Overview of Java

Real-Time Systems, Lecture X

Martina Maggio

18 January 2017

Lund University, Department of Automatic Control
www.control.lth.se/course/FRTN01

Content

[Real-Time Control System: Chapter 6.4]

Introduction

Aim of the Lecture

- Present the basic ideas and concepts behind Java.
- Show how a simple Graphical User Interface (GUI) can be implemented.

Further Sources

- Essentials of the Java programming language Part I[1] and II[2].
- Per Holm, *Objektorienterad programmering och Java*, Studentlitteratur, 1998.
- Arnold and Gosling, *The Java Programming language*, 4th edition[3], 2005.
- Java links and material on the course home page.

[1] http://www.oracle.com/technetwork/java/index-138747.html
[2] http://www.oracle.com/technetwork/java/basicjava2-138746.html
[3] Online from http://www-public.tem-tsp.eu/~gibson/Teaching/CSC7322/

Overview

- Developed at Sun Microsystems (by Gosling et al.);
- Sun acquired Oracle in 2009;
- Syntax from `C` and `C++`, semantics from `Simula`
- General object-oriented language;
- First release in May 1995, Latest release Java 8 in 2014;
- Java 9 scheduled for March 2017;
- Originally intended for consumer electronics;
- Later remarkted as a WWW-language;
- Programs are compiled into bytecode and interpreted by the Java Virtual Machine (JVM), platform independent.

Applications, Applets and Servlets

- Java applications are stand-alone programs;
- Java applets run within Java-enabled browsers;
- Java servlets run within network servers.

The Java Language

- **Simple**: easy to learn, based on today's practice (borrow C++ syntax), small size, automatic garbage collection;
- **Network-oriented**: full support for networked applications, TCP/IP protocols (FTP, HTTP), open, access objects across the net;
- **Portable**: architecture-netural, byte code interpreter;
- (more) **Strongly typed** (than C): compile-time checks, safe references instead of unsafe pointers;
- **Secure**: the bytecode verifier verifies the code before it is interpreted, the security manager in the JVM check access to resources like file system and network.

The Java Language

Hello World

```java
class HelloWorld {
  public static void main(String[] arguments) {
    System.out.println("Hello World, I print!");
  }
}
```

The main method must be declared in the class that is started from the command line (with 'java classname').

Compile and execute:

```
$ javac HelloWorld.java
(creates HelloWorld.class)
$ java HelloWorld
HelloWorld, I print!
```

A source file may contain only one public class. The file should have the same name as the public class with the java extension. The output of the compilation is the file named as the class with the extension `.class`.

Comments

```
1  //  Line comments extend to the end of the line
2
3  /*  A multi-line comments,
4      that continues on multiple lines */
5
6  /** Documentation comment,
7      to be used only immediately before a class or method declaration
8      used by the javadoc tool **/
```

Simple Declarations

```
1   int m, n;          // integer variables
2   double x, y;       // real variables
3   double z = 0.89; // initialization
4   boolean b;         // true or false
5   char ch;           // single character
```

Expressions

```
1   int m, n;          // integer variables
2   double x, y;       // real variables
3   double z = 0.89; // initialization
4   boolean b;         // true or false
5   char ch;           // single character
6
7   n = 3 * (5 + 1);
8   x = y / 1.4;
9   n = m % 8;         // modulo 8
10  b = true;          // = for assignment
11  b = (n == m);      // == for comparison
12  ch = 'x';
```

Explicit Type Casting

```
1  double radians;
2  int degrees;
3
4  degrees = radians * 180 / 3.14; // error:
5  // incompatible types: possible lossy conversion from double to int
6  degrees = (int) (radians * 180 / 3.14); // ok
```

Java Statements

Similar to other languages, statements can be grouped with curly braces: { and }. In Modula-2: `BEGIN` and `END`.

Conditional Statements:
boolean condition within parenthesis, no then keyword

```
1  if (n == 3)
2    x = 3.0;
3  else {
4    x = 4.0;
5    System.out.println("Else");
6  }
```

```
1  if (n != 3) {
2    x = 3.0;
3    y = 7.0;
4  }
5  else {
6    x = 4.0;
7    y = 2.0;
8  }
```

can use logical operators: and (&&), or (||), not (!)

