentalSmartphone phone = new RentalSmartphone("ABC", 2980, 12);
        System.out.println("Name = " + phone.getName());
        System.out.println("Price = " + phone.getPrice());
    }
}
```

RentalSmartphoneの
getPrice()の呼出し

Smartphoneの
getName()の呼出し

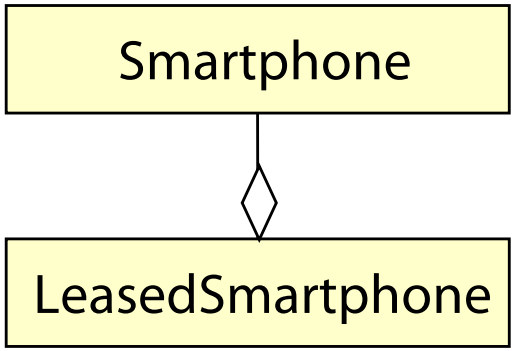
name: “ABC”
price: 2980
period: 12

実行結果

```
Name = ABC
Price = 35760
```

合成(集約)

- 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

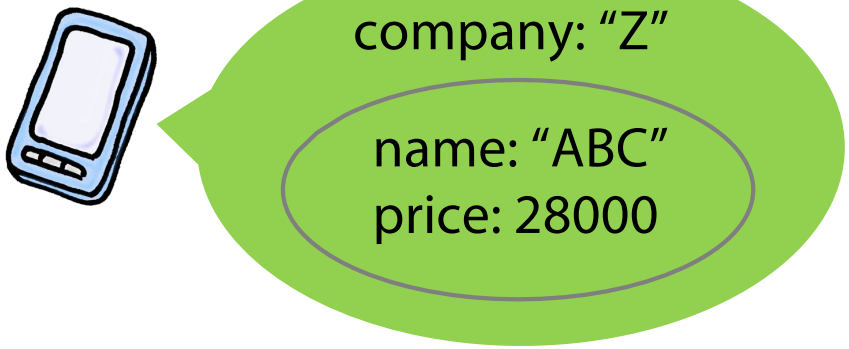
インスタンスの生成とアクセス

```
public class Main3 {
    public static void main(String[] args) {
        Smartphone base = new Smartphone("ABC", 35000);
        LeasedSmartphone phone = new LeasedSmartphone(base, "Z");
        System.out.println("Company = " + phone.getCompany());
        System.out.println("Price = " + phone.getPrice());
    }
}
```

インスタンスの参照値渡し

LeasedSmartphoneのgetPrice()の呼出し
→ SmartphoneのgetPrice()の呼出し

LeasedSmartphoneの
getCompany()の呼出し

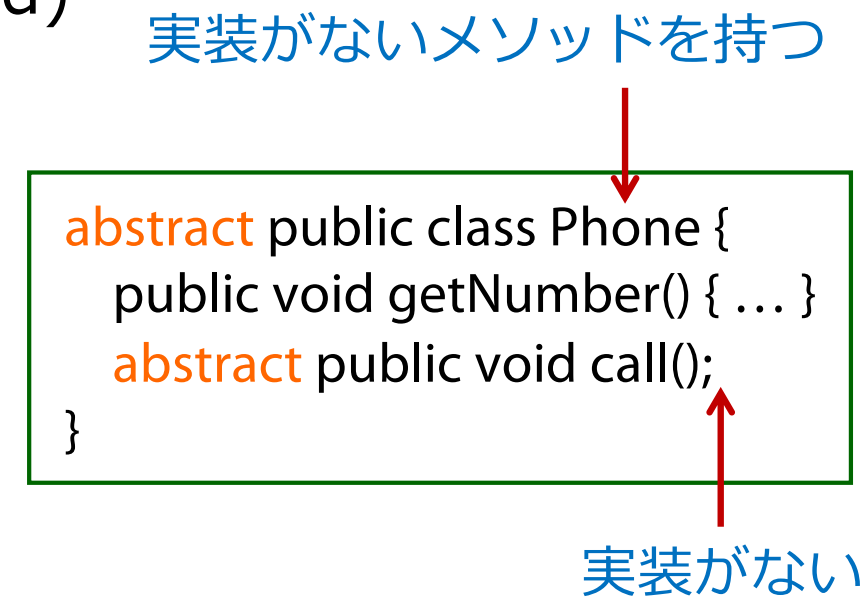


実行結果

