Good practice

v The construction semantics of an object must
make sense
Personalap = newPessoa (); L
Person p = new Person ("António Nunes"); K
Pessoa p = new Pessoa ("António Nunes", 12244, dataNasc); J
v We must give a minimum of public visibility when accessing an object
- Only what is strictly necessary
v Sometimes it makes more sense to create a new object than
that changing existing attributes
Point p1 = new Point (2,3); p1.set (4,5); L

v Join members of the same type
- Do not mix static methods with instance methods
v Declaring variables before or after methods - Do not mix methods, constructors and variables
v Keep constructors together, preferably at the beginning
v If you need to define static blocks, define only one at the beginning or end of the class.
v The order of the members is not important, but following conventions improves the readability of the code

---

Class Relations

v Part of the class modeling process consists of:
- Identify entities applying for classes
- Identify relationships between these entities
v Relationships between classes are easily identified using some real models.
- For example, a Digital Clock and an Analog Clock are both types of Clock (specialization or inheritance).
- A Digital Clock, on the other hand, contains a Battery (composition).
v Relations: - IS-A
- HAS-A

Inheritance (IS-A)

v IS-A indicates specialization (inheritance), that is, when a class is a subtype of another class.

v For example:
- Pinheiro is an (IS-A) tree.
- A Digital Clock is an (IS-A) Clock.
class Watch {/ * ... * /
}
class Digital Watch extends Watch {/ * ... * /
}

Composition (HAS-A)

v HAS-A indicates that a class is composed of objects from another class.

v For example:
- Forest contains (HAS-A) Trees.
- A Digital Watch contains (HAS-A) Battery.

class Stack {/ * ... * /
}
class Digital Watch extends Watch {
Stack p;
/ * ... * /}

Class reuse

v Whenever we need a class, we can:
- Use an existing class that meets the requirements
- Write a new class from "from scratch"
- Reuse an existing class using composition
- Reuse an existing class through inheritance

Inheritance Identification

v Typical signs that two classes have an inheritance relationship
- Have common aspects (data, behavior)
- Have different aspects
- One is a specialization of the other

v Examples:
- Cat is a Mammal - Circle is a Figure - Water is a Drink

Inheritance – Concepts

v Inheritance is one of the main characteristics of OOP
- The CDeriv class inherits, or is derived from, CBase when CDeriv represents a subset of CBase

v Inheritance is represented in the form: class CDeriv extends CBase {/ * ... * /}

v Cderiv has access to CBase data and methods - which are not private in CBase

v A base class can have multiple derived classes but a derived class cannot have multiple base classes
- In Java multiple inheritance is not possible

Constructors with parameters

v In parameterized constructors the base class constructor is the first statement to appear in a derived class constructor.

```
    class Game {
        int num;
Game (int code) {...}
// ...}
class BoardGame extends Game {// ...
BoardGame (int code, int numPlayers) {super (code);
// ...}
}
```

Method Inheritance

v When inheriting methods, we can:

- keep them unchanged

```
class Person {
private String name;
public Person(String n) { name = n; } public String name() { return name; } public String toString() { return "PERSON";}

}
class Student extends Person {

private int nmec;
public Student(String s, int n) { super(s); nmec=n; } public int num() { return nmec; }

}
public class Test {

public static void main(String[] args) { Student stu = new Student("Andreia", 55678); System.out.println(stu + " : " +

stu.name() + ", " + stu.num()); }

}
```

- add new features to it

```
class Person {
private String name;
public Person(String n) { name = n; } public String name() { return name; } public String toString() { return "PERSON";}

}

class Student extends Person {
private int nmec;
public Student(String s, int n) { super(s); nmec=n; } public int num() { return nmec; }
public String toString()

{ return super.toString() + " STUDENT"; } }
```

- redefine them

```
class Person {
private String name;
public Person(String n) { name = n; } public String name() { return name; } public String toString() { return "PERSON";}

}

class Student extends Person {
private int nmec;
public Student(String s, int n) { super(s); nmec=n; } public int num() { return nmec; }
public String toString() { return "STUDENT"; }

}
```

Inheritance and access control

v We cannot reduce the visibility of inherited methods in a derived class
- Methods declared as public in the base class must be public in the subclasses
- Methods declared as protected in the base class must be protected or public in the subclasses. They cannot be private
- Methods declared without access control (default) cannot be private in subclasses
- Methods declared as private are not inherited

Final

v The final classifier indicates "cannot be changed"
v Can be used on:
- Data - final constants int i1 = 9;
- Methods - non-resettable final int swap (int a, int b) {//:
}
- Classes - not inherited final class Rat {// ...
}
v "final" sets constant attributes of primitive types but does not set objects or vectors
- in these cases what is constant is simply the reference to the object

