オブジェクト指向論(Q)

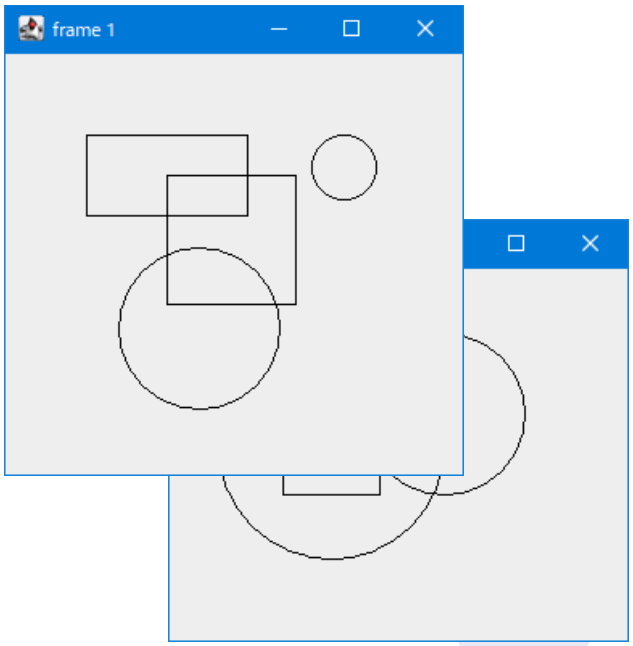
第11回(OOP5)講義

2023/6/19

來村 徳信

今回の講義のテーマと流れ

- 今回の目標
  - 1つのWindowに複数の四角形や円を描画できるようにする.
  - (1) 複数のクラス（四角形と円）を統一的に扱う
    - → 抽象クラス／インタフェース
  - (2) 複数のインスタンスを扱う
    - ArrayList クラスの利用
- 抽象クラス
  - Shape抽象クラス
- インタフェース
  - Drawable インタフェース
  - Listインタフェース(Java)
    - ジェネリクスとメソッドオーバーロード
- 例題プログラム
  - ArrayListを利用した Shape型での図形の保持と描画



抽象クラス(abstract class)

- インスタンス化できないクラス
  - 宣言だけのメソッド(抽象(abstract)メソッド)を含む
  - インスタンスフィールドや普通のメソッド（実装付）も持てる.
    - 後述のインタフェースはこれらを持てない.

```
public abstract class AbstractClassName {
    // フィールド定義
    public abstract retType methodName1(argTypes);
    public methodName2(...) {
        // メソッド定義
      }}
```

- 下位クラスで抽象メソッドの中身を定義する（実装する）
  - extends *AbstractClassName* と宣言（１つのみ）

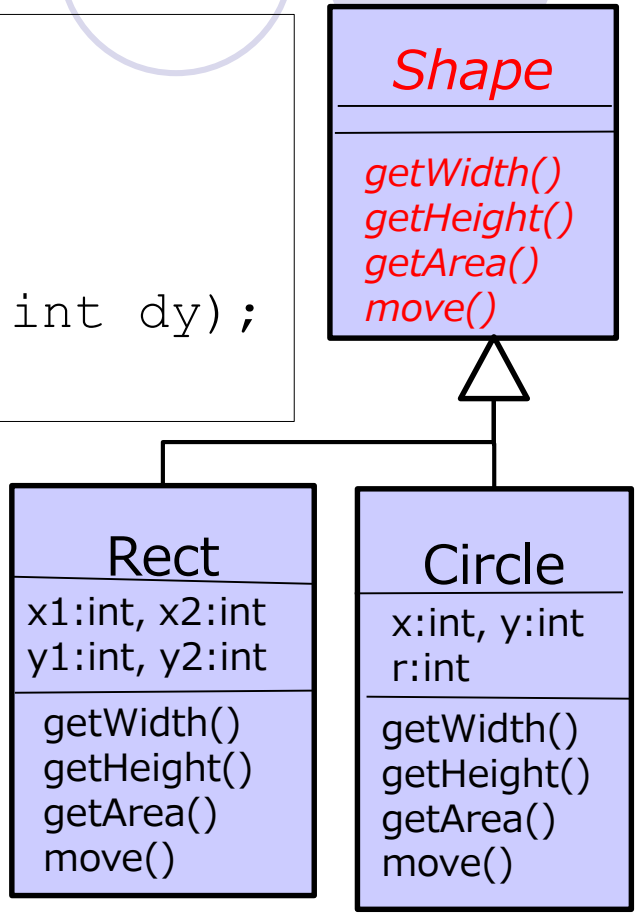
```
public class CName extends AbstractClassName {
    public retType methodName1(argTypes) {
      // メソッドの実装の定義
    }}
```

例：抽象クラス Shape

- 数学的図形一般を表す

```
public abstract class Shape {
    public abstract int getWidth();
    public abstract int getHeight();
    public abstract double getArea();
    public abstract void move(int dx, int dy);
    }
```

UML:
抽象クラス／メソッドは
イタリックで表す



  - メソッド名，引数，戻り値を宣言.
    - 処理内容 {...} がない.
  - 上の例は抽象メソッドの宣言だけで，フィールドやメソッド実装を含まないが，含めることもできる.
    - 例：draw() の実装（後述）

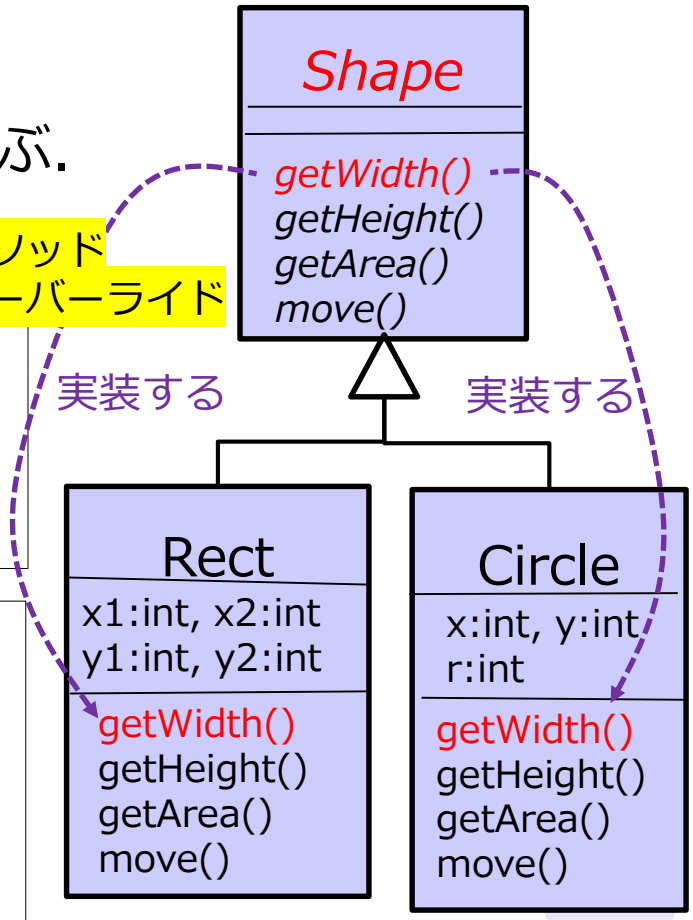
- Shapeの下位クラスはこれらのメソッドを「実装」する．意味が共通している.
  - 処理内容 {...} を記述する.

