CS209

Computer system design and application

Stéphane Faroult
faroult@sustc.edu.cn

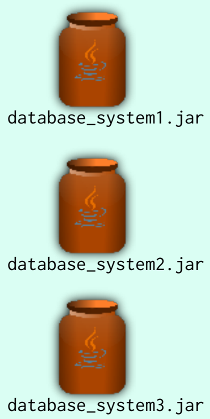
Zhao Yao zhaoy6@sustc.edu.cn
Liu Zijian liuzijian47@163.com
Li Guansong intofor@163.com

Reflection

We have seen last time a few common usages for reflection:

- Locating resources associated with a program when the program can be installed anywhere on a computer
- Reading annotations (usually to generate doc or code)
- And finally something a bit more advanced:

Dynamically loading a class

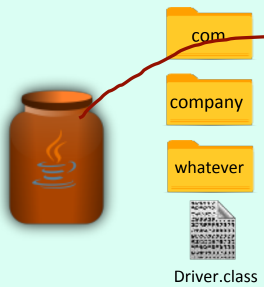


This is particularly useful with database access. Although there is a common language for accessing databases, database providers supply (as java archives) classes that implement the required methods to talk to THEIR system.

Usually the driver has a long complicated name to ensure that there is no conflict (two different drivers cannot have the same name).

Fully qualified class name

`com.company.whatever.Driver`



If the name of the .jar file is included in the CLASSPATH (where the loader looks for .class files), then the program can load the driver of its choice.

```
Class c = Class.forName("com.company.whatever.Driver");
Driver drv = (Driver)c.newInstance();
```

For instance ...

There is a Java graphical tool called Squirrel SQL that uses this to let you query almost any database system, as long as you have the suitable .jar file added to your CLASSPATH. You can switch between very different systems.

Lambda expressions

Our third important topic after annotations and reflection are "Lambda expressions", which were introduced in Java 8 (first released in March 2014). "Lambda expressions" touch on what is called "functional programming", an area which has been recently the object of much interest, even if its roots are more than 100 years old. You'll probably hear about "lambda expressions" and "functional programming" elsewhere than in a Java context.

Nested Classes

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

To explain the benefits of lambda expressions, let's take a look back at classes and interface, and start with nested classes, classes defined inside other classes.

```
class OuterClass {
        private int attr;
 public ...
private class NestedClass {
protected    ...
        }
}
```

YES

If a nested class is declared as public, private or protected it can access the private attributes of the outer class.

```
class OuterClass {
        private int attr;
        ...
 static class NestedClass {
            ...
        }
}
```

NO

This no longer works if it's defined as static, because the attribute only exists when an OuterClass object is created, but NestedClass is accessible without an object.

Depending on the nested class being static or not, you have two different ways to create a nested class object.

```
OuterClass.NestedClass nestedObject =
                   outerObject.new NestedClass();
```

depends on an existing OuterClass object

```
OuterClass.StaticNestedClass nestedObject =
               new OuterClass.StaticNestedClass();
```

independent from any OuterClass object

WHY NESTING?

Grouping

Encapsulation

You can of course question why classes should be nested. This is mostly done as a way of structuring the code, either by grouping software components or for hiding through encapsulation the inner working.

Local Classes

```
class OuterClass {

    ...
    public void doSomething() {
       class LocalClass {
            ...
       }
    }
}
```

You can also have local classes, that are not only defined inside another class, but inside a method.

In the area of Java software engineering, there is also one component that is very much used: interfaces. Interfaces define the behaviour, and how you can "talk" to an object (remember that object oriented programming is mostly about objects exchanging messages).
If a class can only extend (inheritance) one parent class, it can implement multiple interfaces. Java Collections are a rather good example.

Reminder :Interfaces

- abstract *(implicit)*
- define methods that classes MUST implement to conform
- no variable attribute
- constants OK

```
class SomeClass extends ParentClass {

}
```

methods inherited, unless they are abstract

methods must be rewritten

```
class SomeClass implements Interface {

}
```

The only problem with interfaces is that YOU have to rewrite the methods (fortunately one interface rarely defines many methods)

Anonymous Classes

There are many cases when the only things that we are interested in are interface methods. We can of course define a class implementing the interface ...

```
class NamedClass implements Interface {
    ...
}

Interface anObject = new NamedClass(...);
```

... but as the only thing we really want is an "interface object reference", the named class is a bit useless. One such example is a "Comparator" object. We usually just want the compareTo() method.

