オブジェクト指向技術 第12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

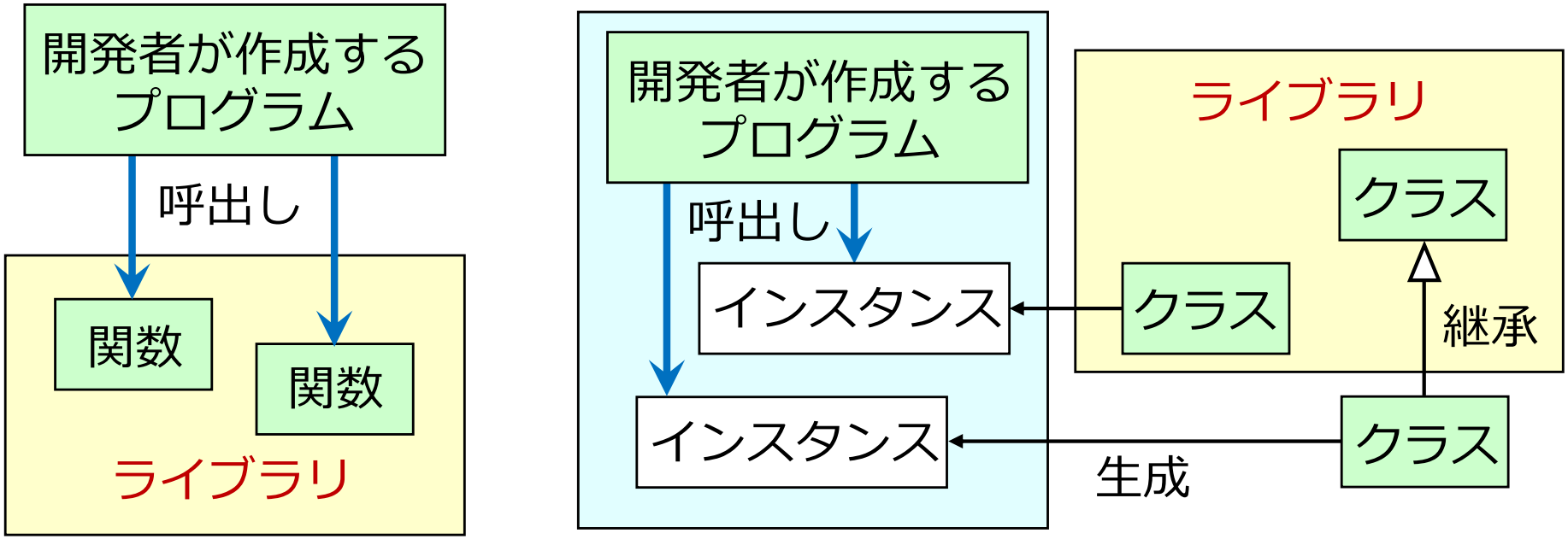
- オブジェクト指向プログラミング
  - Java
  - クラス，メソッド，フィールド
  - パッケージとアクセス制御
  - インスタンスの生成とアクセス
  - 継承，抽象クラス，インタフェース
  - オーバーライド，オーバーロード
  - 配列とコレクション
  - 例外処理，入出力処理
  - GUIプログラミング
  - イベント駆動モデル

GUIプログラミング

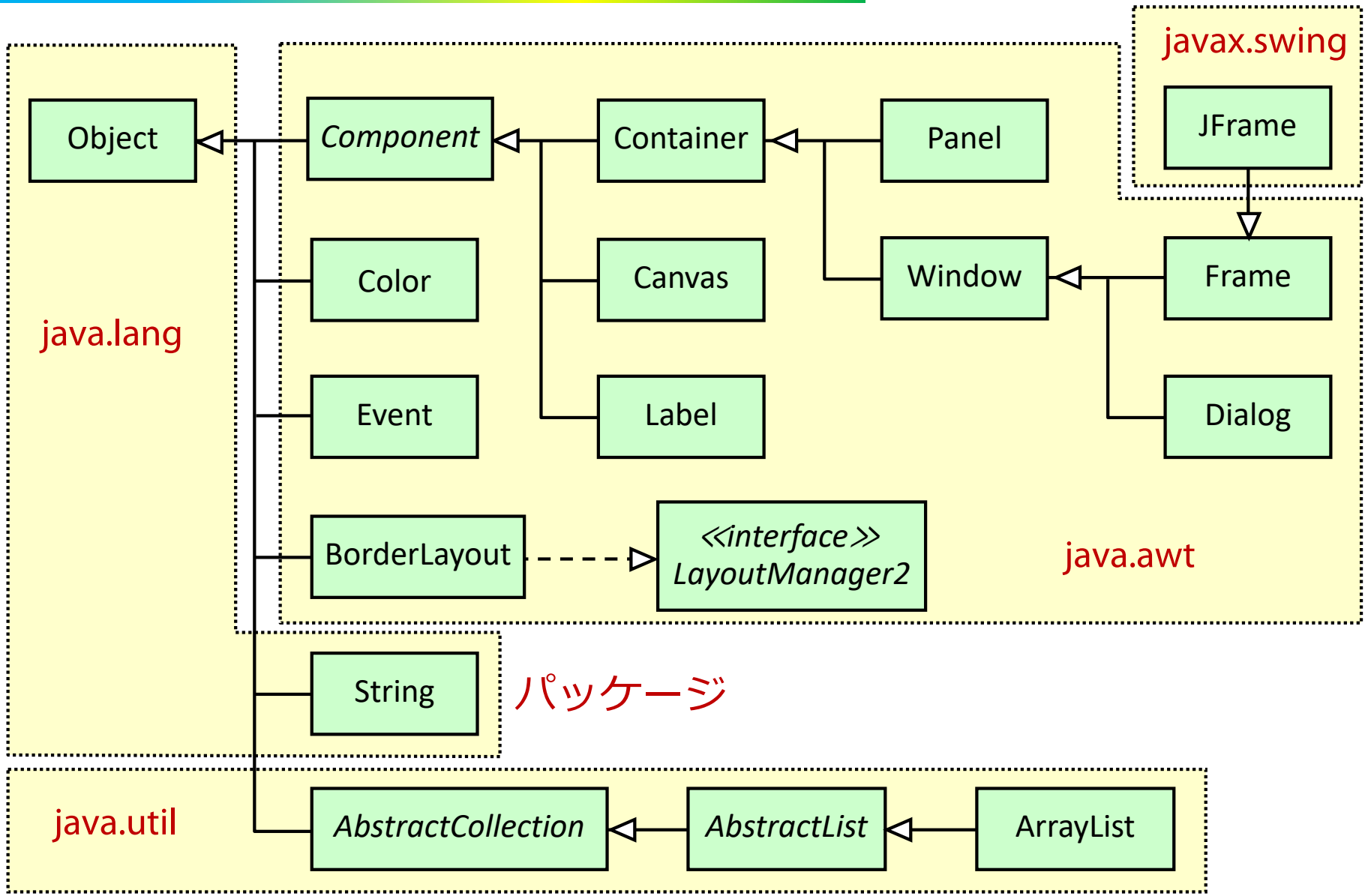
- GUI(Graphical User Interface)
  - グラフィクスを仲介してユーザとプログラムが対話する仕組み
- JavaBeanコンポーネントを用いて構築
  - GUIコンポーネントはウィジェット(widget)とも呼ばれる
- Javaが提供するGUIフレームワーク
  - AWT(Abstract Window Toolkit)
    - 初期から提供されているGUIフレームワーク
  - Swing
    - AWTの拡張
    - プラットフォームで共通して利用可能なGUIフレームワーク
  - JavaFX
    - リッチ・クライアントを作るために用意

