Lecture X: Overview of Java

[6.4 + Java pointers on the home page]

- Language Overview
- GUIs with Swing

Aim of the lecture

- Present the basic ideas and concepts behind Java.
- Show how a simple user interface can be implemented.

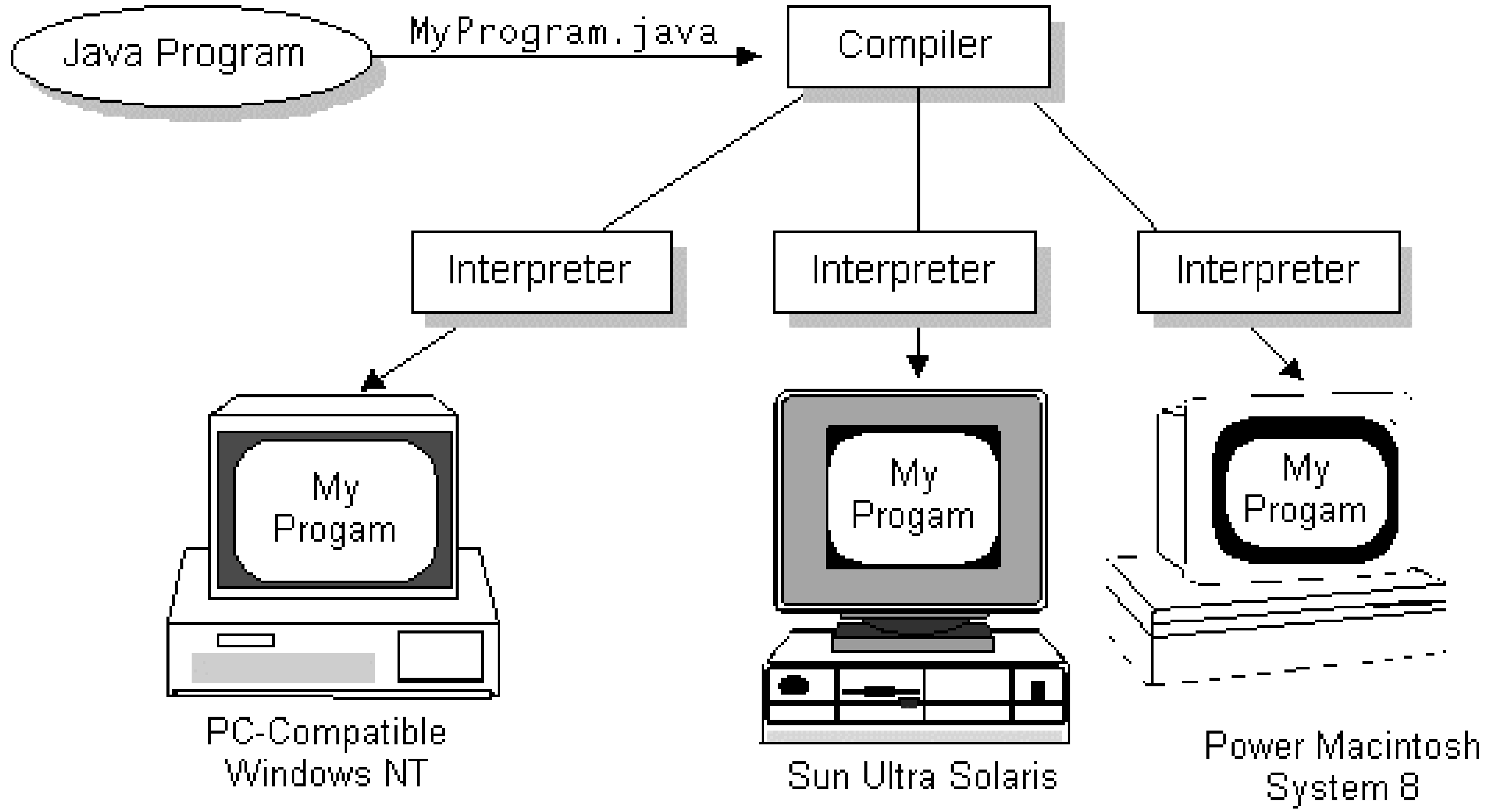
Java 1.4.2 during the lecture

Sources of additional information:

- Sun's Interactive Java Tutorial (local mirror accessible via the course material list on the course homepage)
- Java Programming Language Basics Part1 and Part2.
  `http://developer.java.sun.com/developer/onlineTraining/`
  `Programming/BasicJava1/index.html`
- Per Holm: *Objektorienterad programmering och Java*. Studentlitteratur, 1998.
- Arnold & Gosling: *The Java Programming Language.* (2nd ed.), Addison-Wesley, 1998
- Java links on the course home page
- Java material on the home page of the course in Real-Time Programming.

