Object Oriented Programming (OOP) Concepts: Relationships and Inheritance

Topics

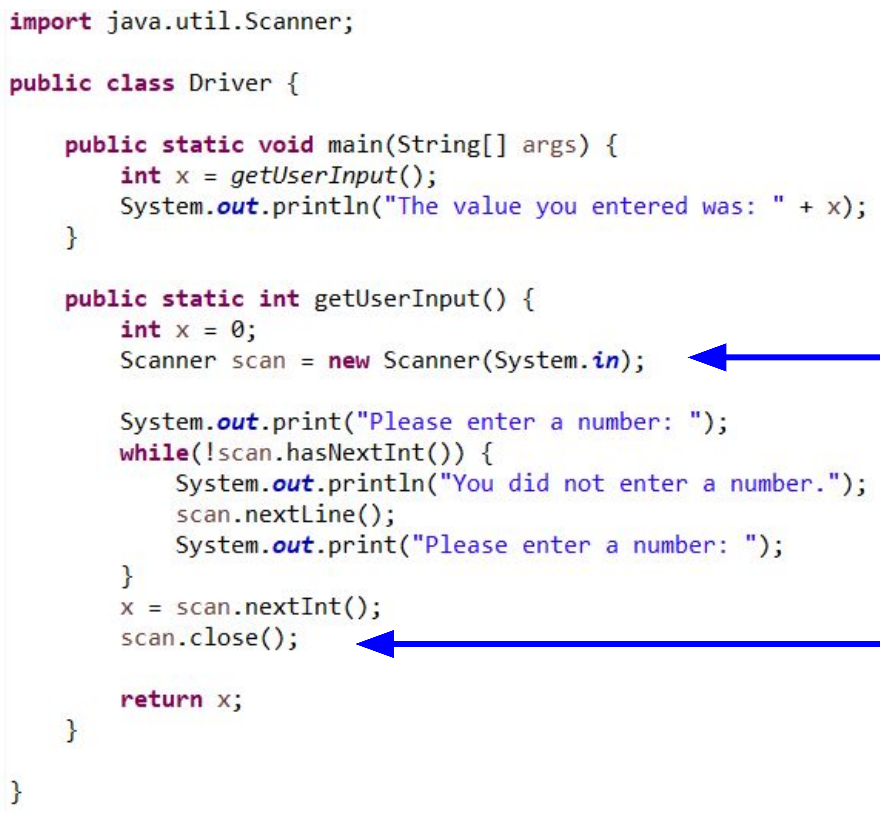
- Dependency (“uses”)
- Association/Composition/Aggregation (“has-a”)
- Inheritance (“is-a”)
- Special Methods

Relationships: Dependency

- **Dependency** - when one Class ***uses*** (i.e., is dependent on) another class in order to perform some action/behavior

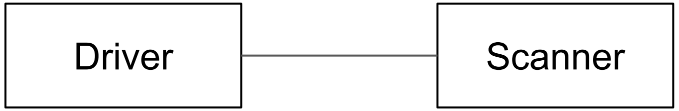
- More specifically, the class does NOT store a field/global variable of the Object/Class you are using (a local variable is fine and usually expected), it simply utilizes the Class/Object for a specific behavior and then no longer needs it

Relationships: Dependency



```
import java.util.Scanner;

public class Driver {

    public static void main(String[] args) {
        int x = getUserInput();
        System.out.println("The value you entered was: " + x);
    }

    public static int getUserInput() {
        int x = 0;
        Scanner scan = new Scanner(System.in);

        System.out.print("Please enter a number: ");
        while(!scan.hasNextInt()) {
            System.out.println("You did not enter a number.");
            scan.nextLine();
            System.out.print("Please enter a number: ");
        }
        x = scan.nextInt();
        scan.close();

        return x;
    }

}
```

UML for Dependency:



This class needs the Scanner class in order to properly perform this method’s behavior (ask for a number and print it to the monitor). So it **depends on** Scanner. Once the specific behavior is completed, then it is no longer needed.