```
Company = Z
Price = 28000
```

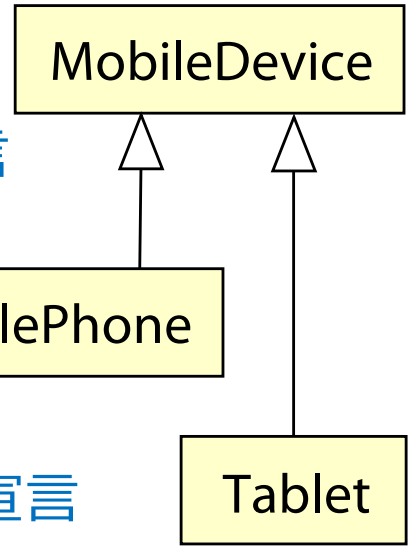
抽象クラス

- 抽象メソッド(abstract method)
  - 実装(中身)がないメソッド
  - シグネチャのみ定義
    - 実装は子孫クラスが決める
  - abstractキーワードを付ける



- 抽象クラス(abstract class)
  - 抽象メソッドを持つクラス
    - 抽象クラスやインタフェースを継承するクラスがすべての抽象メソッドを実装しない場合，そのクラスも抽象クラス
  - abstractキーワードを付ける
  - インスタンス生成できない
    - 実装されていないメソッドが存在するため

抽象クラスと継承の記述例




```
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"); }
}
```

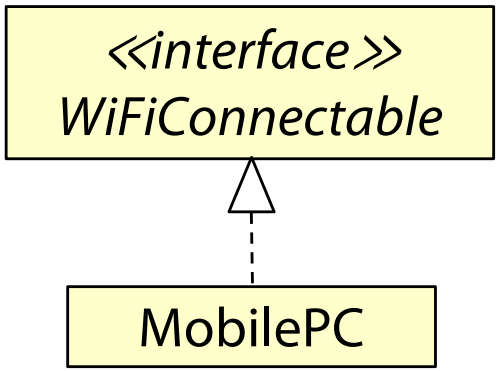
feature()を実装

インタフェースとインタフェース継承

- インタフェース(interface)
  - 抽象メソッドのみで構成
    - Java8以降はデフォルト実装と静的メソッドを定義可能
  - interfaceキーワードを利用
  - publicやabstractは省略可

```
interface WiFiConnectable {
  boolean connect();
}
```

- インタフェース継承
  - クラス宣言にimplementsキーワードを付ける




