発展プログラミング演習II

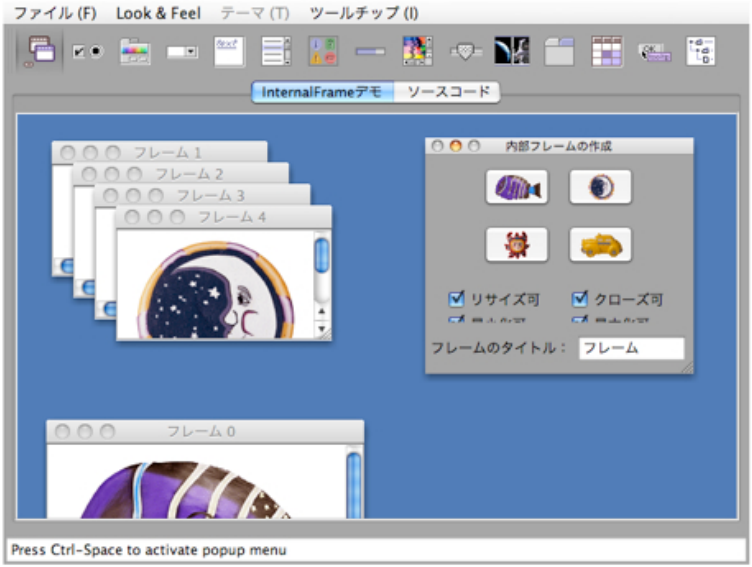
12. GUI

GUI: Graphical User Interface

- WIMP
  - ウィンドウ（Window）
  - アイコン（Icon）
  - メニュー（Menu）
  - ポインティングデバイス（Pointing device）
- GUI部品をマウスで操作する
  - ボタン、テキストエリア、スライダー、etc.

GUI部品の例

- SwingSet2を実行して確認できる



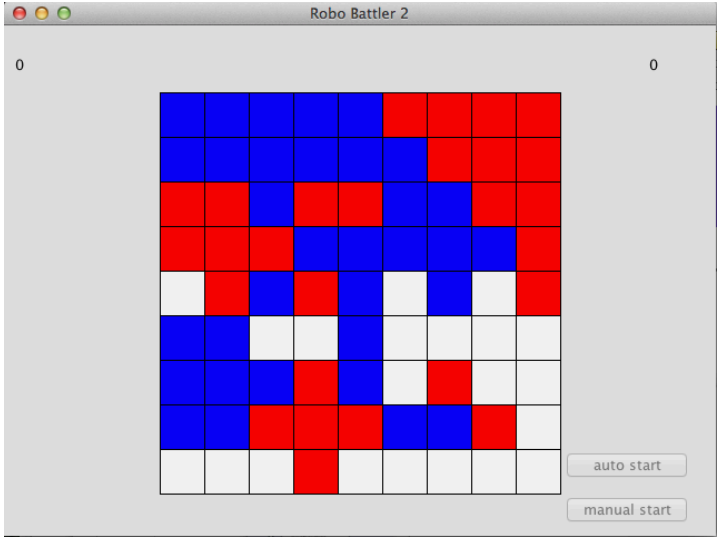
JavaでのGUIライブラリ

- AWT（Abstract Window Toolkit） java.awt
- Swing javax.swing
- SWT（Standard Widget Toolkit）