抽象メソッドの実装

- Shape 抽象クラスは（現時点では）メソッドの宣言のみ
  - 処理内容は書かれていない.
- 下位クラスで抽象メソッドをオーバーライドして，処理内容を実装する.
  - メソッド名，引数，戻り値は同じ.
  - {...} の中身を書く．「実装する」と呼ぶ.

```
public class Rect extends Shape {
   @Override
   public int getWidth() {
     return (this.x2-this.x1);
     }
```

```
public class Circle extends Shape {
   @Override
   public int getWidth() {
     return (this.r*2);
     }
```



Shape抽象クラスの利用

- 直接のインスタンスは作れないので，下位クラスのインスタンスを作る．
  - Rectクラスなどのコンストラクタで new する．
- Shape型の変数に代入できる．
  - 抽象クラスではあるが上位クラスなので，Rect/Circleクラスのインスタンスは，Shape型とみなせる．
- Shapeクラスで宣言されているメソッドを呼べる．
  - インスタンスが直接所属するクラスに応じて，自動的に適切なメソッドが実行される．意味は同じなのでクラスを気にせずに呼べる．

```
Shape s1 = new Rect(10,10,100,150);
Shape s2 = new Circle(200,200,10);
int w1 = s1.getWidth();
int w2 = s2.getWidth();
```

OOP5-A
Main.java

インタフェース

- 抽象メソッドの宣言のみを定義する抽象クラス

```
public interface IntefaceName {
    public abstract methodName1(...);
    public abstract methodName2(...);
    }
```

  - 当然、直接のインスタンスは作れない.
  - インスタンスフィールドやメソッドの実装も定義できない.
    - 抽象クラスとの違い.（後者については例外があるが特殊）
- 継承クラスはメソッドの処理内容を定義（実装）する
  - extends とは別に **implements** *interfaceName* と宣言（複数可）
  - 継承クラスは，interface で宣言されているすべてのメソッドを「実装」しなければならない.

```
public class CName extends SCName implements InterfaceName{
    public methodName1() {
    }
    public methodName2() {
    }}
```

抽象クラスとインタフェースの使い分け

- Java では複数のクラスを extends できない
  - あるクラスの直接の上位クラスは１つだけ.
  - 多重継承はできない.
  - 抽象クラスでは，「 共通 」な属性（フィールド）や処理（メソッド実装）を定義する.
- インタフェースは複数のものを implements できる.
  - extends とは別に
  - implements も複数書ける
  - メソッド群のオーバーライドを実現できる.
  - 「メソッドの 仕様 」＝「 機能仕様 」を表す
    - メソッドの名前，引数，戻り値，その意味
  - 仕様と「 実装 」＝「 実現 の仕方」を分離するために使う
    - インタフェースは共通（使い方は同じ）で，複数の異なる実現方法（実装）でクラスを定義（例：List インタフェース）

例：Drawableインタフェース

- 画面(Graphics)に描画する「機能」を持つ
  - 「画面に描画される（できる）もの」という意味
  - void draw(Graphics g)という抽象メソッドを宣言
  - Drawable インタフェースを implements するクラスは, draw(Graphics g) メソッドを実装しなければならない.

```
import java.awt.Graphics;

public interface Drawable {
    public abstract void draw(Graphics g);
    }
```

Drawableインタフェースによる Shapeクラスの意味の拡張

- 拡張前：数学的な図形
  - int getWidth() で幅を求めることができる
  - double getArea() で面積を求めることができる

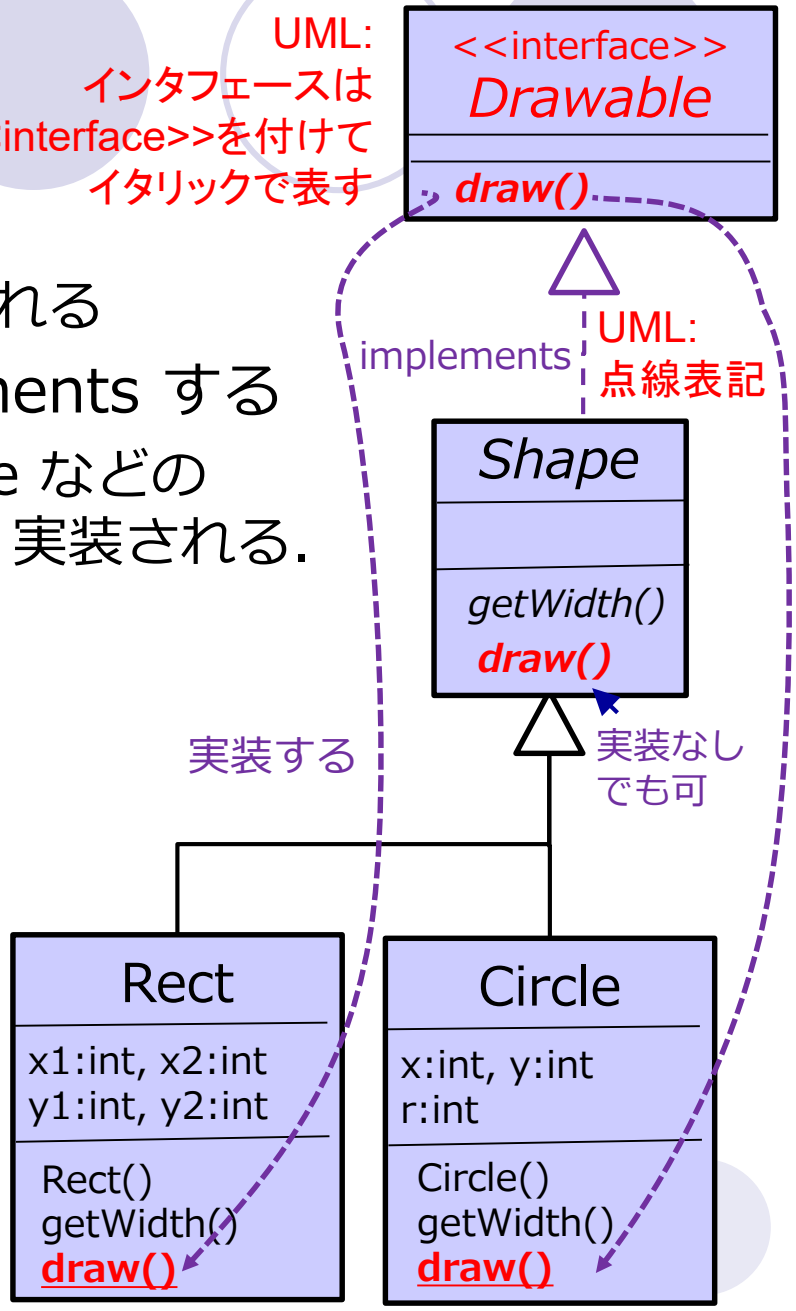
  ※これらは機能というよりは，図形一般が備えている性質，と捉えたので抽象クラスとして定義した.
- 拡張後：「描画される」図形
  - Drawable インタフェースを implements する.
  - void draw(Graphics g) で画面に描画する.

