IST-4-JAV Java Programming
Class 3 - Modeling the structure

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

1. Structuring objects
2. Structuring code
3. Structuring projects

Structuring objects

Inheritance

Object composition

1st approach

- cartesian product (×)
- "tie" fields together

Example

Modeling a (video game) cat:

- age: a very small integer
- hit points: some small integer
- max hit points: some small integer
- rest level: decimal number $\in [0, 1]$

First draft of a Cat

```
class Cat {
    byte age;
    short hitPoints;
    short maxHitPoints;
    float restLevel;
    static short babyMaxHitPoints = 3;
    Cat() {
        this.age = 0;
        this.maxHitPoints = babyMaxHitPoints;
        this.hitPoints = this.maxHitPoints;
        this.restLevel = 1;
    }
}
```

How about domestic cats ?

→ take a `Cat`, and add a name to it ?

```
class DomesticCat {
    Cat innerCat;
    String name;
    DomesticCat(String name) {
        this.name = name;
        this.innerCat = new Cat();
    }
}
```

Not everything is composition !

- `innerCat` ? sounds bad
- we'll eventually add methods: `feed`, `run`...: how will it work for our `DomesticCat` ?

```
void feed() {
    this.innerCat.feed()
}
void run() {
    this.innerCat.run();
}
```

$\Longrightarrow$ with this definition a `DomesticCat` is *not* a `Cat`

Inheriting

conceptually differs from composition

- distinguish a "special case"
- useful when both classes model the same "kind" of real-world objects
- "A *is* a B"

$\Longrightarrow$ perfect for our cats !

Programming: models for structures and behaviours, not just data

Example: Object

inheritance is not rare:

- any class has exactly 1 *parent*/*mother*/*superclass*
- except `Object` ! sits at the bottom
- actually. . . always inherits at least `Object`

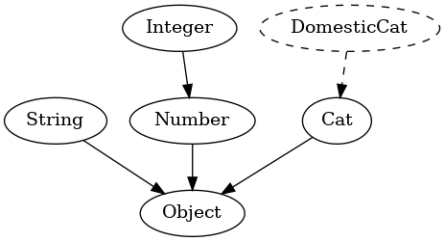

Figure 1: Classes form a tree

Example: Exceptions

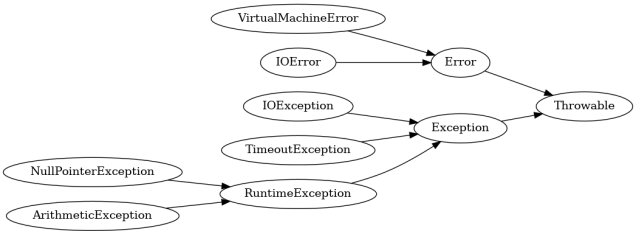

Figure 2: A taxonomy of trouble

- `Throwable`: anything that can be `thrown`
- `Error`: serious "physical" trouble
- `Exception`: recoverable failure
- `TimeoutException`: give up on blocking operation
- `NullPointerException`: tried to dereference `null`

Syntax: `extends`

```
class DomesticCat extends Cat {
    String name;
    DomesticCat(String name) {
        this.name = name;
    }
}
```

declares a new class `DomesticCat` which *inherits* `Cat`: it has everything a `Cat` has !

```
DomesticCat pangur = new DomesticCat("Pangur");
pangur.name; // == "Pangur"
pangur.age; // == 0
pangur.restLevel; // == 1.0
```

Wait ?!

- `pangur.name` got initialized: ok, we did that
- `pangur.age` and `pangur.restLevel` got initialized: why ? also how ?

magic happened: Java called the mother class constructor !

but what would've happened if an argument was needed... ?

Mother/child relationships

Yet another inheritance !

People start caring about the *breed* of their cats:

```
class PureBreedCat extends DomesticCat {
    String breed;
    PureBreedCat(String breed) {
        this.breed = breed;
    }
}
```

```
|  Error:
|  constructor DomesticCat in class DomesticCat cannot be applied to given types;
|    required: java.lang.String
|    found:    no arguments
|    reason: actual and formal argument lists differ in length
```

A hidden call !

```
new DomesticCat()
```

```
|  Error:
|  constructor DomesticCat in class DomesticCat cannot be applied to given types;
|    required: java.lang.String
|    found:    no arguments
|    reason: actual and formal argument lists differ in length
|  new DomesticCat()
```

→ there was a (hidden) call to `DomesticCat();`

Calling the constructor

- the automatically added call takes no argument
- `DomesticCat` doesn't have a constructor with no argument !
- we need to manually call the constructor "above"

```
class PureBreedCat extends DomesticCat {
    String breed;
    PureBreedCat(String name, String breed) {
        super(name);
        this.breed = breed;
    }
}
```

`super`

- *meta* keyword, refer to a class
- depending on the context ! (|| `this`)
- "the parent class"

Look !: call to a function, not a method

Fixed it !

```
PureBreedCat billy = new PureBreedCat("Billy", "siamese");
billy.name; // == "Billy"
billy.breed; // == "siamese"
billy.age; // == 0
billy.restLevel; // == 1.0
```

→ now we know where everything comes from !

