IST-4-JAV Java Programming
Class 4 - Going graphic

Alice BRENON <alice.brenon@liris.cnrs.fr>

Complexity

Simple containers

Implementing interfaces

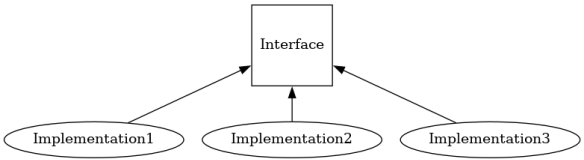


Figure 1: A common structure

One interface, several implementations.

Two implementations

List

- a set of functionalities
- a contract to prove a class can act as a list

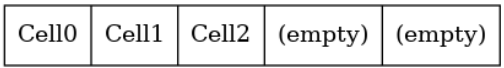
Two implementations

List

- a set of functionalities
- a contract to prove a class can act as a list

→ an interface!

ArrayList



- an array larger than the number of elements
- an index to remember where it stops

LinkedList

- a "head": the element in the current cell
- a "queue": the pointer to the rest of the list

Getting element at index `i`

ArrayList

- check the bounds: $O(1)$
- return cell `i`: $O(1)$

$\Rightarrow O(1)$

LinkedList

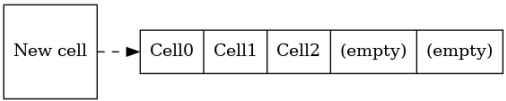
does `i == 0`?

- if yes, get the head:
- otherwise, get the `i-1`th element of the queue

$\Rightarrow O(n)$

Prepending

ArrayList



- create a new array large enough: $O(n)$
- write the new element: $O(1)$
- copy all the other elements after: $O(n)$

$\Rightarrow O(n)$

LinkedList

- create a new cell with the new element pointing to the existing list: $O(1)$

$\Rightarrow O(1)$

Performance comparison

So which one is best?

- if frequent random access is needed: `ArrayList`
- if frequent modification is needed: `LinkedList`

⇒ No "one-size-fits-all", implementation should match the use

In any case

Previously on IST-4-JAV (class 2)...

notion of **type variable**

In any case

Previously on IST-4-JAV (class 2)...

notion of **type variable**

```
List<String> al = new ArrayList<String>();
```

In any case

Previously on IST-4-JAV (class 2)...

notion of **type variable**

```
List<String> al = new ArrayList<String>();
List<String> ll = new LinkedList<String>();
```

Associating values to keys

A common need

- "white pages", phone books...
- Domain Name System
- looking up users in a table by their ID
- finding the selected action in a menu

```
interface Map<K, V> {
    ...
    V get(Object k);
}
```

Association list

```
class Pair<K, V> {
    public K getKey() { ... }
    public V getValue() { ... }
}
```

```
class ArrayList<T> {
    ...
}
```

↓

```
class PhoneBook<K, V> implements Map<K,V> {
    private ArrayList<Pair<K, V>> records;
    PhoneBook (int initialSize) {
        this.records = new ArrayList<>(initialSize);
    }
}
```

Retrieving a number from a key

```
...
V get(Object k) {
    for(Pair<K, V> record: this.records) {
        if(record.getKey().equals(k)) {
            return record.getValue();
        }
    }
    return null;
}
```

- must walk the phonebook until found
- on average, `this.records.size() / 2`
- $\Rightarrow O(n)$

HashMaps

A bit of `ArrayList` and `LinkedList`!

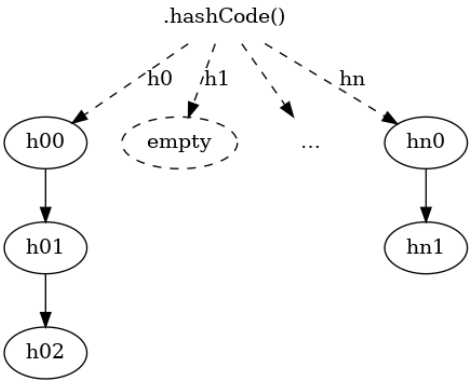


Figure 2: Structure of a HashMap

Properties

A clever implementation

- uses `.hashCode()` on the key: $O(1)$
- each list as long as the number of collisions (if `.hashCode` is good, then few): $O(c)$
- (see birthday problem)

Consequences

- fast access
- fast insertion
- resizing costs when it gets too full (initial capacity / load factor)

Asynchronous programs

Principles

A key distinction

Regular values

- data: raw types, objects...
- can be created dynamically
- can be stored
- passed to functions

Functions

- structural unit
- (similar to loops)
- no dynamic handling

Program flow

- **imperative**: "recipe", sequence of instructions
- **object**: "sections" in the program not executed linearly

→ what about reactions to events?