  ※これは図形にとってオプショナルな「機能」である，と捉えたのでインタフェースとして定義した.

拡張されたクラス図

- Drawable インタフェース
  - 「描画される」という意味
  - draw() メソッドが宣言（のみ）される
- Shape（と下位）クラスが implements する
  - draw() メソッドは，Rect や Circle などの Shape抽象クラスの下位クラスで，実装される.
  - Shape クラスは抽象クラスなので，実装されていなくても許される.
  - Shape型の変数を用いて，draw() を呼び出すことができる. 実際にはインスタンスに応じて、下位クラスのdraw()が実行される.

```
Shape s = new Circle(10,10,5);
s.draw(g);
```



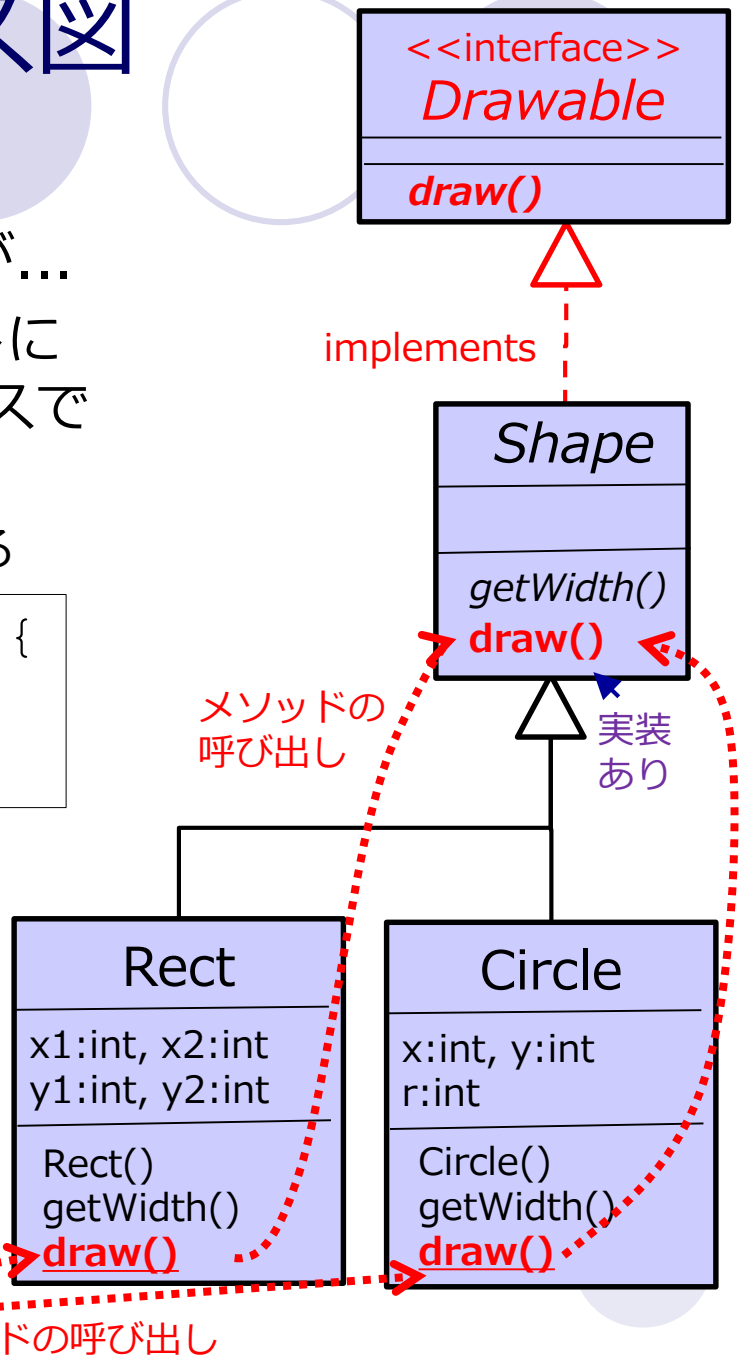
さらに拡張されたクラス図

- Shapeクラスの draw() メソッド
  - 抽象クラスなので実装しなくてもよいが...
  - Rect や Circle などの draw() メソッドに「共通な処理」があれば，Shape クラスで実装することができる.
    - ここでは，描画色の設定をすることにする

```
public void draw(Graphics g) {
   g.setColor(Color.BLACK);
   }
```

  - Shape型変数に向けて呼び出しても，実行されるのは Rect/Circle の draw()
    - 自分の処理の「前」に，Shape クラスの draw() メソッドを呼び出す.
    - オーバーライドされている，上位クラスのメソッドの呼び出し

```
Shape s = new ...
s.draw(g);
```



OOP5-A/B：Drawable & Shape

- Drawable インタフェース

```
public interface Drawable {
    public abstract void draw(Graphics g);
    }
```

- Shape 抽象クラス
  - メソッドの宣言＋draw()の共通部分の実装

```
public abstract class Shape implements Drawable {

  public abstract int getWidth();
  public abstract int getHeight();
  public abstract double getArea();
  public abstract void move(int dx, int dy);

  public void draw(Graphics g) {
    g.setColor(Color.BLACK);
    }
  }
```

