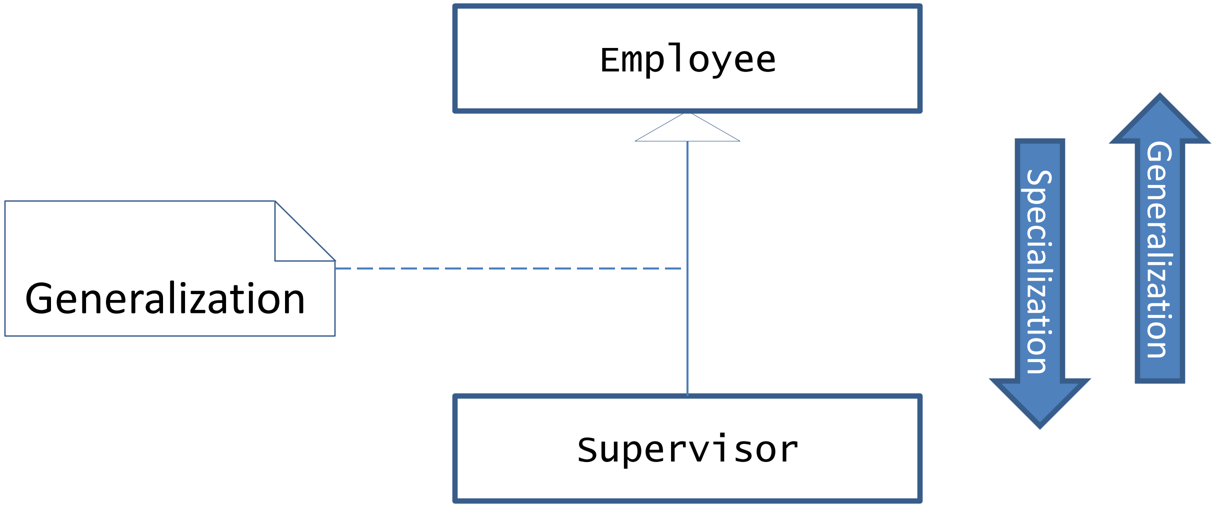
Inheritance

Employee

```java
public class Employee {

    private String name;
    private String ssn;

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

    public String getName() {
        return name;
    }

    public String getSsn() {
        return ssn;
    }

    @Override
    public String toString() {
        return "(" + getName() + ", " + getSsn() + ")";
    }

}
```

Generalization



- A `Supervisor` is an `Employee`.
- An `Employee` can be a `Supervisor`.

Inheritance

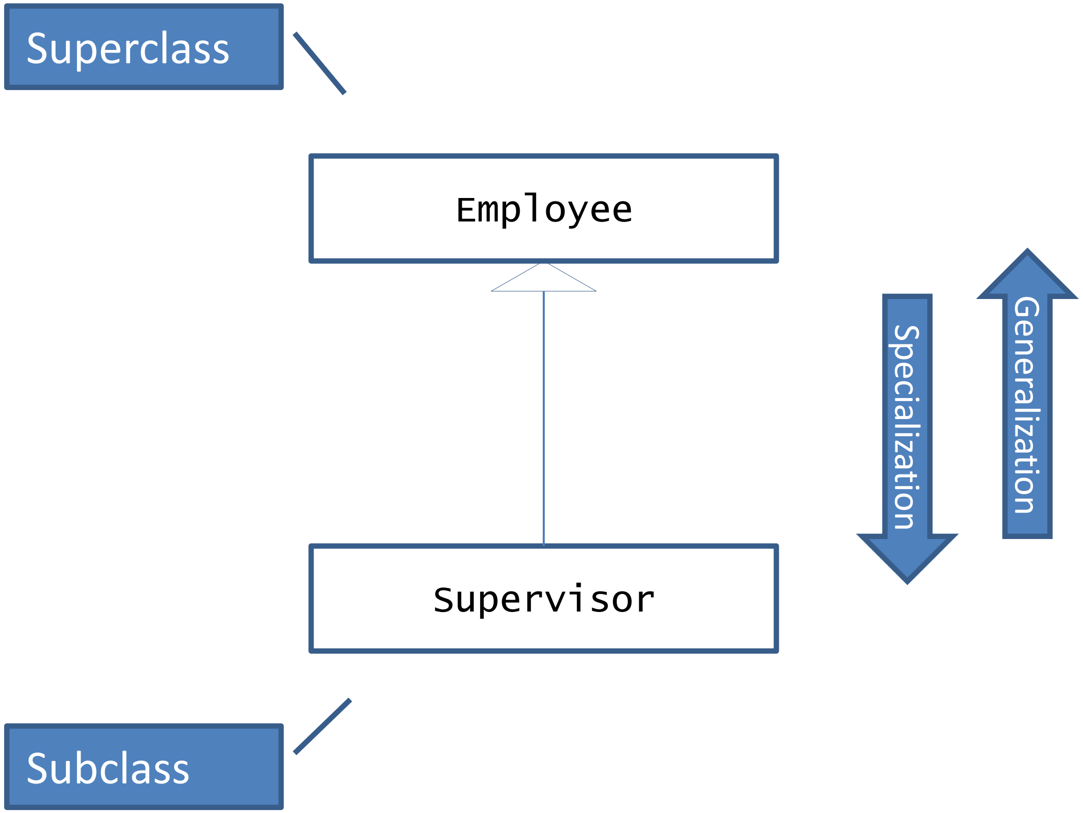
```
public class Supervisor extends Employee {

    private int level;

    public Supervisor(final String name,
                      final String ssn,
                      final int level) {
        …
    }

    public int getLevel() {
        return level;
    }

    @Override
    public String toString() {
        return "(" + getName() + ", " + getSsn() + ", "
               + getLevel() + ")";
    }
}
```

A `Supervisor` is an `Employee`.