Anonymous Classes

Java allows defining an unnamed (ano – nymous = without a name) object that implements all that is required by the interface.

```
Interface anObject = new Interface() {
                     // attribute and method definitions
                   };
```

Very convenient for parameters

Anonymous Classes

```
class NamedClass extends ParentClass {
    ...
}

NamedClass anObject = new NamedClass(...);
```

This works not only with interfaces, but also with inheritance. Children objects can be named ...

Anonymous Classes

```
ParentClass anObject = new ParentClass() {
                     // attribute and method definitions
                   };
```

... or not, if the only thing you are really interested in is a special behaviour of an abstract parent class.

Many examples of this in graphical interfaces (coming soon ...)

```
Button btn = new Button();
btn.setText("Say 'Hi'");
btn.setOnAction(new
          EventHandler<ActionEvent>() {

       @Override
       public void handle(ActionEvent e) {
          System.out.println("Hi!");
       }
   });
```

Single Method

Now, very often interfaces require a single method!

Can be simpler

From Java 8: Lambda expressions

Functional Programming

Programming only with functions, no state stored

In the very common case where your interface requires a single method, you can use lambda expressions. Lambda expressions come from functional programming, where you try not to store any state (which is completely opposed to attributes that store the state of an object ...)

Comes from lambda calculus

Lambda λ (Λ) Greek L – lowercase (uppercase)

"Lambda Calculus" comes from the name of the Greek letter lambda, which is the same as L in the Latin alphabet. Greek letters are much used in mathematics (and from there in physics), and you won't be surprised to learn that lambda calculus comes straight from mathematics.

Comes from lambda calculus

Lambda calculus was developed in the 1930s by Alonzo Church, an American mathematician not as well known (so far) as Alan Turing but with similar concerns.

Alonzo Church
(1903 –1995)

Invented in the 1930s by Alonzo Church a pioneer with Alan Turing of theoretical computing.

What Church was after was a simple notation for mathematical functions, mostly to ease proofs of results (don't underestimate notation, a lot of mathematical progresses came from better notation).

Comes from lambda calculus

Simple notation for functions and applications.

$\lambda x.(4x^3 + 2x + 1)$

function expression (often called M)

Church came out with this, and here is lambda.

"binding" of x (means that x is the variable)

Comes from lambda calculus

Simple notation for functions and applications.

$\lambda x.(4x^3 + 2x + 1)$

$((\lambda x.\, M)\ E) \ \text{->}\ (M[x:=E]))$

This is how giving value E to x is written. Notice the arrow.

β reduction

Simpler way of writing expressions

You are probably unimpressed by lambda expressions. Once again, it's just notation. However, notation often opens whole new vistas. Think of the "0" notation. Envisioning nothing as a computable quantity (first done by Indian mathematicians about 1,500 years ago) opened the door first to equations and then to a lot of mathematical feats. "Cartesian coordinates" linked algebra to geometry. In the case of lambda notation applied to Java programming, it seriously makes programs easier to read – which means fewer bugs.

Lambda expression in Java

Lambda expressions only work with functional interfaces.

Functional interface: only one abstract method

@FunctionalInterface

Method written without its name as

```
(parameter list) -> {method body}
```

Data types optional

If there is only one method to redefine, its name no longer needs to be given.

BENEFIT?

Easily passing a function as parameter

Much less code

Easier to read

As said earlier, using lambda expressions make the code far more readable.

```
Button btn = new Button();
btn.setText("Say 'Hi'");
btn.setOnAction(new
        EventHandler<ActionEvent>() {

        @Override
        public void handle(ActionEvent e) {
            System.out.println("Hi!");
        }
    });
```

Anonymous class

We have seen this expression with an anonymous class.

```
Button btn = new Button();
btn.setText("Say 'Hi'");
btn.setOnAction((e)->{
            System.out.println("Hi!");
        });
```

Much shorter!

As "handle()" is the only method of an event handler, it can also be written like this.

