

Types



Architecture



Real-world projects



# IST-4-JAV Java Programming

## Class 5 - New heights

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



## 1 Types

## 2 Architecture

## 3 Real-world projects

Types



Architecture



Real-world projects



# Types

Types



Architecture



Real-world projects



## Enumerations

# Need

```
class PureBreedCat extends DomesticCat {  
    String breed;  
    ...  
}
```

String ?

- 1 for user: what are the possible values ?
- 2 possible errors (case, typos...)
- 3 need to validate input
- 4 (⇒ what to do with unexpected values ?)  
⇒ type too “large”

# A possible solution

need a finite number of values  
that are

- unique
- easy to compare
- associated to names

⇒ constants of any integral  
type will do !

# A possible solution

need a finite number of values  
that are

- unique
- easy to compare
- associated to names

⇒ constants of any integral  
type will do !

- 1 and 2 are fixed !
- 3 and 4 remain : (

```
final static byte SIAMESE = 0;  
final static byte CHARTREUX = 1;  
final static byte SAVANNAH = 2;  
...
```

# The enum types

- fixed set of **symbolic** constants
- introduced by `enum` instead of `class`
- names separated by `,` (ended by `;`)
- it's a class like any other ! (fields, methods)

```
enum CatBreed {  
    SIAMESE, CHARTREUX, SAVANNAH  
}
```

```
CatBreed breed = CatBreed.SIAMESE;  
breed == CatBreed.SAVANNAH; // false
```

## Advanced example

- all values are constructed statically
- properties can be initialized
- optional constructor can't be public

```
enum BreedType {  
    NATURAL, MUTATION, HYBRID  
}  
  
enum CatBreed {  
    SIAMESE (BreedType.MUTATION),  
    CHARTREUX (BreedType.NATURAL),  
    SAVANNAH (BreedType.HYBRID);  
    private BreedType type;  
    CatBreed (BreedType type) { this.type = type; }  
    BreedType getBreedType() { return this.type; }  
}
```

Types



Architecture



Real-world projects



## Type variables

# Remember containers ?

```
List<Integer>
```

```
List<Double>
```

```
List<String>
```

```
List<List<String>>
```

...

- a “family” of classes (infinite)
- work on any type

# Methods signatures

```
List<Integer> ints = new ArrayList<>();  
List<String> strings = new ArrayList<>();  
for(int i = 0; i < 10; i++) {  
    ints.add(i);  
    strings.add(Integer.toString(i));  
}
```

→ what is the signature of .add ?

# An “input”

```
void add(Integer n) // ??  
void add(String s) // ??
```

- type of their methods depend on the type context
- → need to be named in the class declaration
- “input” of a class, like arguments are the inputs of a function
- **type variables**

# Generics

- an identifier (any variable name)
- inside the “diamond” <>
- usually a single letter:
  - N → “number” type
  - K → “key” in an association
  - V → “value” in an association
  - T → any type
- multiple parameters are comma separated <K, V>

# May appear

## in classes

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

## in methods

as the last modifier, just before the return type

```
public <T> List<T> preview(List<T> full, int size);
```

<T> is not the return type, it's like a flag.

# Co/contravariance again

is `List<Integer>` a “subtype” of `List<Object>` ?

# Co/contravariance again

is `List<Integer>` a “subtype” of `List<Object>` ?

= “if I get a `List<Integer>`, do I have a `List<Object>` ?”

```
List<Integer> ints = new ArrayList<>();  
ints.add(4);
```

## Co/contravariance again

is `List<Integer>` a “subtype” of `List<Object>` ?

= “if I get a `List<Integer>`, do I have a `List<Object>` ?”

`List<Integer> ints = new ArrayList<>();`

`ints.add(4);`

...yes ?

- `Object`
- → `Number`
- → `Integer`

`Object o = ints.get(0);`

# Co/contravariance again

is `List<Integer>` a “subtype” of `List<Object>` ?

= “if I get a `List<Integer>`, do I have a `List<Object>` ?”

```
List<Integer> ints = new ArrayList<>();
```

```
ints.add(4);
```

...yes ?