Overriding

- inheriting methods is good
- but sometimes we want to change their behaviour
- `@Override` pragma: hint for `javac`

```
class Cat {
    ...
    void feed() {
        System.out.println(
            "hunts"
        );
    }
}
```

```
class DomesticCat extends Cat {
    ...
    @Override
    void feed() {
        System.out.println(
            "mews until a human"
            + " feeds it"
        );
    }
}
```

Another use for super

- redeclaring a method in a child class "hides" the parent's implementation
- how to reuse the existing implementation ?
- `super` again !

```
class DomesticCat extends Cat {
    ...
    void feed() {
        if(isHumanAround()) {
            System.out.println("mews a lot");
        } else {
            super.feed();
        }
    }
}
```

Example: Built-in methods

defined in `Object` (so all objects in Java have them)

- `.equals`: object comparison
- `.toString`: object representation
- `.hashCode`: at-a-glance comparison (fast, vs. accurate `.equals`)

→ all *default* implementations meant to be overridden

```
jshell> pangur
pangur ==> DomesticCat@723279cf
```

default `.toString`: *Class name* + `@` + *memory location*

A nicer display

```
class DomesticCat extends Cat {
    String toString() {
        return this.name + ", a cat";
    }
}
class PureBreedCat extends DomesticCat {
    String toString() {
        return this.name + ", a " + this.breed + " cat";
    }
}
jshell> pangur
pangur => Pangur, a cat
jshell> billy
billy => Billy, a siamese cat
```

Overloading

Signature

- the number of arguments
- their order
- their type

Added, not replaced

Java finds functions by their name **and** signature

- classes can have several constructors
- methods can have several implementations
- they can have different *output* types
- the arguments + name must be unique (can't rely on *output*)

Example: feeding

(assuming we never wrote the previous implementation with `super`)

```
class DomesticCat extends Cat {
    void feed(Human h) {
        System.out.println("mews until human feeds it");
    }
}
```

- doesn't replace the default implementation in `Cat` (hunting)
- in the context where a human is passed, mews instead
- a "conditional" without `if` or ternary operator.

Structuring code

Access control

Example: a cat's `name`

can't ask its name to a cat !

Example: a cat's `name`

can't ask its name to a cat !

but

```
pangur.name; // returns a String ("Pangur")
pangur.name = "Marcel"; /* now Pangur has a different name ! */
pangur.name; // "Marcel"
```

The `private` keyword

```
class DomesticCat extends Cat {
    private String name;

    // but still can interact with it !
    void call(String name) {
        if(this.name.equals(name)) {
            System.out.println("mews and comes");
        } else {
            System.out.println(
                "looks away and yawns"
            );
        }
    }
}
```

Why ?

- more realistic
- hides away implementation details
- avoid mistakes
- classes as an "area"
- (remember "structure", not "data" ?)

→ abstraction

Another useful flag: `final`

`Math.PI` → `static final double`
`Math.E`

- prevent from changing a variable
- the variable itself (not what it may refer to ! remember graphs)
- again: numbers, objects...

Not for variables only

- **On methods**: cannot be overridden
- **On class**: cannot be extended

Getters

- by default variables are read-write
- read-only can be achieved by `final`
- "access control": actually read-write, but can't be changed from *outside*
- notion of *view* (there doesn't have to be a corresponding field)

```
class Cat {
    private int age;
    int getAge() {
        return this.age;
    }
}
```

Setters

- useful even for read-write !
- separates feature / implementation

```
class Cat {
    private int age;
    void setAge(int age) {
        if(age > this.age) {
            this.age = age;
        }
    }

    void happyBirthday() {
        this.age++;
    }
}
```

Advanced setters

- side-effects
- control the flow

```
class MovingAnimal {
    private float restLevel;
    private Point at;
    static float range;
    void setAt(Point newAt) {
        float distance = distance(this.at, newAt);
        this.at = newAt;
        this.restLevel *= Math.exp(-distance / range);
    }
}
```

Building blocks

Abstract classes and methods

*What's a **predator** ?*

Abstract classes and methods

*What's a **predator** ?*

- the Idea without a Form
- implements some behaviour
- can't be instantiated, a "draft"
- exists only to be inherited (*factorize* code)

Syntax: `abstract`

- on the class itself
- on the methods without implementation
- incompatible with `final` !

```
abstract class Predator {
    abstract protected void catchPrey();
    protected void feed() {
        this.catchPrey();
        System.out.println("eats it");
    }
}
```

Interfaces

- a "contract": requirements
- a "definition" (→ flexible "type" for classes)
- not meant for instantiation either
- variables are all `static` and `final`

almost *opposite* of abstract classes

Abstract classes

- partial implementation (delay a "choice")
- really a class (inherited like any other)
- (only one mother class, `abstract` or not)