ライブラリ(library)

- 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
  - 手続き(関数)の集まり
    - C言語ライブラリなど
  - オブジェクト指向におけるクラスライブラリ
    - クラス ＝ 独立性の高いモジュール
    - 実装の隠蔽(カプセル化)による置換が容易
    - 既存クラスのインスタンス生成と継承による再利用

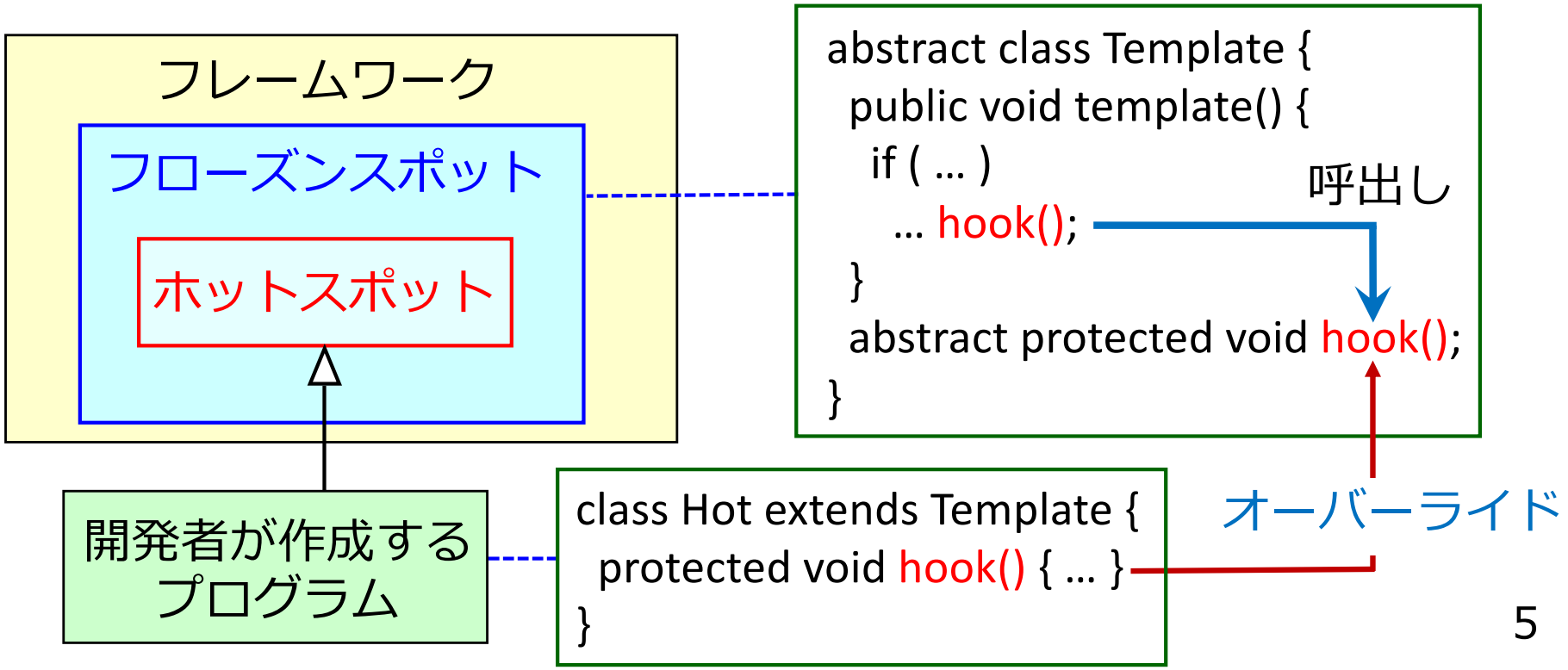


OOライブラリの例(Java)