```
class MobilePC implements WiFiConnectable {
   public boolean connect() { … }
}
```

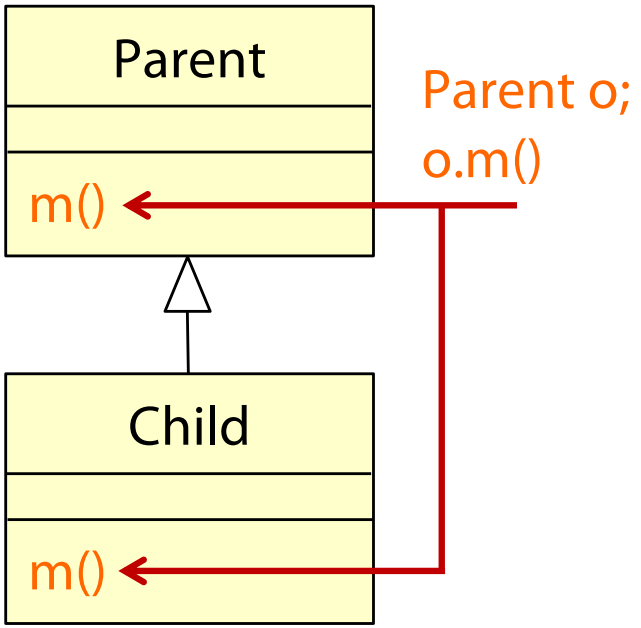
  - インタフェースには継承されていることを書かなくて良い
  - インタフェースを複数継承する場合は "," で区切る

final

- 変数やフィールド宣言時にfinalを付けると
  値の変更を禁止できる
  - 基本型
    - そのフィールドの値は変更できない(初期化が必須)
  - 参照型
    - その参照値を変更できない
      ＝その変数とインスタンスの結び付きは不変
      (インスタンス内部の属性値は変わることがある)
- クラス宣言時にfinalを付けると
  そのクラスの継承を禁止することができる
  - Stringクラスはfinalで定義されている
- メソッド宣言時にfinalを付けると
  そのメソッドのオーバーライドを禁止することができる

多態性と動的束縛

- 多態性(polymorphism)
  - 1つのインスタンスが複数のクラス(型)に属することを可能とする仕組み
  - Childのインスタンスは，Parentのインスタンスと見なせる
  - Parent型の変数oには，Parentのインスタンスと Childのインスタンスが代入できる



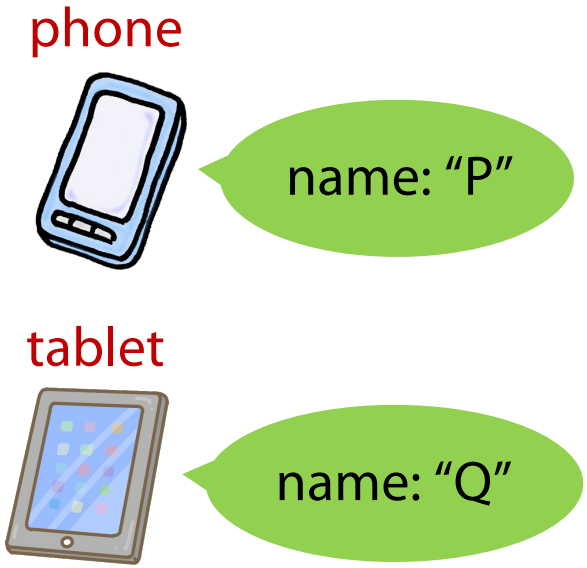
- 動的束縛(dynamic binding)
  - 操作を実行するインスタンスを実行時に受信側で決定する仕組み
  - o.m()と書くだけで，Parentのm()かChildのm()が選択される

多態性と動的束縛の記述例

```
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



実行結果

```
P is Small
Q is Big
P is Small
Q is Big
```

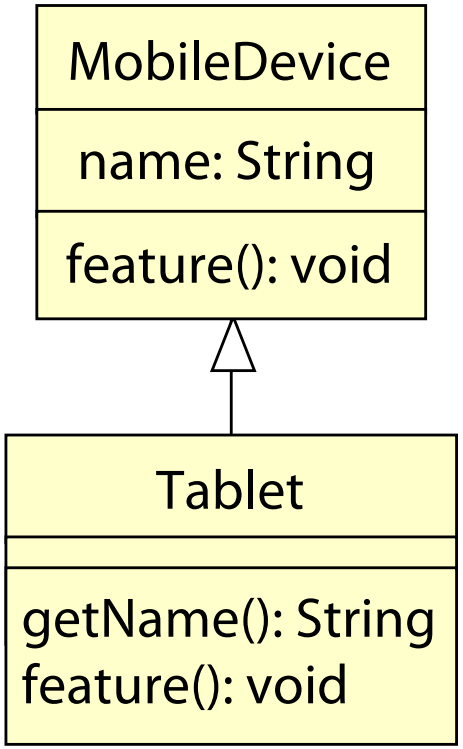
ダウンキャスト

```
public class Main4 {
    public static void main(String[] args) {
        MobileDevice device = new Tablet("Q");
        device.feature();

        // System.out.println(device.getName());

        Tablet tablet = (Tablet)device;
        System.out.println(tablet.getName());
    }
}
```