New method, specific for `Supervisor`.

Overrides the method with the same name in `Employee`.

Generalization



Inheritance

- Subclass specializes superclass
- Members are inherited and keep access category
- Relation is a – References for the superclasse type can refer to subclass variables
- Example

```
Supervisor supervisor = new Supervisor("Guilhermina",
                                       "123456789", 3);

Employee employee = new Supervisor("Felisberto",
                                   "987654321", 5);
```

Inheritance

- Subclass has all superclass properties
- Example:

```
Supervisor supervisor = new
  Supervisor("Guilhermina", "123456789", 3);

Employee employee = new Supervisor("Felisberto",
  "987654321", 5);

String employee_ssn_id_1 = employee.getSsn();

String employee_ssn_id_2 = supervisor.getSsn();
```

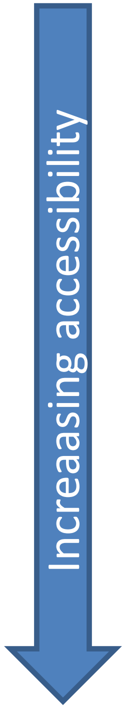
Override

- Subclass method can override superclass method
- Override is a specialization
- Rules
  - Same signature and compatible return type
  - Superclass method cannot be private or final
  - Subclass method must have equal or higher accessibility

Final method cannot be overriden

Access categories

- Members can be:
  - `private` – access only by members of the same class
  - *package-private* (no qualifier) – also accessible to members of classes in the same package
  - `protected` – also accessible to membersof derived classes
  - `public` – universal access



Increaasing accessibility

Class Interfaces

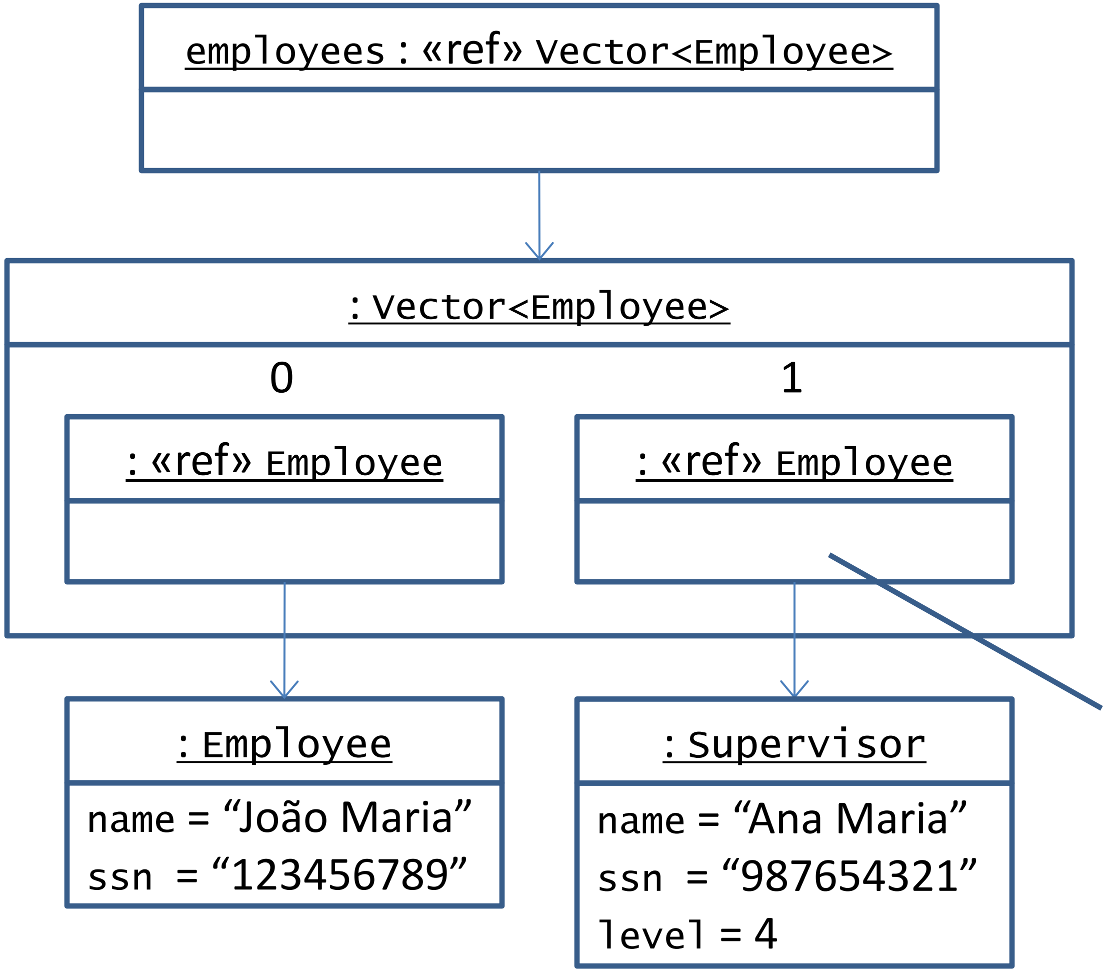
- Within the class itself one can access:
  - Class members and non-private members of the base classes
- In classes of the same package:
  - Non-private members of the class or its super classes
- Ina a derived class:
  - Protected or public members of the class or its super classes
- In other classes:
  - Public members of the class or its super classes