Other common usages

Collections

Lambda expressions are also commonly used for searching data in Collections, as seen in the following example.

```java
class Film {
     private String    title;
     private String    countries;
     private int       year;
     private float     billionRMB;

     // Constructor
     public Film(String title, String countries,
                 int year, float billionRMB) {...}
     // Getters
     ...
     // toString()
     ...
}
```

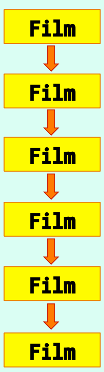
Suppose that a Film class stores box-office information.

```java
ArrayList<Film> films = new ArrayList<Film>();
```

Populate the list from a file

Retrieve information using different conditions

We can build a collection read from a file, and then the problem is how to search this collection. We can search on many different criteria – film title, year of release, country, how much it made so far.



Walk the list

Test each element against a condition

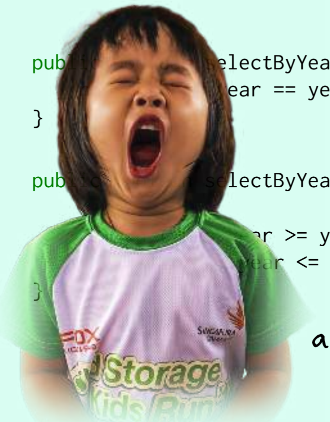
In all cases the process will be the same one – only the condition will change.

1

Add methods to Film

```java
public boolean selectByTitle(String str) {
    return this.title.contains(str);
}
public boolean selectByCountry(String cntry) {
    return this.countries.contains(cntry);
}
```

One option is to add a boolean method that tests every possible condition.



```
pub        electByYear(int year) {
           ear == year;
}

pub        electByYear(int year1,
                        int year2) {
             r >= year1)
             r <= year2);
}
```

and so forth

Very boring code.

2 Use Anonymous Objects

```
interface SelectFilm { boolean test(Film film);}

static void showFilms(SelectFilm tester) {
   for (Film f: films) {
     if (tester.test(f)) {
       System.out.println(f);
     }
   }
}
```

A second solution is to define an interface that implements a "test" boolean method.

2 Use Anonymous Objects

```
showFilms(new SelectFilm() {
            public boolean test(Film f) {
                return f.getYear() == 2014;
            }
         });
```

Anonymous objects allow to define on the fly a suitable test() method that tests for the condition we want.

3 Use Lambda Expressions

```
showFilms((f)-> {return f.getYear() == 2014;});
```

As the preceding interface only defines a single method, it can be called as a lambda expression. As showFilms takes a SelectFilm parameter that only implements a test() method which takes a Film parameter, there is no ambiguity.

Use Lambda Expressions

```
showFilms((f)-> f.getYear() == 2014);
```

In fact, the expression can be further simplified when the returned value can be directly computed, as is the case here.

Lambda Expressions - Variant

Built-in Functional Interfaces

Testing elements in collections is so frequent in practice that there is a built-in functional interface for that.

```
import java.util.function.Predicate;

Predicate<Film> pred
```

method is called "test"

Here is an example of how you can use a Predicate.

Lambda Expressions - Variant

Built-in Functional Interfaces

```
static void filter(Predicate<Film> pred) {
      Film f;
      ListIterator<Film> iter = films.listIterator();
      while (iter.hasNext()) {
         f = iter.next();
         if (pred.test(f)) {
           System.out.println(f);
         }
      }
```

`filter((film)->film.getYear() == 2014);`

There are a few functional interfaces available. Supplier/Consumer are related to multithreading, which we'll see later.

4 Lambda Expressions - Variant

Built-in Functional Interfaces

Predicate<T>	T	→	boolean
Supplier<R>	void	→	R
Consumer<T>	T	→	void
Function<T,R>	T	→	R
UnaryOperator<T>	T	→	T

Streams

And after annotations, reflection and lambda expressions, the fourth interesting new Java feature is called "Streams".

Beware that in spite of the name, it's unrelated to files. It's about chaining processing.