- Object
- → Number
- → Integer

```
Object o = ints.get(0);
```

but

```
ints.set(0, o);
```

```
| Error:
```

```
| incompatible types:
```

```
| java.lang.Object cannot be converted
```

```
| to java.lang.Integer
```

```
| ints.set(0, o)
```

# Wildcards

...but it's a subtype of `List<?>` !

- syntax: ?
- the “unknown type”
- ⇒ allows to get sub/supertypes !

## Remark

no name ⇒ strictly less powerful (!)

## Refine a bit

```
Object first(List<Object> l) {  
    return l.size() > 0 ? l.get(0) : null;  
}
```

→ cannot use on ints

## Refine a bit

```
Object first(List<Object> l) {  
    return l.size() > 0 ? l.get(0) : null;  
}
```

→ cannot use on ints

```
Object first(List<? extends Object> l) {  
    return l.size() > 0 ? l.get(0) : null;  
}
```

# Conversely

```
List<Object> stuff = new ArrayList<>();  
void addInt(List<Integer> l) {  
    l.add(4);  
}  
stuff.add(4); // works  
addInt(stuff); // does not !
```

→ cannot use on stuff

## Conversely

```
List<Object> stuff = new ArrayList<>();  
void addInt(List<Integer> l) {  
    l.add(4);  
}  
stuff.add(4); // works  
addInt(stuff); // does not !
```

→ cannot use on stuff

```
void addInt(List<? super Integer> l) {  
    l.add(4);  
}  
addInt(ints); // still works  
addInt(stuff); // works too !
```

# Bounds

## **extends**

- any type that derives from or implements the given type
- useful on covariant operations
- works for generic types and wildcards

<? **extends** Integer>

<T **extends** Integer>

## **super**

- any type which the given type derives from
- useful on contravariant operations
- only for wildcards ! not generics

<? **super** Integer>

can be combined with &: <T extends Comparable & Number>

## Use case

```
static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> keyExtractor)
```

- $\langle T, U \dots \rangle$ : two type parameters ( $T$  and  $U$ )
- $U$  must be comparable with any type inheriting it
- $\text{Comparator} < T \rangle$ : this returns a comparator (on objects of type  $T$ )
- $\text{keyExtractor}$  a function
  - $\langle ? super T \rangle$  from something  $T$  inherits from
  - $\langle ? extends U \rangle$  to something that inherits  $U$

“If you can map  $T$  to a type  $U$  which you know how to compare, you can compare  $T$ ”

Types



Architecture



Real-world projects



## Error types

# Warning fellow developers

```
int divideByZero(int i) {  
    return i / 0;  
}
```

```
jshell> divideByZero(4)  
| Exception java.lang.ArithmeticException: / by zero  
|     at divideByZero (#14:2)  
|     at (#15:1)
```

*runtime errors vs. static errors*  
→ could we make it a bit safer ?

## Syntax throws

```
int divideByZero(int i) throws ArithmeticException {  
    return i / 0;  
}
```

- throw in the code (statement)
- throws in the type



## Syntax throws

```
int divideByZero(int i) throws ArithmeticException {  
    return i / 0;  
}
```

- throw in the code (statement)
- throws in the type



```
try {  
    ...  
} catch(SomeException e) {  
    ...  
} catch(OtherException e) {  
    ...  
}
```

# A real-life example

```
class FileInputStream extends InputStream {  
    ...  
    FileInputStream(String filePath) throws FileNotFoundException  
    ...  
}  
  
abstract class InputStream {  
    ...  
    int read() throws IOException  
    ...  
}
```

# Implementation

```
String readFile(String filePath) throws FileNotFoundException,  
                                IOException {  
    InputStream input = new FileInputStream(filePath);  
    StringBuffer output = new StringBuffer();  
    int read = input.read();  
    while (read >= 0) {  
        output.append((char) read);  
        read = input.read();  
    }  
    return output.toString();  
}
```

## Remark on the classes hierarchy

- Throwable
- → Exception
- → IOException
- → FileNotFoundException

## Remark on the classes hierarchy

- Throwable
- → Exception
- → IOException
- → FileNotFoundException

```
String readFile(String filePath) throws IOException {  
    ...  
}
```