Overview

- Developed at Sun Microsystems (Gosling et al)
- Syntax from C, C++, semantics from Simula
- General object-oriented language
- First release May 1995. Latest release Java 2 Version 1.6.0.
- Originally intended for consumer electronics
- Re-marketed as a WWW-language
- Programs compiled into byte code
  - interpreted by virtual machine (JVM)
  - compiled into native code
  - sent over Internet
  - platform independent


Java Program
MyProgram.java
Compiler
Interpreter
Interpreter
Interpreter
My Progam
My Progam
My Progam
PC-Compatible Windows NT
Sun Ultra Solaris
Power Macintosh System 8

Applications vs Applets

Java applications:

- stand-alone programs

Java applets:

- runs within Java-enabled browsers (clients)

Java servlets:

- runs within network servers

The Java Language

Aim: A simple, object-oriented, network-oriented, robust, secure, architecture neutral, portable, high-performance, multi-threaded, dynamic language.

Simple:

- easy to learn
- based on today's practice → borrow C++ syntax
- difficult issues omitted, e.g., operator overloading, multiple inheritance, ..
- automatic garbage collection
- small size

Object-oriented:

- the object-oriented features of Simula and C++ with some extensions

Network-oriented:

- full support for networked applications
- TCP/IP protocols (HTTP, FTP)
- open and access objects across the net

Robust:

- strongly typed language
- extensive compile-time checking
- safe references instead of unsafe pointers

Secure:

- the byte code verifier verifies the byte code before it is interpreted
- the class loader is responsible for loading compiled Java classes into the JVM in a safe way
- the security manager handles platform-level security by checking whether or not a program may access platform resources (file system, network, ...)

Comments

```
// A Line comments. Extends to the end of line

/* A multi-line comment
   that continues on multiple lines */

/** Documentation comment. Only immediately
    before a class or method declaration. Used
    by the java-doc tool  */
```

Multi-line comments may not be nested.

Simple Declarations

The usual set of simple types

```
int m, n;      // Integer variables
double x, y;   // Real variables
double z = 0.89; // Initialization
boolean b;
char ch;
```

Numeric Expressions

```
n = 3 * (5 + 1);
x = y / 1.4;
n = m % 8;     // Modulo 8
b = true;
ch = ‘x‘;
```

= for assignment, == for comparison

Type Casting

Explicit type conversion (casting)

```
double radians;
int degrees;
...
degrees = radians * 180 / 3.14;  // Error
degrees = (int) (radians * 180 / 3.14) // OK
```

Java Statements

Similar to other languages.

Statements can be grouped using `'{'` and `'}'` (begin .. end)

Conditional Statements

```
if (n == 3)
    x = 3.0;
```

- no `then` keyword
- boolean condition **within parentheses**

```
if (x != 0)
    y = 3.0 / x;  // semicolon necessary
else
    y = 1;

if (x != 0) {
    y = 3.0 / x;
    x = x + 1;
} else           // NO semicolon
    y = 1;
```

It is common practice to **always** include the braces, also if they only contain one statement.

Logical operators: and – `&&`, or – `||`, not – `!`

While and For statements:

```
double sum = 0.0;
double term = 1.0;
int k = 1;
while (term >= 0.00001) {
    sum = sum + term;
    term = term / k;
    k++;                    // increment k
}

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

```
do {
  x = ...;
  y = ...;
} while (y > 0);
```

Corresponds to a Repeat Until.

Case statements:

```
switch (x) {
  case 1: S1; break;
  case 3: S2; break;
  case 4:
  case 5: S3; break;
  default: S4; break;
}
```

S1, etc can be a sequence of statements.

Object-Oriented Programming

Classes: 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 operations corresponds to a ordinary record (Pascal) or struct (C).

Objects: An *instance* is an executable copy of a class. Another name for instance is object (instance object).

Classes and objects provide:

- modularity
- 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*)

Supports:

- re-usability
- mechanism for organizing and structuring software

Object-Oriented Programming

Polymorphism: The possibility to specify that methods should take parameters that are of superclass type (specify that variables should be of superclass type). In this case it is possible to pass objects that are instances of the superclass or of any of its subclasses as actual parameters to the method. Powerful structuring mechanism.

Classes

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

Attributes:

```
class Turtle {
    private boolean penDown;
    protected int x,y;
}
```

`private`: cannot be accessed 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 outside

A class without any methods can be viewed as a record (struct)

Methods

```
class Turtle {
  // attribute declarations
  private int x,y;

  public void jumpTo(int newX, int newY) {
      x = newX;
      y = newY;
  }
  public int getX() {   // returns an integer
      return x;
  }
}
```

Public means that they can be accessed from the outside.

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

Using Objects

```
Turtle t; // Reference variable
t = new Turtle(100,100);

int a = t.getX();
t.jumpTo(a + 100, 200);
```

Manual memory deallocation not needed.

Class attributes

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

```
class Turtle {
    private boolean penDown;
    protected int x,y;
    static int numTurtles = 0;
}
```

Class Methods

It is also possible for a method to be a class method, rather than an instance method.

Class methods can only access class variables.

To specify that a method is a class method, the keyword `static` is used.

Class methods (and class attributes) are accessible from the class itself, in addition to from each instance of the class. There does not need to be any instances in fact. This is, e.g., the way the mathematical functions in the `Math` class are used.

```
y = Math.sqrt(x);
```

Constructors

A special method that is called when the object is created.

Written like an ordinary method. Has the same name as the class. Has no return type (not even void).

```
public Turtle (int initX, int initY) {
    x = initX;
    y = initY;
    penDown = false;
    numTurtles++;
}
```

It is possible to have multiple methods with the same name as long as the method signatures are different (number and type of arguments).

Often used for constructors.

Compilation

A source file may only contain one public class. The file should have the same name as the public class + the extension `.java`

Compiled using: `javac MyClass.java`

The output (the byte code) will then have the name `MyClass.class`

Executed by JVM through: `java MyClass`

The main method

Statements only within methods in classes. The method that is called by the system when the program is started is called `main`. Must be declared in the class that is started from the command line. Must have the signature below.

The `main` method of the `Turtle` class.

```
public static void main(String[] args) {
    Turtle t = new Turtle(100,200);
    t.right(90);
    while (t.getX() < 150) {
        t.forward(2);
    }
}
```

The `main` method is static, i.e., it cannot directly access instance attributes. The way to do it is to start with creating an instance, and then do the access through that instance.

`args`: command line arguments

Nested Classes

A class that is defined as a member of another class is called a *nested class*.

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 also be *anonymous classes*, i.e. they have no name.

Subclasses

```
class NinjaTurtle extends Turtle {
  // Declarations for NinjaTurtle
}
```

Constructors:

```
public NinjaTurtle(int initX, int initY, String name) {
    super(initX,initY);  // Call constructor of super class
    // more initializations
}
```

Method Overriding

A method in a class overrides any method with the same name and parameters in any superclass.

A subclass cannot override methods that have been declared `final` or methods that have been declared `static`.

It is possible to call overridden methods using the notation `super.method`. Can be used to add functionality to a method rather than replacing it.

Abstract Classes

Methods can be *abstract*. Contain only declarations without any implementation. The method must be implemented in some subclass.

```
public abstract void draw();
```

A class with at least one abstract method is itself called an abstract class and must be declared as `abstract class` rather than just `class`.

An abstract class cannot be instantiated.

It is sometimes useful to define classes to be abstract even if they do not contain any abstract methods (to structure the inheritance hierarchy).

Interfaces

A way of obtaining some of the functionality of multiple inheritance without its problems.

An *interface* defines a protocol, i.e., a set of methods. Any class may then implement the interface. A class may implement several interfaces.

Reference variables may be typed either by class or by interface.

Example

```
interface Drawable {
void draw(SimpleWindow w);
int getWidth();
int getHeight();
}

class Rectangle implements Drawable {
    private int xSide,ySide;

public void draw(SimpleWindow w) {
   int x = w.getX();
   int y = w.getY();
   w.lineTo(x,y+ySide);
   w.lineTo(x+xSide,y+ySide);
   w.lineTo(x+xSide,y);
   w.lineTo(x,y);
}
public int getWidth() {
   return xSide;
}
public int getHeight() {
    return ySide;
}
}

class Person implements Drawable {
    private String name;
public void draw(SimpleWindow w) {
    w.writeText(name);
}
public int getWidth() {
    return 6*name.length();
}
public int getHeight() {
    return 10;
}
}
```

