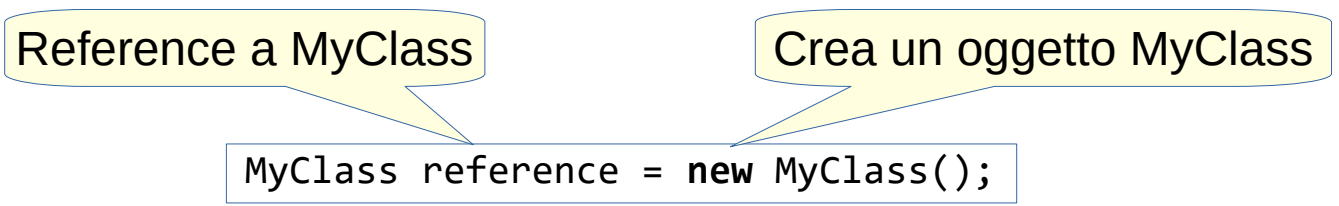
Java SE: OOP

- 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/jse *(modulo 2)*

Classi e oggetti

- Classe:
  - Definita in un package, normalmente in un file che ha il suo stesso nome (.java)
  - Descrive un nuovo tipo di dato, che ha sue proprietà 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



Constructor (ctor)

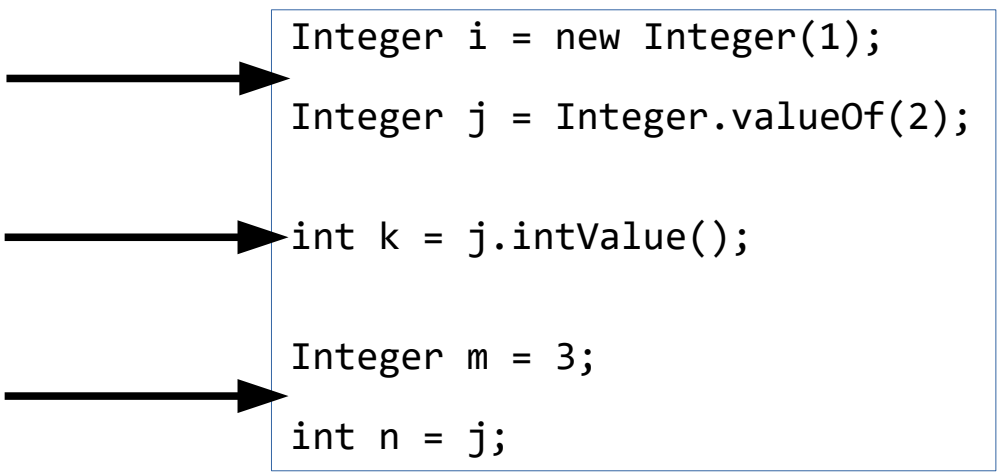
- Metodo speciale, con lo stesso nome della classe
  - Invocato in seguito alla creazione di un oggetto via operatore **new**
  - Ha lo scopo di inizializzare lo stato del nuovo oggetto
- Non ha return type (nemmeno void)
- Ogni classe può avere svariati ctor (*overload*)
  - Devono essere distinguibili in base al numero/tipo dei parametri
- Se una classe non ha ctor, Java ne crea uno di default
  - Senza parametri, non fa niente
- In Java non esiste un distruttore (dtor)
  - L'eliminazione di un oggetto dallo heap è responsabilità del Garbage Collector (gc)

Static Factory Method

- Approccio alternativo e più flessibile al costruttore
  - Può avere un nome significativo, ad esempio
    - My.from(Other) → crea un My from un Other
    - My.getInstance() → ritorna la singola istanza di una classe
    - …
  - 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, Long, Float, Double
- Boxing esplicito
  - Costruttore (deprecato da Java 9)
  - Static factory method
- Unboxing esplicito
  - Metodi definiti nel wrapper
- Auto-boxing
- Auto-unboxing



Alcuni metodi statici dei wrapper

- Boolean
  - **valueOf**(boolean) // reference
  - valueOf(String)
  - **parse**Boolean(String) // primitive
- 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 nasconderne 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

```
int j;

System.out.println(j);
```

- Esplicita per assegnamento (preferita)
  - primitivi: diretto
  - reference: via operatore new

```
int i = 42;

String s = new String("Hello");
```

- Implicita by default (solo member)
  - primitivi
    - numerici: 0
    - boolean: false
  - reference: null

