

# Java SE: OOP

- Classi e oggetti – dati e funzionalità
- Principi di programmazione Object Oriented
- Wrapper di primitivi
- Override e overload
- Ereditarietà
- Interfacce
- Classi astratte
- Progetto di riferimento
  - <https://github.com/egalli64/mpjp> (*modulo 3*)

# Classi e oggetti

- Classe:
  - Ogni classe è definita in un package, in un file che ha il suo stesso nome (.java)
  - Descrive un **nuovo tipo di dato**, che ha variabili e metodi
    - l'accesso ai membri di una classe è indicato con l'operatore di dereferenziazione, il punto “.”
  - Tipicamente sono nomi usati per descrivere il problema che si vuole risolvere
- Oggetto
  - Istanza di una classe, che è il suo modello di riferimento

Reference a MyClass

Crea un oggetto MyClass

```
MyClass reference = new MyClass();
```

# Constructor (ctor)

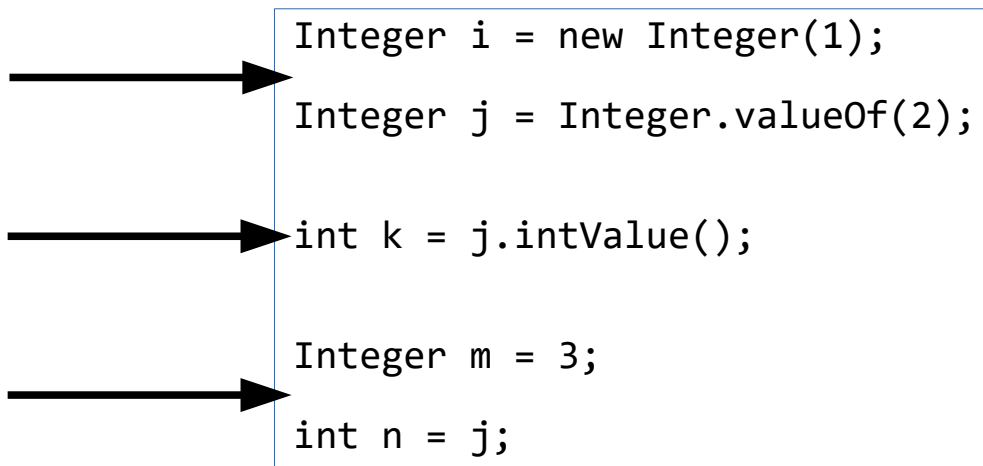
- Metodo speciale, con lo stesso nome della classe, invocato durante la creazione di un oggetto via “new” per inizializzarne lo stato
- Non ha return type (nemmeno void)
- Ogni classe può avere svariati ctor, ognuno dei quali deve essere distinguibile in base al numero/tipo dei suoi parametri
- Se una classe non ha ctor, Java ne crea uno di default senza parametri (che non fa niente)

# Static Factory Method

- Approccio alternativo e più flessibile al costruttore
  - Può avere un nome significativo
  - Può creare un oggetto del tipo richiesto, o derivato, o altro
  - Può incapsulare i passi preparatori alla creazione
  - Permette un maggior controllo sulla creazione

# Wrapper di primitivi

- Controparte reference dei tipi primitivi
  - Boolean, Character, Byte, Short, Integer, Float, Double
- Boxing esplicito
  - Costruttore (deprecato da Java 9)
  - Static factory method
- Unboxing esplicito
  - Metodi definiti nel wrapper
- Auto-boxing
- Auto-unboxing



```
Integer i = new Integer(1);  
Integer j = Integer.valueOf(2);  
  
int k = j.intValue();  
  
Integer m = 3;  
int n = j;
```

# Alcuni metodi statici dei wrapper

- Boolean
  - `valueOf(boolean)`
  - `valueOf(String)`
  - `parseBoolean(String)`
- Integer
  - `parseInt(String)`
  - `toHexString(int)`
- Double
  - `isNaN(double)`
- Character
  - `isDigit(char)`
  - `isLetter(char)`
  - `isLetterOrDigit(char)`
  - `isLowerCase(char)`
  - `isUpperCase(char)`
  - `toUpperCase(char)`
  - `toLowerCase(char)`