- 本授業ではSwing（とAWT）を使う

今年度のお題

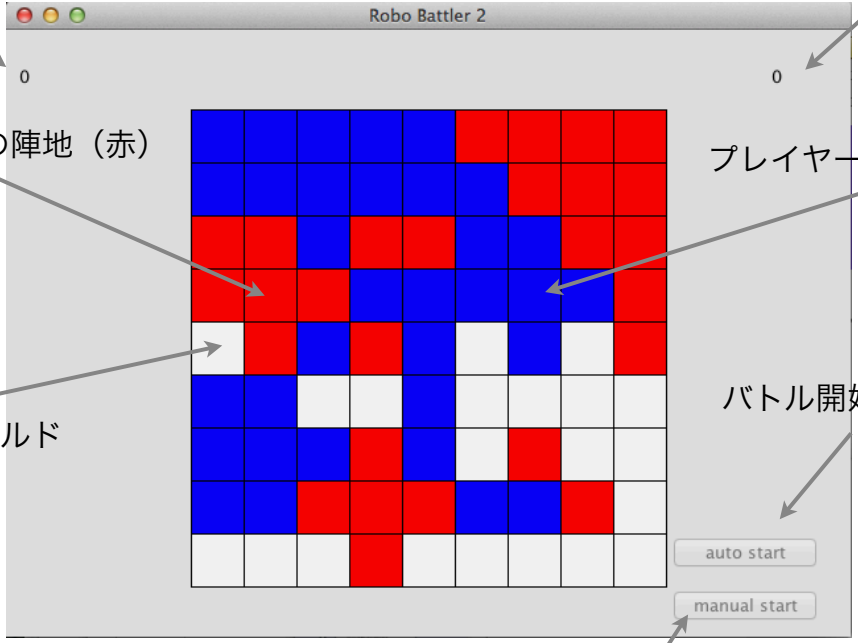
・ オリジナルゲーム「ロボバトラー２」を作る
（と言っても、細かい部分までは作りません）



ロボバトラーの実行方法

- テスト用のディレクトリを作成する
- moodleからrobobattler2.jarとRandomRobo.classをダウンロードして同じディレクトリに入れる
- java -cp robobattler2.jar:. robobattler2/RoboBattler RandomRobo RandomRobo で起動
- manual startボタンでバトル開始、スペースキーで1手ずつ動く
  - auto startボタンは勝敗がつくまで自動的に進む

ゲーム画面の見方



プレイヤー１の勝利数

プレイヤー２の勝利数

プレイヤー１の陣地（赤）

プレイヤー２の陣地（青）

バトルフィールド

バトル開始ボタン（自動）

バトル開始ボタン（手動）

ルール（概要）

- 2体のロボプログラムが戦う（ロボットは出てきません）
- バトルフィールドは9×9の升目
- 交互にミサイルを撃ち、フィールドを自分の陣地にする
  - ミサイルは爆撃範囲の異なる5種類がある
  - 爆撃範囲のマス目は自陣となる
  - ミサイルの順番はバトル開始時にランダムに決まる
- お互いに16発のミサイルを撃ち終わって自陣の多い方が勝ち
- 詳細は説明書を参照のこと

Step1 ウィンドウを表示する

RoboBattler2.java

```
import javax.swing.*;

class RoboBattler2 {

  RoboBattler2() {
    JFrame frame = new JFrame();
    frame.setTitle("Robo Battler");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(640, 480);
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    RoboBattler2 app = new RoboBattler2();
  }
}
```

確認ポイント1

- Javaではすべてのプログラムはクラス内に書く
  - import　パッケージの使用を宣言
- コマンド java クラス名 で指定したクラスの pubic static void main(String[]) メソッドが最初に呼ばれる
- new クラスインスタンスを作る
  →コンストラクタを呼ぶ

確認ポイント2

- メソッドの呼び出し
  オブジェクト式.メソッド名(引数)
  - どんなメソッドが使えるかはAPIリファレンス（javadoc）参照

JFrame

- JFrameクラス：輪郭とタイトルバーを持つことができるトップレベルのウィンドウ
  - setVisibleメソッドで可視状態にするまで表示されないことに注意
- イベント処理ループが自動的に作成され実行されるので、プログラムは勝手に終了しない
  →Step6参照

Step2 描画用の部品を作る

FieldPanel.java

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

class FieldPanel extends JPanel {
  FieldPanel() {
    setBackground(Color.WHITE);
  }
}
```

Step 2. （続き）

RoboBattler2.javaに赤文字部分を追加する