Types



Architecture



Real-world projects



# Architecture

Types



Architecture



Real-world projects



# Patterns

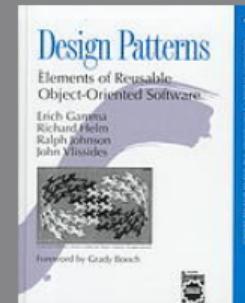
# Origins

## Approach

- how to handle large projects ?
  - experience as a programmer (c.f. chess)
  - similarities from one project to the other
- programming as a *craft* vs. programming as *science*

## A famous book

- “Design Patterns: Elements of Reusable Object-Oriented Software”
- four authors (“gang of four”), (1994)
- huge best-seller, impact on the community
- presents 23 patterns



# Controverse

Very praised in 1994 . . . then progressively criticized

- useless or redundant patterns
- attacks against the concept of “pattern” itself
  - symptoms of “faults” in the language
  - missing features

→ Part of Java’s history

# Singleton

- ensure one instance only
- typical use: logger
- like a global variable + lazy loading

# Singleton example

```
class Logger {  
    private static Logger instance;  
    private Logger() {  
        System.out.println("Created a logger");  
    }  
    public static Logger getInstance() {  
        if(Logger.instance == null) {  
            instance = new Logger();  
        }  
        return Logger.instance;  
    }  
}
```

# Factory

- hide the constructor (“alternative constructors”)
- level some internal complexity
- decouple the consumer class and the choice of a particular implementation

## Factory example: private constructors

```
class Sound {  
    private MP3 mp3Data;  
    private Ogg oggData;  
    private Sound(MP3 mp3Data) {  
        this.mp3Data = mp3Data;  
    }  
    private Sound(Ogg oggData) {  
        this.oggData = oggData;  
    }  
    ...
```

## Factory example: ... with a public method

```
public Sound fromFile(String path) {  
    if (... == ".mp3") {  
        return new Sound(new MP3(path));  
    } else {  
        return new Sound(new Ogg(path));  
    }  
}
```

# Decorator

- add behaviour to a particular instance, not the whole class
- dynamic modification of functionality
- idea of “wrapping” around an existing object
- ≠ inheritance (runtime vs. compile-time)
- (inheritance à la Javascript)
- **circling around the same interface**

## Decorator example: an interface and a base class

```
interface Cat {  
    public String getDescription();  
}  
  
class AnyCat implements Cat {  
    public String getDescription() {  
        return "A cat";  
    }  
}
```

## Decorator example: the class for decorators

```
abstract class CatDecorator implements Cat {  
    private final Cat decoratedCat;  
    public CatDecorator(Cat c) {  
        this.decoratedCat = c;  
    }  
    public String getDescription() {  
        return this.decoratedCat.getDescription();  
    }  
}
```

## Decorator example: and one implementation

```
class WithHat extends CatDecorator {  
    public WithHat(Cat c) { super(c); }  
    public String getDescription() {  
        return super.getDescription() + " with a hat";  
    }  
}
```

# MVC: about

- Model — View — Controller
- even earlier than the “Design patterns” book ! (1978)
- general principle of “Separation of concerns”, modularity, etc.

## Model: example

```
class VolumeState {  
    private float volume = 0.5f;  
    private boolean mute = false;  
    public void changeVolume(float deltaVolume) {  
        float newVolume = this.volume + deltaVolume;  
        this.volume = Math.min(1, Math.max(0, newVolume));  
    }  
    public void toggleMute() {  
        this.mute = !this.mute;  
    }  
    public float getVolume() {  
        return this.mute ? 0 : this.volume;  
    }  
}
```

# Controller

```
class Knob {  
    private VolumeState volumeState;  
    public Knob (VolumeState volumeState) {  
        this.volumeState = volumeState;  
    }  
    public void handle(String command) {  
        switch(command) {  
            case "+": this.volumeState.changeVolume(0.05f); break;  
            case "-": this.volumeState.changeVolume(-0.05f); break;  
            case "m": this.volumeState.toggleMute(); break;  
            default: System.err.println("Unknown: " + command);  
        }  
    }  
}
```

