

発展プログラミング演習II

8.1 多態性とインタフェース

オブジェクト指向の基礎

- ・ クラス
- ・ インタフェース
- ・ メソッド
- ・ フィールド
- ・ 継承
- ・ アクセス制御
- ・ 多態性
- ・ 例外

多態性 polymorphism

- ・ 「複数の形態をとる」
- ・ 同じメッセージで、オブジェクトごとに異なる処理を行わせる
- ・ 具体的には
 - ・ 異なる種類の引数を受け取れるメソッド
 - ・ サブクラスでは異なる動作をするメソッド

多態性の利点

- ・ 同じような挙動をするメソッドの名前を統一できる
 - ・ 例： `System.out.print()`
- ・ 実装が違うサブクラス間で同じ挙動をするメソッドの名前を統一できる
 - ・ 例： `ArrayList`と`LinkedList`
- ・ サブクラスごとに詳細な挙動を変えることができる
 - ・ 例： `Calendar`と`GregorianCalendar`

Java言語での多態性の実現方法

- ・ オーバーロード
- ・ 継承とオーバーライド
- ・ 抽象クラスとインタフェース

オーバーロード overload

- ・ 同一クラス内で、メソッド名が同じでも引数（型、個数）が異なると別のメソッドとして扱われる
- ・ 返値だけ異なるのは不可
- ・ いずれが呼ばれるかは引数の型に応じてコンパイル時に決定される

教科書4.2節参照

発展プログラミング演習II

例題8.1

- 次のプログラムの実行結果を予想せよ

```
class OverloadExample {  
    private double division(double a, double b) {  
        return a / b;  
    }  
  
    private int division(int a, int b) {  
        return a / b;  
    }  
  
    private int division(int a, double b) {  
        return a / (int)b;  
    }  
  
    private int division(double a, int b) {  
        return (int)a / b;  
    }  
  
    public static void main(String[] args) {  
        OverloadExample app = new OverloadExample();  
        System.out.println(app.division(7, 2));  
        System.out.println(app.division(7.2, 2));  
        System.out.println(app.division(7, 1.8));  
        System.out.println(app.division(7.0, 2.0));  
    }  
}
```

発展プログラミング演習II

オーバーロードの利用例

- ・ 多様な引数で同様の動作をするメソッドを提供する
- ・ 例：PrintStream.print()
- ・ 引数の省略（デフォルト値）を許す

・ 例：

```
void method() {method(0, 0);}
void method(int a) {method(a, 0);}
void method(int a, int b) {
    .....
}
```