```
class RoboBattler2 {
  private FieldPanel   FieldPanel;

  RoboBattler2() {
.....

    frame.setSize(640, 480);

    fieldPanel = new fieldPanel();
    fieldPanel.setBounds(140, 60, 361, 361);
    frame.getContentPane().setLayout(null);
    frame.getContentPane().add(fieldPanel);

    frame.setVisible(true);

.....
```

360だとはみ出して線が描かれないので361にしている

確認ポイント

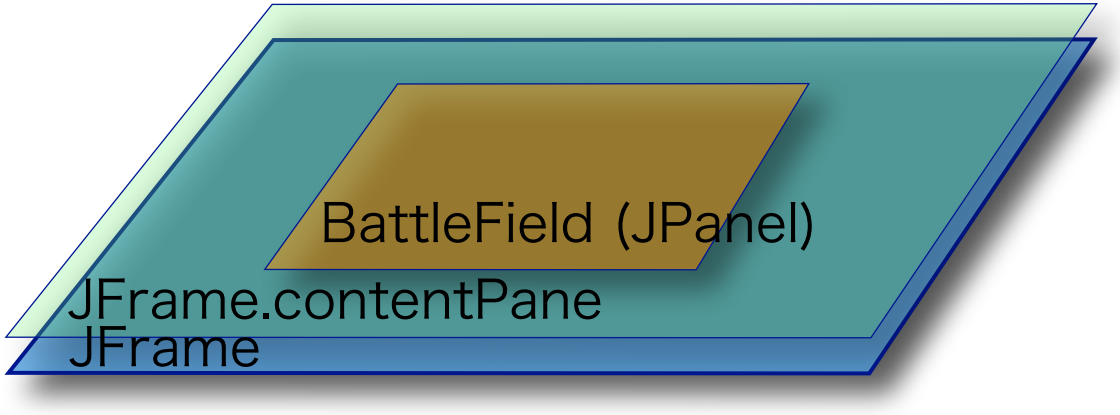
- extends　クラスを継承してサブクラスを作る
  - JPanelをカスタマイズして使用する
    - 具体的には描画処理をオーバーライドする

JPanel

- JPanelクラス：汎用のコンテナ
  - ここでは描画できる板として使用

GUI部品の配置

- 親子構造：親の部品の上に子の部品を配置する(addメソッド)
  - JFrameは特殊。直接には部品を配置できない。JFrame.contentPane上に配置する



Step3 枠を描いてみる

FieldPanel.javaに次のメソッドを追加する

```
.....

   protected void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.setColor(Color.BLACK);
      g.drawRect(0, 0, 360, 360);
   }

.....
```

なぜprotectedが必要か？

確認ポイント

- メソッドのオーバーライド
- protected修飾子

paintComponentメソッド

- 部品を描くために自動的に呼ばれる
  - 表示が必要になったとき
  - 再表示を要求したとき
- Graphics g: 絵を描くための筆（グラフィックコンテキスト）が渡されてくる
- 描きたい内容をこの中にプログラムする
  JPanel#paintComponentをオーバーライド

paintComponentの注意

- プログラムで直接paintComponentメソッドを呼んではいけない
- GUI部品を再描画したい時はrepaintメソッドを使う。
  - 再描画を要求するイベントを登録する。然るべきタイミングでpaintComponentメソッドが呼ばれる。（→Step 6）

Graphicsクラス

- Graphics g : グラフィックコンテキスト
  - setColor(Color c)　c色にする
  - drawRect(int x1, int y1, int width, int height)
    (x1, y1)から幅width高さheightの四角を描く
  - 部品の座標で指定していることに注意!!

オーバーライドの問題

- オーバーライドしたメソッドが呼ばれるので
  上位クラスで定義された描画処理が行われない
  - 背景色の設定 setBackground(Color.WHITE);
- →オーバーライドしたメソッドの最初で
  super.paintComponent(g);
  を呼んでおく

Step4 升目を描く

- Step3の枠線（四角）の中に升目を描く
- まず、縦線を、40ピクセル間隔で描く
  - Graphics.drawLineメソッドを使う
- 同様にして横線も描く

FieldPanel.java

```
.....