## View: example

```
class VolumeBar {  
    private byte width;  
    private VolumeState volumeState;  
    private Knob knob;  
    public VolumeBar(byte width, VolumeState volumeState) {  
        this.width = width;  
        this.volumeState = volumeState;  
        this.knob = new Knob(volumeState);  
    }  
    ...  
}
```

## View: continued

```
...
public void run() {
    while(true) {
        display(this.volumeState.getVolume());
        this.knob.handle(System.console().readLine());
    }
}

public void display(float level) {
    char[] bar = new char[(int) (level * this.width)];
    for(int i = 0; i < bar.length; i++) {
        bar[i] = '#';
    }
    System.out.println(bar);
}
```

Types



Architecture



Real-world projects



# Real-world projects

Types



Architecture



Real-world projects



# Javadoc

# Purpose

- normalize documentation
- make it easy to browse  
(HTML)
- part of the development process

| All Methods       | Static Methods             | Instance Methods                                                           | Concrete Methods | Deprecated Methods |
|-------------------|----------------------------|----------------------------------------------------------------------------|------------------|--------------------|
| Modifier and Type | Method                     | Description                                                                |                  |                    |
| char              | charAt(int index)          | Returns the char value at the specified index.                             |                  |                    |
| InputStream       | chars()                    | Returns a stream of int zero-extending the char values from this sequence. |                  |                    |
| int               | codePointAt(int index)     | Returns the character (Unicode code point) at the specified index.         |                  |                    |
| int               | codePointBefore(int index) | Returns the character (Unicode code point) before the specified index.     |                  |                    |

# Usage

```
/**  
 * Here a general description  
 * of the method  
 * @param name description  
 * @return description  
 */
```

javadoc [PACKAGE | SOURCE-FILE]

(don't forget about visibility !)

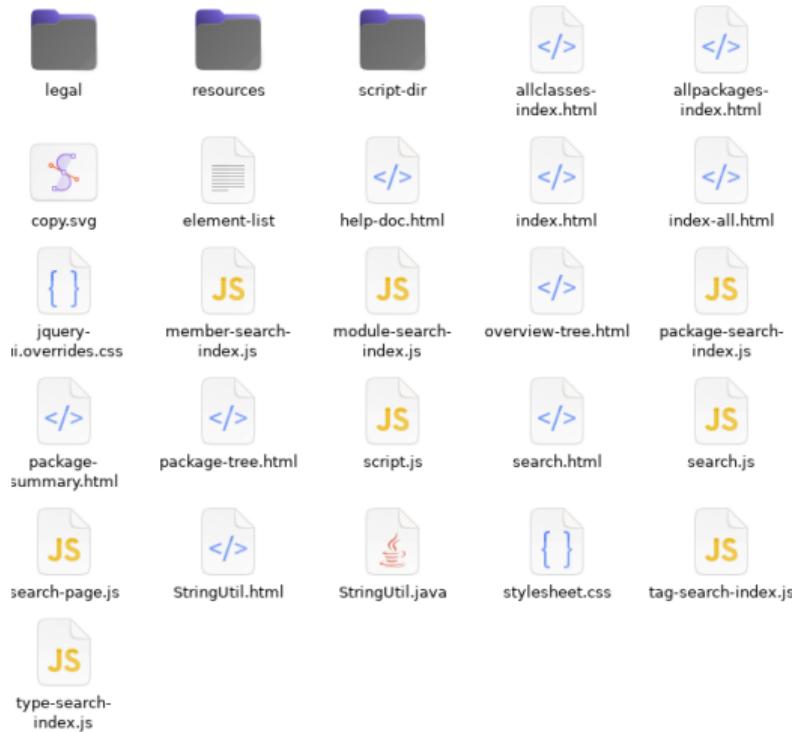
## Tags

- @param
- @return
- @author
- @deprecated
- @throws / @exception

## Example

```
/** A class exposing handy functions to generate Strings */
public class StringUtil {
    /** Replicate a String several times
     *  @param input the initial String
     *  @param n the number of repetitions
     *  @return the initial String concatenated the n times */
    public static String replicate(String input, int n) {
        if(times == 0) {
            return "";
        } else {
            return input + replicate(input, times - 1);
        }
    }
}
```