NOT to be confused with files

~~InputStream~~

~~OutputStream~~

The Idea

When you apply to a string a method that returns a string, you can apply a new method to the result.

```
String str = "now let's have some fun";

"now let's have some fun"  str
"NOW LET'S HAVE SOME FUN"  .toUpperCase()
"NOW LET'D HAVE DOME FUN"  .replace('S','D')
               "DOME"      .substring(15,19)
               "DONE"      .replace('M','N')
```

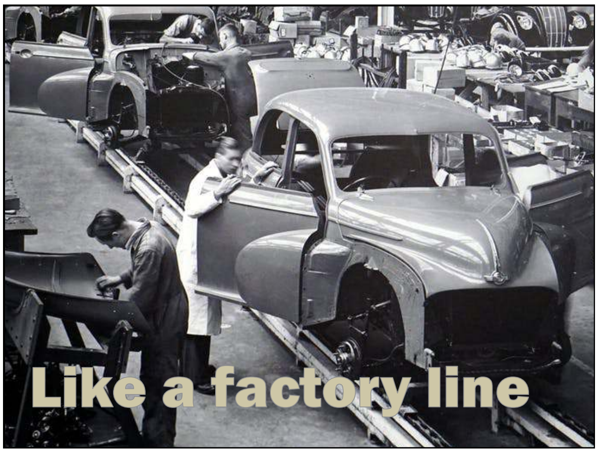
And so forth until you get the result you want.

There is in functional programming a specific term to describe this kind of process.

Because we use functions that return strings, we can chain them.

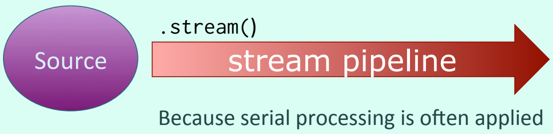
MONAD

structure that represents computations defined as sequences of steps.



Stream

Same idea applied to collections



array
collection
generator function
I/O channel

Because serial processing is often applied to collections of data, the idea is to do something similar in Java – which will look as nothing new to someone who knows SQL, based on similar ideas.

Intermediate operations return a stream

filter
distinct
sorted
map

Terminal operations

foreach
toArray
reduce
count
min, max

With stream operations you have methods that return a stream (which can be fed into something else), and those that don't and terminate the streaming process.

4

Lambda Expressions - Variant

Built-in Functional Interfaces

```
static void filter(Predicate<Film> pred) {
       Film f;
       ListIterator<Film> iter = films.listIterator();
       while (iter.hasNext()) {
          f = iter.next();
          if (pred.test(f)) {
            System.out.println(f);
          }
       }
```

If you remember the preceding filtering of films ...

ArrayList<Film>

```
films.stream()
```

... we can take the collection and turn it into a stream

```
films.stream()
      .filter((film)->film.getYear() == 2014)
```

In that case the filter will be applied to one element at a time.

```
films.stream()
      .filter((film)->film.getYear() == 2014)
      .forEach(System.out::println);
```

We can display any film that "gets through" with a forEach() call (a terminal operation) that applies println() to it. Note the special, unusual notation that specifies the method applied to each element.

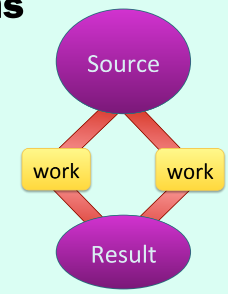
You can insert other intermediate operations before the terminal one, for instance sort the output, if of course Java knows how to sort Film objects. Note that it's FAR more efficient to sort AFTER filtering rather than BEFORE filtering, even if both are possible ...

```
films.stream()
     .filter((film)->film.getYear() == 2014)
     .sorted()
     .forEach(System.out::println);
```

must have a compareTo() method
(implements Comparable<T>)

You can also provide a Comparator

Parallel streams




```
.parallelStream()
```

Like big rivers that reach the sea with a delta, streams can be split into multiple parallel streams but we'll talk about parallelism later. "Data Science" and "Big Data" are full of this.

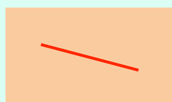
Graphical User Interfaces

An interesting topic is the one of graphical user interfaces (GUI, pronounced Gooey). The programs that you usually write in labs are far uglier than the programs that you use every day: they run in consoles, read from the keyboard, just display text ... So 1970s. Having a nice interface requires quite a lot of coding, but what is interesting is that the logic is very different from the procedural logic you have seen so far (and this logic is the same one with all programming languages and graphical interfaces)

Tons of graphical packages

First of all you don't code everything by yourself, but use functions from packages that you must import when writing your program.