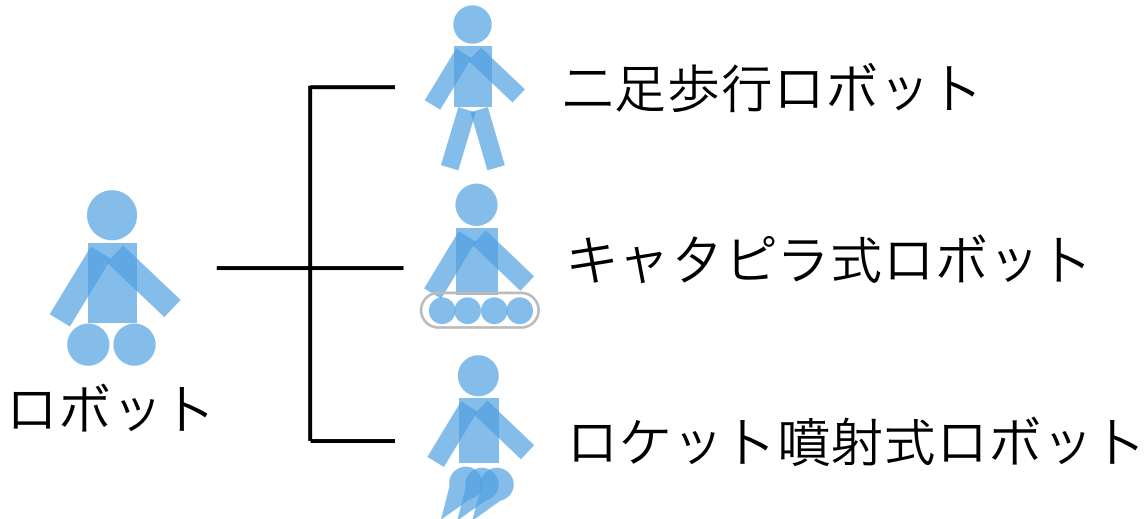

オーバーライド override

- ・ スーパークラスで定義されているメソッドを上書きする
- ・ オーバーライドする側と、される側は同じシグネチャでなければならない
- ・ スーパークラスのメソッドは遮蔽される

教科書6.2節参照

発展プログラミング演習II

オーバーライドの例



- ・ どれも「進む」命令で進むようにしたい
- ・ 移動方法はロボットによって異なる

コーディング例

```
class Robot {  
    ...  
    void move() { // 標準の動き方  
        .....  
    }  
}
```

```
class WalkingRobot extends Robot {  
    ...  
    void move() { // 足を動かして歩く  
        .....  
    }  
}
```

```
class CrawlingRobot extends Robot {  
    ...  
    void move() { // キャタピラで進む  
        .....  
    }  
}
```

```
class JetRobot extends Robot {  
    ...  
    void move() { // ジェット噴射で進む  
        .....  
    }  
}
```

is-a関係

発展プログラミング演習II

move()の呼び出し例

```
Robot[] robos = new Robot[3];
robos[0] = new WalkingRobot();
robos[1] = new CrawlingRobot();
robos[2] = new JetRobot();

for (int i = 0; i < 3; i++) {
    robos[i].move();
}
```

- ・ どのロボットも void move() で進む
- ・ Robotとそのサブクラスを使う側はロボットの違いを意識しなくてよい
- ・ 親クラスの型で参照できることに注意

例題8.2

- 1回の呼び出しで2だけ増加するようにcountメソッドをオーバーライドした、CounterクラスのサブクラスDoubleCounterを作成せよ

```
class Counter {  
    int num = 0;  
  
    void count() {num++;}  
    void setNum(int num) {this.num = num;}  
    int getNum() {return num;}  
}
```

※注意：サブクラスでフィールドにアクセスできるようにするためにはprivate指定をしてはいけない

例題8.3

- 次のプログラムの実行結果を予想せよ

```
class CounterTest {  
    static void callCount(Counter c) {  
        c.count();  
    }  
  
    public static void main(String[] args) {  
        Counter c = new Counter();  
        DoubleCounter dc = new DoubleCounter();  
  
        c.setNum(1);  
        callCount(c);  
        dc.setNum(2);  
        callCount(dc);  
        System.out.println("c = " + c.getNum());  
        System.out.println("dc = " + dc.getNum());  
    }  
}
```

オーバーライドされたメソッドの呼び出し

- ・ オーバーライドした親クラスのメソッドを明示的に呼び出すにはsuperを使う

例：


```
class DoubleCounter extends Counter {  
    void count() {num += 2;}  
    void superCount() {super.count();}  
}
```

- ・ superは子クラスからのみ使用可能
(外部からはsuperは参照できないので)

superを使った処理の例

```
class Robot {  
    void attack() {  
        // 攻撃処理  
        .....  
    }  
    .....  
}
```

```
class DrillRobot extends Robot {  
    void attack() {  
        super.attack(); // 親クラスの攻撃処理  
        // 以下、このクラスの攻撃処理  
        spinDrill();  
        .....  
    }  
    .....  
}
```




- ・ 親クラスの処理に追加する

super()

```
class Robot {  
    Robot(String name)  
    {  
        .....  
    }  
  
    .....  
}
```

```
class DrillRobot extends Robot {  
    void DrillRobot(String name) {  
        super(name);  
        // 以下このクラスの初期化处理  
        .....  
    }  
  
    .....  
}
```