Relationships: Association

- **Association** - when a Class ***has a***nother Class (i.e., is associated with; is encapsulated by; is a part of; etc.) in order to perform its necessary actions/behaviors

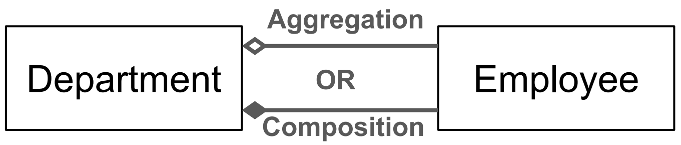
- More specifically, the Class stores a field/global variable of the other Class for continued use/reference

- Can be unidirectional or bidirectional; can be one-to-one, one-to-many, many-to-one, or many-to-many
  - Example: Library and Books

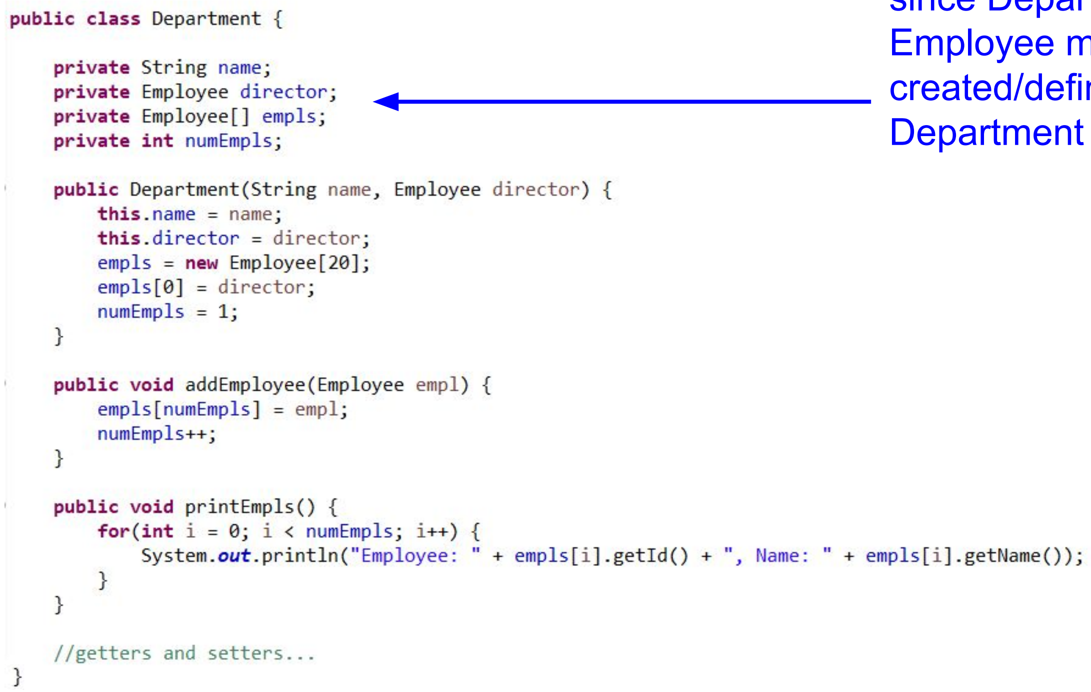
- There are two kinds of Association:
  - Aggregation
  - Composition

Relationships: Association

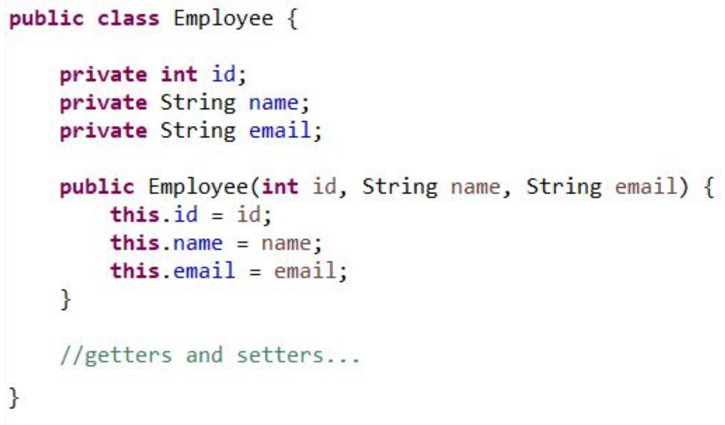
UML for Association:



Department needs Employee for as long as it exists since Department **has** Employee member variables. Employee must be defined before Department can be created/defined/used. It will exist as long as Department exists



```java
public class Department {

    private String name;
    private Employee director;
    private Employee[] empls;
    private int numEmpls;

    public Department(String name, Employee director) {
        this.name = name;
        this.director = director;
        empls = new Employee[20];
        empls[0] = director;
        numEmpls = 1;
    }

    public void addEmployee(Employee empl) {
        empls[numEmpls] = empl;
        numEmpls++;
    }

    public void printEmpls() {
        for(int i = 0; i < numEmpls; i++) {
            System.out.println("Employee: " + empls[i].getId() + ", Name: " + empls[i].getName());
        }
    }

    //getters and setters...
}
```



```java
public class Employee {

    private int id;
    private String name;
    private String email;

    public Employee(int id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }

    //getters and setters...

}
```

Relationships: Association - Aggregation

- **Aggregation** - (a.k.a. “weak association”) means that both Classes/entities can exists without the other

- More specifically, an Object of one Class is NOT created/destroyed based on the creation/destruction of an Object of the other Class
    - Example: Employees and Department

Relationships: Association - Composition

- **Composition** - (a.k.a. “strong association”) means that at least one of the Classes/entities CANNOT exist without the other

- More specifically, an Object of one Class is created/destroyed based on the creation/destruction of an Object of the other Class
  - Example: Car and Engine

- Sometimes Composition’s relationship is referred to as “part of” instead of “has a” (but both are essentially the same)

Relationships: Association - Aggregation vs Composition

- It can be difficult to determine if the association relationship is aggregation or composition
  - Example1: a person (not necessarily a user) and a heart (or any organ inside a human)
  - Example2: a car and an engine

- It often depends on context
  - Example1: an insurance company/database vs. a medical/hospital system
  - Example2: a used car dealership vs. a mechanic repair shop

- Ultimately, the question to ask is: for this system/problem, if one of the two associated Classes is destroyed, *should* the other class also be destroyed? (i.e., *should* an associated Class exist without the other?)

Relationships: Inheritance - A Motivating Example

- Assume we have a system that (among other things) is managing a grocery company's employees
  - The system needs to be able to manage all the necessary data about the employees (e.g., personal info, tax info, ect.) and calculate their paychecks

- Let us say we have 5 different kinds of employees, all with different pay calculations:
  - Store Employee - those who work in the store at an hourly rate
  - Warehouse Employee - those who work in warehouse and product storage who get hourly pay plus hazard pay
  - Distribution Employees - those who deliver products to the warehouses as well as deliver products from warehouses to stores who get an hourly pay plus payment based on the number of miles driven
  - Administrative Employee - those who keep business operations running (e.g., HR, Accounting, Marketing, etc.) and receive a salary
  - Upper Management Employees - those who are regional and company supervisors (e.g., CEO, CFO, CTO, etc.) and receive a salary plus bonus pay

Relationships: Inheritance - A Motivating Example

- We need to store all the necessary data about these 5 types of employees

- There is some data ALL employees have:
  - Personal info: name, email, phone, driver’s license, etc.
  - Tax info: social security number, etc.
  - Company info: company id, position title/name, hire date, etc.

- There is some data only SPECIFIC employees have:
  - Pay info:
    - Store Employee: hourly rate, number of hours worked
    - Warehouse Employee: hourly rate, hazard pay rate, number of hours worked
    - Distribution Employees: hourly rate, number of hours worked rate per mile, number of miles driven
    - Administrative Employee: salary
    - Upper Management Employees: salary, bonus

Relationships: Inheritance - A Motivating Example

- If we create a class for the employees (a great thing to do since that's a LOT of data to manage separately), we have 2 main options:
  - Create separate classes, one for each type of employee
  - Create a single class, that will represent all employees

- There are pros and cons to both approaches (note, there are more than what will be listed, but those listed are the biggest pros and cons)

Relationships: Inheritance - A Motivating Example

- If we create a Class for each type of employee (note, this is only a subset of the data that would need to be recorded):

StoreEmployee
id name email phone driversLicense SSN hireDate hourlyRate hoursWorked

Warehouse Employee
id name email phone driversLicense SSN hireDate hourlyRate hazardPayRate hoursWorked

Distribution Employee
id name email phone driversLicense SSN hireDate hourlyRate hoursWorked mileageRate milesDriven