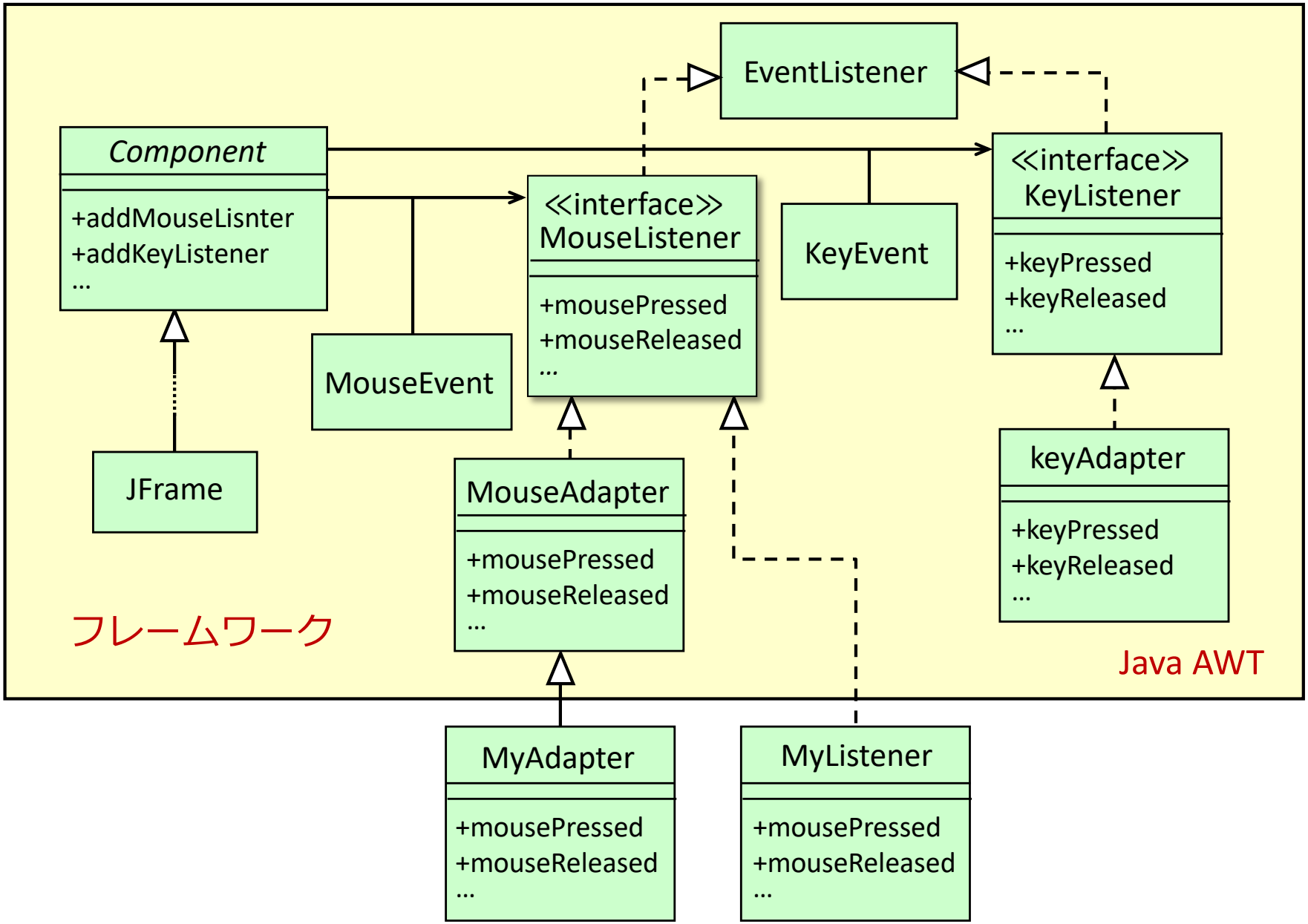


フレームワーク(framework)

- 再利用されるクラスだけでなく，そのインスタンス間の相互作用を内包
  - 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

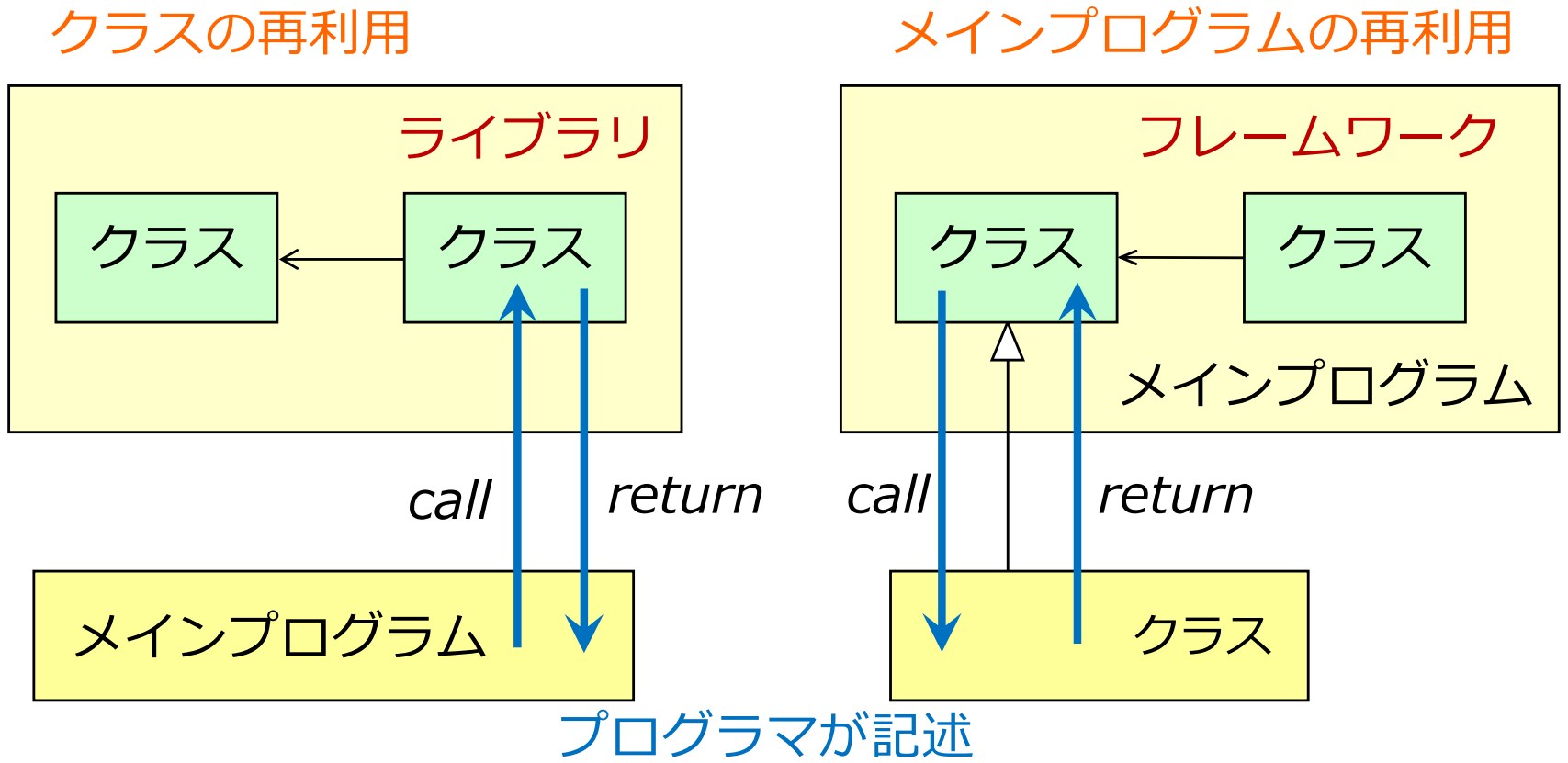


フレームワークの例(Java)



