Lecture 10: Application Programming Interfaces II

Instructor: Musard Balliu, musard@chalmers.se

http://www.cse.chalmers.se/~musard

QUESTIONS?

Plan

- Last time
  1. Robot APIs
- Today's Plan:
  1. Continue with Robot APIs
  2. Graphical User Interfaces

An exploding robot

We'd like to create a bomb (represented by a red `Thing`. If the robot comes up on it, it explodes. How can we do this?

`Bomb`
Solution

More than one robot

We can try a little game by having two robots moving randomly in a city with bombs in it:

Who explodes first?

Just uncomment the lines in the `main` method of `ExplodingRobot`.

Threads

Problem: only one robot moves!

The solution is to use *threads*.

Instances of the interface `Runnable` have a `run` method, which, when started, executes while the rest of the program also continues to run.

In our case, we want the go of the two robots to execute simultaneously.

Threads

Therefore, we need a new kind of robot, which `implements` `Runnable`, and has a `run` method in which we call `go`.

`ThreadedRobot`

GUI and OOP

GUIs are one of the success stories of OOP.

It is relatively easy to see the data and subroutines that "go together" in the components of a GUI.

In this lecture, we examine the main such components provided by Java in the context of providing a GUI for the "exploding robots" application we wrote earlier.

JFrame

The `JFrame` is the main window and is the component that interacts with the graphical environment of the operating system.

The other components will be added to the `JFrame` and be "insulated" from the external environment.

This is a typical example of OO design, illustrating *separation of concerns*, which is a fancy way of saying you shouldn't have to keep track of variables you don't need.

Experimenting with `JFrame`

`JFrame` documentation

An empty `JFrame`

A more reasonable frame

By default, instances of `JFrame` are neither visible, nor, when visible, do they go away when you try to close them. We can extend `JFrame` and have something more reasonable.

A reasonable frame

Adding components to a `JFrame`

Can we put a `City` in a `JFrame`?

Try it!

Adding a `City` to a `JFrame`

The answer is “no”. We can only add instances of `java.awt.Component` to a `JFrame` (or any other instance of `java.awt.Container`), and `City` is not such an instance.

But we can add a `CityView`!

Documentation for `CityView`

Adding a `CityView` to a frame

Problem: the constructor of `CityView` is protected, which means we cannot get a `CityView` directly.

We need to find another way of obtaining a `CityView`. Examining the documentation, we hit upon...

Adding a `CityView` to a frame

RobotUIComponents

We can now add the `CityView` to our frame.

Code

We still have to specify that the frame should be visible, the size, and the behaviour on exit. Here we used `pack()`, which gives the frame the "natural size".

A remark on OO design

Note that the information about visible streets and avenues which we use to construct the city is *not* passed on to `RobotsUIComponents`. Therefore we have to pass it ourselves, explicitely.

This leads to code duplication, so the design of `becker.robots` could be improved.

The best way to do that is to subclass `City`, creating a class `ReasonableCity`, for cities that explicitely store that information, then subclass `RobotUIComponents` to create a class of `ReasonableRobotUIComponents` that extract the information from reasonable cities and use it to construct the city views.

Adding the Start/Stop button

The `CityView` does not provide the start/stop button. We need to obtain it separately.

However, if we try adding to the frame, something unpleasant happens.

Try it!

Using a `JPanel`

When adding several components, we need to use a *panel*.

In line with the idea of separation of concerns, the layout of the various components is taken care of not by the frame, but by a `JPanel`.

The panel uses a *layout manager* in order to keep track of the components.

There are myriad layout managers. The most common, presented for example in Eck's book, are `FlowLayout`, `BorderLayout`, and `GridLayout`. They last two work well if the components are relatively balanced in size, which is exactly the case we don't have.

We end up using a `BoxLayout`.

Try it!

Adding a menu

As always, there are exceptions to what we've just said.

Menus are added to the frame, not to the panel (which makes sense, given you don't normally see menus in the windows, among buttons and such).

We shall add a menu with two items: one for restarting or quitting the game, and one for choosing the number of bombs.

Event-driven programming

At the moment, it's not terribly exciting to use the menus.

When the user fumbles about a component, Java creates an `ActionEvent` object (which depends on the user action). This object is then sent to the event-listeners registered for that component.

Therefore, we need to add event listeners to our menu items.

We start with the easiest: quitting the application.

The listener

Try it!

Adding the other listeners

The Restart listener
Try it!

The pattern is always the same:

- the real work is done in the main class, which has access to all needed data;
- the listener is only used to activate the appropriate method of the main class;
- therefore, the listener is constructed with a reference to the main class.

The Settings listener
Try it!

Inner classes

Because these listeners are in general very flimsy and almost always use a reference to the main class, the Java designers have come up with "inner classes". That is, you can define new types inside other types, with the usual scoping rules: the outer variables and methods are visible inside, so no need to pass references to the main class: the appropriate method can be called directly.

Inner class for `SettingsListener`

I am not a big fan of this idea, for instance because of `this`.

Inner classes

Even more, inner classes can be *anonymous*. Here is the same code with an anonymous animal:

Anonymous classes for settings listeners

In this particular example, the advantages of anonymous classes are not immediately obvious.

Choosing a robot

We want to start by asking the user to choose a robot.

The component to use is `JOptionPane` with the static method `showOptionDialog`.

This component does not get added to the frame or the panel! Moreover, the execution of the application stops while waiting for an answer.

Once the user has made a choice, the game should start directly. We can achieve this by "clicking" the `Start` button from the program.

Adding an option pane

Game over, and restarting

The first robot to trip over a bomb loses. The game should stop, the user should be shown a dialog informing him of the result and asking whether to stop or to restart.

We can stop the game by again "pressing" the `Start` button.

To display the correct result, the robot must know whether it was chosen by the user. To restart or quit the game, the robot must, just like a listener, have a back reference to the main class.

Finally, the "identity" of the robot (i.e., chosen by the user or not), must be initialized in the `chooseRobot()` method of the main class.

The new game robot
The main game class

Homework

- Create the reasonable components discussed in the slide A note on OO design.
- Modify the game so that the robots are placed randomly in the city, but never directly on top of a bomb.
- Read Chapter 6 (minus section 6.2) of Eck's book
- Read section 3.5.2 in Becker's book (about threads)