プログラミング基礎

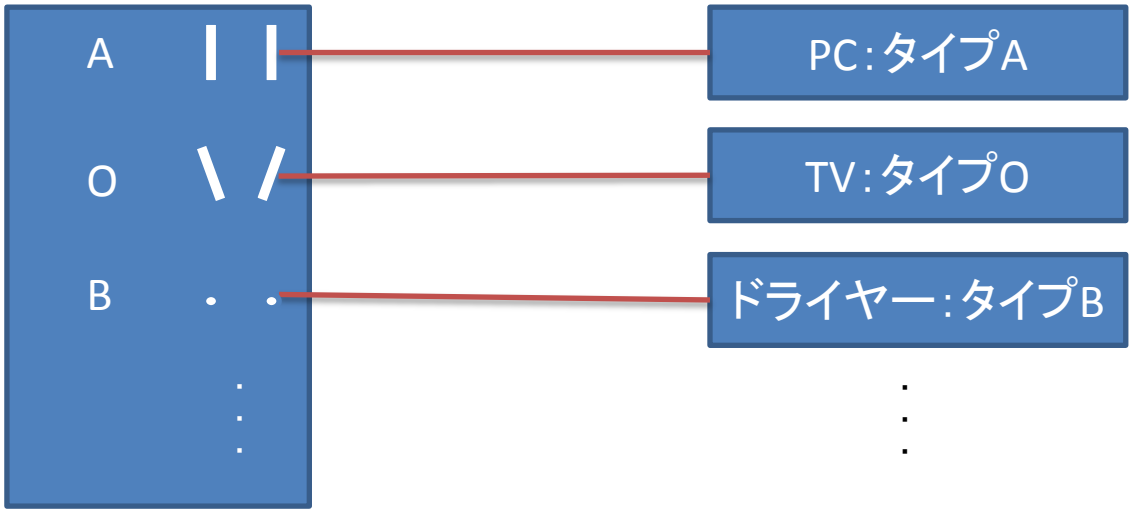
第11回
継承と包含の使い分け
／インタフェース

プログラムを書く前に

- 「コンセントのプラグはタイプAを使う」
  と決まっていなかったら，世の中どうなるか

プログラムを書く前に

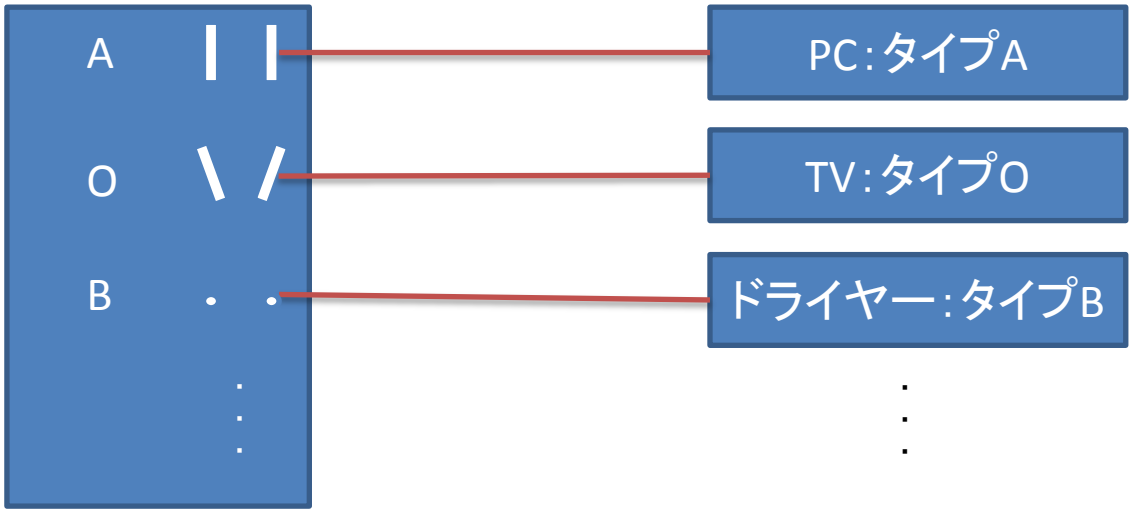
- 「コンセントのプラグはタイプAを使う」
  と決まっていなかったら，世の中どうなるか