制御の逆転(inversion of control)

- フレームワークが制御の流れを管理
  - メインプログラムを内包
  - 呼び出されるクラスを作成



ラベルを持つウインドウの表示

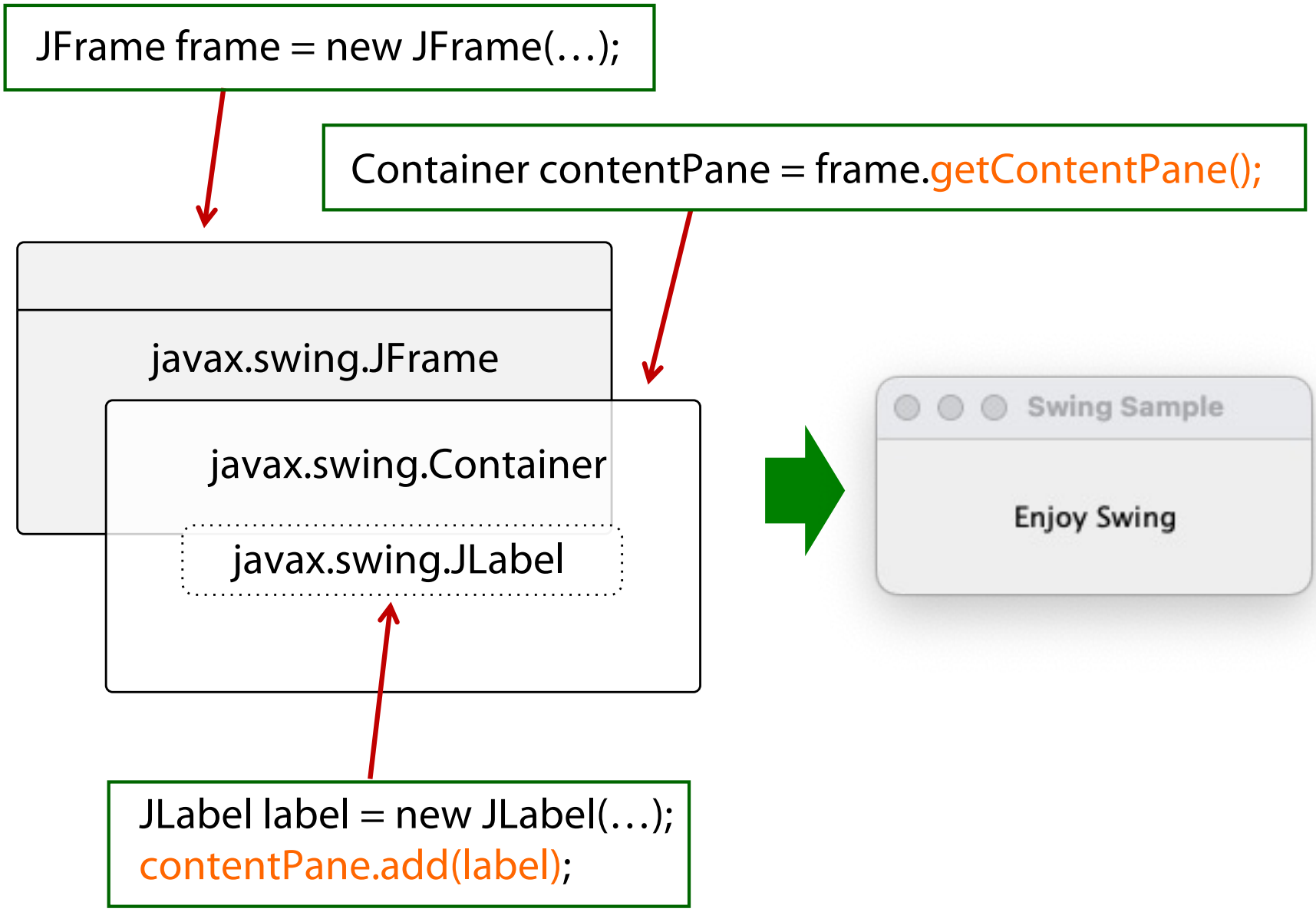
```
import javax.swing.*;
import java.awt.*;

public class Main7 {
   public static void main(String[] args) {

      JFrame frame = new JFrame("Swing Sample");
      Container contentPane = frame.getContentPane();

      JLabel label = new JLabel("Enjoy Swing", JLabel.CENTER);
      contentPane.add(label);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(200, 100);
      frame.setVisible(true);
   }
}
```

