CLASSI ASTRATTE

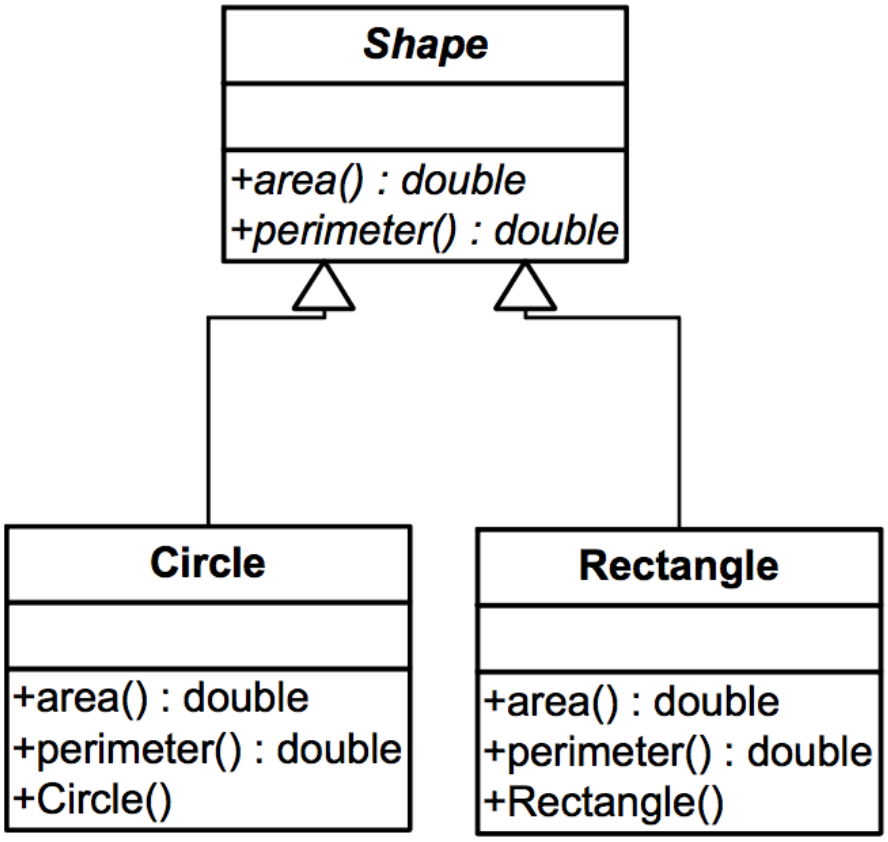
Angelo Di Iorio

Università di Bologna

Classi Astratte

- Ci sono diverse situazioni in cui le classi viste finora non modellano la realtà in modo completamente corretto
- Si consideri la gerarchia in cui le classi `Rectangle` e `Circle` derivano dalla classe `Shape`
- Osservazioni:
  - Non ha molto senso creare istanze di figura geometrica senza sapere di quale figura concreta si tratta
  - Non è possibile calcolare perimetro e area di una figura geometrica senza sapere di quale figura si tratta
  - E' necessario però che sapere calcolare perimetro e area di ogni figura geometrica
- Si potrebbe utilizzare ereditarietà e fornire implementazioni vuote per questi metodi

Esempio figure geometriche



```
public class Shape {
        public double getArea(){ return 0; }
        public double getPerimeter(){ return 0;}
        …
```

```
public class Rectangle extends Shape {
    //variabili e costruttore omessi
    @Override
    public double getArea() {return s1 * s2;}

    @Override
    public double getPerimeter() {return (s1 + s2) * 2;}
…
```

```
public class Circle extends Shape {
        //variabili e costruttore omessi
        @Override
        public double getArea() {return  r * r * 3.14;}

        @Override
        public double getPerimeter() {return 2 * 3.14 * r;}
…
```

Classi Astratte

- Chi garantisce che le sottoclassi implementeranno i metodi per calcolare area e perimetro?
- E' corretto dire che una qualunque figura geometrica ha perimetro e area uguali a 0?
- Abbiamo bisogno di un "segnaposto" che definisce comportamenti in modo incompleto e demanda le sottoclassi a completarli
- Usiamo classi **astratte** che indicano **cosa dovrebbe essere comune alle sottoclassi**