```java
void drawWithBorder(Drawable d, int x,
                    int y, SimpleWindow w); {
  w.moveTo(x,y);
  d.draw(w);
  int width = d.getwidth();
  int height = d.getHeight();
  w.moveTo(x-2,y-2);
  // do 4 calls to w.lineTo to draw the border
}
```

Interface Rules

An interface may only contain method and constant declarations.

All methods in an interface are implicitly public and abstract.

A class may extend only one superclass but it may implement several interfaces.

An interface can be used as a type name.

Interfaces may be extended just like classes, and an interface may extend more than one interface.

Arrays

Similar to objects:

- accessed using reference variables

```
int[] someInts; // Integer array
Turtle[] turtleFamily; // Array of references to turtles

someInts = new int[30]; // Size when the array is created

int i;
for (i = 0, i < someInts.length; i++) {
  someInts[i] = i * i;     // Indices start at 0
}
```

`.length` predefined

Method Parameters

Java uses call-by-value for simple types. It is not possible to pass out any values from formal parameters to actual parameters.

Java uses call-by-reference for objects (really call-by-value since the reference variables behave as pointers). A change to a formal parameter object affects the actual parameter object.

References vs pointers

Java references are the same as safe pointers.

Manipulation of references not allowed (e.g., adding integers)

Compile-time checks guarantees that a reference is initialized before it is used.

Memory allocation

- Static memory allocation - all memory allocated at start-up
- Dynamic memory allocation
  - memory is allocated dynamically from the heap when needed
    - manual memory management
    - the application explicitly allocates memory when needed and deallocates it when no longer used
    - Pascal, Modula-2, C, C++, ...
    - problems: dangling pointers, memory leaks, fragmentation
  - automatic memory management
    - runtime system or OS deallocates memory automatically
    - garbage collection
    - Java
    - problem: takes time and may disturb the real-time application

Java Garbage Collection

Runs as a low-priority thread

Incremental - work is divided into small pieces that are spread out over execution.

Can be explicitly invoked by calling `System.gc()`

Real-time GC methods have been developed (Roger Henriksson, CS, LTH). Is part of Sun's Real-Time Java 2.

Exceptions

An exception object is created when an error occurs in a method.

Contains information about the exception (type, state of the program)

The run-time systems tries to finding some code that handles the exception.

Creating an exception and handing it to the run-time system = *throwing* an exception

The run-time system searches backwards through the call stack of the method until it finds a method containing an appropriate exception handler (he type of the thrown exception should be compatible with the type of the exception handled by the exception handler)

The chosen handler is said to *catch* the exception.

If the runtime system does not find a handler the run-time system and, hence, the Java application terminates.

Catching an exception

An exception handler consists of a `try` block together with at least one `catch` block or one `finally block`

```
try {
  // Code
} catch (Exception e) { // Type of exception handled
  // Code to handle the exception
}

try {
  // Code
} finally {
  // Finally code. For cleanup. Always executed.
}
```

Specifying an exception

Instead of catching an exception, a method can decide to instead simply specify that it may generate a exception.

The keyword `throws` is added to the method signature followed by a comma-separated list of all the exceptions the method throws.

The responsibility of the caller of the method to either catch or specify these exceptions.

Exception types

- Runtime exceptions
  - occur within the run-time system
  - e.g. divide by zero
  - the compiler does not require that these are caught or specified
- Checked exception
  - checked by the compiler
  - requires that the exception is either caught or specified

Exceptions are normally thrown by the run-time system. However, it is possible for an application to throw application-specific errors.

```
throw new Exception1();
```

Common exceptions

Many Java methods used in real-time programs throws checked exception.

For example:

- `wait()` throws InterruptedException
- `sleep()` throws InterruptedException

Try blocks common.

Packages

A collection of related classes, in one or several files.

Unspecified attributes get package visibility, i.e., they are public to other classes in the package, private to other classes.

A class that should be accessible from the outside of the package must be declared public and be written in a separate file. The file should have the name of the class, and it may not contain any other public classes.

Classes declared in files without a package specification belong to the ’unnamed package’

Package Syntax

In file CommandButton.java