Example

```
Vector<Employee> employees =
    new Vector<Employee>();

employees.add(new Employee("João Maria",
                              "123456789"));
employees.add(new Supervisor("Ana Maria",
                                "987654321", 4));
…
for (Employee employee : employees)
    out.println(employee.toString());
```

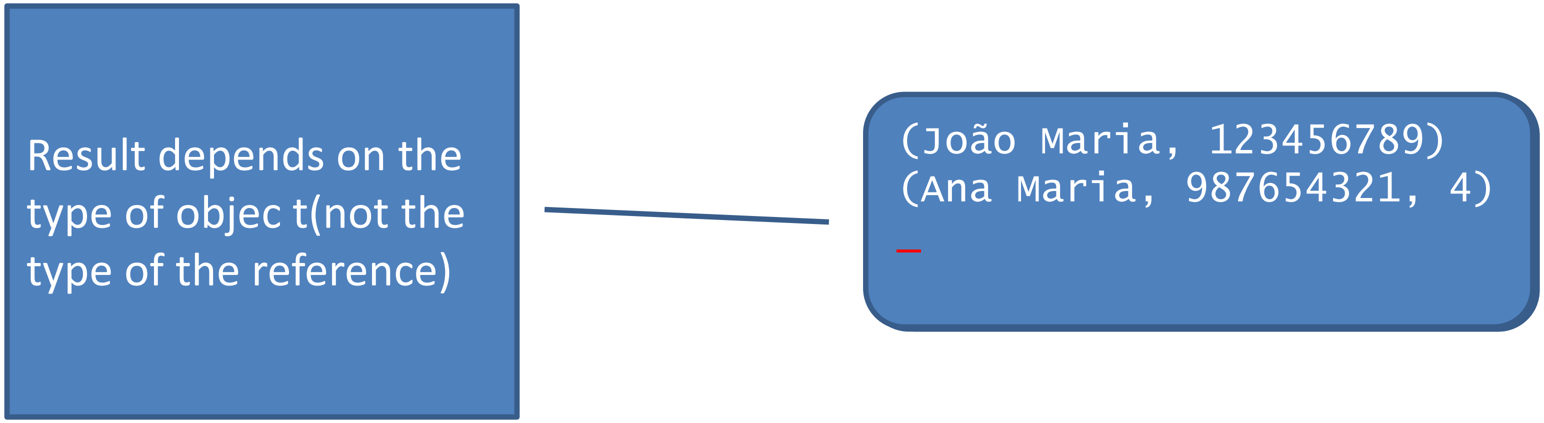
Which toString() will execute?

Organization



Possible because Supervisor is a subclass of Employee, i.e., Supervisor is na Employee.

Result



Result depends on the type of objec t(not the type of the reference)

```
(João Maria, 123456789)
(Ana Maria, 987654321, 4)
_
```

Polimorfism

- Ability of an object to take several forms
  - The form of a member of its own class
  - The form of one of its super classes

- Object can be referenced by a reference of its class or any of its super classes

What is printed?

```
Supervisor supervisor = new Supervisor("Guilhermina",
                                       "123456789", 3);
Employee anEmployee = new Supervisor("Felisberto",
                                     "987654321", 5);
Employee anotherEmployee = new Employee("Elvira",
                                        "111111111");
out.println(supervisor.toString());
out.println(anEmployee.toString());
out.println(anotherEmployee.toString());
```

```
(Guilhermina, 123456789, 3)
(Felisberto, 987654321, 5)
(Elvira, 111111111)
_
```

Polimorfism: operations and methods

- A polimorphic or virtual operation can have several implementations
- An implementation of an operation is a method
- A polimorphic operation can have several implementing methods, one in each class
- All Java operations are polimorphic, except private or final ones
- A class is polimophis if it has at least one polimorphic operation

Object Class

```java
public class Employee extends Object {

    private String name;
    private String ssn;

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

    public String getName() {
        return name;
    }

    public String getSsn() {
        return ssn;
    }

    @Override
    public String toString() {
        return "(" + getName() + ", " + getSsn() + ")";
    }

}
```

All classes that have no superclass are subclasses of `Object`

Static vs dynamic binding

- Binding
  - Association between invocation of an operation and execution of a method
- Static binding
  - Non polimorphic operations invoqued through super
  - Association carried on in compile time
- dinamic binding
  - Polimorphic operations
  - Association carried on in run time