- could the program be in "several locations" at once?
- how could each step anticipate everything that could happen? (and always the same anyway)
- check some central state once in a while?

Use cases

- long (> 100 ms) calls: network
- user interaction (games, anyone?)
- graphical user interfaces (don't want everything to freeze)

How to represent reactions?

Use cases

- long (> 100 ms) calls: network
- user interaction (games, anyone?)
- graphical user interfaces (don't want everything to freeze)

How to represent reactions?

Reactions are functions so... could we pass functions after all?

Functions as values

So what if we could...

- store a function into a variable?
- associate it to a key? (menus...)
- pass it to another function?

```
boolean isEven(int n) {
    return n % 2 == 0;
}
someList.filter(isEven); // Error: cannot find symbol
```

Interfaces!

- classes can have (several!) methods
- passing an object is a way to pass its methods
- only need a convention to find the (unique) method

→ this is called an interface!

```
interface IntPredicate {
    public boolean run(int input);
}
```

Representing a function

- special case: interface with *only one* method to implement
- the class is a simple "wrapper" around it
- conventional name to find it
- (can have other methods but only 1 abstract)

→ it's called a functional interface (pragma `@FunctionalInterface`) !

```
@FunctionalInterface
interface IntPredicate {
    public boolean run(int input);
}
```

isEven as an IntPredicate

```
class IsEven implements IntPredicate {
    public boolean run(int input) {
        return input % 2 == 0;
    }
}

IntPredicate isEven = new IsEven();
isEven.run(2); // true
isEven.run(5); // false
```

Ad-hoc inheritance

- `abstract` as a "debt" in methods
- → "settle the bill"
- you can build *ad-hoc* full-fledged classes from `interface`s and `abstract` ones!

Ad-hoc inheritance

- `abstract` as a "debt" in methods
- → "settle the bill"
- you can build *ad-hoc* full-fledged classes from `interface`s and `abstract` ones!

```
IntPredicate isEven = new IntPredicate() {
    public boolean run(int input) {
        return input % 2 == 0;
    }
}
isEven.run(6); // true
```

Leaving without paying

```
IntPredicate isEven = new IntPredicate();
```

Leaving without paying

```
IntPredicate isEven = new IntPredicate();

|  Error:
|  IntPredicate is abstract; cannot be instantiated
|  IntPredicate isEven = new IntPredicate();
|                        ^-----------------^
```

Still longish

- dropped the empty `class` shell
- instantiate directly as we implement

but

- still several imbricated `{...}`
- have to mind the keywords

we just want to map a `<VARIABLE>` *to* an `<EXPRESSION>` (or `<STATEMENT>`)

Still longish

- dropped the empty `class` shell
- instantiate directly as we implement

but

- still several imbricated `{...}`
- have to mind the keywords

we just want to map a `<VARIABLE>` *to* an `<EXPRESSION>` (or `<STATEMENT>`)

inline function, aka a "λ" from λ-calculus, (Alonzo Church, 1930s)

```
a -> b
```

Syntax

assuming

- `(a, b, ...)` is a tuple of *n* variables (parentheses optional when 1 only)
- `<VALUE>` is an expression
- `<STATEMENT>` is a statement (often of the form `{ ...; ...; ...; }`)

```
(a, b, ...) -> <EXPRESSION>
(a, b, ...) -> <STATEMENT>
```

are values for a given functional interface

Examples

```
(n, m) -> n+m
x -> {System.out.println(x); return -x;}
```

A little bit shorter

The previous example becomes

```
IntPredicate isEven = n -> n % 2 == 0
```

- no mention of `run` any longer

A little bit shorter

The previous example becomes

```
IntPredicate isEven = n -> n % 2 == 0
```

- no mention of `run` any longer
- but still have to mind it!

```
isEven(2);
// Error:
|  cannot find symbol
|    symbol:   method isEven(int)
```

A little bit shorter

The previous example becomes

```
IntPredicate isEven = n -> n % 2 == 0
```

- no mention of `run` any longer
- but still have to mind it!

```
isEven(2);
// Error:
|  cannot find symbol
|    symbol:   method isEven(int)
```

```
isEven.run(2); // true
```

Method reference

- what about existing functions?
- can be wrapped into a λ, but boring:

```
n -> someObject.someMethod(n)
```

- can't invoke regular functions except to apply them
- but you can *refer* to a method with `::`