トップレベル
コンテナを作成

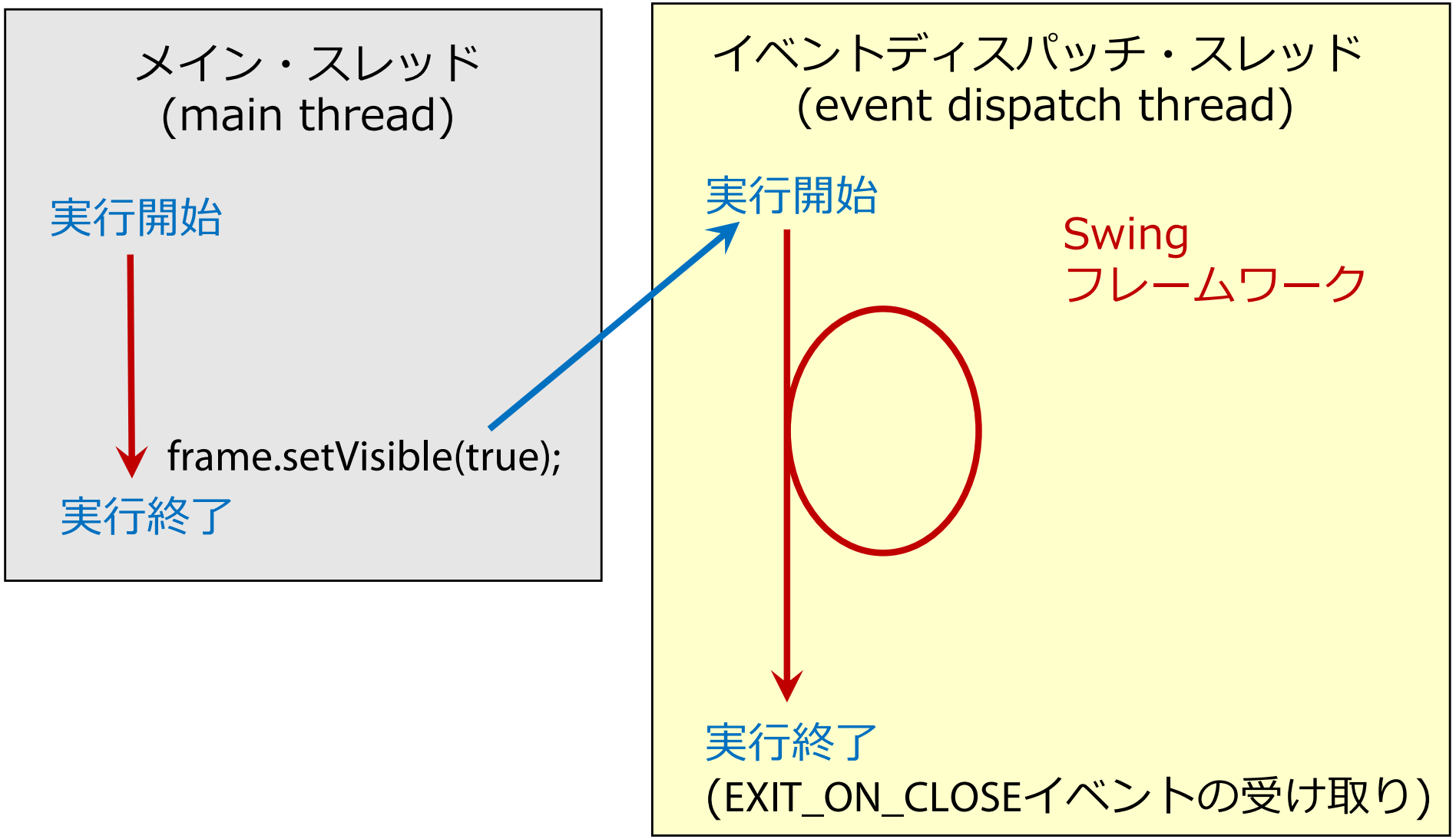
ラベルコンポーネントを
生成し，フレームに追加

フレームを表示

ウインドウの配置


JFrame frame = new JFrame(…);
Container contentPane = frame.getContentPane();
javax.swing.JFrame
javax.swing.Container
javax.swing.JLabel
JLabel label = new JLabel(…);
contentPane.add(label);
Swing Sample
Enjoy Swing

Swingにおける制御の流れ



スレッド(thread)

- プログラムの実行の脈絡
  - 一本の糸のようなもの
  - メモリ空間を共有
  - 協調的並列性
- マルチスレッド(multi-threading)
  - 複数のスレッドが同時に動作

- プロセスやタスク
  - メモリ空間を共有しない
  - 競合的並列性

スレッドの生成と実行(1/2)

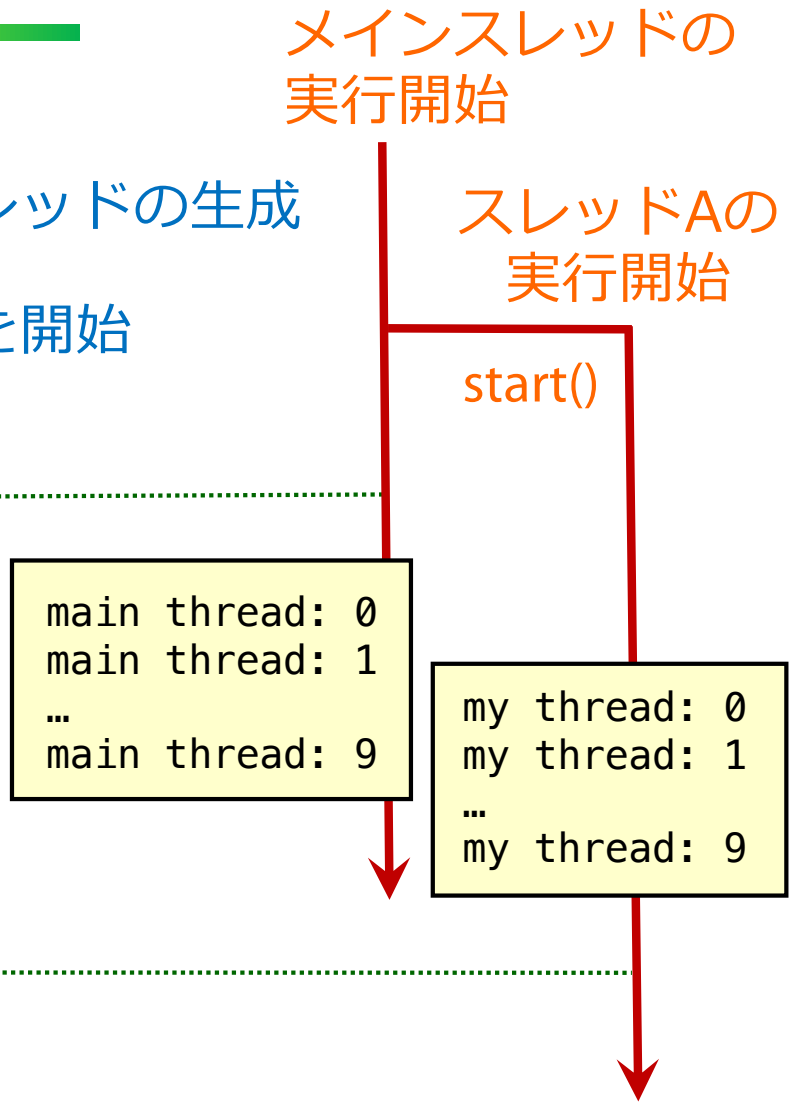
```
public class Main8 {
    public static void main(String[] args) {
        ThreadA thread = new ThreadA();
        thread.start();

        for (int i = 0; i < 10; i++) {
            System.out.println("main thread: " + i);
        }
    }
}

class ThreadA extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("my thread: " + i);
        }
    }
}
```