# Generated files



# The main page (index.html)

The screenshot shows a web-based interface for viewing Java code. At the top, there's a navigation bar with tabs: PACKAGE (which is selected and highlighted in orange), CLASS, TREE, INDEX, and HELP. Below the navigation bar, there's a breadcrumb trail: PACKAGE: DESCRIPTION | RELATED PACKAGES | CLASSES AND INTERFACES. The main content area has a title "Unnamed Package" followed by a horizontal line. Below this, there's a table-like structure with two columns: "Classes" (which is also highlighted in orange) and "Description". A single row is visible, showing "StringUtil" under "Class" and "A class exposing handy functions to generate Strings" under "Description".

| Classes    | Description                                          |
|------------|------------------------------------------------------|
| StringUtil | A class exposing handy functions to generate Strings |

no package, and javadoc resents it!

# The class itself

| All Methods                                                                          | Static Methods                 | Concrete Methods                 |
|--------------------------------------------------------------------------------------|--------------------------------|----------------------------------|
| Modifier and Type                                                                    | Method                         | Description                      |
| static String                                                                        | replicate(String input, int n) | Replicate a String several times |
| <b>Methods inherited from class java.lang.Object</b>                                 |                                |                                  |
| clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait |                                |                                  |
| <b>Constructor Details</b>                                                           |                                |                                  |
| <b>StringUtil</b>                                                                    |                                |                                  |
| public StringUtil()                                                                  |                                |                                  |
| <b>Method Details</b>                                                                |                                |                                  |
| <b>replicate</b>                                                                     |                                |                                  |
| public static String replicate(String input, int n)                                  |                                |                                  |
| Replicate a String several times                                                     |                                |                                  |
| <b>Parameters:</b>                                                                   |                                |                                  |
| input - the initial String                                                           |                                |                                  |
| n - the number of repetitions                                                        |                                |                                  |
| <b>Returns:</b>                                                                      |                                |                                  |
| the initial String concatenated the n times                                          |                                |                                  |

Types



Architecture



Real-world projects



# Classpath

# In java

- projects grow
- several dependencies (libraries)

**Where to find the code ?**

# In java

- projects grow
- several dependencies (libraries)

## Where to find the code ?

current directory (.) but not very flexible (compilation)

# In java

- projects grow
- several dependencies (libraries)

## Where to find the code ?

current directory (.) but not very flexible (compilation)

### Cross-tool concern

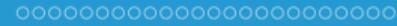
- jshell
- javac
- java
- javadoc
- jar

# From the shell

- list of “code bases”: folders or jar
- :-separated ( $\equiv \$PATH$ )
- equivalent option from all the ecosystem
  - --class-path, -classpath, -cp
  - same syntax

```
echo $CLASSPATH  
"path/to/some/library:path/to/another:current/project"
```

Types



Architecture



Real-world projects



JAR

# Archives

- contain the code
- easily distributed
- (lighter)
- independent from packages

## Syntax ( $\equiv$ tar)

- -t: list
- -c: create
- -x: extract

```
jar cvf MyApp.jar *.class
```

```
jar xvf MyApp.jar
```

```
jar tvf MyApp.jar | less
```

# Using libraries

*Main.java*

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(StringUtil.replicate(args[0], 3));  
    }  
}
```

```
javac Main.java
```

# Using libraries

*Main.java*

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(StringUtil.replicate(args[0], 3));  
    }  
}
```

```
javac Main.java
```

```
Main.java:3: error: cannot find symbol  
        System.out.println(StringUtil.replicate(args[0], 3));  
                           ^  
symbol:   variable StringUtil  
location: class Main
```

# Using it

assuming ../library.jar contains StringUtil

```
javac --class-path ../library.jar Main.java  
java --class-path "../library.jar:." Main test
```

Let's create a JAR

```
jar --create -f ../main.jar Main.class
```

we can run the app

```
java -cp "../library.jar:../main.jar" Main test
```