Administrative Employee
id name email phone driversLicense SSN hireDate salary

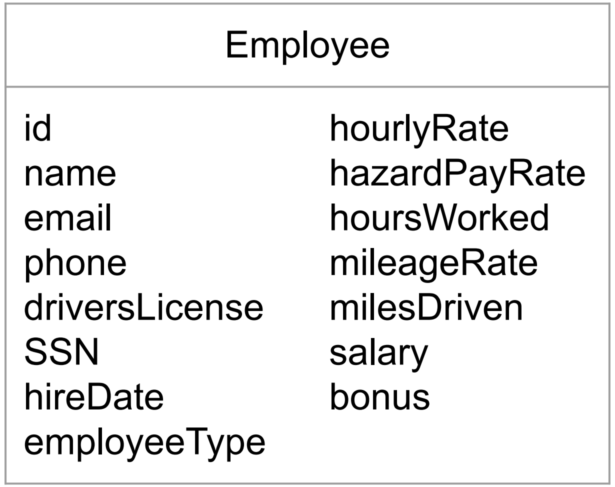
Upper Management Employee
id name email phone driversLicense SSN hireDate salary bonus

Relationships: Inheritance - A Motivating Example

Pros	Cons
● High degree of Modularity (easy to manage data and calculate paychecks) ● Only the data that is necessary is Encapsulated in the Class (no unused variables/methods are defined for a class)	● Multiple instances of the same data and same methods are present in the system (i.e., since there are multiple instances of the same data/behaviors, if something changes then we must change it for all of them, increasing the chance of human error and bugs) ● We must have special methods and data structures (e.g. Arrays) for each different type of employee (e.g., if we want to have an Array to hold/manage all employees, we must have 5 separate Arrays, one for each type of employee that must be managed/checked individually)

Relationships: Inheritance - A Motivating Example

- If we create a single Class for all employee (note, this is only a subset of the data that would need to be recorded):



Employee

id
name
email
phone
driversLicense
SSN
hireDate
employeeType
hourlyRate
hazardPayRate
hoursWorked
mileageRate
milesDriven
salary
bonus

Relationships: Inheritance - A Motivating Example

Pros	Cons
● Only a single set of data and methods exist (so if something changes in the system then it only needs to be updated once) ● We can manage all employees through a single variable type (e.g., if we want to have an Array to hold/manage all Employees, we can since they are all under a single data type; we also only need a single method with the “Employee” parameter type instead of 5 separate/overloaded methods with different parameter types)	● Low degree of modularity (i.e., for many behaviors we will need to first check what employeeType this Employee is and then decide how to proceed; e.g., calculating pay; this can lead to human error and bugs if not done carefully as it can easily become very complicated) ● No Employee will use all the data we have set aside for it (i.e., we will be wasting space since no Employee will use all the different payment data)

Relationships: Inheritance - A Motivating Example

- So which one do we choose? Is one better than another?
  - Neither is really better than the other and both aren't really great options…
- How about another idea. What we really want is a way we can use either whenever we want?
  - We want to use the single class version when we want to manage all employees through a single "unified" type
  - We want to use the multi class version when we want high modularity and separation of different behaviors
- **Inheritance** - when one class ***is a***nother class (i.e., "is *also* another class")
- **Polymorphism** - the ability for an Object to change data types (i.e., take on different forms/types)
  - Through this, we can have a class change between the "general" single-class version OR the "specialized" multi-class version at our discretion
  - Note: This also includes all the rules/definitions/implementations surrounding *casting* for Classes

Relationships: Inheritance - Generalization and Specialization

- **Generalization** - a “higher order” entity/Class created from the common data/behavior of a set of specialized entities/Classes

- **Specialization** - a “lower order” entity/Class created from the individual data/behavior of a specific type of a more general entity/Class