Java Statements

While, For and DoWhile Statements:

```
1  double sum = 0.0;
2  double term = 1.0;
3  int k = 1;
4  while (term >= 0.00001) {
5    sum = sum + term;
6    term = term / k;
7    k++;
8  }
```

```
1  double sum = 0.0;
2  int i;
3  for (i = 1; i <= 100; i++) {
4    sum = sum + 1.0 / i;
5  }
```

```
1  double x = 0.0;
2  double y = 20.0;
3  do {
4    x = x + 1.0;
5    y = y / x;
6  } while (y > 3.0);
```

Java Statements

Case statement:

```
1   switch (x) {
2     case 1:  y = 0;
3              z = 1;
4              break;
5     case 2:  y = 1;
6              z = 0;
7              break;
8     case 3:  y = 1;
9              z = 1;
10             break;
11    default: y = 0;
12             z = 0;
13             break;
14  }
```

Object-Oriented Programming

- **Class**: a structure that defines the data (state) and the operations (methods) that can be performed on that data (the behavior). Abstract data type. A class without methods corresponds to an ordinary record (Pascal) or struct (C).
- **Object**: an *instance* (or object) is an executable copy of a class.

Classes and objects provide **modularity** and **information hiding**.

Object-Oriented Programming

- **Inheritance**: a class inherits state and behavior from its superclass (single inheritance) or superclasses (multiple inheritance). Subclasses can add state and behavior. Subclasses can *override* (redefine) inherited state and behavior. In case of single inheritance the classes form an *inheritance tree* (class hierarchy).

Inheritance supports **re-usability** and provides a mechanism for organizing and structuring software.

Object-Oriented Programming

- **Polymorphism**: a subtype is a datatype that is related to another datatype (the supertype) by some notion of substitutability, meaning that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype. If `Rectangle` is a subtype of `Figure`, any term of type `Rectangle` can be safely used in a context where a term of type `Figure` is expected.

There is much more to polymorphism (for example parametric polymorphism) but in object-oriented programming polymorphism is usually *subtype polymorphism*.

Class: Attributes

A **class** declaration contains a set of attributes (fields or instance variables) and functions (methods). Attributes:

```
1  class Turtle {
2    // attributes declaration
3    private boolean penDown;
4    protected int x, y;
5  }
```

- `private` cannot be accessed from outside the class
- `protected` can be accessed from within the class and all its subclasses but not from the outside
- `public` can be accessed from the outside

Class: Methods

A **class** declaration contains a set of attributes (fields or instance variables) and functions (methods). Methods:

```
1  class Turtle {
2    // attributes
3    protected int x, y;
4
5    // methods
6    public void jumpTo (int newX, int newY) {
7      x = newX; y = newY;
8    }
9    public int getX () { return x; }
10 }
```

- `public` means they can be accessed from outside

Class: Constructors

A **constructor** is a special method that has no return type and is called when an object is created. It is possible to have multiple methods with the same name and different parameter sets (often used for constructors) — as long as the signatures are different.

```
1  class Turtle {
2    // attributes
3    protected int x, y;
4
5    // constructors
6    public Turtle () { x = 0; y = 0; } // no params
7    public Turtle (boolean initial) { // one boolean
8      if (initial) { x = 5; y = 5; }
9      else { x = -5; y = -5; }
10   }
11   public Turtle (int startX, int startY) { // two int
12     x = startX; y = startY;
13   }
14 }
```

Class: Example

```
1   class Turtle {
2     protected int x, y; // attributes
3
4     // constructor
5     public Turtle (int startX, int startY) {
6       this.jumpTo (startX, startY);
7     }
8     // methods
9     public void jumpTo (int newX, int newY) { x = newX; y = newY; }
10    public int getX () { return x; }
11  }
12
13  class Main {
14    public static void main(String[] args) {
15      Turtle t = new Turtle(10, 10);
16      a.jumpTo(t.getX(), 5);
17    }
18  }
```

- the object that the method belongs to can be accessed using `this`

Class: Static Attributes

By preceding the declaration of an attribute with `static` the attribute becomes a class variable, all the instances share the same copy of the class variable.

```
1  class Turtle {
2    // attributes
3    protected int x, y;
4    static public int numTurtles = 0;
5    // methods
6    public Turtle (int startX, int startY) {
7      x = startX; y = startY;
8      numTurtles++;
9    }
10 }
11 class Main {
12   public static void main(String[] args) {
13     Turtle t1 = new Turtle(9, 5); Turtle t2 = new Turtle(10, 10);
14     System.out.println("num = " + t1.numTurtles); // num = 2
15     System.out.println("num = " + t2.numTurtles); // num = 2
16   }
17 }
```