Example

```
class Arithmetic {
    ...
    public static boolean isEven(int input) {
        return input % 2 == 0;
    }
    public static boolean isOdd(int input) { ... }
    public static boolean isPrime(int input) { ... }
    ...
}
IntPredicate[] predicates = {Arithmetic::isEven,
                             Arithmetic::isPrime};
predicates[0].run(2); // true
Arithmetic.isEven(2); // true
```

Generalizing a bit

```
@FunctionalInterface
interface Function<I, O> {
    public O run(I input);
}

class Multiple {
    int i;
    public Multiple(int i) {
        this.i = i;
    }
    public boolean divisible(int j) {
        return j % this.i == 0;
    }
}
```

Full example

Remember?

```
someList.filter(isEven); // Error: cannot find symbol
```

Full example

Remember?

```
someList.filter(isEven); // Error: cannot find symbol

List<Integer> smallerThan10 = new LinkedList<Integer>();
for(int i = 0; i < 10; i++) {
    smallerThan10.add(i);
}

Multiple by3 = new Multiple(3);
smallerThan10.removeIf(by3::divisible)
smallerThan10; // smallerThan10 ==> [1, 2, 4, 5, 7, 8]
```

Graphical User Interfaces

Libraries

AWT

Basic tooling

- older
- events (key presses)
- notion of `Component`
- `Graphics` surface to draw
- definition of `Layout`

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

Swing

More advanced

- more recent
- heavily depends on AWT anyway
- advanced widgets: dialogs, etc
- easier styling

```
import javax.swing.*;
import javax.swing.colorchooser.*;
import javax.swing.plaf.*; // pluggable look-and-feel
```

General idea

a tree of components

- you plug components into containers (also components)
- recursively up to the root window
- components can appear only once

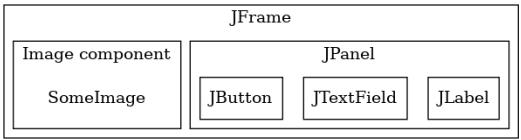


Figure 3: Example window structure

The 2D API

- within `AWT`
- useful for:
  - geometric primitives
  - text
  - images

A different approach

- you implement `Component`
- a `Graphics` object is passed around in the `paint` method
- you draw to it

Documentation

Packages

https://docs.oracle.com/javase/7/docs/api/overview-summary.html

Tutorials to AWT's 2D API

https://docs.oracle.com/javase/tutorial/2d/overview/index.html

Tutorials to Swing components

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

Swing examples

https://docs.oracle.com/javase/tutorial/uiswing/examples/components/index.html

Simple Swing example

Minimal window

```
import javax.swing.*;
public class EmptyWindow {
  private static void createAndShowGUI() {
    JFrame frame = new JFrame("Some window title");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
  }
  public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> createAndShowGUI());
  }
}
```

Translation

```
import javax.swing.*;


public class EmptyWindow {
...
}
```

access to

- JFrame
- SwingUtilities

a class for our program

Translation

```
private static void createAndShowGUI() {
  JFrame frame = new JFrame("Some window title");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.pack();
  frame.setVisible(true);
}
```

a function to draw our window

- A window is a `JFrame`, set its title
- make the program end when the window closes
- find a size that works for components
- show the window (yes!)

Translation

```
public static void main(String[] args) {
  SwingUtilities.invokeLater(() -> createAndShowGUI());
}
```

- still a program like any other, needs the usual `main`
- schedule a rendering (see the lambda?)
- where did the control flow go?

With a panel

```
import java.awt.GridLayout;
  ...
  private static void createAndShowGUI() {
    ...
    JPanel jpanel = new JPanel(new GridLayout(2, 2));
    frame.add(jpanel);
    ...
  }
```

With a couple widgets

```
...
private static void createAndShowGUI() {
  ...
  JLabel label = new JLabel("Hey there!");
  JTextField input = new JTextField();
  JButton submit = new JButton("click me");
  jpanel.add(label);
  jpanel.add(input);
  jpanel.add(submit);
  ...
}
```

Add reactions to events

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
  ...
  private static void createAndShowGUI() {
    ...
    submit.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("Received " + e);
      }
    });
    ...
  }
```

Display image

```
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.*;
import javax.imageio.ImageIO;
class ImageViewer extends Component {
  private BufferedImage img;
  ImageViewer(String path) {
    try {
      img = ImageIO.read(new File(path));
    } catch (IOException e) {}
  }
  public void paint(Graphics g) {
    g.drawImage(this.img, 0, 0, null);
  }
}
```