Final methods

- Subclass is not required to override methods
- Superclass can forbid override using final
- Reasons for using final:
  - Behavior must not be overridden, the code in that method must run every time the operation is called

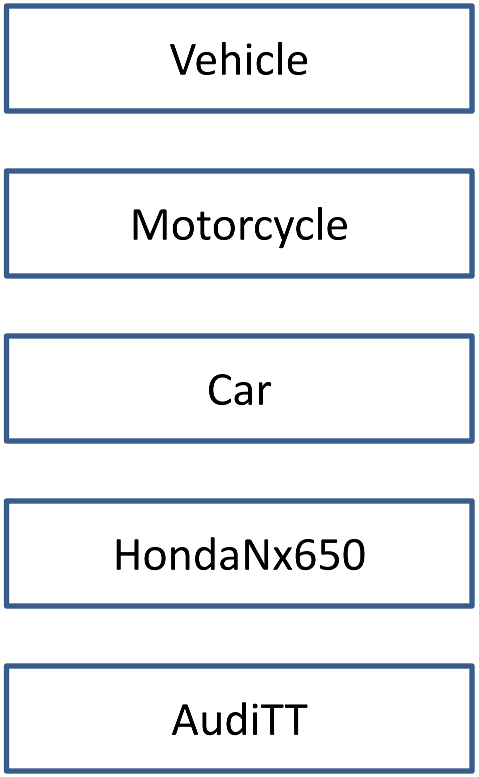
Superclass access

```
public class Base {

    public String className() {
        return "Base";
    }

}

public class Derived extends Base {

    @Override
    public String className() {
        return “Derived”;
    }

    public void testCalls() {
        Base base = (Base)this;

        out.println("Through this:  " + this.className());
        out.println("Through base:  " + base.className());
        out.println("Through super: " + super.className());
    }
}
```

```
Through this:  Derived
Through base:  Derived
Through super: Base

_
```