    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLACK);
        g.drawRect(0, 0, 359, 359);

        for (int x =          ; x <          ; x +=      ) {
            g.drawLine(x,          , x,          );
        }
    }

.....
```

確認ポイント

- for文の書き方
- ローカル変数
  カウンタ変数のような局所的に使う変数は
  使う範囲だけで有効なように宣言するべき

Step5 得点を表示する

- 得点を表示する＝値を文字で描く
- 文字を描くには
  - GUI部品に直接描画する
  - 専用のGUI部品を使う
    （ラベル、テキストフィールドなど）
- ここではJLabelを使う

ラベル

- 短いテキスト文字列の表示領域
  - 画像も表示可能
- 入力イベントには反応しない
  ＝ユーザは内容を変更できない
- setTextメソッドを使って内容を書き換える
  ことはできる

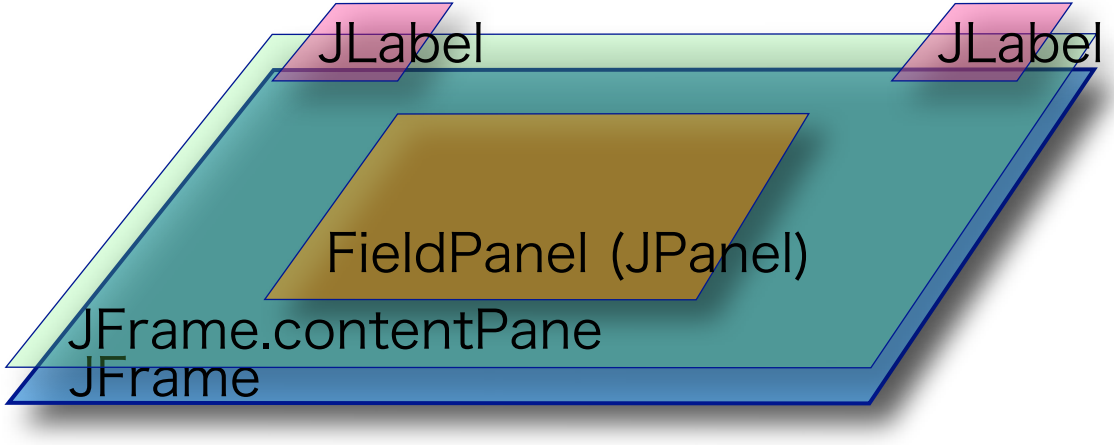
ラベルを追加する

RoboBattler2.javaに赤文字部分を追加する

```
class RoboBattler2 {
  private FieldPanel   fieldPanel;
  private JLabel[] scoreLabels = new JLabel[2];

  RoboBattler2() {
.....

    scoreLabels[0] = new JLabel("0");
    scoreLabels[0].setBounds(10, 10, 50, 50);
    frame.getContentPane().add(scoreLabels[0]);
    scoreLabels[1] = new JLabel("0");
    scoreLabels[1].setBounds(580, 10, 50, 50);
    frame.getContentPane().add(scoreLabels[1]);

    frame.setVisible(true);
}
```

GUI部品の配置（再掲）

- 親子構造：親の部品の上に子の部品を配置する(addメソッド)
- コンテナ：他の部品を配置するための部品



Step6 ボタンをつける

- 戦闘開始ボタンをつける
  - GUI部品のボタンを追加する
  - ボタンがクリックされたときの処理を書く

Step6.1 ボタンを追加する

RoboBattler2.javaに赤文字部分を追加する

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

class RoboBattler2 {
  private FieldPanel   fieldPanel;
  private JLabel[] scoreLabels = new JLabel[2];
  private JButton manualStartButton;

  RoboBattler2() {
.....

    manualStartButton = new JButton("manual start");
    manualStartButton.setBounds(500, 420, 120, 30);
    manualStartButton.setFocusable(false);
    frame.getContentPane().add(manualStartButton);

    frame.setVisible(true);
}
```