Classi Astratte

- Una classe si dice **astratta** se dichiara almeno un metodo senza fornire la sua implementazione
- Questa implementazione **DEVE** essere fornita da ogni sua sottoclasse **concreta**
- La keyword `abstract` permette di dichiarare un metodo astratto
- Se una classe Java ha un metodo astratto, anche solo uno, deve essere dichiarata astratta (con `abstract`)

```
Public abstract class Shape {
        abstract public double getArea();
        abstract public double getPerimeter();
```

Non c'è body!

```
public class Rectangle extends Shape {
    //variabili e costruttore omessi

    @Override
    public double getArea() {return s1 * s2;}

    @Override
    public double getPerimeter() {return (s1 + s2) * 2;}
…
```

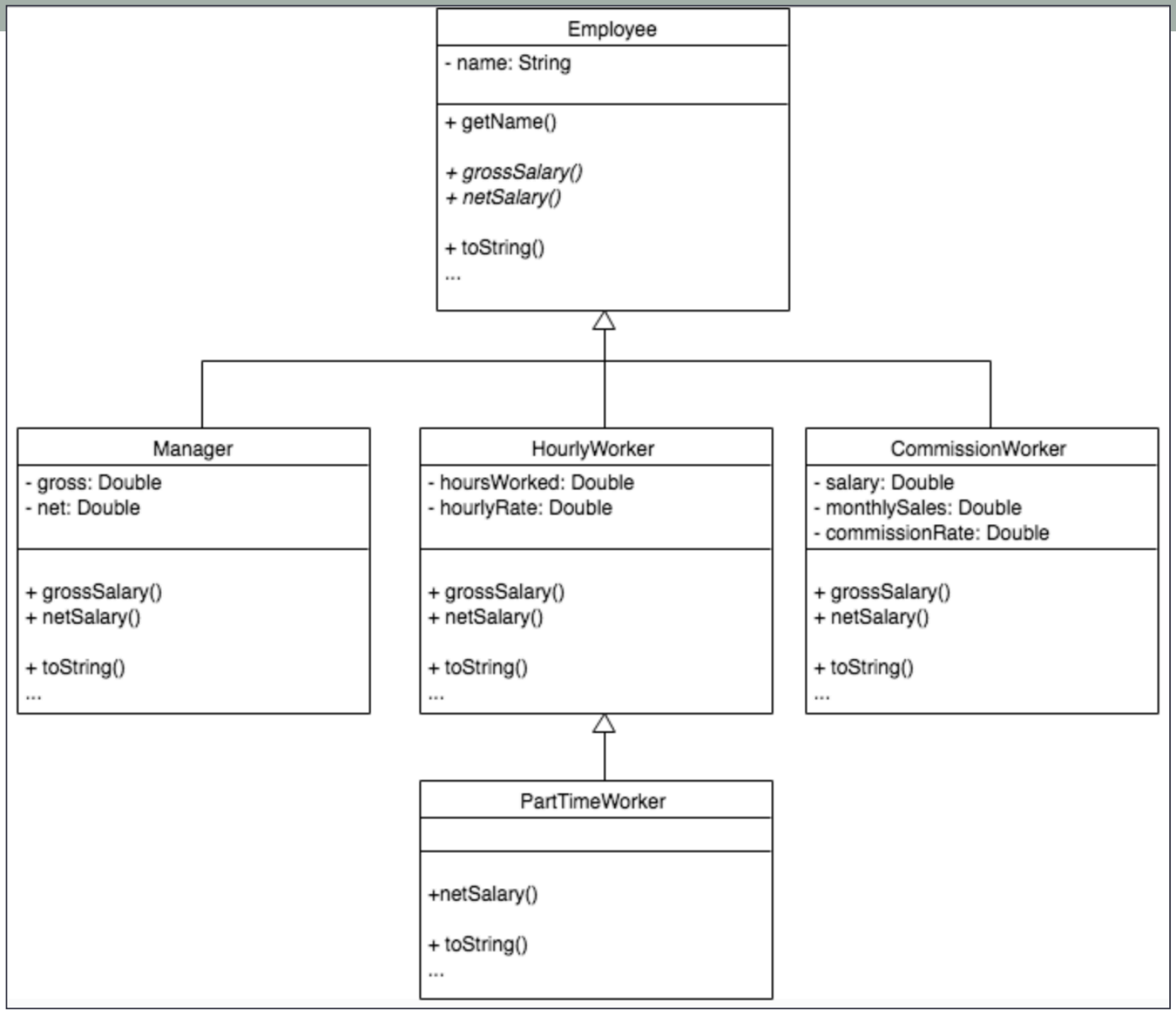
```
public class Circle extends Shape {
        //variabili e costruttore omessi

        @Override
        public double getArea() {return  r * r * 3.14;}

        @Override
        public double getPerimeter() {return 2 * 3.14 * r;}
…
```