Analysis

- Vehicle
- Motorcicle
- Car
- Honda NX 650
- Audi TT



Vehicle

Motorcycle

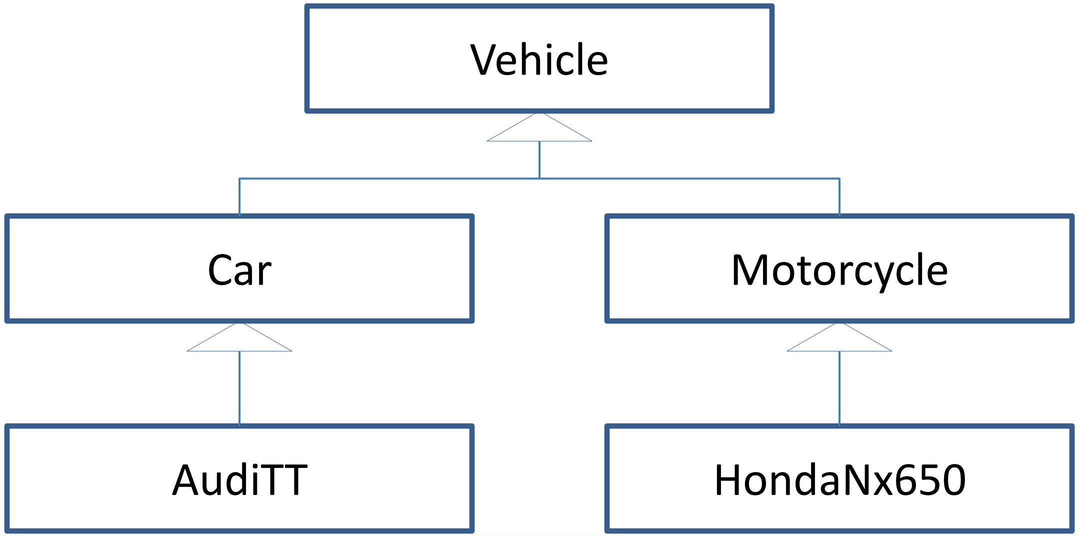
Car

HondaNx650

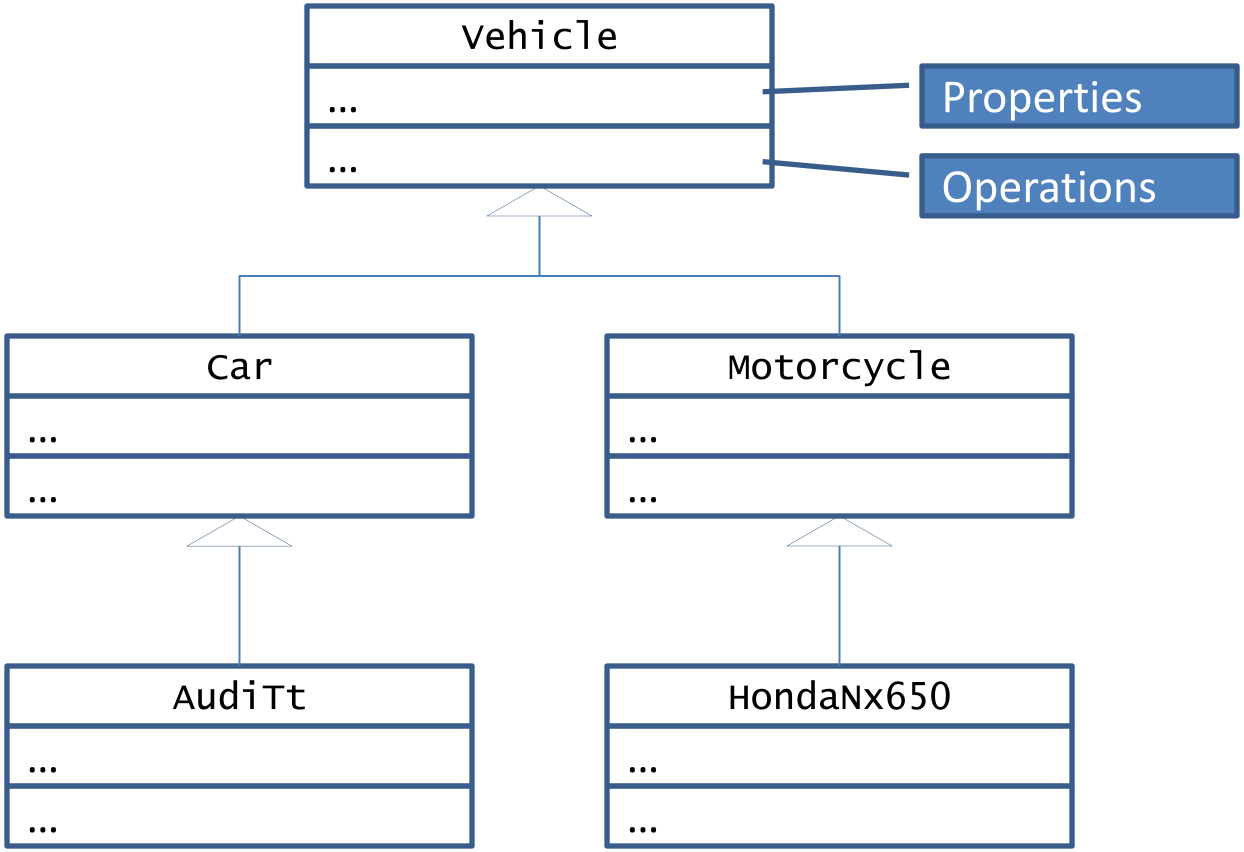
AudiTT

Analysis: relations

- A car is a vehicle
- A motorcicle is a vehicle
- A Honda NX 650 is a motorcicle
- An Audi TT is a car



Design


Vehicle
…
…
Properties
Operations
Car
…
…
Motorcycle
…
…
AudiTt
…
…
HondaNx650
…
…

Implementation

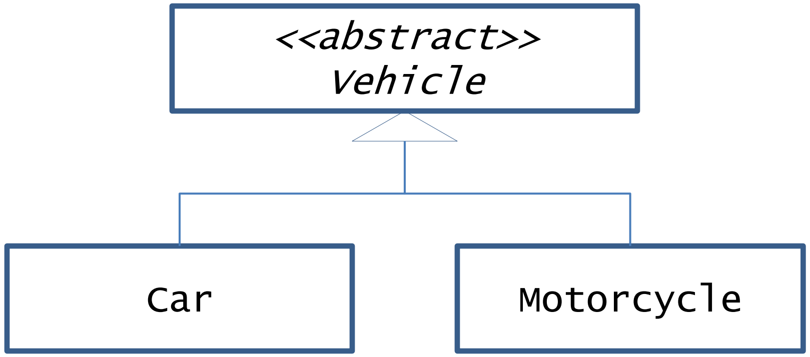
```
public class Vehicle {
    …
}

public class Car extends Vehicle {
    …
}

public class Motorcycle extends Vehicle {
    …
}

public class HondaNx650 extends Motorcycle {
    …
}

public class AudiTT extends Car {
    …
}
```

Abstract and concrete

- Abstract – No instances in the problem
- Concreto – Has instances in the problem
- Depending on the domain…
  - Veihcle and Car: abstract; Audi TT concrete
  - Vehicle abstract; Car and Audi TT concrete