---

Inheritance - Good Practices

v Program for the interface and not for the implementation
v Look for aspects common to various classes and promote them to a base class
v Minimize relationships between objects and organize related classes within the same package
v Use inheritance judiciously - whenever possible, favor composition

---

Methods common to all objects

v In Java, all classes are derived from the super class java.lang.Object
v Methods of this class: - toString ()
- equals ()
- hashcode () - finalize ()
- clone ()
- getClass () - wait ()
- notify ()
- notifyAll ()

---

toString ()

Circle c1 = new Circle (1.5, 0, 0); System.out.println (c1);
c1.toString () is automatically invoked
Circle @ 1afa3

- the toString () method must always be redefined to behave according to the object
public class Circulo {// ....
@Override
public String toString () {
return "Centro: (" + centro.x () + "," + centro.y () + ")" + "Radius:" + radius;
}

---

equals ()

v The expression c1 == c2 checks whether references c1 and c2 point to the same object
- If c1 and c2 are automatic variables, the previous expression compares values
v The method tests whether two objects are equal
Circle p1 = new Circle (0, 0, 1);
Circle p2 = new Circle (0, 0, 1); System.out.println (p1 == p2); // false System.out.println (p1.equals (p2)); // false (why?)
v equals () must be redefined whenever objects of this class can be compared
- Circle, Point, Complex ...

v Equality properties
- reflective: - symmetrical: - transitive:
x.equals (x) àtrue x.equals (y) ßày.equals (x)
x.equals (y) AND y.equals (z) àx.equals (z)
v We must respect the Object.equals (Object o) signature
public class Circulo {// ...
@Override
public boolean equals (Object obj) {// ...}
}
v Problems
- What if 'obj' is null?
- What if you reference an object other than a circle?

---

hashCode ()

v Whenever the equal () method is rewritten, hashCode () must also be
- Equal objects must return equal hash codes
v The purpose of the hash is to help identify any
object through an integer
// Circulo.hashCode () - Very simple example !!! public int hashCode () {
return radius * centro.x () * centro.y (); }
// ..
Circle c1 = new Circle (10,15,27); Circle c2 = new Circle (10,15,27); Circle c3 = new Circle (10,15,28);
- Building a good hash function is not trivial. Other sources are recommended for its construction

---

Summary - Why inheritance?

v Many real objects have this characteristic
v Allows you to create simpler classes with more watertight and better defined features
- We should avoid classes with very "extensive" interfaces
v It allows to reuse and extend interfaces and code
v Lets you take advantage of the polymorphism

Upcasting and downcasting
Handgeschrieben Seite 24

Polymorphism

v Base idea:
- the type declared in the reference need not be exactly the same as the type of the object it points to - can be of any derived type
Circle c1 = new Target (...); Object obj = new Circle (...);
v Polymorphic reference
- T ref1 = new S (); // OK as long as the whole S is a T

v Polymorphism is, together with Inheritance and Encapsulation, one of the fundamental characteristics of OOP.
- Different shapes with similar interfaces.
v Other designations:
- Dynamic binding, late binding or run-time binding
v This feature allows us to take more advantage of the heritage.
- We can, for example, develop an X () method with CBase parameter with the guarantee that it accepts any argument derived from CBase.
- The X () method is only resolved in execution.
v All methods (with the exception of final ones) are late binding.
- The final attribute associated with a function, prevents it from being redefined and simultaneously gives an indication to the compiler for static linking (early binging) - which is the only way of linking in languages with C.

Generalization

v Generalization consists of improving the classes of a problem in order to make them more general.
v Forms of generalization:
v Make the class as comprehensive as possible
to cover the widest range of entities.
class ZooAnimal;
v Abstract different implementations for
similar operations in abstract classes in a
upper level.
ZooAnimal.draw ();
v Gather behaviors and characteristics and make them
rise as far as possible in the class hierarchy.
ZooAnimal.weight;

Abstract classes

v A class is abstract if it contains at least one abstract method.
- An abstract method is a method whose body is not
defined.
public abstract class Forma {
// can define constants
public static final double DOUBLE_PI = 2 * Math.PI;
// can declare abstract methods public double area (); public double perimeter ();
// can include non-abstract methods
public String aka () {return "euclidean"; }}
v An abstract class is not instantiable.
Form f; // OK. We can create a reference to Form f = new Form (); // Mistake! We cannot create Forms

v In an inheritance process, the class is no longer abstract only when it implements all abstract methods.
public class Circulo extends Forma {
protected double r;
public double area () {return Math.PI * r * r;
}
public double perimeter () {return DOUBLE_PI * r;
}}
Form f;
f = new Circle (); // OK! We can create Circles
