1

MORE JAVA!

Toy.java

2

```
/**  A class representing toys that have a button */
public class Toy {
    private String name;
    private int ageLow, ageHigh;
    public Toy(string name, int ageLow, int ageHigh) { …}
    public void pushTheButton() { …}
}
```

Metodo

3

- Construttore

```
public Toy(String name, int ageLow, int ageHigh) {
    this.name = name;
    this.ageLow = ageLow;
    this.ageHigh = ageHigh;
}
```

- **this**? Permette di accedere alle variabili di istanza dell'oggetto corrente.
  - **this** is a name for the "object in which it appears"(java documentation)

"this"

4

```
/public class Toy {
    private String name;
    private int ageLow, ageHigh;
    public Toy(String name, int ageLow, int ageHigh) {
        this.name = name;
        this.ageLow = ageLow;
        this.ageHigh = ageHigh;
    }
}
```

Campi di un oggetto

5

- Creiamo un oggetto t1,
  - t1 = **new** Toy("ActionWarrior", 10, 15) crea l'oggetto invocando il metodo costruttore Toy con gli opportuni parametri
  - t1.PushTheButton() invoca il metodo push-button dell'istanza della classe Toy, oggetto t1

Costruttore e tipi

6

- Il metodo costruttore della classe Toy non restituisce alcun valore: Usato per inizializzare l'ambiente locale che mantiene le informazioni sull'oggetto creato
- PushTheButton restituisce **void**

  Opera per effetti laterali.

Ereditarieta'

7

- In Java, ogni classse e' una sotticlasse di un'altra classe
  - La classe Object e' la classe da cui ereditano tutte le classi (la radice della gerarchia). Possiede diversi metosi
    - toString(), Equals(), ….
- Le sottoclassi possono definire nuovamente questi metosi oppre ereditarli. Questo vale per ogni livello della gerarchia di ereditarieta'.
  - t1.toString() restituisce la stringa corrispondente al tipo di Toy e il nomet1.
  - Potremmo ridefinire toString() per restituire anche altre informazioni (esempio eta')

Static vs metodi di istanza

8

- Il metodo main e' dichiarato static.
- Metodi e campi possono essere dichiarati static.
- Static : il metodo o il campo sono condivisi tra tutte le istanze di oggetti della classe
  - Java Tutorial "A static method can't "see" instance methods or fields of an object unless you specify the object instance"

9

```
public class Toy {
    private String name;
    private int ageLow, ageHigh, myId;
    private static int nextId= 0; // first unassigned id
    public Toy(string name, int ageLow, int ageHigh) {
        this.name = name;
        this.ageLow = ageLow;
        this.ageHigh = ageHigh;
        myId = nextId++;
    }
}
```

Overriding "toString"

10

- Class Object defines toString, so every object of every class contains toString.
  - toString in Object: prints name@Address
  - Most classes override toString()
  - toString() in an object usually returns a string that contains values of the fields of the object, printed in a nice way.

```
@Override      // An "attribute": tells Eclipse what we intend
public string toString() {
    return this.name + ":" + this.value;
}
```

Overridden Methods

11

Suppose a class overrides method m

- It is useful to be able to call the parent version. E.g. maybe you still want to print the Name@Address using Object.toString()
- In subclass, call overridden method using super.m()

```
Public @Override String toString() {
      return super.toString() + ": " + name + ", price=" + price;
}
 .... "ns@0xAF402: Hotel Bates, price=37.50"
```

Constructors

12

Java automatically calls two.toString()
It works: class Thing inherits Object.toString().

- Called to create new instances of a class.
- A class can define multiple constructors
- Default constructor initializes all fields to default values (0, false, null…)

```
class Thing {
    int val;
    Thing(int val) {
        this.val = val;
    }
    Thing() {
        this(3);
    }
```

```
Thing one = new Thing(1);
Thing two = new Thing(2);
Thing three = new Thing();
System.out.println("Thing two = " + two);
```

Example

Prints: Csuper constructor called.
Constructor in A running.

13

```
public class CSuper {
   public CSuper() {
        System.out.println("CSuper constructor called.");
   }
}
public class A extends CSuper {
   public A() {
        super();
        System.out.println("Constructor in A running.");
    }
    public static void main(String[] str) {
          ClassA obj = new ClassA();
     }
}
```

What are local variables?

14

- Local variable: variable declared in method body
- Not initialized, you need to do it yourself!

```
class Thing {
    int val;
    public Thing(int val) {
        int undef;
        this.val = val + undef;
     }
     public Thing() {
         this(3);
     }
}
```

What happens here?