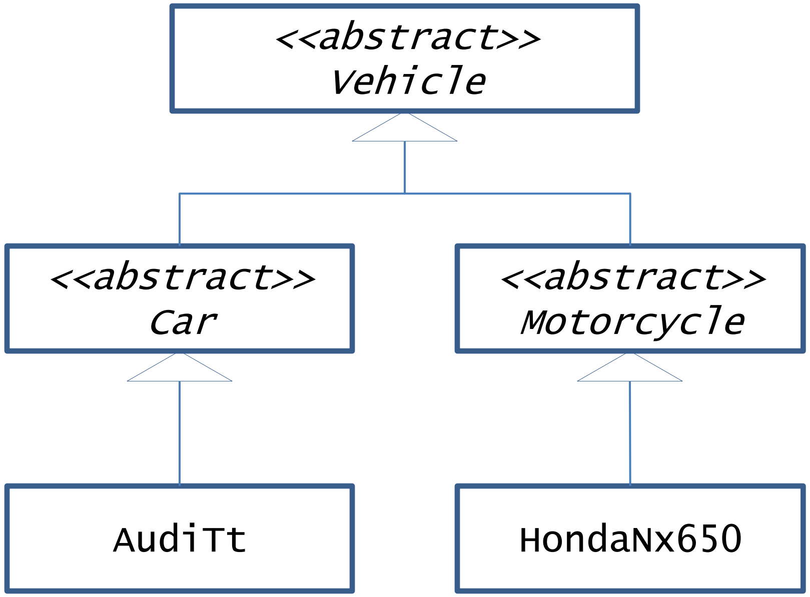
Analysis and design

Hipothesis 1



Concrete classes are usually leafs

Hipothesis 2



Implementation: hipothesis 1

```
public abstract class Vehicle {
    …
}

public class Car extends Vehicle {
    …
}

public class Motorcycle extends Vehicle {
    …
}
```

Implementation: hipothesis 2

```
public abstract class Vehicle {
    …
}

public abstract class Car extends Vehicle {
    …
}

public abstract class Motorcycle extends Vehicle {
    …
}

public class HondaNx650 extends Motorcycle {
    …
}

public class AudiTt extends Car {
    …
}
```

Abstract classes

- When an operation is qualified as `abstract` it is merely a declaration of an operation

- A non-abstract operation includes its definition

- A class containing an abstract operation must be abstract

- An abstract class must have the qualifier `abstract`

Abstract classes

- An abstract class cannot be instantiated
- A subclass of an abstract class can only be concrete if it defines all abstract methods of its superclass

Toolbox: Position

```java
public class Position {

    private double x;
    private double y;

    public Position(final double x, final double y) {
        this.x = x;
        this.y = y;
    }

    public final double getX() {
        return x;
    }

    public final double getY() {
        return y;
    }

}
```

Toolbox : Size

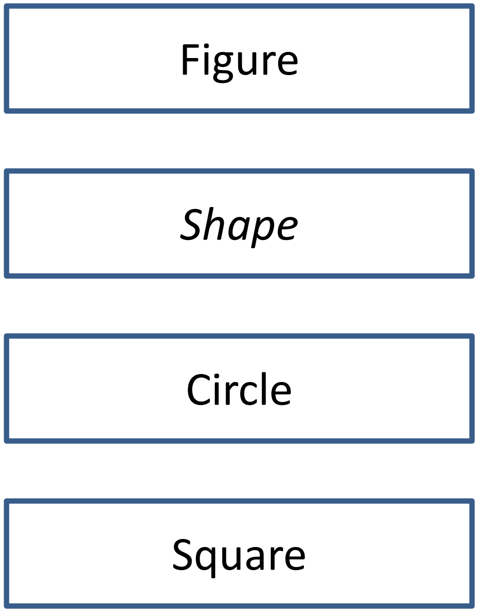
```
public class Size {

    private double width;
    private double height;

    public Size(final double width,
                final double height) {
        this.width = width;
        this.height = height;
    }

    public final double getWidth() {
        return width;
    }

    public final double getHeight () {
        return height;
    }
}
```

Toolbox : Box

```
public class Box {

    private Position topLeftCornerPosition;
    private Size size;

    public Box(final Position topLeftCornerPosition,
               final Size size) {
        this.topLeftCornerPosition = topLeftCornerPosition;
        this.size = size;
    }

    public final Position getTopLeftCornerPosition() {
        return position;
    }

    public final Size getSize() {
        return size;
    }
}
```