Class: Static Methods

The keyword `static` can be used also for methods. Static methods (class methods) can only access static variables (class variables). Class methods are accessible from the class itself (in addition to from each instance of the class). An example is the `Math` class.

```
1  double x = 9.0;
2  double y = Math.sqrt(x); // 3.0
```

Nested Classes

A class that is defined as a member of another class is called a **nested class**. It has unlimited access to its enclosing class members, even if they are declared private. A nonstatic nested class is known as an **inner** class (the most usual form of nested classes). Inner classes can be anonymous (have no name). An inner class can only be accessed via an instance of the outer class.

```
1  class OuterClass {
2    // attributes
3    private String name;
4    // methods
5
6    class InnerClass {
7      // attributes
8      // methods
9      public void printName() {
10       System.out.println("Attribute name of the outer class: " + name);
11     }
12   }
13 }
```

Subclasses

A subclass `extends` a superclass:

```
1  class NinjaTurtle extends Turtle {
2    // attributes
3    String name;
4
5    // constructors
6    public NinjaTurtle (int initX, int initY, String initName) {
7      super(initX, initY); // call constructor of super class
8      name = initName;
9    }
10 }
```

Subclasses

A subclass can override methods of a superclass (not possible for `static` or `final` ones):

```
1  class Turtle {
2    ...
3    public Turtle(int initX, int initY);
4    public void jumpTo (int newX, int newY) { x = newX; y = newY; }
5  }
6
7  class NinjaTurtle extends Turtle {
8    String name; // attributes (+ attributes in superclass)
9
10   // constructor (+ constructors in superclass)
11   public NinjaTurtle (int initX, int initY, String initName) {
12     super(initX, initY); name = initName;
13   }
14   public void jumpTo (int newX, int newY) { // method overriding
15     super.jumpTo(newX, newY); // it is possible to call the superclass
16     System.out.println(name + " jumped to a new position");
17   }
18 }
```

Abstract Classes

Methods can be **abstract**. They then contain only declarations without any implementation. The method must be implemented in the the subclasses. A class with an abstract method is an abstract class and must be decleared `abstract`. An abstract class cannot be instantiated, but it is possible to instantiate a subclass that implements the abstract methods. Sometimes it is useful to declare a class abstract even if there is no abstract methods, to structure the hierarchy.

```
1  abstract class Drawable {
2    public abstract void draw();
3  }
```

Interfaces

A way of obtaining some of the functionality of multiple inheritance without is problems. An interface defines a set of methods. Any class may implement that interface (and a class can implement several interfaces). Reference variables may be typed either by class or by interface.

- An interface may contain only methods and constant declarations.
- All methods are implicitly `public` and `abstract`.
- A class can implement several interfaces, but can extend only another class.
- An interface can be used as a type name.
- Interfaces can be extended like classes. An interface can extend more than one interface.

Example

```
1  interface Drawable {
2    void draw (SimpleWindow w);
3    int getWidth();
4    int getHeight();
5  }
6
7  class Rectangle implements Drawable {
8    private int xSide, ySide;
9
10   public void draw(SimpleWindow w) {
11     int x = w.getX(); int y = w.getY();
12     w.lineTo(x, y + ySide);           // draw the first line
13     w.lineTo(x + xSide, y + ySide);   // draw the second line
14     w.lineTo(x + xSide, y);           // draw the third line
15     w.lineTo(x, y);                   // close the rectangle
16   }
17   public int getWidth() { return xSide; }
18   public int getHeight() { return ySide; }
19 }
```

Example

```
1  interface Drawable {
2    void draw (SimpleWindow w);
3    int getWidth();
4    int getHeight();
5  }
6
7  class Person implements Drawable {
8    private String name;
9
10   public void draw(SimpleWindow w) {
11     w.writeText(name);
12   }
13   public int getWidth() {
14     return 6 * name.lenght();
15   }
16   public int getHeight() {
17     return 10;
18   }
19 }
```