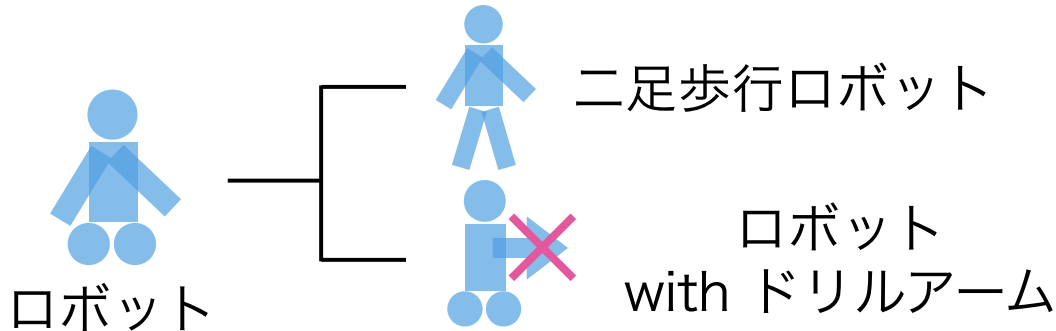


- ・ 親クラスのコンストラクタの呼び出し
- ・ 子クラスのコンストラクタの先頭で必ず呼び出さなければならない
- ・ 引数が無い場合は省略可能

オーバーライドの禁止

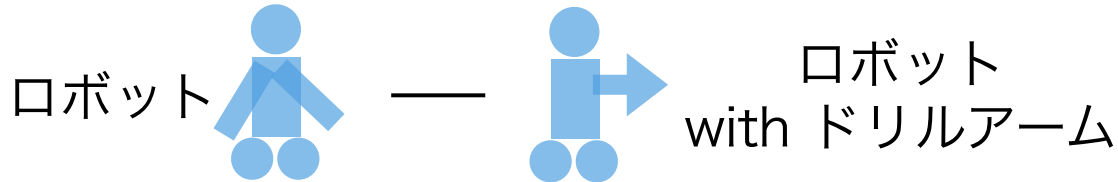
- ・ final 修飾子の付いているメソッドはオーバーライドできない

オーバーライドを禁止したい例



- ・ 腕は勝手に変更して欲しくない場合
→ 腕関係のメソッドをfinalにしてしまう

オーバーライドを禁止したい例（続き）



- ・ ロボットクラスに次のメソッドがあるとする
 - ・ 腕を動かすメソッド `raiseArms`
 - ・ 移動するメソッド `move`
- ・ 移動するときには腕を前後に動かす
- ・ ドリルアーム付きロボットは腕を動かすとドリルを回転させたい

コーディング例

```
class Robot {  
    ...  
    void raiseArms() {  
        .....  
    }  
    void move() { // デフォルトの動き  
        .....  
        raiseArms();  
    }  
}
```



```
class DrillRobot extends Robot {  
    ...  
    void raiseArms() {  
        .....  
        spinDrills(); // ドリルを回転させる  
    }  
}
```

- move()の中でraiseArms()を呼んでいる
- DrillRobotのmove()を呼ぶとドリルも回転する

対策例

```
class Robot {  
    ...  
    final void raiseArms() {  
        .....  
    }  
    void move() { // デフォルトの動き  
        .....  
        raiseArms();  
    }  
}
```



```
class DrillRobot extends Robot {  
    ...  
    void drillAttack() {  
        raiseArms();  
        spinDrills(); // ドリルを回転させる  
    }  
}
```

- raiseArms()のオーバーライドを禁止する
- DrillRobotでは腕を動かすと同時にドリルを回転させるメソッドを別に用意する
- そもそもクラス設計が適切か見直すべき

オーバーライドとアクセスレベル

- ・ オーバーライドするメソッドのアクセスレベルは、オーバーライドされる側と同等かそれよりもゆるくなくてはならない
- ・ public > protected > (指定無し)
- ・ privateは??? (考えてみよう)