Interfaces

- what, not how
- not inherited, *implemented*
- (no restriction on number)

Syntax: `interface` then `implements`

```
interface Animal {
    int getAge();
    void feed();
}
```

"We call `Animal` any object which. . . "

```
class Cow implements Animal {
    private int age;
    int getAge() {
        return this.age;
    }
    void feed() {
        System.out.println("grazes");
    }
}
```

"A cow is an Animal, and here's the proof"

Bridges between interfaces and abstract classes

Extending

- an `interface` can be inherited (`extends`, same syntax)
- sort of prerequisites, set inclusion: all birds are animals

Partial implementation

- a class may `implement` an interface but not all its methods
- makes unimplemented `abstract` $\Longrightarrow$ whole class `abstract`

Default implementation

- maybe confusing ($\approx$ regular methods in `abstract`)
- perfect use case: augmenting interface backwards-compatibly

What about typing ?

Subtype vs. Derived type

Subtype

- substitutability
- `T1` < `T2`
- `T1` can replace `T2` everywhere
- ex: `int` < `long`

Derived type

- inheritance (everything above)
- "special case"

subtype $\overset{?}{\iff}$ derived type

Covariance / Contravariance

assuming `A` < `B`

→

- if I *have* an `A`
- then I have a `B`
- covariance

←

- if I *need* an `A`
- a `B` might not be enough
- contravariance

- functions: left or right of ↦ ?
- programs as games: "whose turn ?"

Contravariance reverses subtyping

Answer

- objects have methods
- methods are functions
- hence, can be contravariant

```
class Cat {
    Cat mate(Cat partner) {
        ...
    }
}

class PureBreedCat extends DomesticCat {
    PureBreedCat mate(PureBreedCat partner) {
        ...
    }
}
```

Structuring projects

Isolating parts

Packages

- applications start growing
- avoid name conflicts
- structuring things also document them
- reuse some parts
- developed by different organisations

→ packages

Creating: `package`

```
package name.of.the.package;
```

statement must be added to each file

- arbitrary names (usually "company"'s domain name)
- hierarchically left to right (opposite from internet domain names)
- all lowercase
- use _ to fix invalid names (reserved words, – and other special characters)

Virtual Path / Filesystem Path

- path to files should reflect packages hierachy
- lowercase for packages → directory
- CamelCase for classes → files

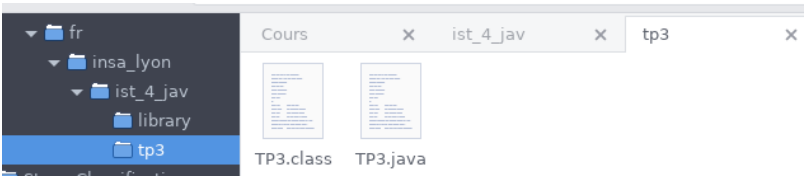

Figure 3: Two example packages

Using: `import`

```
import name.of.the.package.SomeClass; // one class
import name.of.the.package.*; // all its classes
```

- you can only import classes, not packages
- no "subpackages": prefix on names $\neq$ anything on package
- $*$ isn't a regex, and (see above) will catch only classes

More about visibility

`public`, at last !

- control the boundaries at the package level
- default: nothing outside the package
- (`jshell`: same temporary package)

`public`: entirely visible

`protected`: only in subclasses ("backstage" access)

How ?

- modifiers applied to
  - fields
  - methods
  - classes

Keywords / visibility

keyword	class	package	subclass	outside world
`private`	Y	N	N	N
(nothing)	Y	Y	N	N
`protected`	Y	Y	Y	N
`public`	Y	Y	Y	Y

Good practice

"Need to know" basis $\Longrightarrow$ start `private`, and grant access as needed

Model the reality

- who/what could get the information ?
- could it be changed from *outside* ?
- is it a convenience or something fundamental ?

A subtle remark

- `private` is very restrictive (the class to itself)
- → makes sense only *inside* a class

A subtle remark

- `private` is very restrictive (the class to itself)
- → makes sense only *inside* a class
- `protected` differs from default only for inheritance
- → makes sense only *inside* a class

Example

Source code

fr/insa_lyon/ist_4_jav/library/SomeLibrary.java

```java
package fr.insa_lyon.ist_4_jav.library;

public class SomeLibrary {
    public static String greet = "Hi there !";
}
```

Source code

fr/insa_lyon/ist_4_jav/class3/Main.java

```
package fr.insa_lyon.ist_4_jav.class3;
import fr.insa_lyon.ist_4_jav.library.SomeLibrary;

public class Main {
    public static void main(String[] args) {
        System.out.println(SomeLibrary.greet);
    }
}
```

Compiling

- `javac` works on files
- `java` works on class names
- (use autocomplete to let `java` guide you)

```
$ javac fr/insa_lyon/ist_4_jav/class3/Main.java
$ java fr.insa_lyon.ist_4_jav.class3.Main
Hi there !
```