MobileDeviceにgetName()は
定義されていないので，
コンパイルエラー

ダウンキャスト

TabletのgetName()を呼出し



実行結果

```
Q is Big
Q
```

静的な型と動的な型

```
MobilePhone device1
  = new MobilePhone("P");

Tablet device2
  = new Tablet("Q");

MobileDevice device3
  = new Tablet(”R");
```

device1の静的な型はMobilePhone
device1の動的な型はMobilePhone
device2の静的な型はTablet
device2の動的な型はTablet
device3の静的な型はMobileDevice
device3の動的な型はTablet

```
MobileDevice device = new Tablet("Q");
device.feature();
```

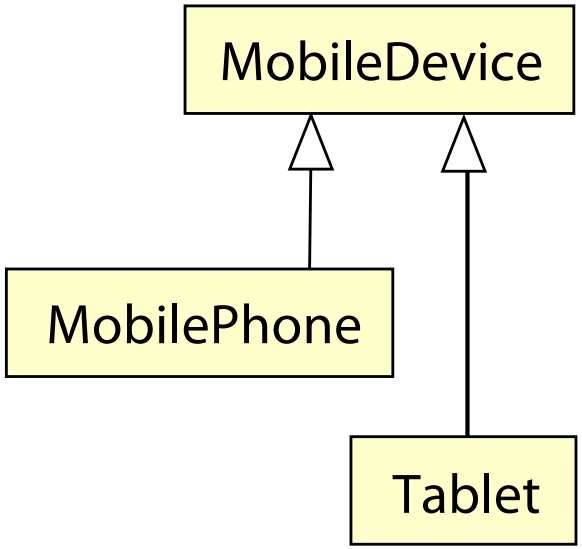
deviceの動的な型で
呼出先が決定

```
void method(MobileDevice device) { … }
void method(Table table) { … }

MobileDevice device = new Tablet("Q");
method(device);
```

deviceの静的な型で
呼出先が決定

instanceof 演算子

- インスタンスの(動的な)型を調べる




```
MobilePhone m = new MobilePhone("M");
System.out.println(m instanceof MobilePhone);
System.out.println(m instanceof MobileDevice);

Tablet t = new Tablet("T”);
// System.out.println(t instanceof MobilePhone);
System.out.println(t instanceof Tablet);

MobileDevice d = new MobilePhone("M");
System.out.println(d instanceof MobilePnone);
System.out.println(m instanceof MobileDevice);
```

練習問題(Exec2)

```
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 "Disc! " + super.getInfo();
    }
}
```

```
public class Exec2 {
    public static void main(String[] args) {
        PC pc1 = new PC(250);
        System.out.println(pc1.getInfo());  // (1)

        DiscountedPC pc2 = new DiscountedPC(250);
        System.out.println(pc2.getInfo());  // (2)

        PC pc3 = new DiscountedPC(250);
        System.out.println(pc3.getInfo());  // (3)

        PC pc4 = pc1;
        System.out.println(pc4.getInfo());  // (4)

        DiscountedPC pc5 = (DiscountedPC)pc3;
        System.out.println(pc5.getInfo());  // (5)
    }
}
```

```
$250
```

```
$200!
```

```
$200!
```

```
$250
```

```
$200!
```

練習問題(Exec3)

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

class UsedPC {
    private PC pc;
    public UsedPC(PC pc) { this.pc = pc; }
    void cutPrice() {
        pc.price = pc.price / 2;
    }
    String getInfo() {
        return pc.getInfo();
    }
}
```

```
public class Exec3 {
    public static void main(String[] args) {
        PC pc1 = new PC(250);
        System.out.println(pc1.getInfo());  // (1)

        UsedPC pc2 = new UsedPC(pc1);
        System.out.println(pc2.getInfo());  // (2)

        pc2.cutPrice();
        System.out.println(pc2.getInfo());  // (3)

        pc1.price = 80;
        System.out.println(pc1.getInfo());  // (4)
        System.out.println(pc2.getInfo());  // (5)
    }
}
```

$250

$250

$125

$80

$80

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

次回の講義の最初に小テストを行います