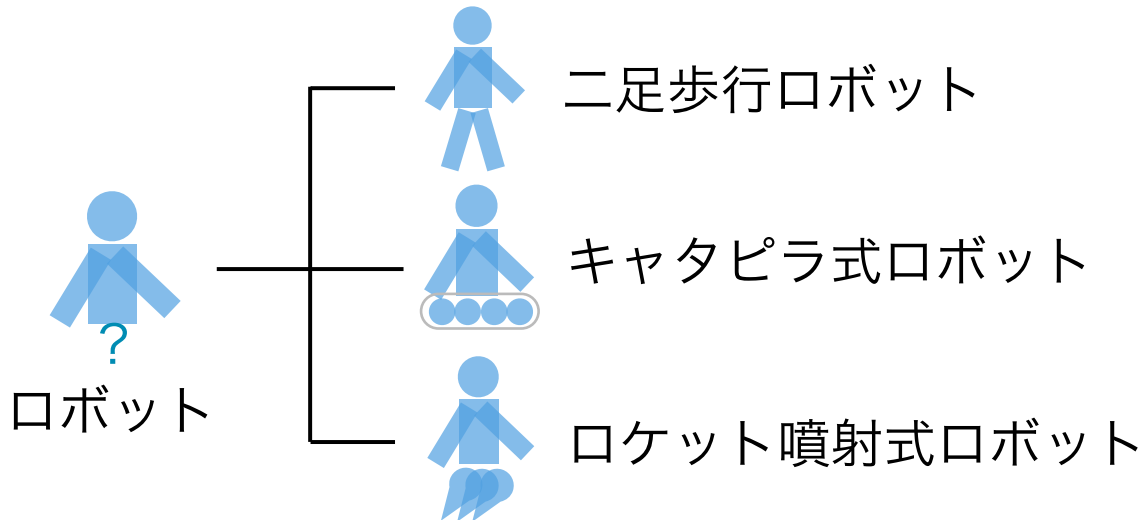
抽象クラス

- ・ 抽象メソッドを持つクラス
- ・ 抽象メソッド：
 - ・ メソッド名、引数と返値の型だけ定義
 - ・ 具体的な実装は持たない
- ・ 抽象クラス、抽象メソッドは修飾子abstractで宣言する
- ・ インスタンスは作成できない