Rect クラス

- Drawable インタフェースを実装するクラス1
  - Shape 抽象クラスの下位クラス
  - draw メソッドを実装（処理を定義）
    - Graphics.drawRect で四角形を描画．中身は前回とほぼ同じ．
    - 違い：最初に Shapeクラスの draw() を呼び出す．

```
public class Rect extends Shape {
...
  public Rect(int x1, int y1, int x2, int y2) {
    this.x1 = x1; this.y1 = y1;
    this.x2 = x2; this.y2 = y2;
    }
  @Override
  public void draw(Graphics g) {
    (7)_____;  <-------  Shape クラスの draw(g) を呼び出す
    g.drawRect(this.x1, this.y1,
               this.getWidth(), this.getHeight());
    }
```

Circle クラス

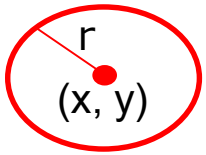
- Drawable インタフェースを実装するクラス２
  - drawメソッドを実装する（処理内容を定義する）

```
public class Circle extends Shape {
  private int x, y, r;

  public Circle(int cx,int cy,int cr) {
    this.x = cx; this.y = cy; this.r = cr;
    }

  @Override
  public void draw(Graphics g) {
    (7)_____;
    (8)_____(
      (9)_____,
      (10)_____,
      this.getWidth(),
      this.getHeight());
    }
```

円の中心座標(x,y)と半径r を表す



Shape クラスの draw(g) を呼び出す
(Rectクラスの(7)と同じ)

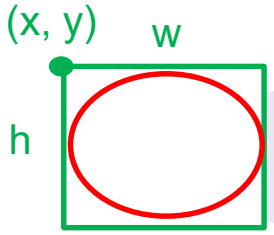
楕円を描画する Graphics クラスの drawOval を使う．引数は（左上X座標，左上Y座標，横の幅，縦の高さ）

※(9),(10)はメソッドを呼び出さずに，演算で求める．
※円の左上x,y座標が負の値になる可能性は考慮しなくよい．

Graphics クラス (java.awt.)

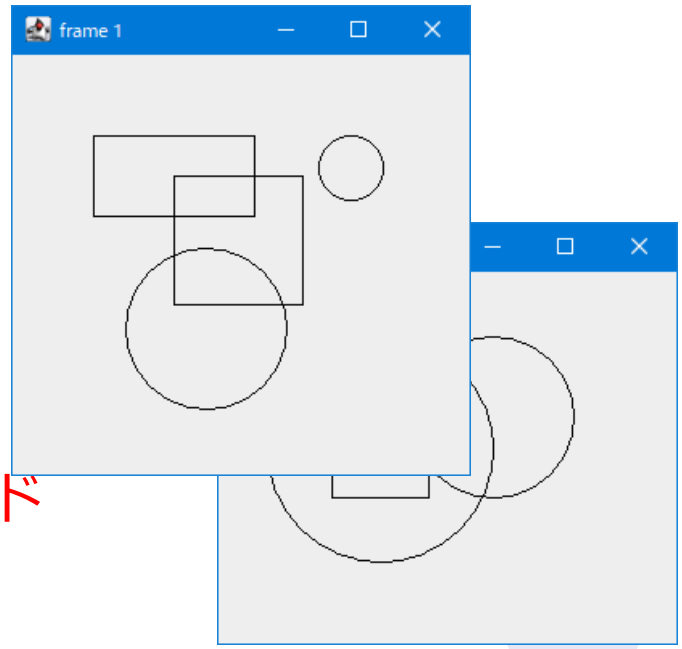
再掲+α

- 描画対象を表す
  - https://docs.oracle.com/javase/jp/8/docs/api/java/awt/Graphics.html
  - 現在の描画コンテキストを（フィールドとして）持つ
    - 例：現在の色
- 基本的な描画メソッドを持つ
  - void drawLine(int x1, int y1, int x2, int y2)
    - 点 (x1, y1) と点 (x2, y2) を結ぶ線を現在の色を使って描く.
  - void drawRect(int x, int y, int width, int height)
    - 指定された矩形の輪郭を描く.
  - void drawOval(int x, int y, int width, int height)
    - 指定された矩形に収まる楕円の輪郭を描く.
    - (x,y)が左上座標，width が幅，heightが高さ
  - void setColor(Color c)
    - 現在の色を、指定された色に設定する.



今日の講義のテーマと流れ（再掲１）

- 今回の目標
  - 1つのWindowに複数の四角形や円を描画できるようにする.
  - (1) 複数のクラス（四角形と円）を統一的に扱う
    - → 抽象クラス／インタフェース
  - (2) 複数のインスタンスを扱う
    - ArrayList クラスの利用
- 抽象クラス
  - Shape抽象クラス
- インタフェース
  - Drawable インタフェース
  - Listインタフェース(Java)
    - ジェネリクスとメソッドオーバーロード
- 例題プログラム
  - ArrayListを利用した Shape型での図形の保持と描画



インタフェースの例：java.util.List インタフェース

- 「順序を持った要素の集まり」を操作するメソッド群
  - リストの要素は指定したクラス*EC*のインスタンス(への参照)
  - 順序があるので「集合」とは異なる→ コレクション（参考）
- 宣言されているメソッドの例
  - boolean add(*EC* e)：リストの最後に要素 e （*EC*というクラスのインスタンス）を追加する．成功すれば true を返す．
  - boolean add(int index, *EC* e) ：リストのindex番目に，要素 e (*EC*クラスのインスタンス）を挿入する．後ろにずれる．
  - *EC* get(int index)：リストのindex番目の要素を返す．
  - int size()：リストの要素の数を返す．
- 実装クラスの例（他にもある．末尾の「参考」を参照）
  - java.util.ArrayList クラス
    - Listインタフェースを配列で実装したクラス．可変長の配列を提供

ジェネリクス（総称型）

- List/ArrayList は総称型である.
  - 要素には，指定されなければ，最上位のObjectクラスの下位クラス（つまり全てのクラス）のインスタンスが入る.
  - 危険なので標準的に，宣言時に *List<EC>* と書いて，要素を *EC クラス*（とその下位クラス）のインスタンスに限定する.
    - 宣言 `ArrayList<EC> list1;` または `List<EC> list1;`
      インタフェース型のクラス変数も作れる（こちらが普通）
    - 作成 `list1 = new ArrayList<>();`
      コンストラクタの呼び出し．Listではダメ
    - 宣言＆作成
      ```
      List<EC> list1
      = new ArrayList<>();
      ```
      ダイヤモンドオペレータ <EC>と書くのと同じ．省略記法だが標準的.
  - *EC*クラス（の下位クラス）のインスタンス（のみ）を要素として，追加(add)・置換(set)できる.
    - 要素の追加 `list1.add(e)`
      リストの末尾に，*EC*クラスのインスタンスeを，要素として追加
    - 他のクラスのインスタンスを入れようとすると，エラーになる．実行時に生じる例外の場合もある.