答え：プラグ形状に応じたコンセントの口がたくさん増える

プログラムを書く前に

- 「コンセントのプラグはタイプAを使う」
  と決まっていなかったら，世の中どうなるか



このままでは何が問題か？
⇒コンセントは，各家電製品のプラグ形状
をすべて把握していなければならない

プログラムを書く前に

- 多少強引だがプログラムに直してみる

```
class Socket
{
   void connect(PC pc){pc.plugA();}
   void connect(TV tv){tv.plugO();}
   void connect(Drier d){d.plugB();}
}
```

何が問題か？
⇒Socketクラスは家電製品のプラグ形状をすべて把握しなければならない
⇒家電製品の種類が増えるごとにconnectメソッドを書き加えなければいけない

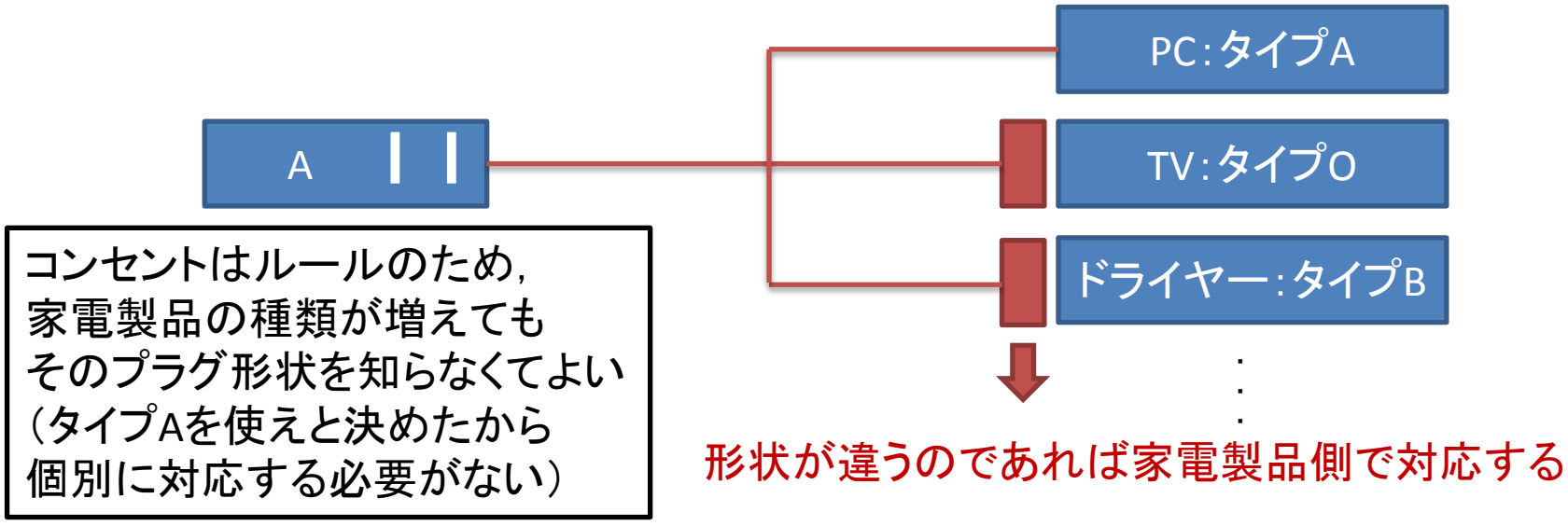
```
class PC
{
   void plugA(){println("起動する");}
}

class TV
{
   void plugO(){println("点ける");}
}

class Drier
{
   void plugB(){println("風を出す");}
}
```