教科書7.1節参照

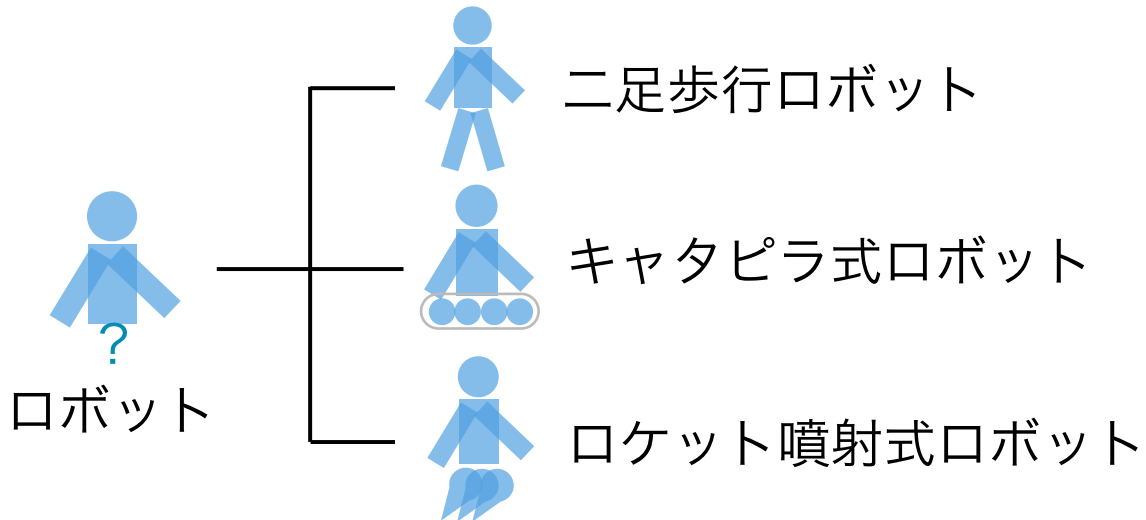
発展プログラミング演習II

抽象クラスが必要な例



- ・ ロボットクラスでは「進む」動きを決めておきたくないが、サブクラスでは必ず「進む」動きを用意させたい

抽象クラスが必要な例



- ・ 「進む」ためのmoveメソッドはサブクラスで必ず実装させたいが、ロボットクラスでは実装したくない

コーディング例

```
abstract class AbstractRobot {  
    ...  
    abstract void move();  
}
```

```
class WalkingRobot extends AbstractRobot {  
    ...  
    void move() { // 足を動かして歩く  
        .....  
    }  
}
```

```
class CrawlingRobot extends AbstractRobot {  
    ...  
    void move() { // キタピラで進む  
        .....  
    }  
}
```

```
class JetRobot extends AbstractRobot {  
    ...  
    void move() { // ジェット噴射で進む  
        .....  
    }  
}
```

発展プログラミング演習II

例題8.4(1 / 7)

- 2つのクラスPachiLightとMachiMaxを、上位クラスとなる抽象クラスPachiを加えて再設計せよ

例題8.4(2/7)

```
class Pachilight {
    private static final double NYUUSHOU = 0.15;
    private static final double OOATARI = 1.0 / 100.0;
    private static final double RENCHAN = 1.0 / 2.5;
    private static final int    OOATARI_DEDAMA = 400;

    private int    tama = 0;

    void setTama(int tama) {this.tama = tama;}
    int getTama() {return tama;}

    void play() {
        if (tama <= 0) return;
        tama--;
        if (Math.random() < NYUUSHOU) chusen();
    }

    void chusen() {
        if (Math.random() < OOATARI) {
            ootari();
        }
    }

    void ootari() {
        System.out.println("大当たり");
        tama += OOATARI_DEDAMA;
        if (Math.random() < RENCHAN) {
            System.out.print("連荘!! ");
            ootari();
        }
    }
}
```

例題8.4(3/7)

```
class Pachimax {
    private static final double NYUUSHOU = 0.15;
    private static final double OOATARI_NORMAL = 1.0 / 350.0;
    private static final double OOATARI_KAKUHEN = 1.0 / 35.0;
    private static final double KAKUHEN = 0.7;
    private static final int OOATARI_DEDAMA = 1400;

    private int tama = 0;
    private boolean kakuhen = false;

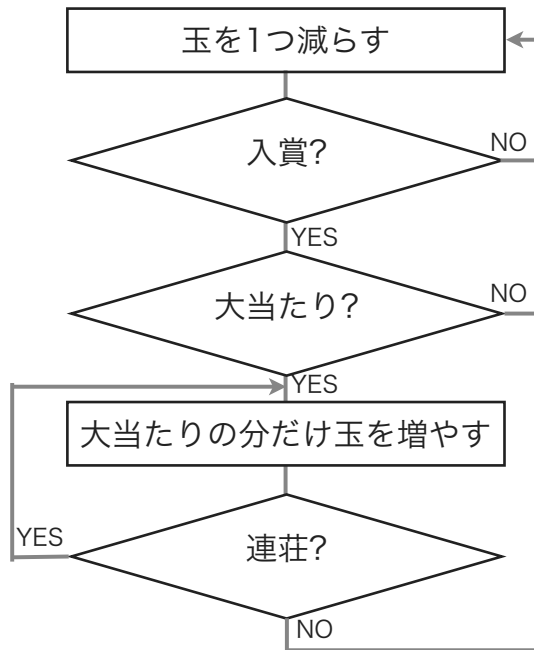
    void setTama(int tama) {this.tama = tama;}
    int getTama() {return tama;}

    void play() {
        if (tama <= 0) return;
        tama--;
        if (Math.random() < NYUUSHOU) chusen();
    }

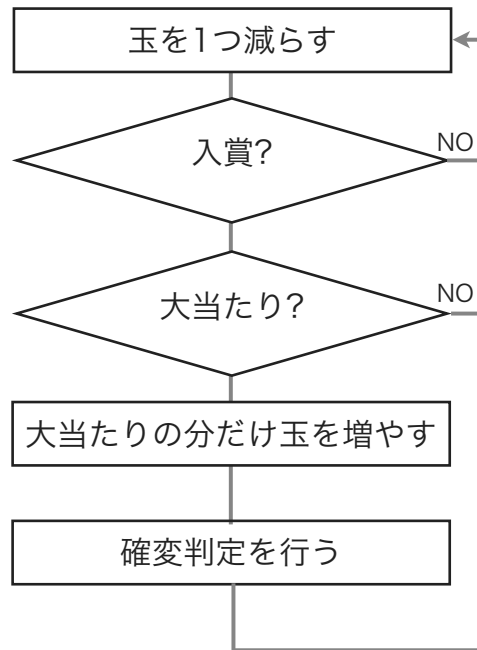
    void chusen() {
        double kakuritsu = kakuhen ?
            OOATARI_KAKUHEN : OOATARI_NORMAL;
        if (Math.random() < kakuritsu) {
            ootari();
        }
    }

    void ootari() {
        System.out.println("大当たり");
        tama += OOATARI_DEDAMA;
        if (Math.random() < KAKUHEN) {
            System.out.println("  確変!!");
            kakuhen = true;
        } else {
            System.out.println("  確変終了");
            kakuhen = false;
        }
    }
}
```