Example

```
1  interface Drawable {
2    void draw (SimpleWindow w);
3    int getWidth();
4    int getHeight();
5  }
6
7  // it is possible to write methods that work with every instance of
8  // a Drawable, like this drawWithBorder
9  void drawWithBorder (Drawable d, int x, int y SimpleWindow w) {
10   w.moveTo(x, y);
11   d.draw(w);
12   int width = d.getWidth();
13   int height = d.getHeight();
14   w.moveTo(x-2, y-2);
15   // do 4 calls to w.lineTo() to draw the border
16 }
```

Arrays

Arrays are similar to objects, accessed using reference variables.

```
1  int[] someInts; // integer array
2  Turtle[] turtleFamily; // array of references to turtles
3
4  someInts = new int[30]; // specify size when the array is created
5
6  int i;
7  for (i = 0, i < someInts.lenght; i++) { // lenght is predefined
8    someInts[i] = i * i; // indices start at 0
9  }
```

Call-by-value vs. Call-by-reference

- **Call-by-value**: when a parameter is passed to a method, Java passes it by value. For simple types, this means that the value of the `int` or `double` is passed and the method cannot modifies the value seen by the callee.
- In case an object is passed, things are more tricky. You can see it as if the object passed was a pointer, pointing to the address in which the object is stored. The pointer cannot be changed by the method, but the content of what is pointed by it can be changed. The pointer is passed with a **call-by-value** but its reference can be modified. So in some sense it can be seen as the **call-by-reference** of C.

Example

```
1  class Dog {
2    String name;
3    public Dog(String name) { this.name = name; }
4    public String getName() { return name; }
5    public void setName(String name) { this.name = name; }
6  }
7  class Main {
8    public static void main( String[] args ){
9      Dog aDog = new Dog("Max");
10     foo(aDog);
11     Boolean a = aDog.getName().equals("Fifi"); // ?
12     Boolean b = aDog.getName().equals("Max"); // ?
13   }
14   public static void foo(Dog parameter) {
15     Boolean c = parameter.getName().equals("Fifi"); // ?
16     Boolean d = parameter.getName().equals("Max"); // ?
17     parameter.setName("Fifi");
18     Boolean e = parameter.getName().equals("Fifi"); // ?
19     Boolean f = parameter.getName().equals("Max"); // ?
20   }
21 }
```

Example

```
1  class Dog {
2    String name;
3    public Dog(String name) { this.name = name; }
4    public String getName() { return name; }
5    public void setName(String name) { this.name = name; }
6  }
7  class Main {
8    public static void main( String[] args ){
9      Dog aDog = new Dog("Max");
10     foo(aDog);
11     Boolean a = aDog.getName().equals("Fifi"); // true
12     Boolean b = aDog.getName().equals("Max"); // false
13   }
14   public static void foo(Dog parameter) {
15     Boolean c = parameter.getName().equals("Fifi"); // false
16     Boolean d = parameter.getName().equals("Max"); // true
17     parameter.setName("Fifi");
18     Boolean e = parameter.getName().equals("Fifi"); // true
19     Boolean f = parameter.getName().equals("Max"); // false
20   }
21 }
```

Example

```
1  class Dog {
2    String name;
3    public Dog(String name) { this.name = name; }
4    public String getName() { return name; }
5    public void setName(String name) { this.name = name; }
6  }
7  class Main {
8    public static void main( String[] args ){
9      Dog aDog = new Dog("Max");
10     foo(aDog);
11     Boolean a = aDog.getName().equals("Fifi"); // ?
12     Boolean b = aDog.getName().equals("Max"); // ?
13   }
14   public static void foo(Dog parameter) {
15     Boolean c = parameter.getName().equals("Fifi"); // ?
16     Boolean d = parameter.getName().equals("Max"); // ?
17     parameter = new Dog("Fifi");
18     Boolean e = parameter.getName().equals("Fifi"); // ?
19     Boolean f = parameter.getName().equals("Max"); // ?
20   }
21 }
```

Example

```
1  class Dog {
2    String name;
3    public Dog(String name) { this.name = name; }
4    public String getName() { return name; }
5    public void setName(String name) { this.name = name; }
6  }
7  class Main {
8    public static void main( String[] args ){
9      Dog aDog = new Dog("Max");
10     foo(aDog);
11     Boolean a = aDog.getName().equals("Fifi"); // false
12     Boolean b = aDog.getName().equals("Max"); // true
13   }
14   public static void foo(Dog parameter) {
15     Boolean c = parameter.getName().equals("Fifi"); // false
16     Boolean d = parameter.getName().equals("Max"); // true
17     parameter = new Dog("Fifi");
18     Boolean e = parameter.getName().equals("Fifi"); // true
19     Boolean f = parameter.getName().equals("Max"); // false
20   }
21 }
```

References

- **References** are the same as pointers in C.
- Manipulation of references is not allowed (for example adding integers).
- Compile-time checks guarantee that a reference is initialized before it is used.