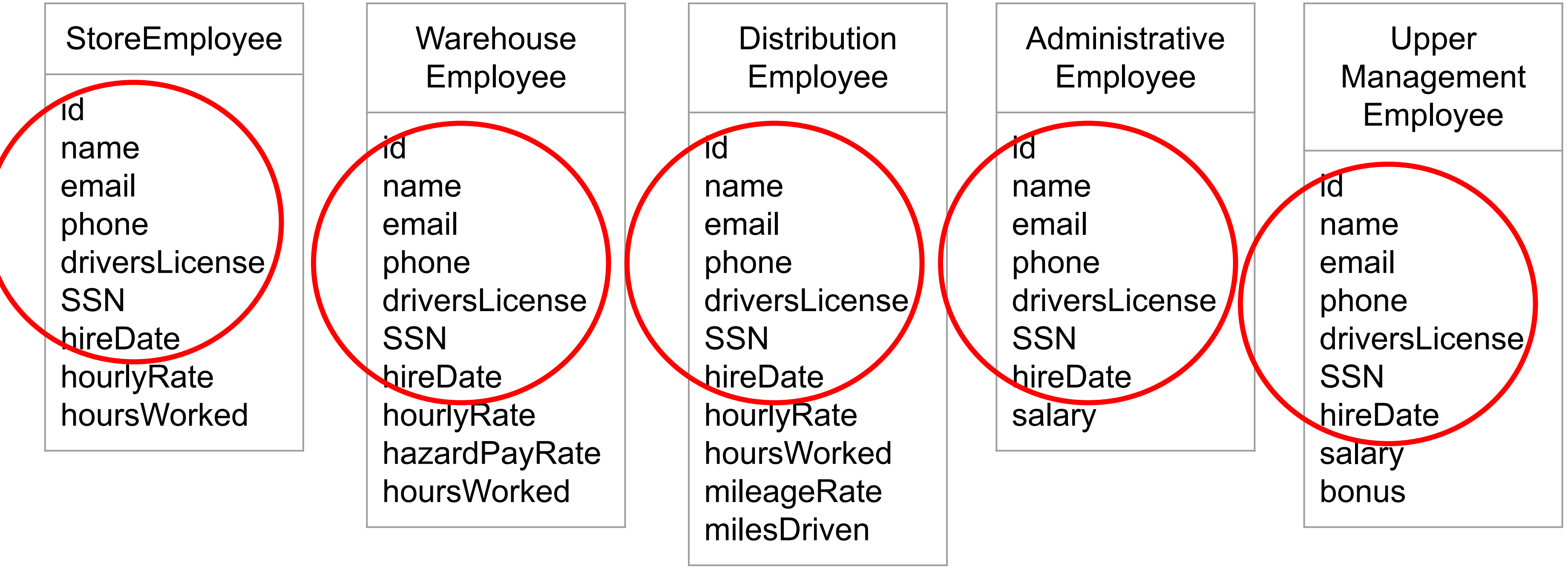
Relationships: Inheritance - Generalization and Specialization

- Generalization - find all common data/behaviors to create a new Class

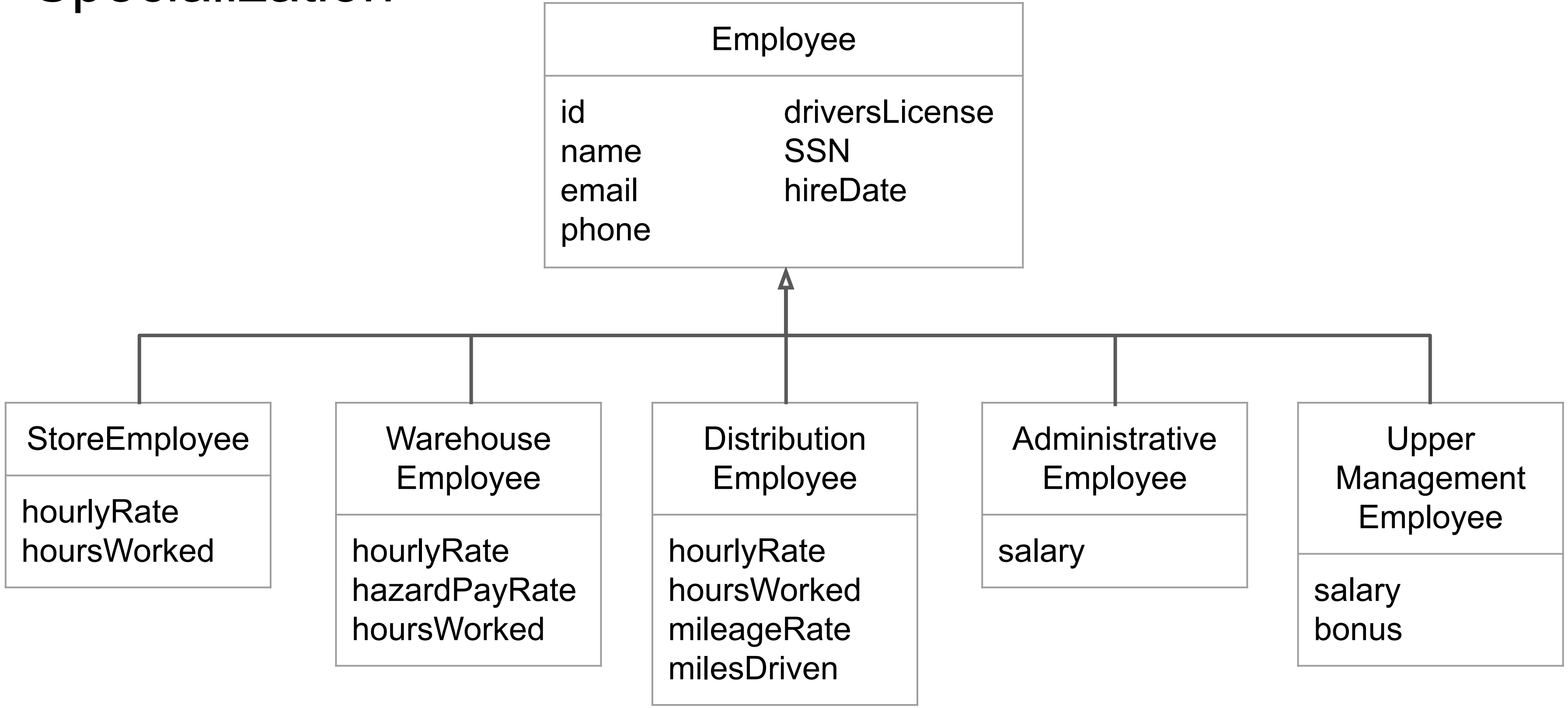
StoreEmployee	Warehouse Employee	Distribution Employee	Administrative Employee	Upper Management Employee
id	id	id	id	id
name	name	name	name	name
email	email	email	email	email
phone	phone	phone	phone	phone
driversLicense	driversLicense	driversLicense	driversLicense	driversLicense
SSN	SSN	SSN	SSN	SSN
hireDate	hireDate	hireDate	hireDate	hireDate
hourlyRate	hourlyRate	hourlyRate	salary	salary
hoursWorked	hazardPayRate	hoursWorked		bonus
	hoursWorked	mileageRate		
		milesDriven		