Listの全要素へアクセスする for文

- 例：要素のクラスが *EC* である **list1** の場合

```
List<EC> list1 = new ArrayList<>();
```

← 前のスライドと同じ

- (1) for 文
  - 順番を表す変数（例：i) を0からListのサイズまで増加させて，変数iでListの要素にアクセスする.

```
for (int i=0; i<list1.size(); i++) {
    EC e = list1.get(i);
    if (e != null) e.method();
    }
```

- (2) for each 文
  - 「:」の右にListの変数名（コレクション型(後述)，配列も可）を書く．左側の要素型（例：*EC*型）の変数(例：e)に，Listの全要素がひとつづつ順番に取り出される.

```
for (EC e : list1) {
    if (e !=null) e.method();
    }
```

ECクラスのmethodという名前のメソッドを e に向けて呼び出す場合

練習：String型のArrayList の使用

- 宣言と生成

```
import java.util.List;
import java.util.ArrayList;
```

  - String 型を要素とするList型の変数 ***strList*** を宣言し，ArrayListのインスタンスを生成して，それへの参照を代入する

```
(1)_______ strList = new (2)_____________;
```

- 要素の追加
  - “abc”, “xyz” を順にリストの最後に要素として追加する.

```
(3)_______("abc"); (3)_______("xyz");
```

- 要素の取り出し
  - For each 文で，***strList*** の全要素をString型の変数***se***に順番に取り出し，大文字に変換した結果を resStr に連結する.

```
String resStr = "";
for ((4)_______ : (5)_______) {
   resStr += (6)_________;
   }
System.out.println(resStr);
```

基礎的文法：文字列の利用

- 文字列型：String
  - （実はクラスなので，大文字始まり）
  - “abc” のように “ … “ で値を生成できる.
  - 演算子+で連結できる.
    - 他の型の変数を + で文字列型に連結すると，
      その変数の値が，自動的に文字列に変換される.
  - System.out.println()でコンソール出力できる
  - Stringクラスのメソッド例：
    - **String toUpperCase()**：文字列を大文字に変換して返す.

参考

このように値の変更を行う場合は，StringBuilder / StringBuffer クラスを使うことが推奨されている

追加

```
int n = 10;
String str1 = "value of n = ";
str1 += n;
str1 = str1.toUpperCase();
System.out.println(str1);
```

文字列リテラルの生成と代入

変数nは整数型だが，その値を表す文字列に自動的に変換される

文字列を連結する演算子

追加

String のインスタンス str1 へ**toUpperCase()** を呼び出す.

大文字に変換された文字列が戻るので，代入する.

メソッドオーバーロードとライド

- メソッドオーバーロード（多重定義） NEW
  - １つのクラス内に，同じメソッド名の，異なる引数（数と型）を持つような複数のメソッドを定義すること.
  - どのような引数で呼び出されたか（引数の数や型）によって，どのメソッドが実行されるかが決まる.
    - 引数を省略すると，デフォルト処理される場合が多い.
    - 例1：Listインタフェースのaddメソッド
    - 例2：複数のコンストラクタ
- メソッドオーバーライド 復習
  - 下位クラスのメソッドが，上位クラスの，同名・同じ引数・同じ戻り値のメソッドを，上書きすること.
  - インスタンスが直接所属するクラスによって，どのクラスのメソッドが実行されるかが決まる. NEW
- 合わせて多態性(ポリモーフィズム，polymorphism)と呼ばれる.

再掲+α

オーバーロードの例1：java.util.List インタフェース

- 「順序を持った要素の集まり」を操作するメソッド群
  - リストの要素は指定したクラス*EC*のインスタンス(への参照)
  - 順序があるので「集合」とは異なる→ コレクション（参考）
- 宣言されているメソッドの例
  - boolean add(*EC* e)：リストの最後に要素 e （*EC*というクラスのインスタンス）を追加する．成功すれば true を返す．
  - boolean add(int index, *EC* e) ：リストのindex番目に，要素 e （*EC*クラスのインスタンス）を挿入する．後ろにずれる
  - *EC* get(int index)：リストのindex番目の要素を返す．
  - int size()：リストの要素の数を返す．
- 実装クラスの例（他にもある．末尾の「参考」を参照）
  - java.util.ArrayList クラス
    - Listインタフェースを配列で実装したクラス．可変長の配列を提供

例2:コンストラクタのオーバーロード

- 例：Rectのコンストラクタ
  - コンストラクタの一行目で暗黙的に super() が呼ばれている
    - 上位クラスのコンストラクタ．最上位クラスでインスタンス生成
  - 引数4つ（必須な値すべてが引数）
  - 引数2つ（デフォルト値を使って, 同じクラスのコンストラクタを **this()** で呼び出す）

```
public class Rect {
   private int x1, x2, x2, y2;

   Rect(int x1, int y1, int x2, int y2) {
      // ここで super() が暗黙的に呼ばれる.
      this.x1 = x1; this.y1 = y1;
      this.x2 = x2; this.y2 = y2;
      }
   Rect(int x1, int y1) {
      this(x1,y1,x1+100,y1+100);
      }
```

左上座標のみが指定されたときのコンストラクタ

デフォルト値として幅・高さ 100 の四角形を生成するとした場合

上のRect(int,init,int,int)を呼び出す．
"Rect" ではない（コンパイルエラーになる）

コンストラクタの定義

- クラス名と同名のメソッド（前回と同じ）
- 通常，下位クラスのコンストラクタは，上位クラスのコンストラクタを super(...) で呼び出したあと，独自な処理を行う．

```
public class Rect {
  private int x1, y1, x2, y2;

  public Rect(int x1,int y1,int x2,int y2) {
    this.x1 = x1; this.x2 = x2;
    this.y1 = y1; this.y2 = y2;
    }}
```

```
public class RndRect extends Rect {
 private int r;

 public RndRect(int x1,int y1,int x2,int y2,int r) {
    super(x1,y1,x2,y2);
    this.r=r;
    }}
```