Caso di studio: Employee

- Modelliamo il seguente scenario: in un'azienda ci sono quattro tipi di dipendenti, i cui stipendi (lordo e netto) sono calcolati in modo diverso
- Per ogni dipendente è utile sapere:
  - Nome
  - Stipendio lordo
  - Stipendio netto

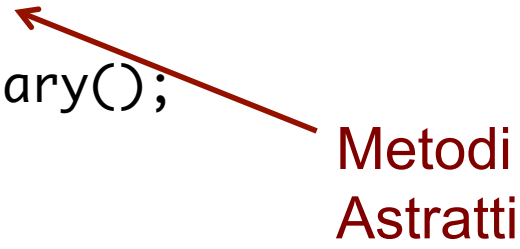
Caso di studio: Employee

- **Manager**: stipendio lordo mensile da cui un 10% viene trattenuto per ottenere il salario netto
- **HourlyWorker**: stipendio lordo mensile calcolato sul numero di ore lavorative; stipendio netto ottenuto trattenendo il 5%
- **PartTimeWorker**: simile al lavoratore ad ore ma senza trattenute
- **CommissionWorker**: riceve uno stipendio base con un bonus aggiuntivo (premio produzione). Da questo stipendio va trattenuto il 10% per ottenere lo stipendio netto


Employee
- name: String
+ getName()
+ grossSalary()
+ netSalary()
+ toString()
...
Manager
- gross: Double
- net: Double
+ grossSalary()
+ netSalary()
+ toString()
...
HourlyWorker
- hoursWorked: Double
- hourlyRate: Double
+ grossSalary()
+ netSalary()
+ toString()
...
CommissionWorker
- salary: Double
- monthlySales: Double
- commissionRate: Double
+ grossSalary()
+ netSalary()
+ toString()
...
PartTimeWorker
+netSalary()
+ toString()
...

Classe Employee

```
public abstract class Employee {

	private String name;

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

	public String getName() {
		return name;
	}

	abstract public double grossSalary();

	abstract public double netSalary();

}
```



Manager.java

```
public class Manager extends Employee {

	private double gross;
	private double net;

	public Manager(String name, double salary) {
		super(name);
		gross = salary;
		net = 0.9 * gross;
	}

	public double grossSalary() {
		return gross;
	}

	public double netSalary() {
		return net;
	}

	public String toString() {
		return "Manager[" + "name = " + getName() + ",
			gross = " + gross + ", net = " + net + "]";
	}
}
```

HourlyWorker.java

```
public class HourlyWorker extends Employee {
	private double hoursWorked;
	private double hourlyRate;

	public HourlyWorker(String name, double hoursWorked,
double hourlyRate) {
		super(name);
		this.hoursWorked = hoursWorked;
		this.hourlyRate = hourlyRate;
	}

	public double grossSalary() {
		return hoursWorked * hourlyRate;
	}

	public double netSalary() {
		return this.grossSalary() * 0.95;
	}

	public String toString() {
		return "HourlyWorker[" + "name = " + getName() + ",
	gross = " + grossSalary()  + ", net = " + netSalary() + "]";
	}
}
```

CommissionWorker.java

```
public class CommissionWorker extends Employee{
        private double salary;
        private double monthlySales;
        private double commissionRate;


        public CommissionWorker(String name, double salary, double
monthlySales, double commissionRate) {
                super(name);
                this.salary = salary;
                this.monthlySales = monthlySales;
                this.commissionRate = commissionRate;
        }

        public double grossSalary() {
                return salary + monthlySales * commissionRate / 100;}


        public double netSalary() {return this.grossSalary() * 0.9;}

        public String toString() {
                return "CommissionWorker[" + "name = " + getName() + ",
                gross = " + grossSalary() + ", net = " + netSalary() + "]";
        }
}
```

Cicli polimorfi

- Una **classe astratta** è un **tipo**
- Classi astratte e polimorfismo possono essere sfruttati per implementare in modo “naturale” comportamenti ripetuti su oggetti diversi ma appartenenti alla stessa gerarchia (polimorfi)
- Esempio: scriviamo un programma che inizializza un array di dipendenti, stampa lo stipendio di ognuno e calcola la spesa totale per gli stipendi
- Usiamo un “**polymorphic loop**” su oggetti della classe padre
- Il binding dinamico di Java richiamerà i metodi opportuni per calcolare gli stipendi in base al ruolo