例題8.4(4/7) 処理の流れ



PachiLight



PachiMax 発展プログラミング演習II

例題8.4(5/7) 考え方

- PachiLightとPachiMaxで共通の部分をPachiに持たせる
- private指定してあると子クラスからもアクセスできないことに注意
- PachiLightとPachiMaxで異なる部分で、共に実装している必要のあるメソッドを抽象メソッドとして定義する
- PachiLightとPachiMaxを、Pachiを継承するように修正する

例題8.4(5/7) Pachiクラス

- ・ 現在の玉数を保持するフィールド
およびアクセッサメソッド
- ・ 1発分の当たり判定を行うplayメソッド
 - ・ 所定の確率で大当たり判定を行う
- ・ 以下の2つは継承したクラスで実装する
 - ・ 大当たり判定を行うchusenメソッド
 - ・ 大当たり時の処理を行うooatariメソッド

例題8.4 (7/7)

- 実装したプログラムの動作を、PlayPachi.javaで確認せよ

```
class PlayPachi {
    static void simulate(Pachi pachi, int num) {
        // 引数の回数だけpachiをプレイする
        for (int i = 0; i < num; i++) {
            pachi.play();
        }
    }

    public static void main(String[] args) {
        Pachi pachi1 = new PachiMax();
        Pachi pachi2 = new PachiLight();

        pachi1.setTama(5000); // 最初は5000発
        simulate(pachi1, 10000); // 10000回プレイしてみる
        System.out.println("pachi1 = " + pachi1.getTama());

        pachi2.setTama(5000);
        simulate(pachi2, 10000);
        System.out.println("pachi2 = " + pachi2.getTama());
    }
}
```

課題8.1

- ・ 例題8.4に加えて次の仕様のPachiMiddleを、Pachiクラスのサブクラスとして実装せよ
 - ・ 大当たり確率 1/200
 - ・ 大当たり時の出玉 800個
 - ・ 大当たり時直後、玉を消費せずに75回分大当たり抽選を行う（時短）
 - ・ 時短中に大当たりした場合、残り回数処理は任意とする
- ・ PlayPachi.javaにPachiMilddleをテストするコードを追加して動作を確認せよ

抽象クラスの利点と注意点

- ・ 抽象クラスはインスタンス化できない
 - ・ 複数種類のサブクラスをまとめて扱える
 - ・ サブクラスで必ず実装すべきメソッドを定義できる
 - ・ 不完全なためインスタンス化されると困るクラスは抽象化するとよい
- ・ インスタンス化するサブクラスでは、**すべての**抽象メソッドは具体的に実装される必要がある
- ・ 抽象メソッドのままだとサブクラスも抽象クラスになる