```
package Gui;

public class CommandButton {
...
}
```

Package Names

Standard packages

- java.xxx (e.g. java.awt and java.awt.event)

Java extension packages

- javax.xxx (e.g., javax.swing)

User-defined packages:

- globally unique names
- convention: reversed Internet domain name followed by local directory structure (e.g., `se.lth.cs.realtime` and `se.lth.control.realtime`)

All files belonging to the same package should be stored in a directory with the name of the package.

Package Access

1. Explicit naming

```
B = new Gui.CommandButton();
```

2. Import one class

```
import Gui.CommandButton;
B = new CommandButton();
```

3. Import many classes

```
import Gui.*;
B = new CommandButton();
```

Standard Packages

java.lang – Object, Class, String, ...

Java.io – Streams and random-access files

java.awt – Abstract Window Toolkit

java.applet – Applet

java.util – Collections, Date, Time, ...

java.net – Sockets, Telnet, URLs, ...

...

Swing

Class package for graphical user interface implementation that replaces the older AWT (Abstract Window Toolkit)

Supports buttons, menus, scrollbars, ...

All class names begins with J (JButton, JFrame, ...)

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

Top-Level 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

Simplifies the positioning of components in window

JPanel:

- a panel (pane)

Other examples a JScrollPane and JTabbedPane.

Panes are themselves components

Atomic Components

Selfsufficient entities that presents information to the user or implements some user control

- JButton - button
- JTextField - editable textfield
- JLabel - uneditable text field
- JSlider - slider
- PlotComponent - local plotter class

See

`http://www.java.sun.com/docs/books/tutorial/uiswing/components/components.html`

Layout Management

Controls the layout of components in an intermediate container.

BorderLayout:

- five areas: north, south, east, west, 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 movement, mouse click, ...) is considered an event.

The Swing system informs the Java application about an event by creating an object of class `AWTEvent`.

Several subclasses:

- `ActionEvent` – generated when an event that is specific for a certain component occurs (e.g. mouse click)
- `TextEvent` – generated when the contents of a text component is changed
- ...

Listeners

When an event occurs, the Java system calls a method in a listener object.

A listener object is an object that implements a listener interface.

`ActionListener`

- contains `actionPerformed(ActionEvent e)`
- called when an `ActionEvent` object is generated

`TextListener`

- contains `textValueChanged(TextEvent e)`
- called when a `TextEvent` is created

Several components can listen to the same event.

Achieved by calling the `addActionListener` method of an event source object with the listener object as the argument.

Event Methods

All event classes contain the method:

`Object getSource()`: – return the source object of the event

Some event classes contain event-specific methods:

```
char getKeyChar();      // KeyEvent

int getX(); int getY(); // MouseEvent
```

Setting up an event handler

Three bits of code:

1. declare that the event handler class implements a listener interface

```
public class MyClass implements ActionListener {
```

2. register an instance of the event handler class as a listener upon on or more components

```
someComponent.addActionListener(instanceOfMyClass);
```

3. code that implements the methods in the listener interface

```
public void actionPerformed(ActionEvent e) {
  // code that reacts to the action
}
```

Event Dispatching Thread

Event-handling code executes in a single thread, the *event-dispatching thread*. Ensures that each event handler will finish executing before the next one executes.

Rule: Once a Swing component has been realized (painted on screen), all code that might affect or depend on the state of the component should be executed by the event-dispatching thread.

A Swing top-level window is realized by calling one of the method `setVisible(true)`, `show`, `pack`

Exception to the rule

A few methods are thread-safe.

An Application's GUI can often be constructed and shown in the main thread.

Listener lists can be modified from any thread.

Execute code in the event-dispatching thread

Many application need to perform non-user-event-driven GUI work after the GUI has been created:

- programs that must perform length initialization operations before they can be used
- programs whose GUI must be updated as a result of some event generated from the application - common in control applications.

The `SwingUtilities` class contains two methods to help:

- invokeLater: registers some code to be executed in the event-dispatching thread. Returns immediately, without wait.
- invokeAndWait: Waits for the code to execute.

Often implemented using anonymous classes implementing runnable