スレッドの生成

スレッドの実行を開始

メインスレッドで実行

スレッドAで実行



スレッドの生成と実行(2/2)

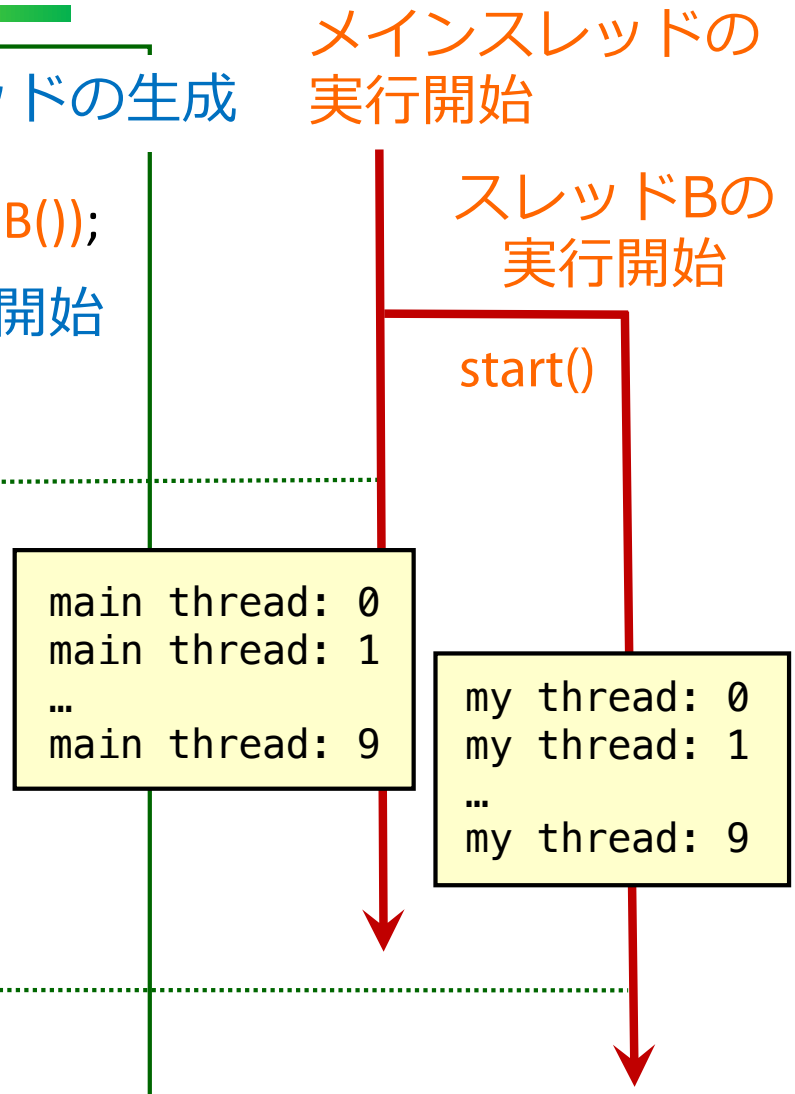
```
public class Main9 {
    public static void main(String[] args) {
        ThreadA thread = = new Thread(new ThreadB());
        thread.start();

        for (int i = 0; i < 10; i++) {
            System.out.println("main thread: " + i);
        }
    }
}

class ThreadB implements Runnable {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("my thread: " + i);
        }
    }
}
```

スレッドの生成

スレッドの実行を開始

メインスレッドで実行

スレッドBで実行



GUIコンポーネントの配置

- レイアウトマネージャを用いて配置を指定

```
public class Main10 {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Swing Sample");
      Container contentPane = frame.getContentPane();

      JPanel panel = new JPanel();
      panel.setLayout(new BorderLayout());
      contentPane.add(panel);

      JLabel label = new JLabel("Enjoy Swing.");
      panel.add(label, BorderLayout.NORTH);
      JButton button = new JButton(”Please Push");
      panel.add(button, BorderLayout.CENTER);
      …
   }
}
```