Memory Allocation

For **static** memory allocation, all the memory is allocated at start-up.
For **dynamic** memory allocation there are two alternatives:

- memory is allocated dynamically from the heap when needed (`Pascal`, `Modula-2`, `C`, `C++`, ...):
  - manual memory management,
  - the application explicity allocates memory when needed and deallocates it when no longer use it,
  - problem: dangling pointers, memory leaks, fragmentation;
- automatic memory management (`Java`):
  - runtime system or OS deallocates memory automatically,
  - garbage collection,
  - problem: takes time and may disturb the real-time application.

Java Garbage Collector

The Garbage Collector in Java:

- runs as a **low-priority** thread,
- it is **incremental**: work is divided into small pieces that are spread out during the execution,
- can be explicitly invoked using `System.gc()`,
- real-time garbage collectors have been developed (for example by Roger Henriksson) and are part of Sun's Real-Time Java 2.

Exceptions

Exceptions happen when error occurs. The exceptions contain information about the execution (type, state of the program). During the execution, the JVM tries to find code that handles the exception (it `throws` the exception to a method that can handle it – the runtime systems searchers backwords through the call stack of the method until it finds a method containint appropriate code to handle the exception – the handler has a `catch` for the exception – if no appropriate handler is found, the application terminates with an error).

Exceptions

Exception handling code is composed by at least a `try` and a `catch` or a `try` and a `finally` (or all three).

```
1  class Main {
2    public static void main(String[] arguments) {
3      try {
4        // code that can throw the exception
5        System.out.println("try");
6      } catch (Exception e) {
7        // code to handle the exception of type Exception
8        System.out.println("catch");
9      } finally {
10       // code that is executed anyway (used for cleanup)
11       System.out.println("finally");
12     }
13   }
14 }
```

Exceptions

The keyword `throw` can be used to raise an exception. A method can signal that it may throw an exception in its signature, by using `throws` and a list of comma-separeted types of exception.

```
1  class Main {
2    public static void main(String[] arguments) {
3      try {
4        System.out.println("try");
5        uselessMethod(15); // prints "I am happy"
6        uselessMethod(5); // throws the exception
7      } catch (Exception e) {
8        e.printStackTrace(); // prints the call stack
9        System.out.println("catch");
10     }
11   }
12   public static void uselessMethod(int parameter) throws Exception {
13     if (parameter < 10) throw new Exception("I wanted something else");
14     else System.out.println("I am happy");
15   }
16 }
```

Exceptions

Types of exceptions:

- `RuntimeException`
  exception that occurrs within the Runtime system like division by zero, the compiler does not require that these are caught or specified.
- Checked Exceptions
  checked by the compiler, requires that the exception is caught or specified. For example `wait()` and `sleep` can throw an `InterruptedException`.

It is possible to define new types of exceptions (not needed for this course).

Packages

Java code is often organized in **packages**. A package is a collection of related classes. Attributes with unspecified visibility get `package` visibility (they are public to other classes in the package and private to classes that do not belong to the package). A class that should be accessible from outside the package must be declared public. Classed declared in files without a package specification belong to a '`default`' package. All files belonging to the package should be stored in the same directory (named with the name of the package).

File `Rectangle.java`:

```
1  package Drawings;
2
3  public class Rectangle {
4    private double lenght, height;
5    public Rectangle(double l, double h) { lenght = l; height = h; }
6    public double getArea() { return height * lenght; }
7  }
```

Package Names

- `java.xxx`: standard java packages
  (`java.awt`, `java.awt.event`)
- `javax.xxx`: java extension packages
  (`javax.swing`)
- User-defined packages: globally unique names
  convention: reversed Internet domain name followed by local directory structure (`se.lth.cs.realtime`, `se.lth.control.realtime`)

Standard Packages

- `java.lang`: Object, Class, String, ...
- `java.io`: Streams, Random-Access Files
- `java.applet`: Applet
- `java.awt`: Abstract Window Toolkit
- `java.util`: Collections, Date, Time, ...
- `java.net`: Sockets, Telnet, URLs, ...