Low level graphics



You have low–level packages with functions (called "primitives") for performing tasks such as drawing a rectangle, a line or a curve.

High level graphics

You also have high–level packages that use the previous ones to draw for instance buttons, and automatically change them when they are clicked — this is what we'll talk about.

Historically several packages in Java

1995 **AWT** (Abstract Window Toolkit)

Looks like other applications on the system

Dec 1996 **Java Foundation Classes** Swing

Looks the same on all systems

In Java, several packages allow you to code a GUI. The first one was AWT, followed by "Java Foundation Classes" quicky renamed "Swing".

Historically several packages in Java

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

Swing relies on AWT, and whenever you code a Swing application you also need to import classes from AWT, as well as from other packages for images.

Swing
AWT

Historically several packages in Java

A new package, JavaFX, was introduced in 2008.

2008 **JavaFX** `import javafx.*;`

2007

1990s, early 2000s

android

Officially adopted by many Smartphones

JavaFX, with which you import classes from a single package (but many subpackages) supports other devices than computer screens for which AWT and Swing were written — mobile phones in particular. It also allows to define the looks of applications in external files called "style sheets" or "CSS" files (CSS means "Cascading Style Sheet" — 'cascade' is French for 'Waterfall'), a technique borrowed from web programming. However, because software has a long life, there is a lot of Swing around, Swing is still much in use and will probably stay around for quite a while. It's good to know both Swing and JavaFX (they aren't VERY different, class names change, basic ideas are the same).

Historically several packages in Java

2008 **JavaFX** `import javafx.*;`

	Model	Data Management	
Application	**View**	User Interface	Visual Elements Looks
	Controller	Logic	

JavaFx applications often follow a popular structure known as "Model/View/Controller" (or MVC) in which data management, user interface and logic are clearly separated.

Event-driven programming

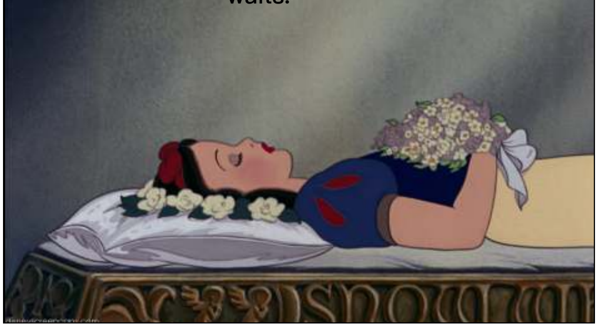
Whichever package you are using, and even whichever programming language you are using (what I'm saying about Java is also true in C/C++ or Python for example), programming graphical interfaces is a very different kind of programming that what you have done so far, and is called event–driven programming.

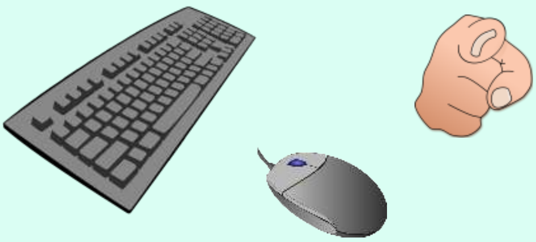
a Graphical Application is a big loop ...

You don't have to code the loop, it's performed for you by the graphical package functions. Basically, you draw things on the screen, display them, and run a loop that does nothing but wait. What is it waiting for? Simply for the user who (presumably) is sitting in front of the screen to do something (other than head–scratching).

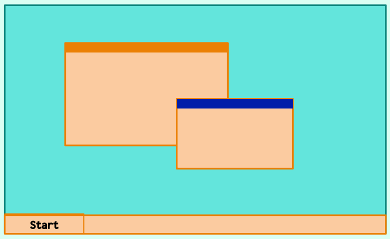
WAIT

Your application sleeps and waits.