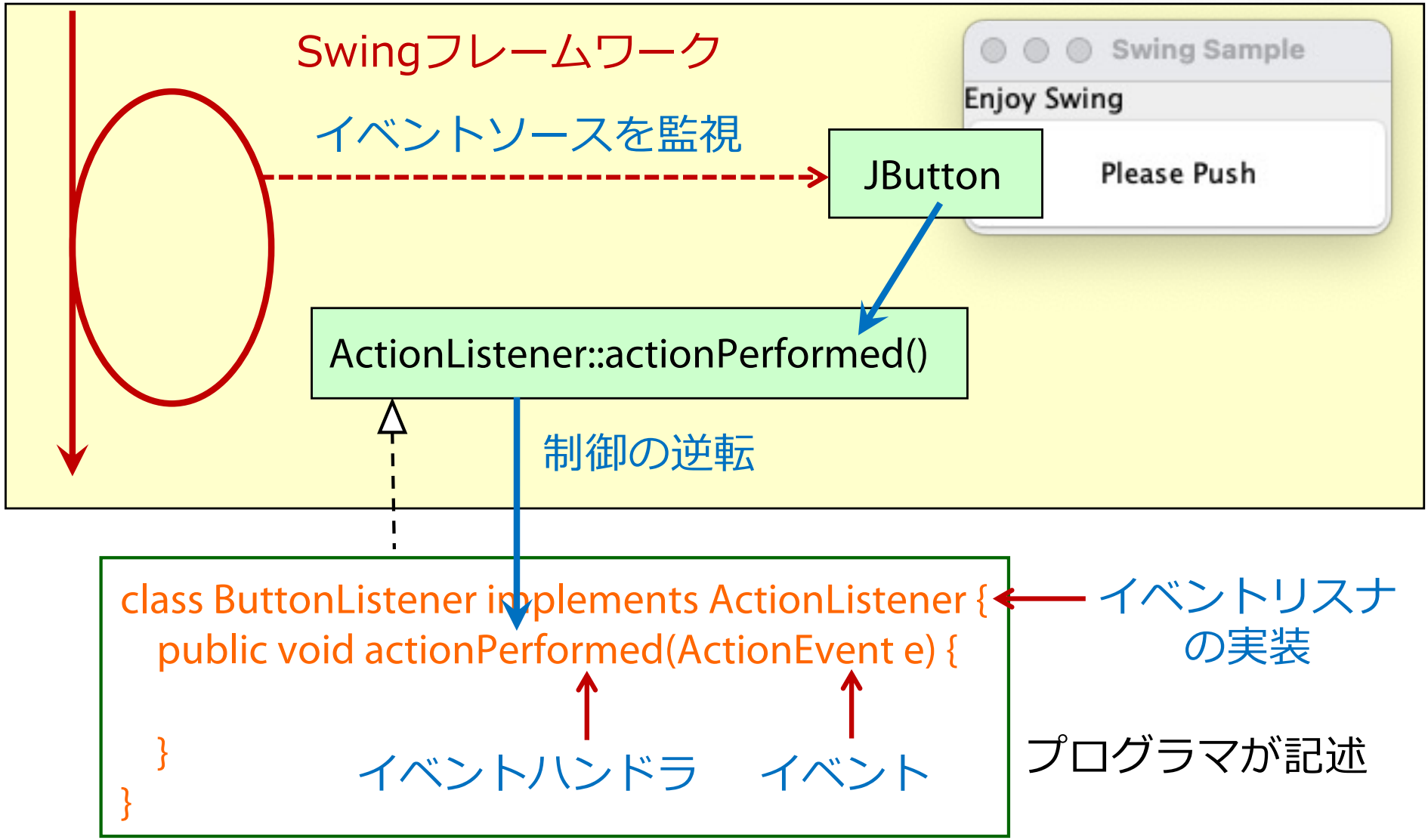
レイアウトマネージャを
生成 & 指定

位置を指定

イベント駆動モデル

- 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
  - 特にGUIプログラムにおいて利用される
  - イベントごとに処理を記述
  - 処理を直接呼び出さない
    - あらかじめリスナ(イベントハンドラ)を登録
      → プログラムのメインループの中でイベントを監視
      → イベント発行時に登録されたイベントハンドラを呼び出し
- いろいろなイベント駆動プログラムの例
  - ユーザイベントに対する処理
    - ボタンクリック、マウスクリック、キーボード入力、描画
  - センサの利用(組込みソフトウェア)
  - 他のプロセス，スレッドからのメッセージ

イベント駆動モデルの例



GUIイベントの処理の例

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Main11 {
    public static void main(String[] args) {
        …
        JButton button = new JButton("Please Push");
        contentPane.add(button);
        ButtonListener bl = new ButtonListener();
        button.addActionListener(bl);
        …
    }
}
```

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

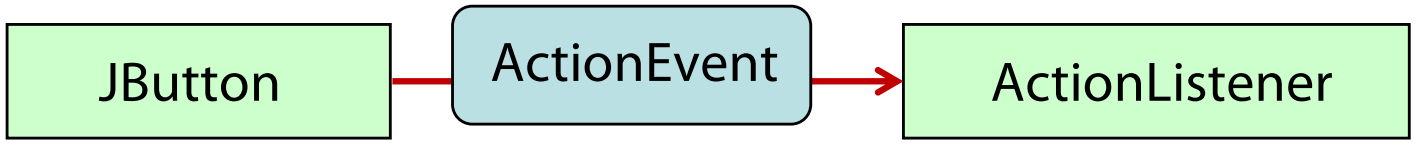
イベントソースを作成

イベントリスナの生成

イベントソースの登録

Swingにおける様々なイベント

- ボタンクリック



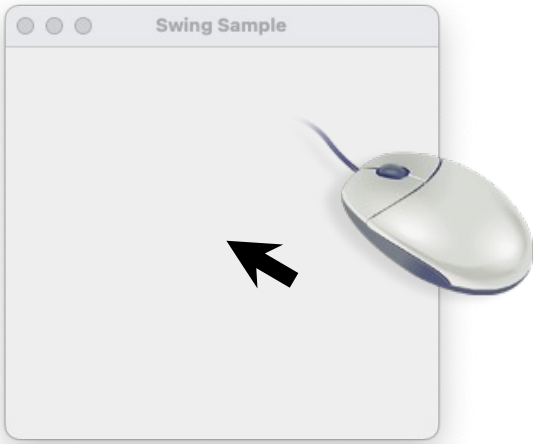
- マウスクリック

- キーボード入力

- 描画 (リスナ登録不要)

マウスイベント処理の例

```
public class Main12 {
    public static void main(String[] args) {
        …
        MouseCanvas canvas = new MouseCanvas();
        contentPane.add(canvas);
        …
    }
}
```

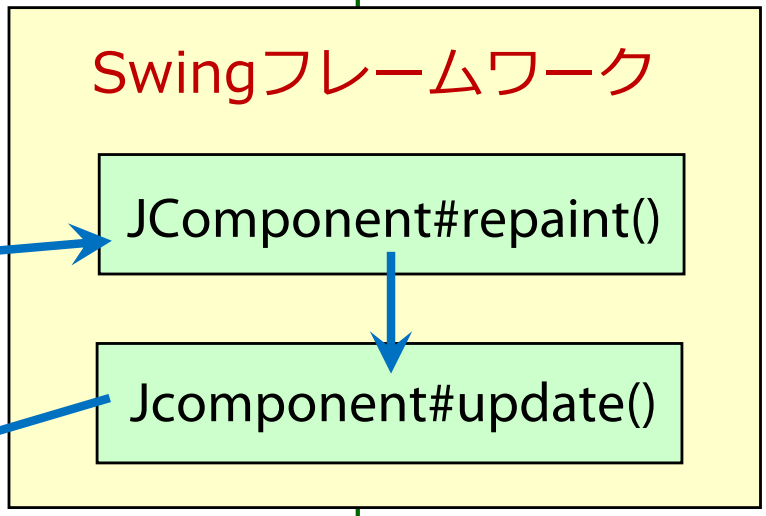



```
class MouseCanvas extends JPanel implements MouseListener {
    MouseCanvas() {
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent e) {
        System.out.println(e.getX() + " " + e.getY());
    }
    …
}
```

← イベントソースの登録

グラフィクスの描画