```
Runnable updateAComponent = new Runnable() {
  public void run() {component.doSomething();}
};
SwingUtilities.invokeLater(updateAComponent);
```

Can be called from, e.g., a Regul or Opcom thread.

Swing Example

A periodic thread that generates a sine-wave.

The sinewave data is sent to a Swing-based GUI that plots the sine wave using a PlotComponent.

The GUI also implements two sliders that makes it possible to change the amplitude and frequency of the sine wave.

A monitor object used for the communiction between the GUI and sine wave class. Implemented as an internal class of the sine wave class.

Swing Example

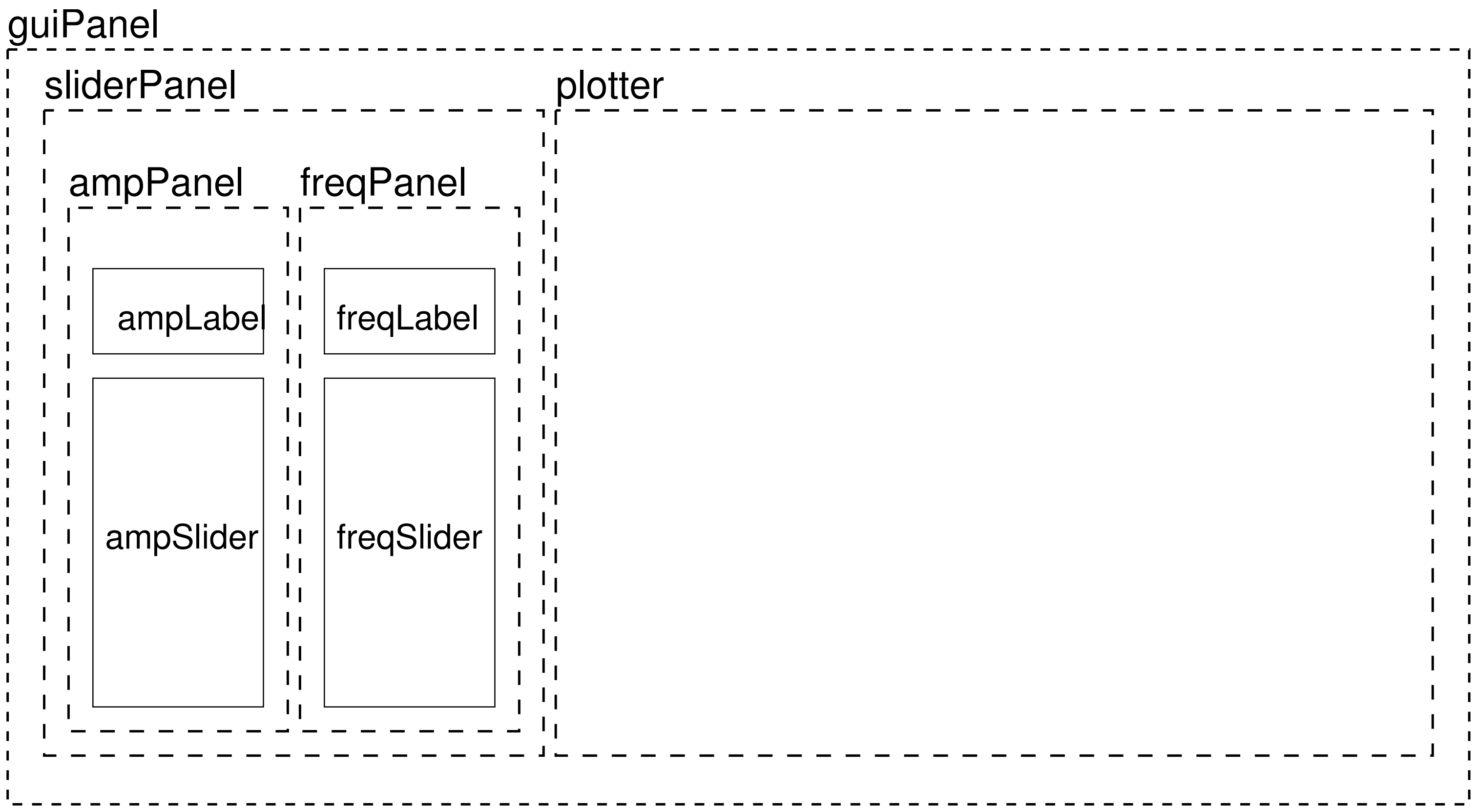
Classes:

- Main – Separarate main class used during start-up.
- Sinus – the sine wave generator
  - Monitor – Internal class inside Sinus
- Opcom – GUI

User threads:

- main thread during start-up
- Sinus – extends the Thread class
- Swing event-dispatching thread

Swing Panels


guiPanel
sliderPanel
plotter
ampPanel
freqPanel
ampLabel
freqLabel
ampSlider
freqSlider