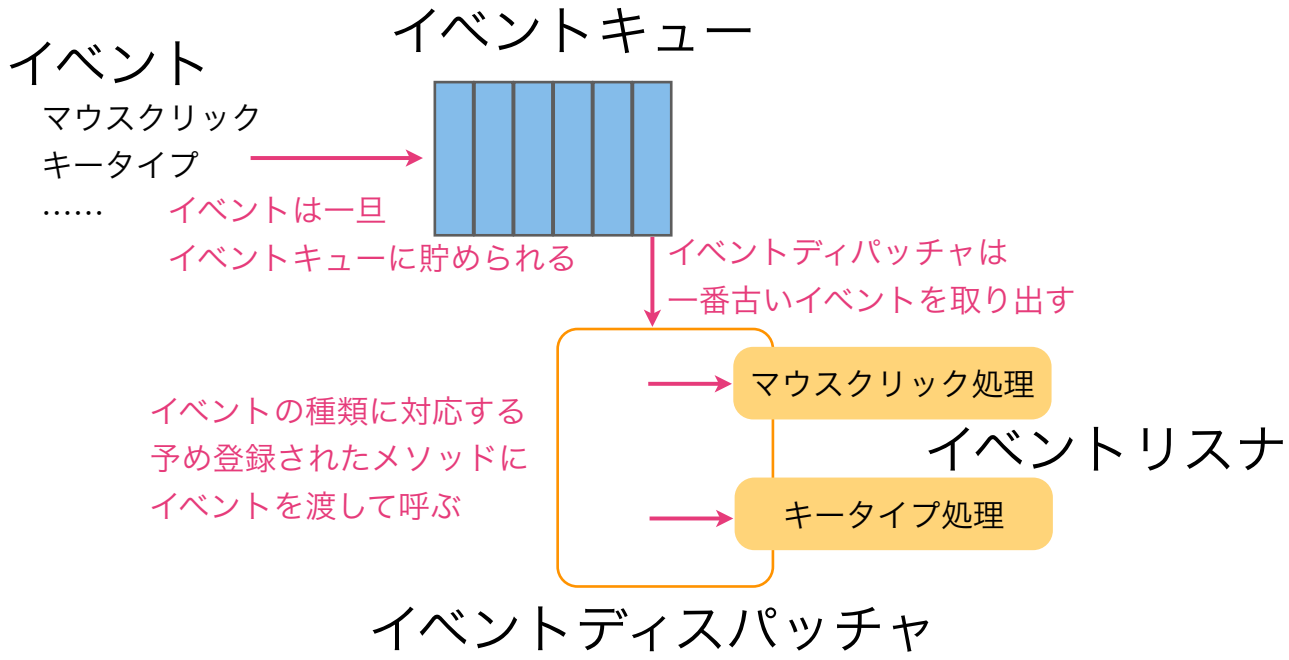
イベント駆動 event-driven

- ユーザの操作やプログラムの実行によって発生したイベントに対応して処理を行う実行形式
- イベント event：
  マウスクリックなどの事象
- イベントループ event-loop：
  イベントの発生を待ち続ける
  - 自動的に用意される
  - プログラムを明示的に終了する必要がある