抽象クラスとアダプタ*

- ・ 抽象メソッドを沢山持つ抽象クラスのサブクラスは実装が大変
- ・ 実装が必要なメソッドを空の定義にしておく
- ・ 必要なメソッドのみサブクラスでオーバーライドする
- ・ インスタンス化の防止はできない
- ・ 例：KeyListenerに対するKeyAdapter

インタフェース interface

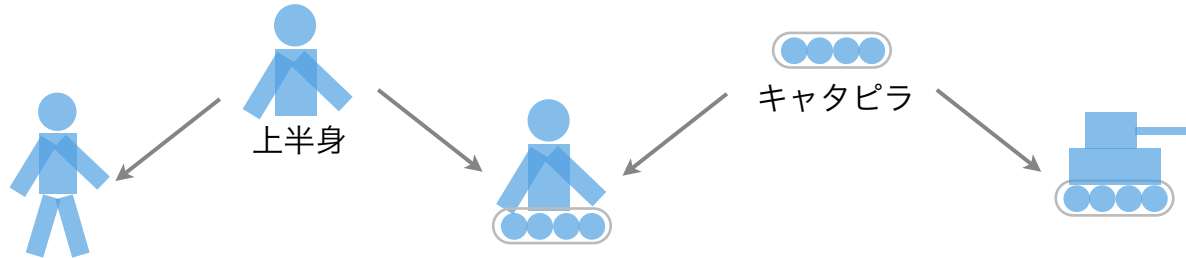
- ・ 抽象メソッドしか持たないクラスのようなもの
 - ・ クラスではない。インタフェース。
 - ・ コンストラクタがない
=オブジェクトは作れない
 - ・ メソッドは自動的にabstract public扱い
 - ・ フィールドは定数のみ持つことが可能
(自動的にpublic static final扱い)

教科書7.2節参照

発展プログラミング演習II

多重継承

- ・ 多重継承＝複数の上位クラスから継承



- ・ Java言語では多重継承はサポートされていない
- ・ インタフェースを代わりに使うことができる
- ・ 複数のインタフェースを実装可能

インタフェースの定義例

```
interface RobotArms {  
    int MAX_DEGREE = 90;  
    int MIN_DEGREE = -90;  
  
    void raiseArms(int degree);  
    void downArms(int degree);  
    void upArms(int degree);  
}
```

- それぞれ修飾子が省略されていることに注意
(何が省略されているか考えてみよう)

インタフェースの実装例

```
interface RobotArms {  
    int MAX_DEGREE = 90;  
    int MIN_DEGREE = -90;  
  
    void raiseArms(int degree);  
    void downArms(int degree);  
    void upArms(int degree);  
}
```

```
interface Drill {  
    void spinDrill();  
    void stopDrill();  
}
```

```
class DrillRobot extends Robot implements  
RobotArms, Drill {  
    .....  
    public void raiseArms(int degree) {.....}  
    public void downArms(int degree) {.....}  
    public void upArms(int degree) {.....}  
    public void spinDrill() {.....}  
    public void stopDrill() {.....}  
    .....  
}
```

- ・ 複数のインタフェースを実装可能
- ・ すべての抽象メソッドを実装する必要あり
- ・ 修飾子に注意
- ・ 上位クラスも継承可能（ただし1つのみ）

発展プログラミング演習II

インタフェースでの参照

- ・ インタフェースを実装したクラスインスタンスはインタフェース型の変数で参照可能

```
class DrillRobot extends Robot implements
RobotArms, Drill {
    .....
    public void raiseArms(int degree) {.....}
    public void downArms(int degree) {.....}
    public void upArms(int degree) {.....}
    public void spinDrill() {.....}
    public void stopDrill() {.....}
    .....
}
```

```
{
    .....
    DrillRobot robot = new DrillRobot();
    robot.raiseArms(45);
    drillAttack(robot);
    .....
}

void drillAttack(Drill drill) {
    drill.spinDrill();
    .....
    drill.stopDrill();
}
```