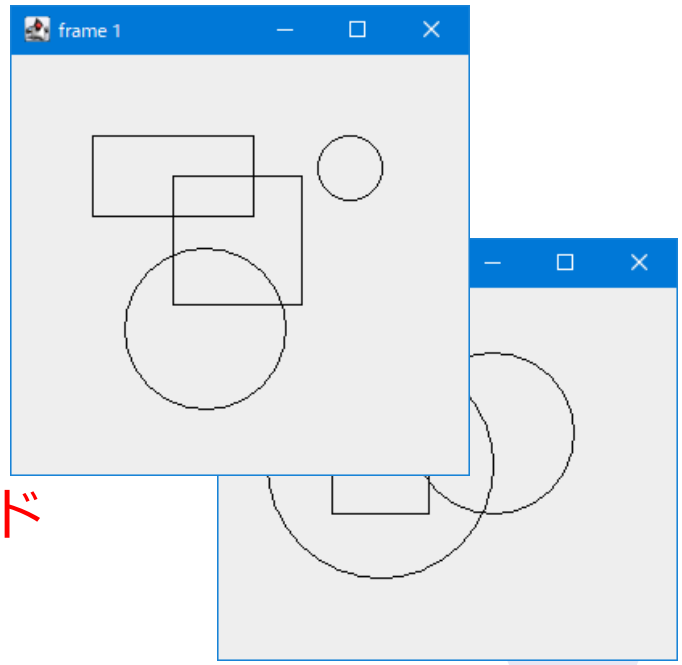
追加された仮引数

上位クラスのコンストラクタの呼び出し（特殊構文．この位置）

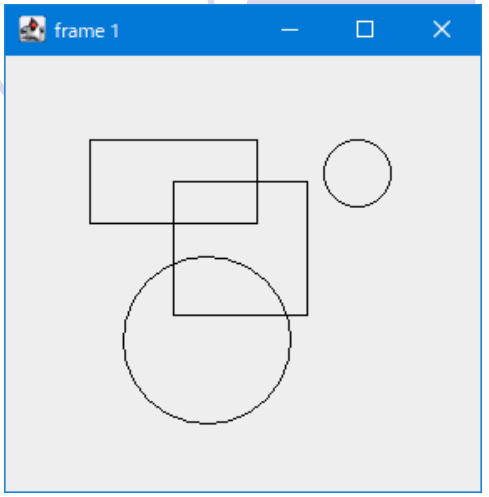
このクラス固有の処理

今日の講義のテーマと流れ（再掲2）

- 今回の目標
  - 1つのWindowに複数の四角形や円を描画できるようにする.
  - (1) 複数のクラス（四角形と円）を統一的に扱う
    - → 抽象クラス／インタフェース
  - (2) 複数のインスタンスを扱う
    - ArrayList クラスの利用
- 抽象クラス
  - Shape抽象クラス
- インタフェース
  - Drawable インタフェース
  - Listインタフェース(Java)
    - ジェネリクスとメソッドオーバーロード
- 例題プログラム
  - ArrayListを利用した Shape型での図形の保持と描画