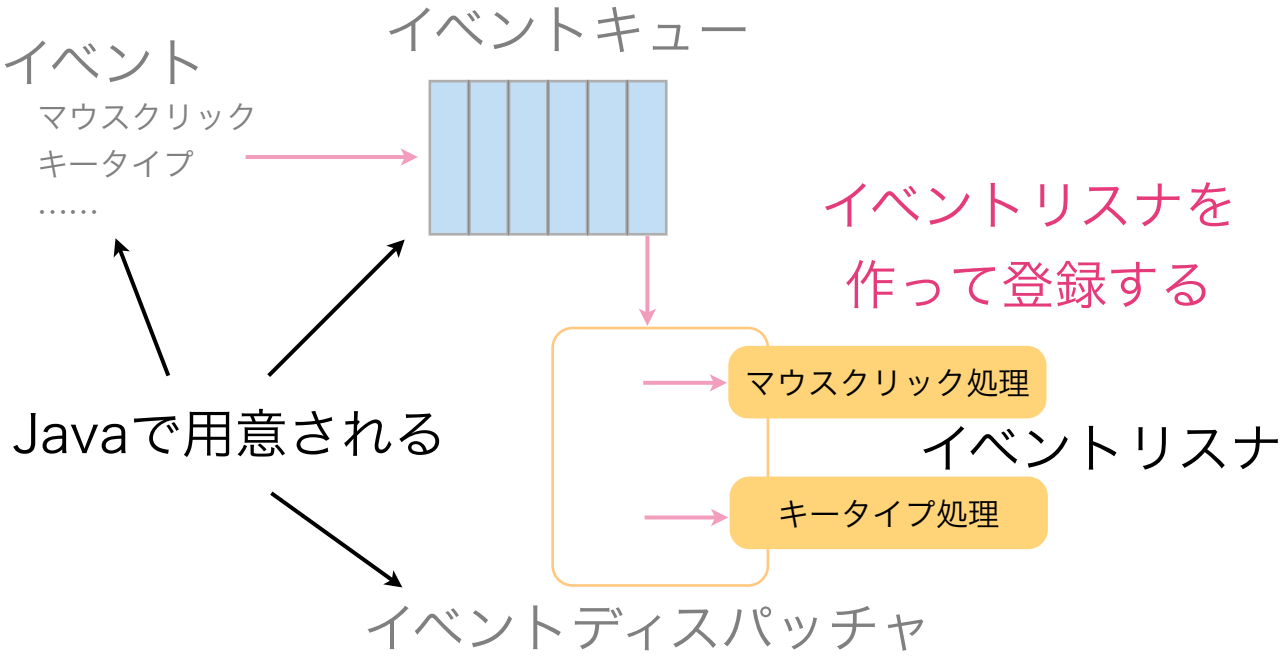
イベント駆動（続き）

- イベントハンドラ event-handler：
  イベントが発生したときに実行する処理
- イベントキュー event queue：
  処理待ち中イベントのバッファ
- イベントディスパッチャ event-dispatcher：
  発生したイベントに応じてイベントハンドラを呼ぶ仕組み