# Lo “scope” delle variabili

- Vita limitata al blocco che le contiene
- Member (field, property)
  - di istanza (default)
    - stato dell'oggetto
  - di classe (static)
- Locali (automatiche)
  - Esistenza limitata a un metodo o a un blocco interno
  - Caso particolare, la variabile di ciclo nel loop for, definita subito prima del blocco relativo
- Una variabile locale non può nascondere un'altra locale. Potrebbe però nascondere una proprietà (ma non si fa!)

```
public class Scope {  
    private static int staticMember = 5;  
    private long member = 5;  
  
    public void f() {  
        long local = 7;  
        if (staticMember == 2) {  
            float local = 0.0F;  
            short inner = 12;  
            staticMember = 1 + inner;  
            member = 3 + local;  
        }  
    }  
  
    public static void main(String[] args) {  
        double local = 5;  
        System.out.println(local);  
        staticMember = 12;  
    }  
}
```

# Inizializzazione delle variabili

- Finché non viene inizializzata una variabile non può essere usata – errore di compilazione
- Esplicita per assegnamento (preferita)
  - primitivi: diretto
  - reference: via operatore new
- Implicita by default (solo member)
  - primitivi
    - numerici: 0
    - boolean: false
  - reference: null

```
int i = 42;  
  
String s = new String("Hello");
```

```
private int i;           // 0  
private boolean flag;    // false  
private String t;        // null
```



# Tre principi OOP

- Incapsulamento per mezzo di classi
  - Visibilità pubblica (metodi) / privata (proprietà)
- Ereditarietà in gerarchie di classi
  - Dal generale al particolare
- Polimorfismo
  - Una interfaccia, molti metodi (override)

# Access modifier per data member

- Aiuta l'incapsulamento
  - Privato
- Dubbio
  - Protetto
- Normalmente sconsigliati
  - Package (default)
  - Pubblico

Static initializer

Costruttore

```
public class Access {  
    private int a;  
    protected short b;  
    static double c;  
    // public long d;  
  
    static {  
        c = 18;  
    }  
  
    public Access() {  
        this.a = 42;  
        this.b = 23;  
    }  
  
    // ...  
}
```

# Access modifier per metodi

- Pubblico
- Package
  - usi speciali
- Protetto / Privato
  - helper

```
public class Access {  
    // ...  
  
    static private double f() {  
        return c;  
    }  
  
    void g() {  
        f();  
    }  
  
    public int h() {  
        return a / 2;  
    }  
}
```

# interface

- Cosa deve fare una classe, non come deve farlo (fino a Java 8)
- Una class “implements” una interface
- Un’interface “extends” un’altra interface
- I metodi sono (implicitamente) public
- Le eventuali proprietà sono costanti static final

# interface vs class

```
interface Barker {  
    String bark();  
}  
  
interface BarkAndWag extends Barker {  
    int AVG_WAGGING_SPEED = 12;  
  
    int tailWaggingSpeed();  
}
```

```
public class Fox implements Barker {  
    @Override  
    public String bark() {  
        return "yap!";  
    }  
}
```

extends vs implements

```
public class Dog implements BarkAndWag {  
    @Override  
    public String bark() {  
        return "woof!";  
    }  
  
    @Override  
    public int tailWaggingSpeed() {  
        return BarkAndWag.AVG_WAGGING_SPEED;  
    }  
}
```

# L'annotazione Override

- Annotazione: informazione aggiuntiva su di un elemento
- `@Override`
  - Annotazione applicabile solo ai metodi, genera un errore di compilazione se non esiste un “super”-metodo ridefinibile
- **Override**: il metodo definito nella classe derivata ha la stessa signature e tipo di ritorno di un metodo super (che non può essere final). La visibilità dell'override non può essere più estesa di quella del metodo super
- **Overload**: metodi con stesso nome ma signature diversa
- Signature di un metodo: nome, numero, tipo e ordine dei parametri