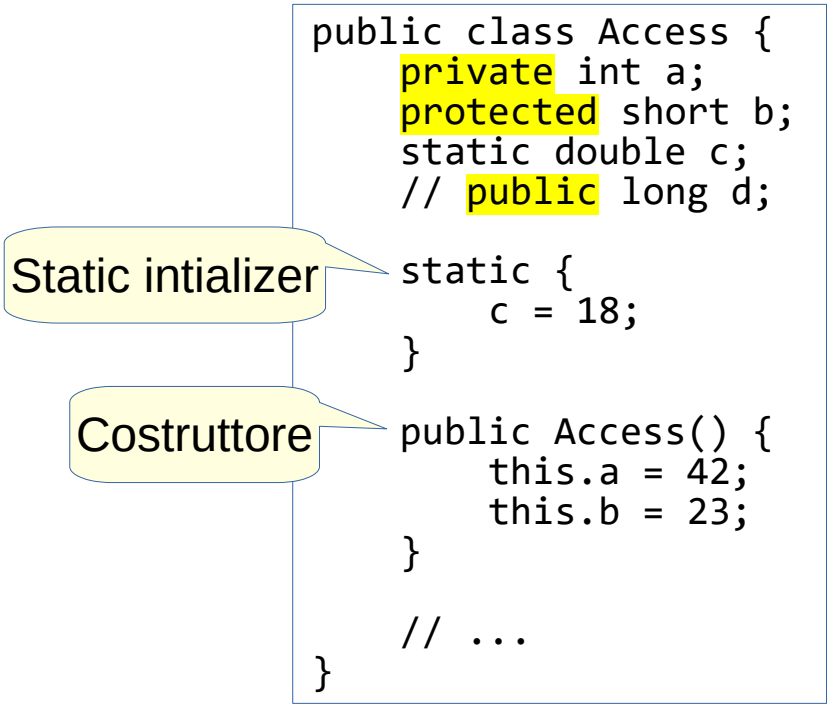
```
private int i;           // 0

private boolean flag;    // false

private String t;        // null
```

Alcuni principi OOP

- **Incapsulamento**
  - Raggruppamento di dati e funzionalità in una classe
    - Astrazione: selezione tra possibili membri in base al problema particolare
    - Coesione: si mira a mantenere una forte correlazione interna
  - Visibilità pubblica (metodi) / privata (proprietà) dei suoi membri
- **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



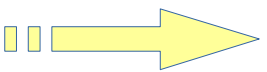
Access modifier per metodi

- Pubblico
  - Uso normale
- Package
  - Casi particolari (test, …)
- Protetto / Privato
  - Helper
  - Ctor per impedire l'istanziazione

```
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 (o più) 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 WaggingBarker extends Barker {
    int DEFAULT_WAG_COUNT = 3;

    String wag();
}
```

extends vs implements

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

```
public class Dog implements WaggingBarker {
    @Override
    public String bark() {
        return "Woof";
    }

    @Override
    public String wag() {
        StringBuilder sb = new StringBuilder();

        // …

        return sb.toString();
    }
}
```

L’annotazione Override

- Annotazione: dà informazioni aggiuntive a un elemento
- @Override
  - Annotazione applicabile solo ai metodi
  - Causa 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

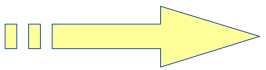
abstract class

- Una classe abstract non può essere instanziata
- 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

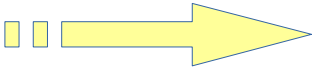
- Single inheritance: una sola superclass
- Implicita derivazione dalla classe base **Object** by default 
- Una subclass può essere usata al posto della sua superclass (is-a)
  - Per ogni classe X si può scrivere Object object = new X();
- Una subclass può aggiungere proprietà e metodi a quelli ereditati dalla superclass (attenzione a non nascondere proprietà della superclass con lo stesso nome!)
- I membri public delle superclass – meno i costruttori – sono parte dell'interfaccia della classe
  - I membri private delle superclass non possono essere acceduti direttamente dalla classe corrente
- Subclass transitivity: C subclass B, B subclass A → C subclass A

La classe Object

- Classe concreta definita in java.lang, raramente usata direttamente
- Confronto tra istanze via **equals**(Object)
  - Se *dobbiamo* ridefinirlo, assicurarsi che sia→riflessivo, simmetrico, transitivo, consistente
  - Perché la classe sia usabile in hash table, va ridefinito anche **hashCode**()
- Rappresentazione di una istanza per log / debug via **toString**()
  - Per gli array si usa il metodo statico Arrays.toString(array)
- Creazione di un clone di una istanza→compito complesso e delicato
  - La classe deve implementare l'interfaccia Cloneable e ridefinire il metodo **clone**()
- Confronto tra due istanze via Comparable.compareTo()
  - Non è un metodo di Object, ma dell'interfaccia Comparable

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

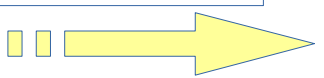


Esempio di ereditarietà

```
public class Pet {
    private String name;

    public Pet(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

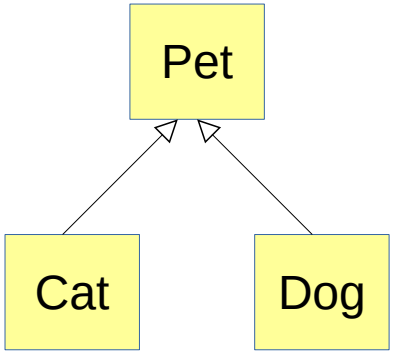
```
Dog tom = new Dog("Tom");

String name = tom.getName();
double speed = tom.getSpeed();
```

```
public class Dog extends Pet {
    private double speed;

    public Dog(String name) {
        this(name, 0.0);
    }

    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 { // ...