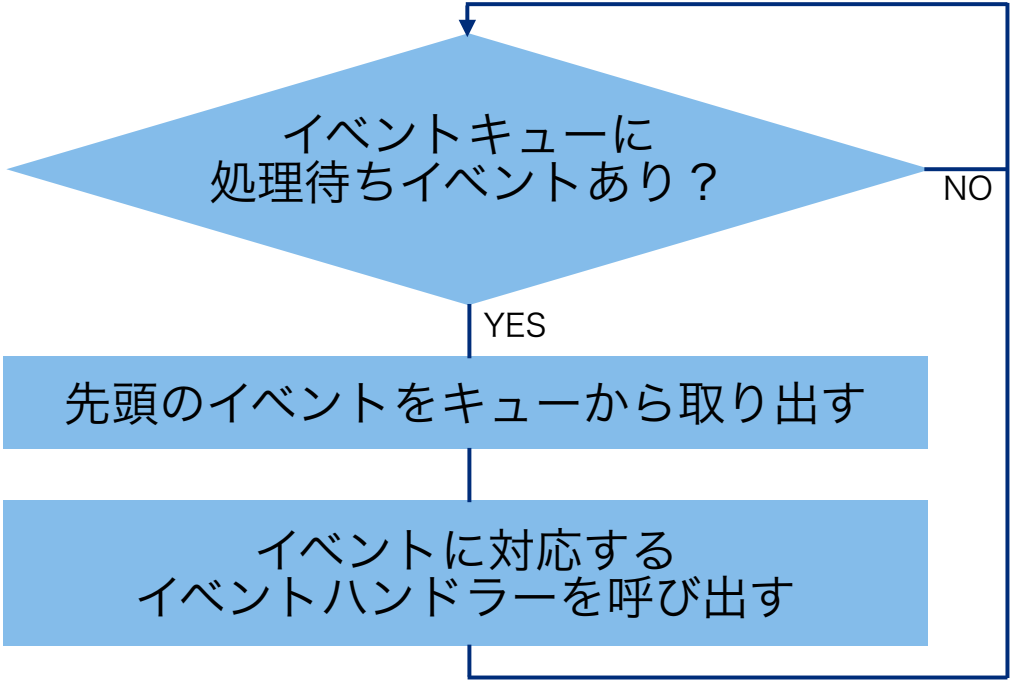
図解・イベント処理


イベントキュー
イベント
マウスクリック
キータイプ
……
イベントは一旦
イベントキューに貯められる
イベントディパッチャは
一番古いイベントを取り出す
イベントの種類に対応する
予め登録されたメソッドに
イベントを渡して呼ぶ
マウスクリック処理
キータイプ処理
イベントリスナ
イベントディスパッチャ

プログラマがするべきこと



イベントループの基本構造



主なイベントの種類

- （java.awt.eventパッケージを参照）
- MouseEvent
  マウス操作（マウスボタン、移動など）
- KeyEvent
  キーボード操作
- WindowEvent
  ウィンドウの状態変化
- ActionEvent
  コンポーネントが定義するアクション
  （例：ボタンが押された）

イベントリスナー

- イベントハンドラーのためのインタフェース
- MouseEvent
  → MouseListener, MouseMotionListener
- KeyEvent
  → KeyListener
- WindowEvent
  → WindowListener, WindowFocusListener, WindowStateListener
- ActionEvent
  → ActionListener

ボタンの処理を追加する

- JButton#addActionListenerに
  ActionListenerインタフェースを実装した
  クラスオブジェクトを引数で渡して登録する
- ActionListener#actionPerformedに
  処理内容を追加する
  - 現時点では何かメッセージを表示しておく

Step6.2 イベントリスナの登録

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

.....

RoboBattler2() {
  .....
  manualStartButton.setFocusable(false);
  manualStartButton.addMouseListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.out.println("battle start!!”);
    }
  });
  frame.getContentPane().add(manualStartButton);
}
.....
```

- 無名クラス

確認ポイント

- GUI部品の作成、配置、イベントリスナの登録

Step 7 ゲームの機能を追加する

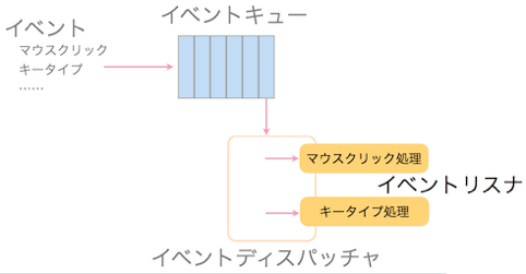
- ゲームの進行に必要なクラスを追加
  - BattleManager　戦闘を管理するクラス
  - Field　陣地の状態を管理するクラス
  - Robo　ロボの抽象クラス
  - Missile　ミサイルの抽象クラス
    - LineMissile, XMissile, ...などのサブクラス
- パッケージ化しておく
- （この段階のプログラムはまだ未完成）

パッケージ

- パッケージ名を宣言する
  - ソースコードの先頭に `package パッケージ名;`
- ディレクトリをパッケージの階層と同じように作成する
- コンパイルする `javac パッケージのディレクトリ/.../ソースファイル名`

Step8 キーイベント処理を追加する

RoboBattler2.javaのコンストラクタに
赤文字部分を追加する