＊「コンセント」は和製英語，Outlet<米>またはSocket<英>

プログラムを書く前に

- 「コンセントのプラグはタイプAを使う」というルールを決める

⇒コンセントは必ずタイプAを使うからあとは家電製品側でなんとかしろ

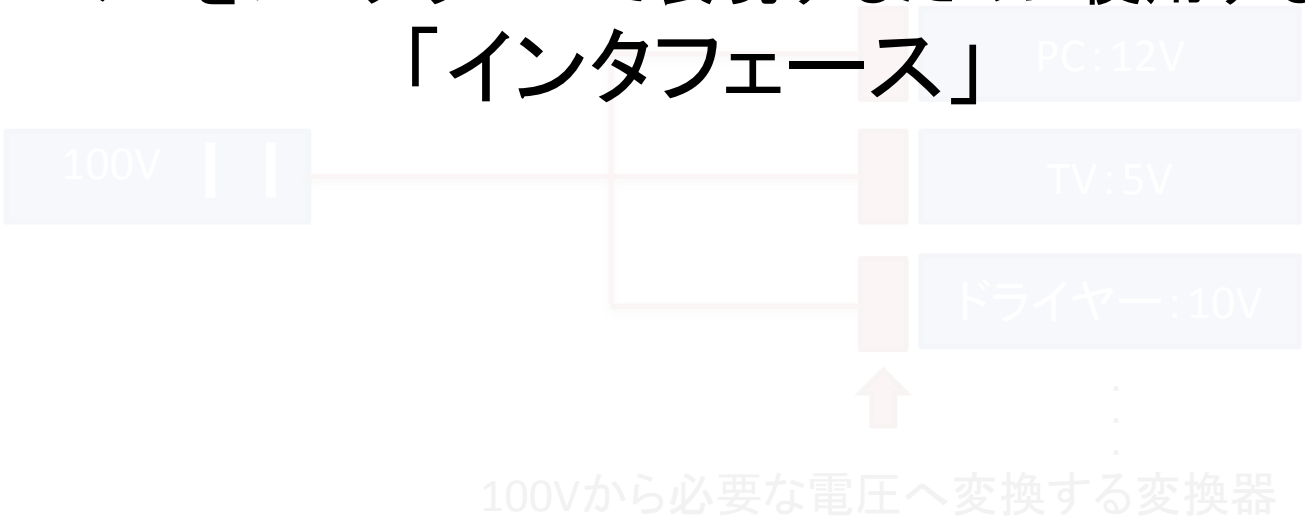


プログラムを書く前に

- 「コンセントのプラグはタイプAを使う」というルールを決める

⇒コンセントは必ずAC100Vを供給するからあとは家電製品側でなんとかしろ

”ルール”をプログラム上で表現するために使用するのが
「インタフェース」



インタフェースの書き方

「コンセントのプラグはタイプAを使う」
⇒「コンセントにつなげる家電製品はタイプA のプラグが接続できる」

```
interface TypeAPlugable
{
    public void plugA();
}
```

「このインタフェースを実装したクラスは～ができる」
という意味.
インタフェース名は～ableが多い

「～ができるためには，このメソッドを実装すること」という意味で，戻り値の型，メソッド名，引数を書く
（{～;}は書かない）

このインタフェースでは，
「このインタフェースを実装したクラスはタイプAのプラグを接続できる」
ということ

タイプAのプラグを接続できるのであれば，
このインタフェースを実装するクラス内で
必ずplugAメソッドを実装しろ，ということになる

インタフェースの使い方

- 多少強引だがプログラムに直してみる

```
class Socket
{
    void connect(TypeAPlugable p){p.plugA();}
}
```

```
interface TypeAPlugable
{
    public void plugA();
}
```

TypeAPlugableインタフェースを実装したクラス

```
class PC implements TypeAPlugable
{
    public void plugA(){println("起動する");}
}
```

実装されたplugAメソッド

```
class TV implements TypeAPlugable
{
    public void plugA(){plugO();}
    void plugO(){println("点ける");}
}
```

```
class Drier implements TypeAPlugable
{
    public void plugA(){plugB();}
    void plugB(){println("風を出す");}
}
```

プログラムを書く前に

- 多少強引だがプログラムに直してみる

```
class Socket
{
   void connect(TypeAPlugable p){p.plugA();}
}
```

```
interface TypeAPlugable
{
   public void plugA();
}
```

```
class PC implements TypeAPlugable
{
   public void plugA(){println("起動する");}
}
```

```
class TV implements TypeAPlugable
{
   public void plugA(){plugO();}
   void plugO(){println("点ける");}
}
```

```
class Drier implements TypeAPlugable
{
   public void plugA(){plugB();}
   void plugB(){println("風を出す");}
}
```

```
class Toaster implements TypeAPlugable
{
   public void PlugA(){PlugBF();}
   void PlugBF{println("焼く");}
}
```

TypeAPlugableインタフェースが
ルールの役割を果たす．そのため，将来
家電製品の種類が増えたとしても，
Socketクラスはそのプラグ形状を気にしなくてよい
⇒Socketクラスにプログラムを
　書き加える必要がない