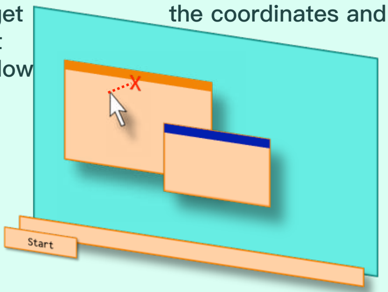


An event can be anything. A key pressed, a move of the mouse, a finger swiping a touch screen, somebody jumping in front of a webcam … Anything that can be translated into an electrical signal reaching the computer.



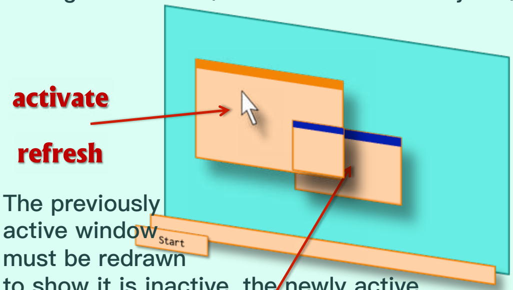
Your graphical application will run in a Window Manager that also reacts to events. For instance you may have an active window and an inactive window on the screen.

The user may move the mouse and click it outside the active window. The Windows management system will get the coordinates and discover that another window is at this location.



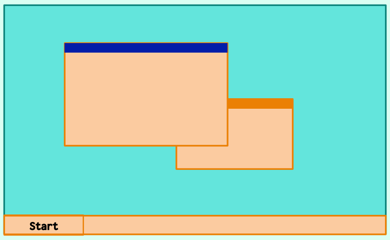
This is an event.

That means that the inactive window has to been "brought forward" (we have a stack of objects).

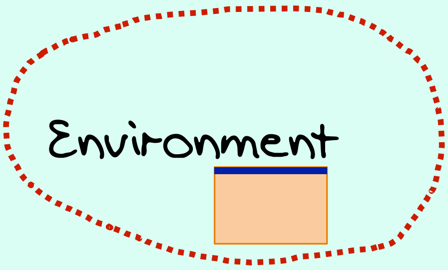


The previously active window must be redrawn to show it is inactive, the newly active window must be redrawn too, including what was previously hidden **deactivate** by the other window.

And finally your screen will look like this.



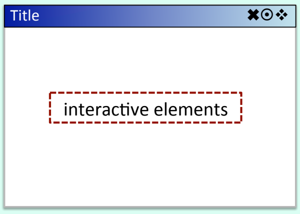
All this work is performed by the Windows Manager and doesn't require your writing a single line of code.

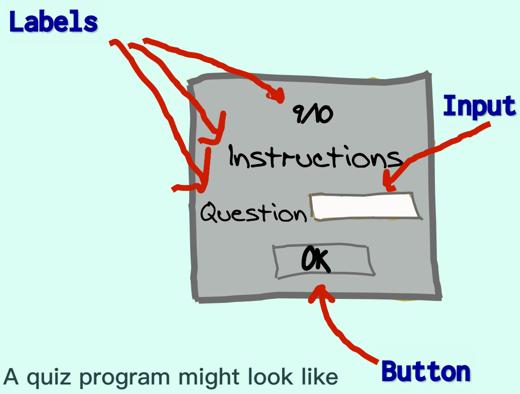


You must always keep in mind that when you are writing a graphical user interface it is running within an environment that manages windows (including resizing, destruction and so forth).

When you design your application you must decide on what the user will see: will your window have a title, will it be resizable, which elements will the user interact with in the window?

What does the user see?



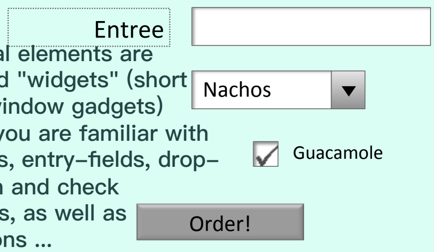


A quiz program might look like this. Clicking the button evaluates the answer.

Your program must prepare everything in advance and, like a conjuror, only reveal elements at the right time when they should be visible.

Window gadget
Widget

Visual elements are called "widgets" (short for window gadgets) and you are familiar with labels, entry–fields, drop–down and check boxes, as well as buttons ...