# abstract class

- Una classe abstract non può essere istanziata
- Un metodo abstract non ha body
- Una classe che ha un metodo abstract deve essere abstract, ma non viceversa
- Una subclass di una classe abstract
  - o implementa tutti i suoi metodi abstract
  - o è a sua volta abstract

# Relazioni tra classi/interfacce

- Ereditarietà (**is-a**) keyword **extends** e **implements**
  - extends
    - (Sub)classe o interfaccia che ne estende un'altra
    - Eredita proprietà e metodi da super
      - p. es.: Mammal superclass di Cat e Dog
  - implements
    - (Sub)classe che implementa un'interfaccia
- Aggregazione (**has-a**)
  - Classe che ha come proprietà un'istanza di un'altra classe
  - p. es.: Tail in Cat e Dog

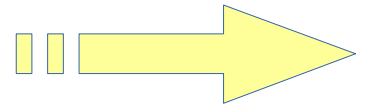


# Ereditarietà in Java

- Single inheritance: una sola superclass
- Implicita derivazione dalla classe base **Object** by default
  - `public boolean equals(Object)` // Confronto tra oggetti: riflessivo, simmetrico, transitivo, consistente
  - `public String toString()` // Rappresentazione dell'oggetto (per logging) ma: `Arrays.toString(array)`
- Una subclass può essere usata al posto della sua superclass (is-a)
  - per ogni classe X si può scrivere `Object object = new X();`
- Una subclass può aggiungere proprietà e metodi a quelli ereditati dalla superclass (attenzione a non nascondere proprietà della superclass con lo stesso nome!)
- Costruttori e quanto nella parte private della superclass non è ereditato dalla subclass
- Subclass transitivity: C subclass B, B subclass A  $\rightarrow$  C subclass A

# this vs super

- **this** è una reference all'oggetto corrente
- **super** indica al compilatore che si intende accedere ad un membro di una *superclass* dal contesto corrente
- ctor → ctor: (primo statement)
  - **this()** – nella classe
  - **super()** – nella superclass



# Esempio di ereditarietà

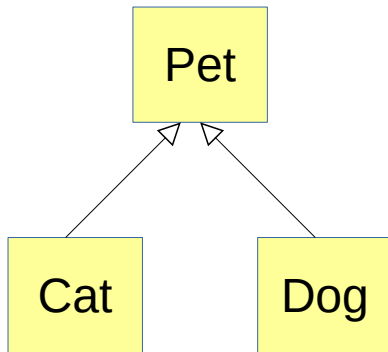
```
public class Pet {  
    private String name;  
  
    public Pet(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
Dog tom = new Dog("Tom");  
  
String name = tom.getName();  
double speed = tom.getSpeed();
```

```
public class Dog extends Pet {  
    private double speed;  
  
    public Dog(String name) {  
        this(name, 0);  
    }  
  
    public Dog(String name, double speed) {  
        super(name);  
        this.speed = speed;  
    }  
  
    public double getSpeed() {  
        return speed;  
    }  
}
```

# Reference casting

- Upcast: da subclass a superclass (sicuro)
- Downcast: da superclass a subclass (rischioso)
  - Protetto con l'uso di **instanceof**



```
// Cat cat = (Cat) new Dog(); // Cannot cast from Dog to Cat

Pet pet = new Dog("Bob");
Dog dog = (Dog) pet;      // OK here, but unsafe
Cat cat = (Cat) pet;      // trouble at runtime
if(pet instanceof Cat) { // OK
    Cat tom = (Cat) pet;
}
```

# Final

- Costante primitiva

```
final int SIZE = 12;
```

- Reference che non può essere riassegnata

```
final StringBuilder sb = new StringBuilder("hello");
```

- Metodo di istanza che non può essere sovrascritto nelle classi derivate

```
public final void f() { // ...
```

- Metodo di classe che non può essere nascosto nelle classi derivate

```
public static final void g() { // ...
```

- Classe che non può essere estesa

```
public final class FinalSample { // ...
```