演習

- 前述のプログラムを動かしてみよ：プログラム名：SocketTest

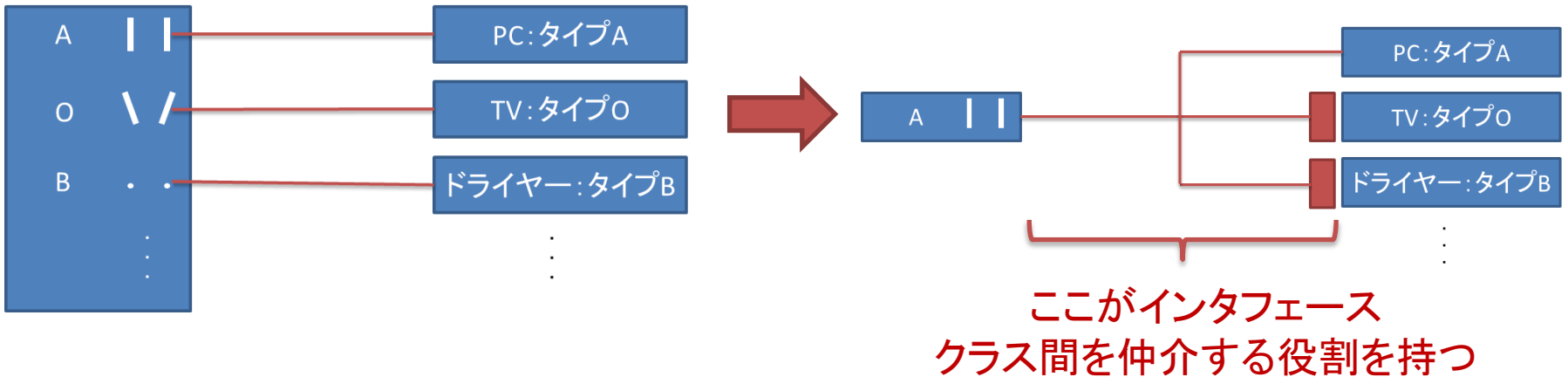
```
//Socketクラス，TypeAPlugableインタフェース，
// TypeAPlugableインタフェースを実装したPC，TV，Drierクラス

public class SocketTest
{
   static public void main(String[] args)
   {
      Socket s = new Socket();
      PC pc = new PC();
      TV tv = new TV();
      Drier d = new Drier();
      s.connect(pc);
      s.connect(tv);
      s.connect(d);
   }
}
```

実行例

インタフェースの役割

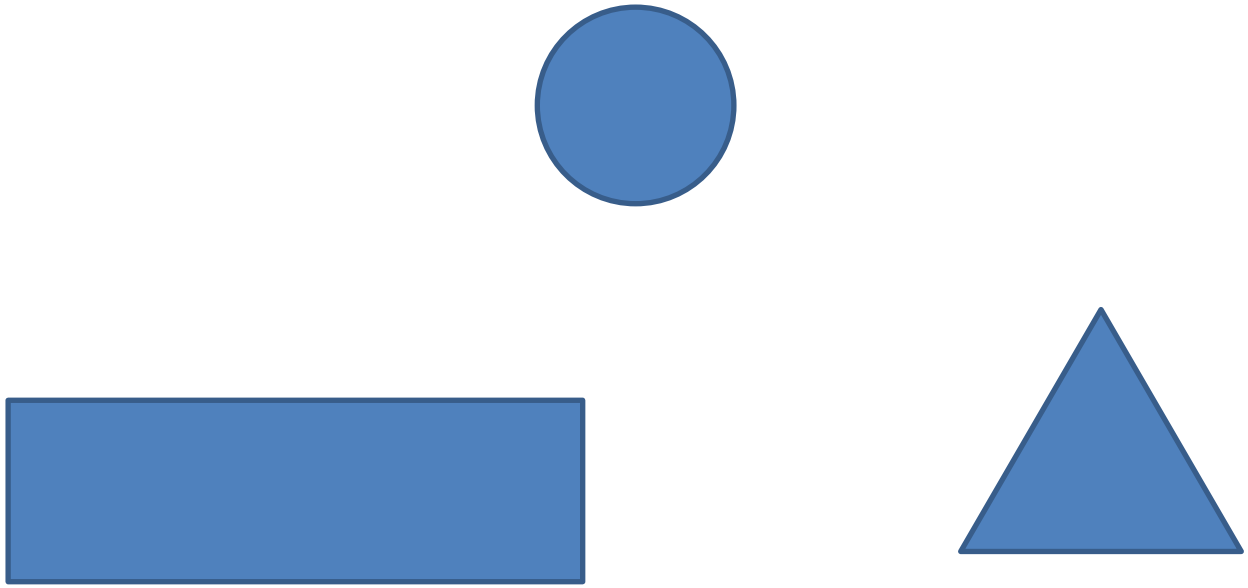
- ルールを決めることで
  ＝インタフェースを利用することで
  クラス同士のつながり（⇒依存度）を減らす