15

- If you access an object using a reference that has a **null** in it, Java throws a NullPointerException.
- Thought problem: what did developer intend?

```
class Thing {
    RoomMate myFriend;

    Thing(int val) {
            myFriend.value = val;
     }
}
```

16

- Suppose a is of type A and b is of type B
  - … and A has static field myAVal,
  - ….and B has field myBVal.
- Suppose we have static initializers:

  public static int myAVal = B.myBVal+1;

  public static int myBVal = A.myAVal+1;

17

- What happens depends on which class gets loaded first. Assume program accesses A.
  - Java "loads" A and initializes all its fields.
  - Now, static initializers run. A accesses B. So Java loads B and initializes all its fields.
  - Before we can access B.myBVal we need to initialize it.
- B sets myBVal = A.myAVal+1 = 0+1 = 1
- Next A sets A.myAVal = B.myBVal+1=1+1=2
- (Only lunatics write code like this but knowing how it works is helpful)

Some Java « issues »

18

- An overriding method cannot have more restricted access than the method it overrides

```
class A {
    public int m() {...}
}
class B extends A {
    private @Override int m() {...} //illegal!
}

A foo = new B();  // upcasting
foo.m();          // would invoke private method in
                  // class B at runtime
```

… a nasty example

19

```
class A {
   int i = 1;
   int f() { return i; }
}
class B extends A {
   int i = 2;                          // Shadows variable i in class A.
   int @Override f() { return -i; }    // Overrides method f in class A.
}
public class override_test {
   public static void main(String args[]) {
      B b = new B();
      System.out.println(b.i);         // Refers to B.i; prints 2.
      System.out.println(b.f());       // Refers to B.f(); prints -2.
      A a = (A) b;                        // Cast b to an instance of class A.
      System.out.println(a.i);         // Now refers to A.i; prints 1;
      System.out.println(a.f());       // Still refers to B.f(); prints -2;
   }
}
```

The "runtime" type of "a" is "B"!

Information Hiding

20

- What "information" do classes hide?
  "Internal" design decisions.

```
public class Set {
    …
  public void add(Object o) …

    public boolean contains(Object o) …

    public int size() …
}
```

- Class'es interface: everything in it that is externally accessible

Encapsulation

21

- By hiding code and data behind its interface, a class encapsulates its "inner workings"
- Why is that good?
  - Can change implementation later without invalidating the code that uses the class

```
class LineSegment {
  private Point2D p1, p2;
  ...
  public double length() {
    return p1.distance(p2);
  }
}
```

```
class LineSegment {
  private Point2D p;
  private double length;
  private double phi;
  ...
  public double length() {
    return length;
  }
}
```

Degenerate Interfaces

22

- Public fields are usually a Bad Thing:

```
class Set {
    public int count = 0;

  public void add(Object o) ...

    public boolean contains(Object o) ...

    public int size() ...
}
```

- Anybody can change them; the class has no control

23

- Interface should "hint" at its behavior

Bad:

```
public int product(int a, int b) {
    return a*b > 0 ? a*b : -a*b;
}
```

Better:

```
/** Return absolute value of a * b */
public int absProduct(int a, int b) {
    return a*b > 0 ? a*b : -a*b;
}
```

- Names and comments matter!

24

- Unexpected side effects are a Bad Thing

```
class MyInteger {
  private int value;
  ...
  public MyInteger times(int factor) {
    value *= factor;
    return new MyInteger(value);
  }
}
...
MyInteger i = new MyInteger(100);
MyInteger j = i.times(10);
```

Developer trying to be clever. But what does code do to i?

"DRY" Principle

25

- Don't Repeat Yourself
- Nice goal: have each piece of knowledge live in one place
- But don't go crazy over it
  - DRYing up at any cost can increase dependencies between code

Refactoring

26

- Refactor: improve code's internal structure without changing its external behavior
- Most of the time we're modifying existing software
- "Improving the design after it has been written"
- Refactoring steps can be very simple:

```
public double weight(double mass) {
  return mass * 9.80665;
}
```

```
static final double GRAVITY = 9.80665;
public double weight(double mass) {
  return mass * GRAVITY;
}
```

- Other examples: renaming variables, methods, classes

Why is refactoring good?

27

- If your application later gets used as part of a Nasa mission to Mars, it won't make mistakes
- Every place that the gravitational constant shows up in your program a reader will realize that this is what they are looking at
- The compiler may actually produce better code

Common refactorings

28

- Rename something
  - Eclipse will do it all through your code
  - Warning: Eclipse doesn't automatically fix comments!
- Take a chunk of your code and turn it into a method
  - Anytime your "instinct" is to copy lines of code from one place in your program to another and then modify, consider trying this refactoring approach instead...
  - ... even if you have to modify this new method, there will be just one "version" to debug and maintain!

Refactoring & Tests

29

- Eclipse supports various refactorings
- You can refactor manually
  - Automated tests are essential to ensure external behavior doesn't change
  - Don't refactor manually without retesting to make sure you didn't break the code you were "improving"!
- More about tests and how to drive development with tests next week