Package Access

- Explicit Naming

```
1  Gui.CommandButton button = new Gui.CommandButton();
```

- Import One Class

```
1  import Gui.CommandButton;
2  CommandButton button = new CommandButton();
```

- Import a Full Package

```
1  import Gui.*;
2  CommandButton button = new CommandButton();
```

Graphical Interfaces: Swing

Swing is a class package for graphical user interfaces implementation. It supports buttons, menus, scrollbars and more. All the class names begin with J (`JButton`, `JFrame`, ...).

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
```

Containers

Every program that presents a swing GUI contains at least one top-level swing container:

- JFrame: implements a single main window
- JDialog: implements a secondary, "pop-up" window

Intermediate containers simplify the positioning of components in the window:

- JPanel
- JScrollPane
- JTabbedPane

Atomic Components

Self-sufficient entities that present information to the user or implement some user control[4]

- JButton – a button
- JTextField – editable text field
- JLabel – uneditable text field
- JSlider – a slider
- PlotComponent – local plot class

[4] https://docs.oracle.com/javase/tutorial/uiswing/components/index.html

Layout Management

Controls the layout of components in an intermediate container.

- `BorderLayout`: composed by five areas (north, south east, west and center), default layout for every content pane;
- `BoxLayout`: puts components in a single row or column;
- `FlowLayout`: lays out components from left to right, starting new rows if necessary. default layout for JPanel;
- `GridLayout`: places components in a cell grid (matrix), all of the same size.

Event Handling

Each user action (key press, mouse movements, mouse clicks) is considered an event. The Swing system informs the Java application about an event by creating an object of class `AWTEvent`. The class has several subclasses, among which:

- `ActionEvent`: generated when a specific event for a certain component occurs (like mouse clicks);
- `TextEvent`: generated when the content of a text component is changed.

Listeners

When an event occurs, the runtime system calles a method in a listener object. A listener is an object that `implements` a listener interface. For example:

- `ActionListener`: contains `actionPerformed(ActionEvent e)` and is called when an `ActionEvent` object is generated;
- `TextListener`: contains `textValueChanged(TextEvent e)` and is called when a `TextEvent` object is generated.

Several components can listen to the same event. This is achieved by calling the method `addActionLinstener` witht the listener object as argument.

Event Methods

All event classes contain the method `Object getSource()` that returns the source object of the event. Some event classes contain event-specific methods like:

```
char getKeyChar(); // Key event
int getX(); int getY(); // Mouse event
```

Setting up Event Handlers

The event handler class `implements` a listener interface:

```
1  public class MyEventHandler implements ActionListener {
```

One should register an instance to the event handler class as a listener upon one or more components:

```
1  MyEventHandler handler = new MyEventHandler();
2  component.addActionListener(handler);
```

In the listener interface, one should implement the `actionPerformed` method:

```
1  public void actionPerformed(ActionEvent e) {
2    // code to react to the action
3  }
```

Event Dispatching Thread

Event-handling code executes in a single thread to make sure that events are processed sequentially and the method handling the reaction to an event terminates before handling the next action. This single thread is called *event-dispatching thread*.

Rule: All the code that might affect or depend on the state of a component should be executed by the event-dispatching thread.

Execute code in the event-dispatching thread

Applications usually need to perform (non user-related) operations on the graphical user interface after it is created (for example because of initialization or because of application-related events that do not depend on the user). Swing has two methods that can be used in this case:

- `invokeLater`: registers code to be executed by the event-dispatching thread and terminates,
- `invokeAndWait`: wait until the code executes.

Swing Example

A periodic thread that generates a sine wave. The data is sent to a GUI that plots the sine wave using a `PlotComponent`. The GUI also has two sliders that allow to change frequency and amplitude of the sine wave.

- Classes: `Main`, `Sinus` (the sine wave generator, `Monitor` (internal class of `Sinus`, `Opcom` (the GUI).
- User Threads: `main`, `Sinus` (extends the thread class), Swing event-dispatching thread.

Swing Example: Panels