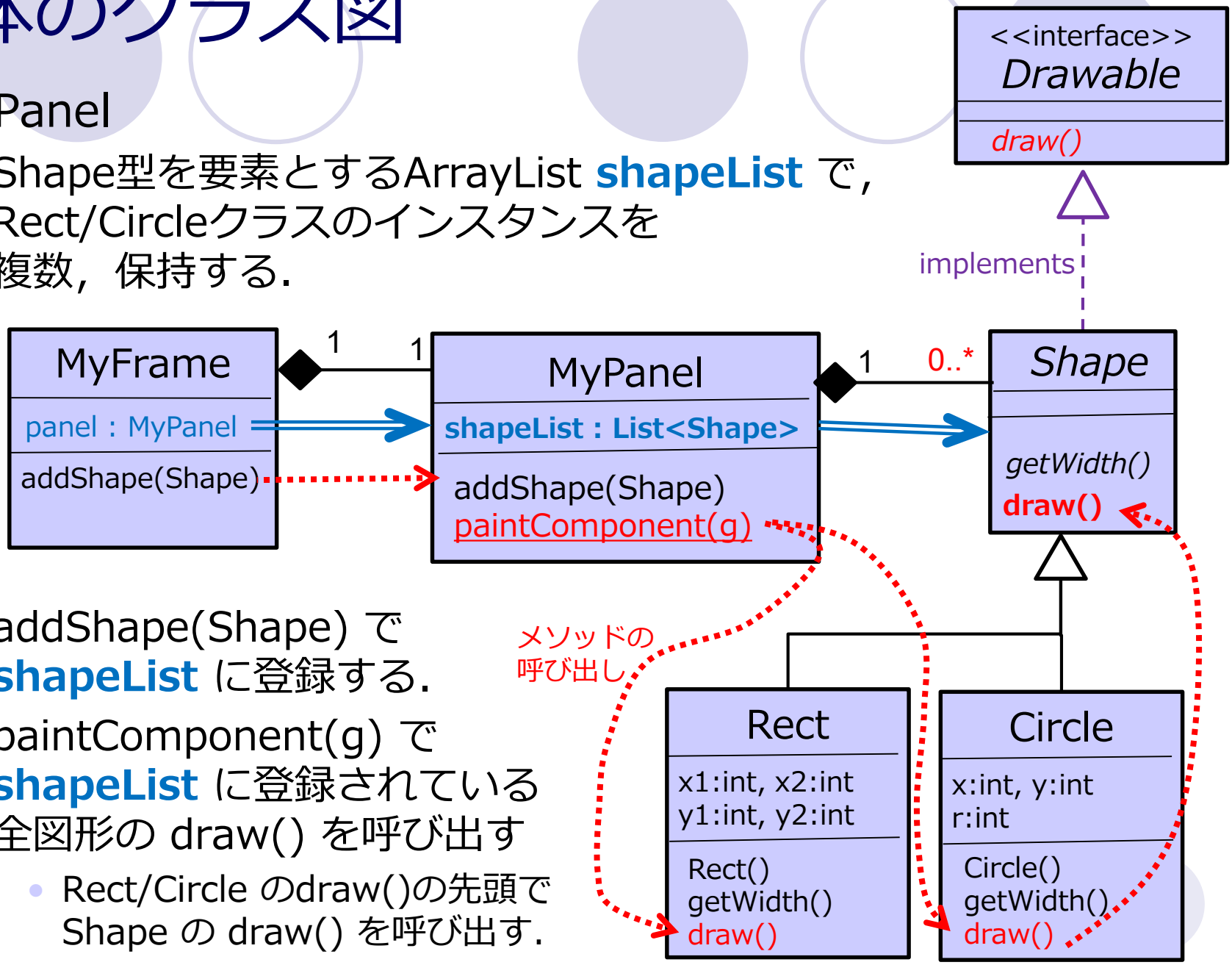
複数の図形の描画



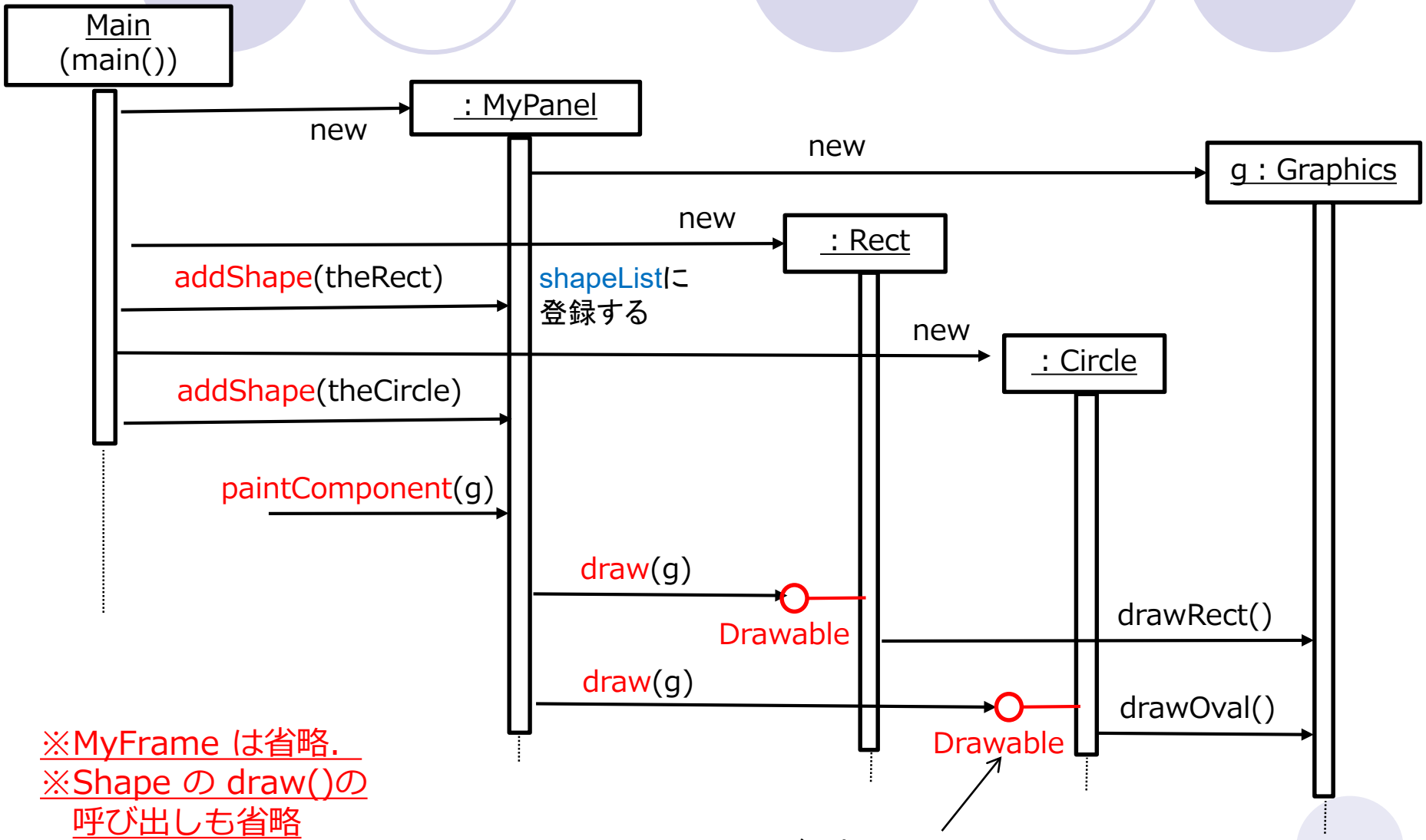
- MyPanel クラスのフィールドとして，Shape型を要素とする ArrayList ***shapeList*** を用意する.
  - Shape型の下位クラスである RectやCircle クラスのインスタンスを格納できる.
- 図形の登録：addShape(Shape s)
  - 呼ばれるたびに，s を ***shapeList*** に追加する.
  - s は実際には，Shape 抽象クラスの下位クラスである，Rect/Circleクラスのインスタンス
- 図形の描画：paintComponent(Graphics g)
  - ***shapeList*** に登録されている全ての要素（Shape型）をひとつずつShape型変数 ***se*** に取り出し，各 ***se*** へ draw(g) を呼び出す.
  - インスタンスに応じて，RectまたはCircleクラスのdraw() が呼び出される.
    - その中で Shape クラスの draw() を呼び出す

全体のクラス図

- MyPanel
  - Shape型を要素とするArrayList **shapeList** で，Rect/Circleクラスのインスタンスを複数，保持する.
  - addShape(Shape) で **shapeList** に登録する.
  - paintComponent(g) で **shapeList** に登録されている全図形の draw() を呼び出す
    - Rect/Circle のdraw()の先頭で Shape の draw() を呼び出す.



メッセージの流れ：登録と描画



Mainクラス main() メソッド

- Shapeの下位クラスのインスタンスを作り，addShape() で(MyFrameを通して）MyPanel に登録する.

```
public class Main {
  public static void main(String[] args){
    MyFrame.setUI(); // set cross-platform UI

    MyFrame mf1 = new MyFrame("frame 1",50,50,300,300);
    Shape s = new Rect(50,50,150,100);
    mf1.addShape(s);
    mf1.addShape(new Rect(100,75,180,155));
    mf1.addShape(new Circle(120,170,50));
    mf1.addShape(new Circle(210,70,20));
    mf1.makeVisible();

    MyFrame mf2 = new MyFrame("frame 2",400,100,300,270);
    mf2.addShape(new Rect(70,50,130,140));
    mf2.addShape(new Circle(170,90,50));
    mf2.addShape(new Circle(100,110,70));
    mf2.makeVisible();
  }}
```

図形データが揃った後で表示状態にする.

MyFrame：登録

- addShape(Shape) メソッド
  - 自分の MyPanel インスタンス (panel) へ転送するだけ
  - 前回とほぼ同じ．メソッド名と引数が変わっただけ．

```
public class MyFrame extends JFrame {
   private MyPanel panel = null;

   public MyFrame(String title) {
      super(title);
      ...
      this.panel = new MyPanel();
      this.add(this.panel, BorderLayout.CENTER);
      ...
      }

   public void addShape(Shape s) {
      if (this.panel != null) this.panel.addShape(s);
      }
...
```

MyPanel(1)：登録

```
import java.awt.*;
import javax.swing.*;
import java.util.List;
import java.util.ArrayList;

public class MyPanel extends JPanel {
    private (1)_____ shapeList = null;

    public MyPanel() {
        super();
        this.shapeList = new (2)_____;
        }

    public void addShape(Shape s) {
        (3)_____;
        this.repaint();
        }
```