```
class MouseCanvas extends JPanel implements MouseListener {
    private int x = -1, y = -1;
    MouseCanvas() { addMouseListener(this); }

    public void mouseClicked(MouseEvent e) {
        x = e.getX(); y = e.getY();
        repaint();
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (x > 0 && y > 0) {
            g.setColor(Color.blue);
            g.drawOval(x - 15, y - 15, 30, 30);
        }
    }
}
```

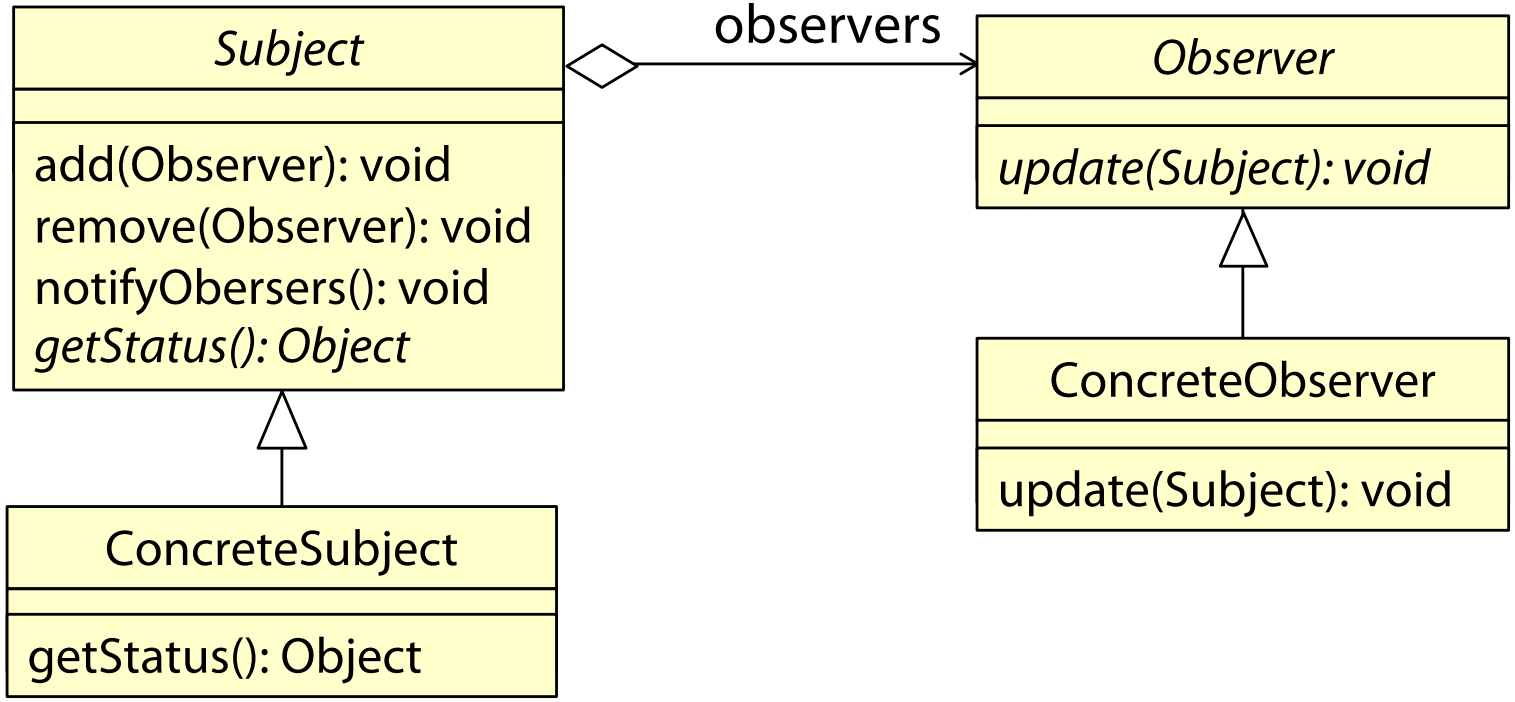


paintComponent()の再定義

Graphicsインスタンス
の取得

Observer Pattern

- イベント駆動モデルの標準的な実装方法
- パターンの概要
  - オブジェクトの状態変化を通知する
  - 被験者(*Subject*)と観測者(*Observer*)で構成



Observer Patternの適用例(1/3)

```
import java.util.*;

public abstract class Subject {
   protected List<Observer> observers = new ArrayList<>();

   public void add(Observer o) {
      observers.add(o);
   }
   public void remove(Observer o) {
      observers.remove(o);
   }
   public void notifyChange() {
      for (Observer o : observers) {
         o.update(this);
      }
   }
   public abstract Object getStatus();
}
```

Observerの追加

Observerの削除

追加されているObserverの
updateを呼び出す

```
public abstract class Observer {
  public abstract void update(Subject s);
}
```

Observer Patternの適用例(2/3)

```
public class Counter extends Subject {
   private int value;

   public Counter() {
      value = 0;
   }
   public void inc() {
      value++;
      notifyChange();
   }
   public void reset() {
      value = 0;
      notifyChange();
   }
   public Object getStatus() {
      return value;
   }
}
```

Subjectから呼び出される

保持する値が変化したことを通知

```
public class NumericalObserver
  extends Observer {
   public void update(Subject s) {
      int value = (int)s.getStatus();
      System.out.println(value);
   }
}
```

```
public class GraphicObserver
  extends Observer {
   public void update(Subject s) {
      int value = (int)s.getStatus();
      for (int i = 0; i < value; i++) {
         System.out.print('*');
      }
      System.out.println();
   }
}
```

Observer Patternの適用例(3/3)

```
public class Main14 {
   public static void main(String[] args) {
      Counter counter = new Counter();
      NumericalObserver o1 = new NumericalObserver();
      GraphicObserver o2 = new GraphicObserver();
      counter.add(o1);
      counter.add(o2);

      counter.inc();
      counter.inc();
      counter.inc();

      counter.remove(o2);
      counter.reset();
      counter.inc();
      counter.inc();
   }
}
```

Subjectの作成

Observerの作成

Observerの作成

Observerの追加

Observerの削除

実行結果

```
1
*
2
**
3
***
```

```
0
1
2
```

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

次回以降は小テストは行いません