- 依存度を減らすことで
  - コードの量を抑える
  - 一度完成したクラスを後から修正しなくてよくなる



インタフェース活用例

- 「さまざまな図形の合計面積を求めたい」



インタフェース活用例

- 「さまざまな図形の合計面積を求めたい」

インタフェースを用いない場合

```
static int shapesArea(Circle[] cs, Triangle[] ts, Square[] rs)
{
   int area = 0;
   for(int i = 0; i < cs.length; i++)
   {
      area += cs[i].area();
   }
   for(int i = 0; i < ts.length; i++)
   {
      area += ts[i].area();
   }
   for(int i = 0; i < rs.length; i++)
   {
      area += rs[i].area();
   }
   return area;
}
```

```
class Circle
{
   int r;
   Circle(int rr){r=rr;}
   int area(){return r * r * 3;}
}

class Triangle
{
   int b; int h;
   Triangle(int bb, int hh){b=bb; h = hh;}
   int area(){return (b * h) / 2;}
}
class Square
{
   int w; int h;
   Square(int ww, int hh){w=ww; h = hh;}
   int area(){return w * h;}
}
```

インタフェース活用例

- 「さまざまな図形の合計面積を求めたい」

インタフェースを用いない場合

```
static int shapesArea(Circle[] cs, Triangle[] ts, Square[] rs)
{
   int area = 0;
   for(int i = 0; i < cs.length; i++)
   {
      area += cs[i].area();
   }
   for(int i = 0; i < ts.length; i++)
   {
      area += ts[i].area();
   }
   for(int i = 0; i < rs.length; i++)
   {
      area += rs[i].area();
   }
   return area;
}
```