```
RoboBattler() {
.....

  frame.addKeyListener(new KeyAdapter() {
      public void keyTyped(KeyEvent e) {
        System.out.println("key typed: " + e.getKeyChar());
        manager.step();
        fieldPanel.repaint();
      }
    });

  frame.setVisible(true);
}
```

アダプタクラス

- イベントリスナインタフェースの問題点：
  インタフェースを実装するクラスは
  すべてのメソッドを実装しなくてはいけない
- アダプタクラス
  - 空のメソッドを提供
  - プログラマは継承して使う
  - 必要なメソッドのみオーバーライドすればよい

Step9 自分のロボを作る

- Roboは抽象クラス
- Roboクラスを継承し、
  actionメソッドを実装する

javadoc

- /** */の形式のコメントの内容を元に
  ドキュメントを作成してくれる
- オプションを付けなければprotectedとpublic
  のクラスとメンバのみが対象

.jarファイル

- アーカイブファイル
- 作成コマンド

```
jar -cf アーカイブファイル名 アーカイブするファイル
```

- ライブラリ化

RandomRobo.java

```
import robobattler2.*;

public class RandomRobo extends Robo {
  protected void action(robobattler2.Missile missile) {
    int x = (int)(Math.random() * Field.SIZE);
    int y = (int)(Math.random() * Field.SIZE);
    fire(x, y);
  }
}
```

何も考えず適当に撃つだけのロボ

コンパイルと実行

- コンパイル

```
javac -cp robobattler2.jar RandomRobo.java
```

- 実行

```
java -cp robobattler2.jar:. robobattler2/RoboBattler2 RandomRobo RandomRobo
```

- 手動バトル開始ボタンで戦闘開始、
  何かキーを押すと互いに1手ずつ動く

Step10 自動化する*

- キーを毎回押さなくても
  自動的に進むようにしたい
- 自動バトル開始ボタンを追加する

スレッド thread

- プログラムの制御の流れ
- Java言語では一つのアプリケーションが
  複数のスレッドを利用できる
- それぞれのスレッドは並行して処理を進める
  - 同期に注意。複数のスレッドが同じデータの
    値を同時に書き換えたらまずいことになる**

スレッドの作成方法

- 2つの方法（Threadクラスの説明を参照）
  - Threadクラスを継承し、
    runメソッドをオーバーライドする
  - Runnableインタフェースを実装し、
    runメソッドを実装する
- ここでは後者の方法を説明する

基本構造

```
.....

class RoboBattler2 implements Runnable {
....
    void start() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        // スレッドの処理内容を書く

    }
}
```

- runメソッドを実装する
  - publicが必ず付くことに注意
- Threadに自分を渡してインスタンスを作り、startを呼ぶ

runメソッドの基本内容

```
public void run() {
    try {
        manager.step();
        battleField.repaint();
        Thread.sleep(INTERVAL_TIME);
    } catch (InterruptedException ie) {
        break;
    }
}
```

- Thread#sleep　指定された時間（ミリ秒）待つ
  - 例外InterruptedExceptionを投げうる
    → try - catch で囲む

自動実行を止めるために

```
public void run() {
    // スレッドの処理内容を書く

    Thread thisThread = Thread.currentThread();
    while (thisThread == thread) {
        try {
            manager.step();
            fieldPanel.repaint();
            Thread.sleep(INTERVAL_TIME);
        } catch (InterruptedException ie) {
            thread = null;
            break;
        }
    }
}

void stop() {
    thread = null;
}
```

- Thread#stopを使ってはいけない

自動実行の開始と停止

- 開始
  - 自動戦闘開始ボタンのイベントハンドラでstartを呼ぶ
- 停止
  - 戦闘終了時にはsetScoreがBattleManagerから呼ばれるので、この中でstopを呼ぶ