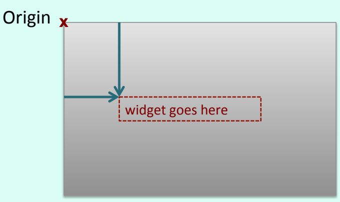


Containers

Container → widget 1 → widget 2 → widget 3

Something that you are probably not familiar with is the notion of "container". If you see widgets, you don't see containers that are nothing more than linked lists of widgets and (more about this soon) other containers.

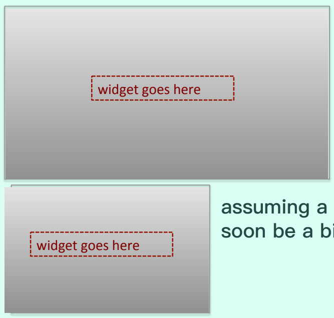
If you have a fixed-size window, things are easy. You can say "I want this widget to appear at these coordinates relative to the upper-left corner of the window".



Unfortunately, the easy case isn't the most common one.

Some containers are fixed

Most aren't



Usually people can (and do) resize windows and you want a "fluid" layout. If you give absolute coordinates assuming a given window size, it will soon be a big mess.

Boxes Containers are here for solving these issues. Boxes come in two flavors, and display widgets next to each other (with some padding in–between) in only one direction.

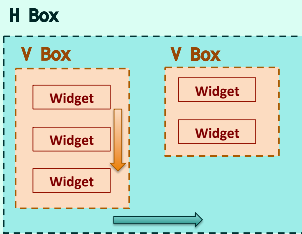
Horizontal boxes

Vertical boxes

Grids Other frequently used containers are grids, which allow you to place widgets at relative (row, column) coordinates rather than distance coordinates.

The great thing with containers is that they can be nested: you can have vertical boxes in horizontal ones or in grids, or the reverse.

Flickr: Jeff Hitchcock



For instance you can have two vertical boxes (each one showing widgets vertically) and add them to a horizontal box (side by side). When the window is resized, the global layout is respected and it still looks (more or less) as intended.

CALLBACK

function associated with an event

The last important idea to understand with graphical user interfaces is the one of "callbacks", often called "handlers" in Java, which is the name given to a function associated with an event. For instance, clicking a button might trigger a search inside a database. This is a function that you write, and associate with the button.

Predefined events

- destroy window
- button press/release
- key press/release
- focus in/out
- move in/out

and so forth

Predefined events are very, very numerous. You only handle those that matter to you. You often must perform a number of checks when the window is destroyed (for instance a text editor will ask you whether you want to save your changes)

OK, so how does it work with Java?

DEMO

The demo is simply a window that moves away every time you try to click the button (merely to irritate people)

What is shown here is the JavaFx version but a Swing version is also available.

Life of a javafx application

Create an instance of the **Application** class

The program class must extend Application

A JavaFX application derives from the Application class in the JavaFx package. It means that it automatically inherits standard attributes and methods.

Life of a javafx application

Create an instance of the **Application** class

Call the **init()** method

Does nothing by default

JavaFx will also automatically call a function called init(). By default, this function does nothing. You can write your own version, and connect to a network or a database, or read a parameter file.

Life of a javafx application

Create an instance of the **Application** class

Call the **init()** method

Call the **start(javafx.stage.Stage)** method

Window

MUST be rewritten

What you must write is a function called "start()" that takes a "Stage" (the name given to windows in JavaFx) as parameter. The function adds the widgets to the window and defines how it looks, and how widgets will react.

Life of a javafx application

Create an instance of the **Application** class

Call the **init()** method

Call the **start(javafx.stage.Stage)** method

Wait for the application to finish:

the application calls **Platform.exit()**

or window closed

You must write the event handlers you need, and nothing else — JavaFx will run the application until it calls an exit routine (perhaps associated with a "Quit" button) or it receives the event "Window destroyed".

Life of a **javafx** application

Create an instance of the **`Application`** class

Call the **`init()`** method
Call the **`start(javafx.stage.Stage)`** method
Wait for the application to finish:
 the application calls **`Platform.exit()`**
 or window closed
Call the **`stop()`** method It will then call a stop() method where you can undo what you have done in init() — disconnect for instance from a database or network. Like with init(), rewriting stop() is optional.