Anaylsis

- Figure
- Form (abstract)
- Circle
- Square



Figure

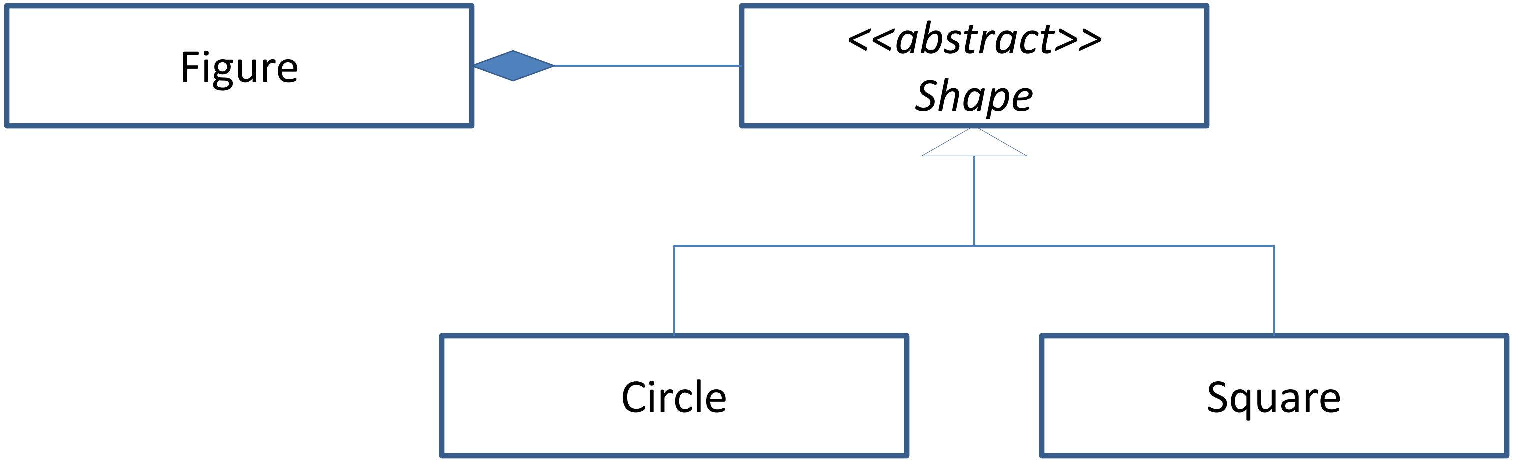
Shape

Circle

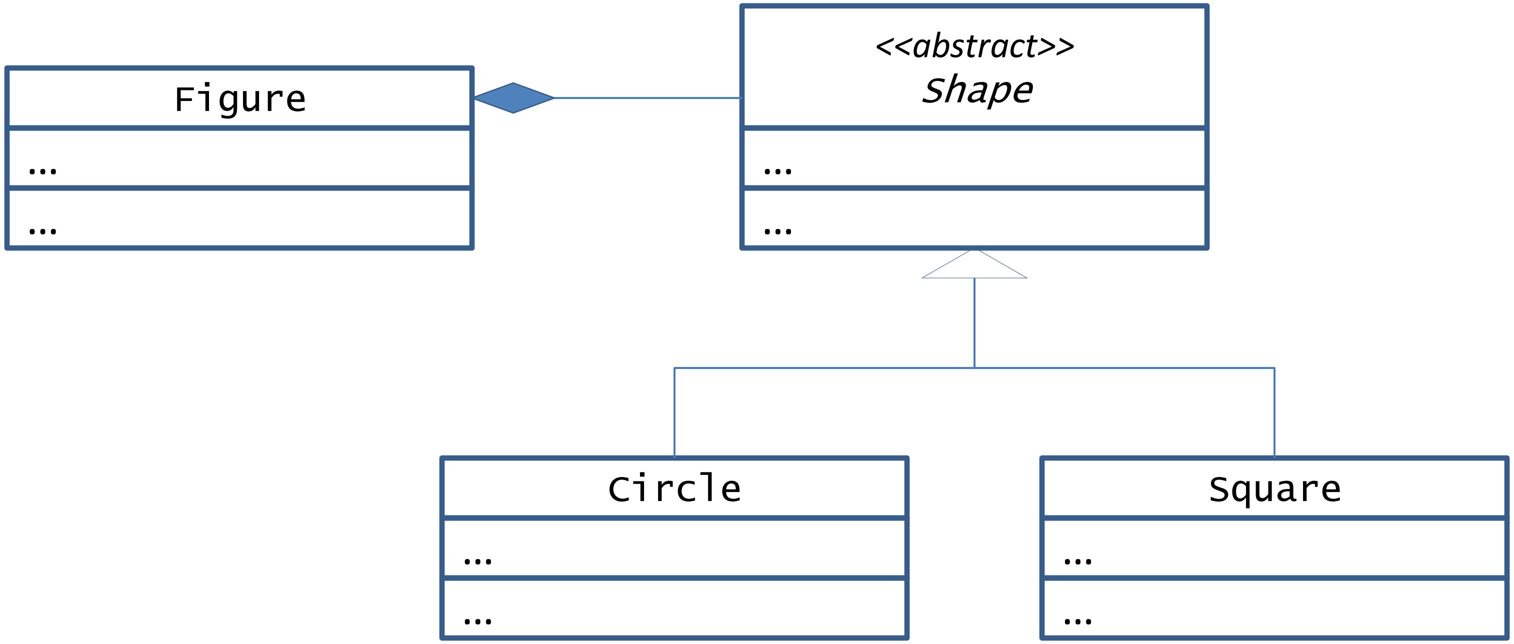
Square

Analysis

- A figure is composed of Forms
- A Circle is a Form
- A Square is a Form



Design



Implementation

```
public class Figure {
    private Vector<Shape> shapes;
    …
}

public abstract class Shape {
    …
}

public class Circle extends Shape {
    …
}

public class Square extends Shape {
    …
}
```

Implementation: Shape

```
public abstract class Shape {

    private Position position;

    public Shape(final Position position) {
        this.position = position;
    }

    public final Position getPosition() {
        return position;
    }
    public abstract double getArea();
    public abstract double getPerimeter();
    public abstract Box getBoundingBox();

    public void moveTo(final Position newPosition) {
        position = newPosition;
    }
}
```

Area of a “form”?

Abstract operations

Implementation: Circle

```
public class Circle extends Shape {

    private double radius;

    public Circle(final Position position,
                  final double radius) {
        super(position);
        this.radius = radius;
    }

    public final double getRadius() {
        return radius;
    }

    …
```

A Circle is a Shape the class Circle inherits from Shape.

Only necessary the extra attribute, center position is hinherited from Shape.