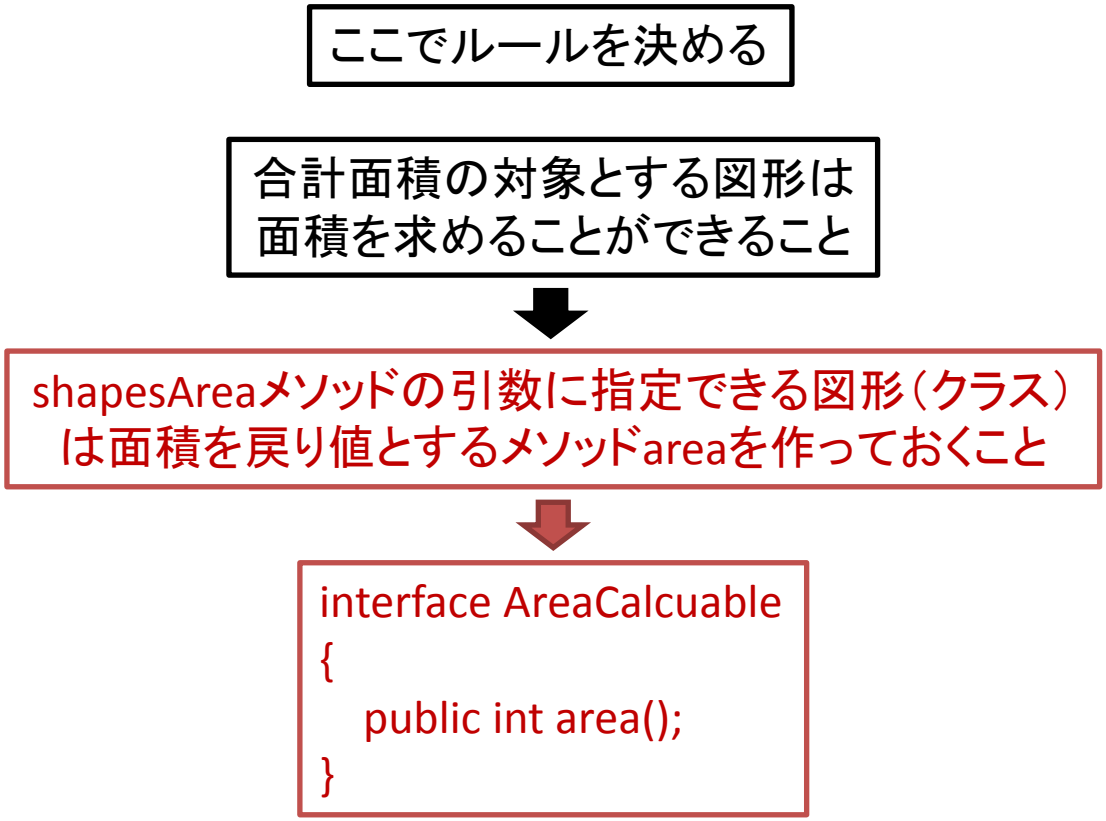
メソッドが
・図形の種類
・各図形の数
を把握しなければならない

図形の種類が増えたら
書き直し

```
class Circle
{
   int r;
   Circle(int rr){r=rr;}
   int area(){return r * r * 3;}
}

class Triangle
{
   int b; int h;
   Triangle(int bb, int hh){b=bb; h = hh;}
   int area(){return (b * h) / 2;}
}
class Square
{
   int w; int h;
   Square(int ww, int hh){w=ww; h = hh;}
   int area(){return w * h;}
}
```

インタフェース活用例

- そこでインタフェースを用いる



ルールをプログラムで示した，インタフェースの完成

インタフェース活用例

- 「さまざまな図形の合計面積を求めたい」

インタフェースを用いた場合

```
interface AreaCalcuable
{
   public int area();
}
```

```
static int shapesArea(AreaCalcuable[] acs)
{
   int area = 0;
   for(int i = 0; i < acs.length; i++)
   {
      area += acs[i].area();
   }
   return area;
}
```

```
class Circle implements AreaCalcuable
{
   int r;
   Circle(int rr){r=rr;}
   public int area(){return r * r * 3;}
}

class Triangle implements AreaCalcuable
{
   int b; int h;
   Triangle(int bb, int hh){b=bb; h = hh;}
   public int area(){return (b * h) / 2;}
}
class Square implements AreaCalcuable
{
   int w; int h;
   Square(int ww, int hh){w=ww; h = hh;}
   public int area(){return w * h;}
}
```

インタフェース活用例

- 「さまざまな図形の合計面積を求めたい」

インタフェースを用いた場合

```
interface AreaCalcuable
{
   public int area();
}
```

```
static int shapesArea(AreaCalcuable[] acs)
{
   int area = 0;
   for(int i = 0; i < acs.length; i++)
   {
      area += acs[i].area();
   }
   return area;
}
```

引数に指定された
AreaCalcuableインタフェース
の配列に，実際どんなクラスの
インスタンスがあるのか
メソッドは知らない

しかし，とにかく
いずれのインスタンスも
面積を求めることができる
のは分かっているので，
合計面積が計算できる

```
class Circle implements AreaCalcuable
{
   int r;
   Circle(int rr){r=rr;}
   public int area(){return r * r * 3;}
}

class Triangle implements AreaCalcuable
{
   int b; int h;
   Triangle(int bb, int hh){b=bb; h = hh;}
   public int area(){return (b * h) / 2;}
}
class Square implements AreaCalcuable
{
   int w; int h;
   Square(int ww, int hh){w=ww; h = hh;}
   public int area(){return w * h;}
}
```

```
interface AreaCalcuable
{
   public int area();
}


class Circle implements AreaCalcuable
{
   int r;
   Circle(int rr){r=rr;}
   public int area(){return r * r * 3;}
}

class Triangle implements AreaCalcuable
{
   int b; int h;
   Triangle(int bb, int hh){b=bb; h = hh;}
   public int area(){return (b * h) / 2;}
}
class Square implements AreaCalcuable
{
   int w; int h;
   Square(int ww, int hh){w=ww; h = hh;}
   public int area(){return w * h;}
}
```

```
public class AreaTest
{
   public static void main(String[] args)
   {
      AreaCalcuable acs[] = new AreaCalcuable[3];
      Circle c = new Circle(1);
      Triangle t = new Triangle(2, 3);
      Square r = new Square(1, 2);
      acs[0] = c;
      acs[1] = t;
      acs[2] = r;
      int area;
      area = shapesArea(acs);
      System.out.println("合計面積:" + area);
   }

   static int shapesArea(AreaCalcuable[] acs)
   {
      int area = 0;
      for(int i = 0; i < acs.length; i++)
      {
         area += acs[i].area();
      }
      return area;
   }
}
```