図形が増えたから再描画が必要なので，
描画イベントをリクエストする．
自動的に，paintComponent() が呼ばれる．

（続く）

MyPanel(2)：描画

- void paintComponent()
  - ArrayList **shapeList** の全要素を，Shape型変数**se**に取り出し，draw() を呼ぶ.
    - Shape型変数**se**の各インスタンスは実際にはRect/Circleのインスタンスであり，自分を描画するような draw()メソッドを必ず持つことが「保証」されている.
  - MyPanelやShapeはそれぞれの描画方法を知らない.
    - 「カプセル化」．図形の種類が増えても，MyPanel やShapeを変更する必要がない！
      - Shape クラスの draw() は共通部分の定義

```
public void paintComponent(Graphics g){
  super.paintComponent(g);
  for ((4)__________ : (5)________________) {
    if (se != null) (6)____________;
    }
  }
```

Drawable & Shape

- Drawable インタフェース

```
public interface Drawable {
    public abstract void draw(Graphics g);
    }
```

- Shape 抽象クラス
  - メソッドの宣言＋draw()の共通部分の実装

```
public abstract class Shape implements Drawable {

  public abstract int getWidth();
  public abstract int getHeight();
  public abstract double getArea();
  public abstract void move(int dx, int dy);

  public void draw(Graphics g) {
    g.setColor(Color.BLACK);
    }
  }
```

Rect クラス：描画

- Drawable インタフェースを実装するクラス1
  - Shape 抽象クラスの下位クラス
  - draw メソッドを実装（処理を定義）
    - Graphics.drawRect で四角形を描画．中身は前回とほぼ同じ．
    - 違い：最初に Shapeクラスの draw() を呼び出す．

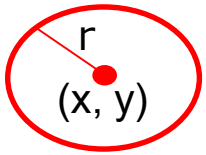
```
public class Rect extends Shape {
...
  public Rect(int x1, int y1, int x2, int y2) {
    this.x1 = x1; this.y1 = y1;
    this.x2 = x2; this.y2 = y2;
    }
  @Override
  public void draw(Graphics g) {
    (7)_____;  <-------  Shape クラスの draw(g) を呼び出す
    g.drawRect(this.x1, this.y1,
               this.getWidth(), this.getHeight());
    }
```

Circle クラス：描画

- Drawable インタフェースを実装するクラス2
  - drawメソッドを実装する（処理内容を定義する）

```
public class Circle extends Shape {
  private int x, y, r;
  public Circle(int cx,int cy,int cr) {
    this.x = cx; this.y = cy; this.r = cr;
    }
  @Override
  public void draw(Graphics g) {
    (7)_____;
    (8)_____(
      (9)_____,
      (10)_____,
      this.getWidth(),
      this.getHeight());
    }
```

円の中心座標(x,y)と半径r を表す



Shape クラスの draw(g) を呼び出す
(Rectクラスの(7)と同じ)

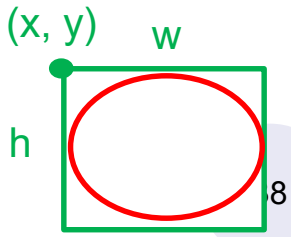
楕円を描画する Graphics クラスの drawOval を使う．引数は（左上X座標，左上Y座標，横の幅，縦の高さ）

※(9),(10)はメソッドを呼び出さずに，演算で求める．
※円の左上x,y座標が負の値になる可能性は考慮しなくよい．

Graphics クラス (java.awt.)

再掲

- 描画対象を表す
  - https://docs.oracle.com/javase/jp/8/docs/api/java/awt/Graphics.html
  - 現在の描画コンテキストを（フィールドとして）持つ
    - 例：現在の色
- 基本的な描画メソッドを持つ
  - void drawLine(int x1, int y1, int x2, int y2)
    - 点 (x1, y1) と点 (x2, y2) を結ぶ線を現在の色を使って描く.
  - void drawRect(int x, int y, int width, int height)
    - 指定された矩形の輪郭を描く.
  - void drawOval(int x, int y, int width, int height)
    - 指定された矩形に収まる楕円の輪郭を描く.
    - (x,y)が左上座標，width が幅，heightが高さ
  - void setColor(Color c)
    - 現在の色を、指定された色に設定する.



参考

コレクションフレームワーク

- オブジェクトの集まりを扱うクラス群(java.util.)
  - https://docs.oracle.com/javase/jp/8/docs/technotes/guides/collections/overview.html

例：

- List インタフェース
  - 順序がある．インデックスを指定してアクセスできる．
- Set インタフェース
  - 順序がない．要素は重複しない．
- Map インタフェース
  - キーと値の組．順序がない．
- それぞれいくつかの実装クラスがある．
  - 使えるメソッドは同じ．
  - 異なる特徴をもつ．例：ある操作に対する速度が異なる．

参考

Listインタフェースの実装クラス

- Listインタフェース
  - 順序がついた要素のまとまり
  - add(e), add(2,e), get(2) など「順番」を指定して，要素を出し入れできる.
- 実装クラス(java.util.)のいろいろ
  - ArrayListクラス
    - 配列をつかって実装している. （デフォルトの大きさは10）
    - サイズをオーバーすると新たな配列を作る.
    - 途中への挿入が遅い（ずらさないといけないから）.
  - LinkedListクラス
    - 要素自身が前後の要素への参照を持つことで実現
    - 途中への挿入が早い（つなぎなおすだけだから）.

参考：Java の配列

- 特殊なオブジェクトと捉えられる．
- 宣言（どちらの書き方でもよいが，上が現在の標準的）
  - 要素の型[] 配列名．例：int[] *intArray*;　Rect[] *rectArray*;
  - 要素の型 配列名[]．例：int *intArray*[];　Rect *rectArray*[];
- 生成：実行時に固定長
  - new 要素の型[要素数]　例：int[] *intArray*= new int[10];
  - 配列の領域が（動的に）確保される．
    - 要素の値は要素型のデフォルト値(0やnull)で初期化される．
- アクセス
  - 配列名[添え字] でアクセスできる．添え字は0から．
    - 例：*intArray*[2] = 10;　*rectArray*[0] = new Rect(...);
  - 配列の長さは，配列名.length で得られる．読み取り専用のフィールド．例：*intArray.length*

今回の講義のまとめ

- 今回の目標
  - 1つのWindowに複数の四角形や円を描画できるようにする.
  - (1) 複数のクラス（四角形と円）を統一的に扱う
    - → 抽象クラス／インタフェース
  - (2) 複数のインスタンスを扱う
    - ArrayList クラスの利用
- 抽象クラス
  - Shape抽象クラス
- インタフェース
  - Drawable インタフェース
  - Listインタフェース(Java)
    - ジェネリクスとメソッドオーバーロード
- 例題プログラム：
  - ArrayListを利用した Shape型での図形の保持と描画