A little help from the super-class…

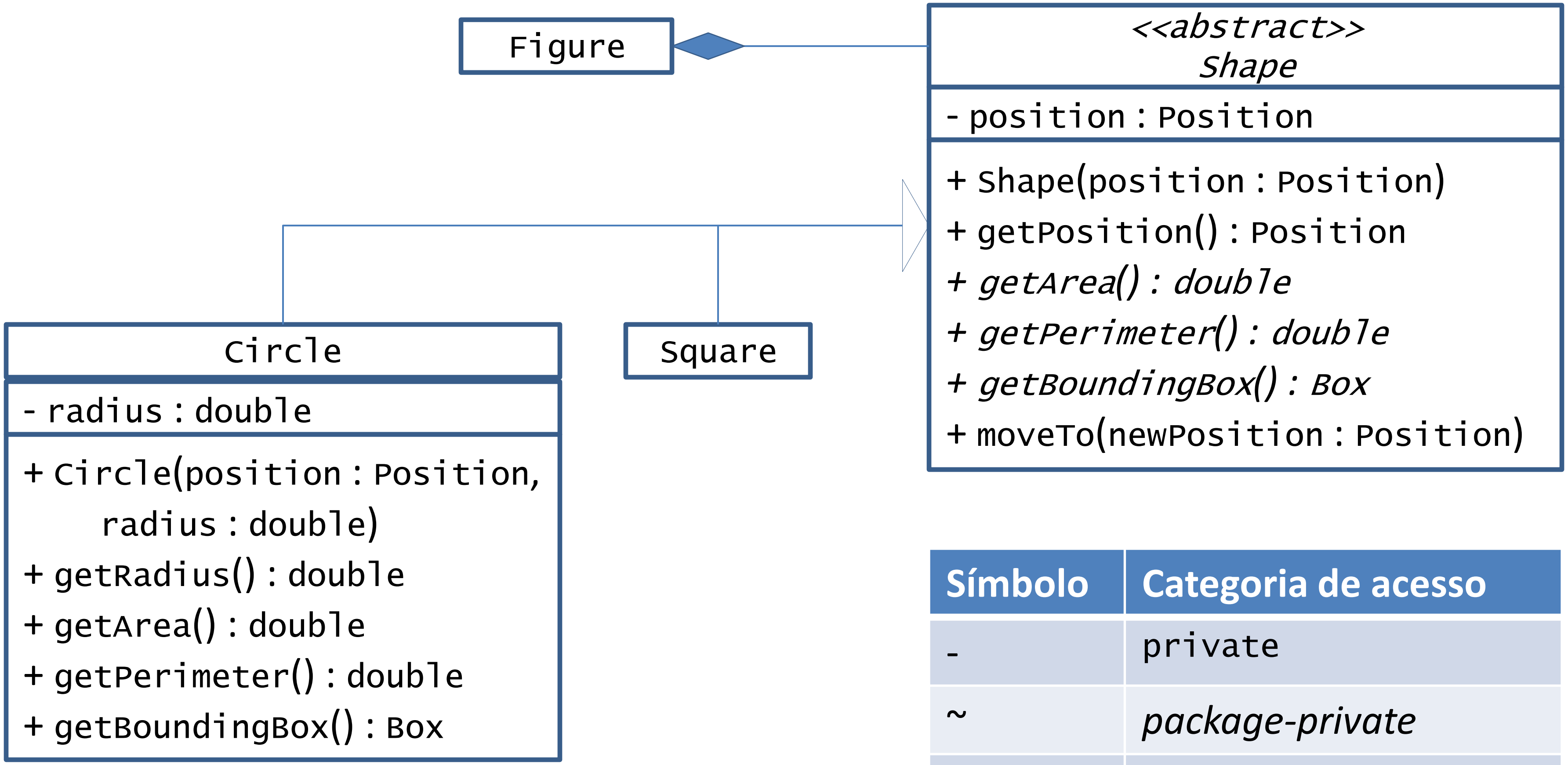
Implementation: `Circle`

```
    …

    @Override
    public double getArea() {
        return Math.PI * getRadius() * getRadius();
    }

    @Override
    public double getPerimeter() {
        return 2.0 * Math.PI * getRadius();
    }

    @Override
    public Box getBoundingBox() {
        return new Box(
            new Position(getPosition().getX() - getRadius(),
                         getPosition().getY() - getRadius()),
            new Size(2.0 * getRadius(), 2.0 * getRadius())
        );
    }
}
```

Area of a cirlcle: π × r^2.

Each abstract opertion of Shape must be defined.

Detailed design



Figure

<<abstract>>
Shape

- position : Position

+ Shape(position : Position)
+ getPosition() : Position
+ *getArea() : double*
+ *getPerimeter() : double*
+ *getBoundingBox() : Box*
+ moveTo(newPosition : Position)

Circle

- radius : double

+ Circle(position : Position,
radius : double)
+ getRadius() : double
+ getArea() : double
+ getPerimeter() : double
+ getBoundingBox() : Box

Square

Símbolo	Categoria de acesso
-	private
~	*package-private*
#	protected
+	public

References

- Y. Daniel Liang, *Introduction to Java Programming*, 7.ª ed., Prentice-Hall, 2010.

Summary

- Inheritance