演習

- 以下のmainメソッドが動くように，AreaCalcuableインタフェースを実装したTrapeziumクラスを追加せよ
（プログラム名：AreaTest）

```
public static void main(String[] args)
  {
      AreaCalcuable acs[] = newAreaCalcuable[4];
      Circle c = new Circle(1);
      Triangle t = new Triangle(2, 3);
      Square r = new Square(1, 2);
      Trapezium tr = new Trapezium(2, 3, 1); //引数は上底，下底，高さ
      acs[0] = c;
      acs[1] = t;
      acs[2] = r;
      acs[3] = tr;
      int area;
      area = shapesArea(acs);
      System.out.println("合計面積：" + area);
  }
```

他のクラスではやらない内容（＝テスト範囲外）

使用頻度が高い例

- Javaに初めから用意されているインタフェース

java.lang.Comparable

⇒インタフェース名の通り，このインタフェースを実装したクラスは，
「比較できる(comparable)」ことになる
⇒ここで言う比較とは，2つのインスタンスを比較した時，大小がわかる
ということ
⇒大小さえ分かれば，並び替え（ソート）ができる

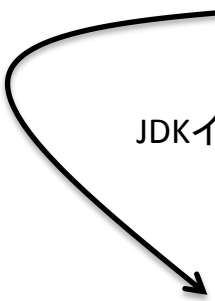
他のクラスではやらない内容（＝テスト範囲外）

使用頻度が高い例

以下がComparableインタフェース
（＊Javaに初めから用意されているので以下を書く必要はない）

```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

JDKインストール先，src.zip/java/lang/Comparable.javaより抜粋



インスタンスの大小を比較できるようにしたいのであれば，
compareToメソッドを実装しろ，ということ

⇒どのようにクラスでcompareToメソッドを実装するのか？
⇒説明書を参照
http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/java/lang/Comparable.html

他のクラスではやらない内容（＝テスト範囲外）

使用頻度が高い例

説明書を要約すると…

引数oは比較する相手のインスタンス

```
public int compareTo(T o)
{
   もし自分が小さい（oのほうが大きい）場合は
   return -1;（説明書では負の値）
   もし自分==oであれば
   return 0;
   もし自分が大きい（oのほうが小さい）場合は
   return 1;（説明書では正の値）
}
```

という風に実装しろ，ということ

他のクラスではやらない内容（＝テスト範囲外）

使用頻度が高い例

java.lang.Comparableの使用例：

```
class Circle implements java.lang.Comparable
{
   int r;
   Circle(int rr){r=rr;}
   int area(){return r * r * 3;}
   //半径を基に大小を決定する
   public int compareTo(Object o){
      Circle oc = (Circle)o;
      if (this.r < oc.r){
         return -1;
      }else if (this.r == oc.r){
         return 0;
      }else{
         return 1;
      }
   }
}
```

```
public class SortTest
{
   public static void main(String[] args)
   {
      Circle[] cs = new Circle[3];
      cs[0] = new Circle(5);
      cs[1] = new Circle(7);
      cs[2] = new Circle(2);
      printCircles(cs);
      java.util.Arrays.sort(cs);
      System.out.println("Sorted!");
      printCircles(cs);
   }

   static void printCircles(Circle[] cs)
   {
      for(int i = 0; i < cs.length; i++)
      {
         System.out.println("[" + i + "]:r = " + cs[i].r);
      }
   }
}
```

他のクラスではやらない内容（＝テスト範囲外）

使用頻度が高い例

java.lang.Comparableの使用例：

```
java.util.Arrays.sort(cs);
```

Javaでは初めからArraysクラスというものが用意されている
⇒配列に対して各種操作を行うクラス

Arraysクラスに配列を昇順に並び替えるsortメソッドがある
⇒sortメソッドは，「とにかく大小の比較ができるクラス
（＝**Comparableインタフェースが実装してあるクラス**）
の配列なら何でも昇順に並び替えるよ」
というメソッド

（実際にどのようにsortメソッドが並び替えを行っているか気になる人は，
プログラムを見ることが可能 JDKインストール先/src.zip/java/util/Arrays.java）