EmployeesDemo.java

```
public class EmployeesDemo {

    private Employee[] staff;

    private double totalGrossSalary;
    private double totalBenefits;
    private double totalNetSalary;

    public void doTest() {

        Employee[] staff = new Employee[5];

        staff[0] = new Manager("Fred", 800);
        staff[1] = new Manager("Ellen", 700);
        staff[2] = new HourlyWorker("John", 37, 13.50);
        staff[3] = new PartTimeWorker("Gord", 35, 12.75);
        staff[4] = new CommissionWorker("Mary", 400, 15000, 3.5);

    // continua in slide successiva
```

EmployeesDemo.java

```
// continua doTest() da slide precedente

totalGrossSalary = 0.0;
totalNetSalary = 0.0;

for (int i = 0; i < staff.length; i++) {
      totalGrossSalary = totalGrossSalary + staff[i].grossSalary(
      totalNetSalary = totalNetSalary + staff[i].netSalary();
      System.out.println(staff[i]);
};

totalBenefits = totalGrossSalary - totalNetSalary;

System.out.println("Total gross salary: " + totalGrossSalary);
System.out.println("Total benefits: " + totalBenefits);
System.out.println("Total net salary: " + totalNetSalary);


...
```

Output doTest()

```
Manager[name = Fred, gross = 800.0, net = 720.0]
Manager[name = Ellen, gross = 700.0, net = 630.0]
HourlyWorker[name = John, gross = 499.5, net = 474.525]
PartTimeWorker[name = Gord, gross = 446.25, net = 446.25]
CommissionWorker[name = Mary, gross = 925.0, net = 832.5]
Total gross salary: 3370.75
Total benefits: 267.4749999999999
Total net salary: 3103.275
```

Vantaggi del poliformismo

1. Non è necessario sapere nulla circa i tipi di impiegati durante la scrittura del loop. Il sistema run-time conosce il tipo chiama la versione corretta del metodo
2. Se si aggiungono nuovi tipi di impiegati alla gerarchia non è necessario fare modifiche al “polymorphic loop” che calcola gli stipendi

ESERCIZI

Esercizio motori

Implementare le classi Java utili per descrivere il motore di un'automobile.

Ogni motore è caratterizzato da:

- *cilindrata (intero)*
- *numero_cilindri (intero)*

Da queste informazioni è possibile derivare la potenza (espressa in cavalli, di tipo double) in base al tipo di motore.

Esistono tre tipi di motore:

- *benzina* – potenza: (cilindrata/numero_cilindri) * 0.1
- *diesel* – potenza: (cilindrata/numero_cilindri) * 0.2
- *metano* – potenza: ((cilindrata * 0.8)/numero_cilindri) * 0.25

Definire la classe astratta `Motore` e le opportune classi concrete. Implementare una classe test per verificare il funzionamento delle classi e dei metodi.

Esercizio motori (2)

- Aggiungere alla classe di test un metodo che prende in input un vettore di motori e restituisce la media tra le potenze dei motori nel vettore

Esercizio animali

Implementare le classi Java e modellare questa situazione (ovviamente semplificata):

Ogni animale ha un certo numero di zampe e fa un verso:

- *Il gatto è quadrupede e miagola*
- *Il cane è quadrupede e abbaia*
- *Il tacchino è bipede e goglotta*

Definire le classi `Animale`, `Quadrupede`, `Bipede`, `Cane` e `Tacchino` e i metodi get per leggere il verso dell'animale (stringa) e il numero di zampe (intero); implementare il metodo `toString()`.

Implementare una classe test per verificare il funzionamento delle classi e dei metodi.

Esercizio animali (2)

Aggiungere i metodi per gestire ulteriori informazioni:

- *Ogni esemplare di animale ha un nome e un anno di nascita*. Per semplicità l'età si calcola come il numero di anni trascorsi dalla data corrente, allo stesso modo per tutte le specie.

- E' possibile confrontare un animale con un altro (anche di specie diversa) in base alla loro età in anni. La classe `Animale` espone quindi un metodo `piuGrandeDi(Animale a)` che restituisce *true* se l'istanza su cui è invocata ha un età maggiore dell'istanza passata come parametro, *false* altrimenti.