Relationships: Inheritance - Generalization and Specialization

- Generalization - find all common data/behaviors to create a new Class

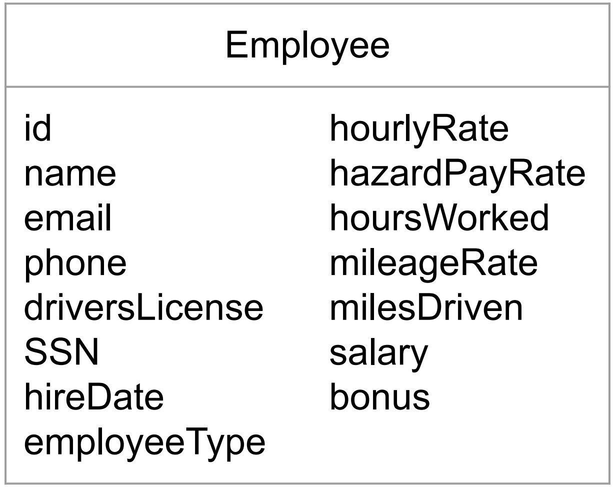


Relationships: Inheritance - Generalization and Specialization



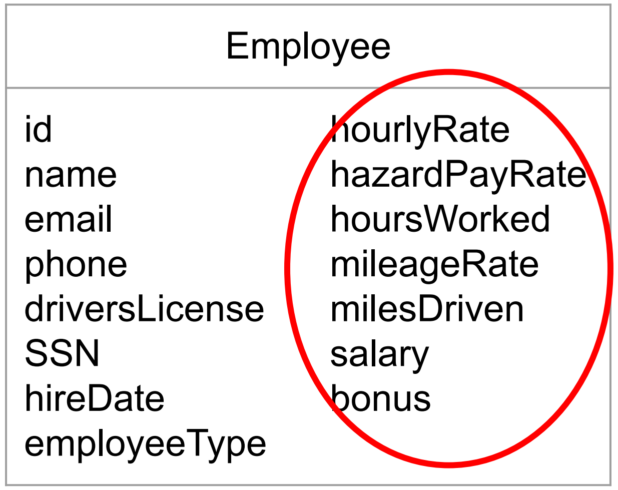
Relationships: Inheritance - Generalization and Specialization

- Specialization - find all individual data/behaviors to create new Classes

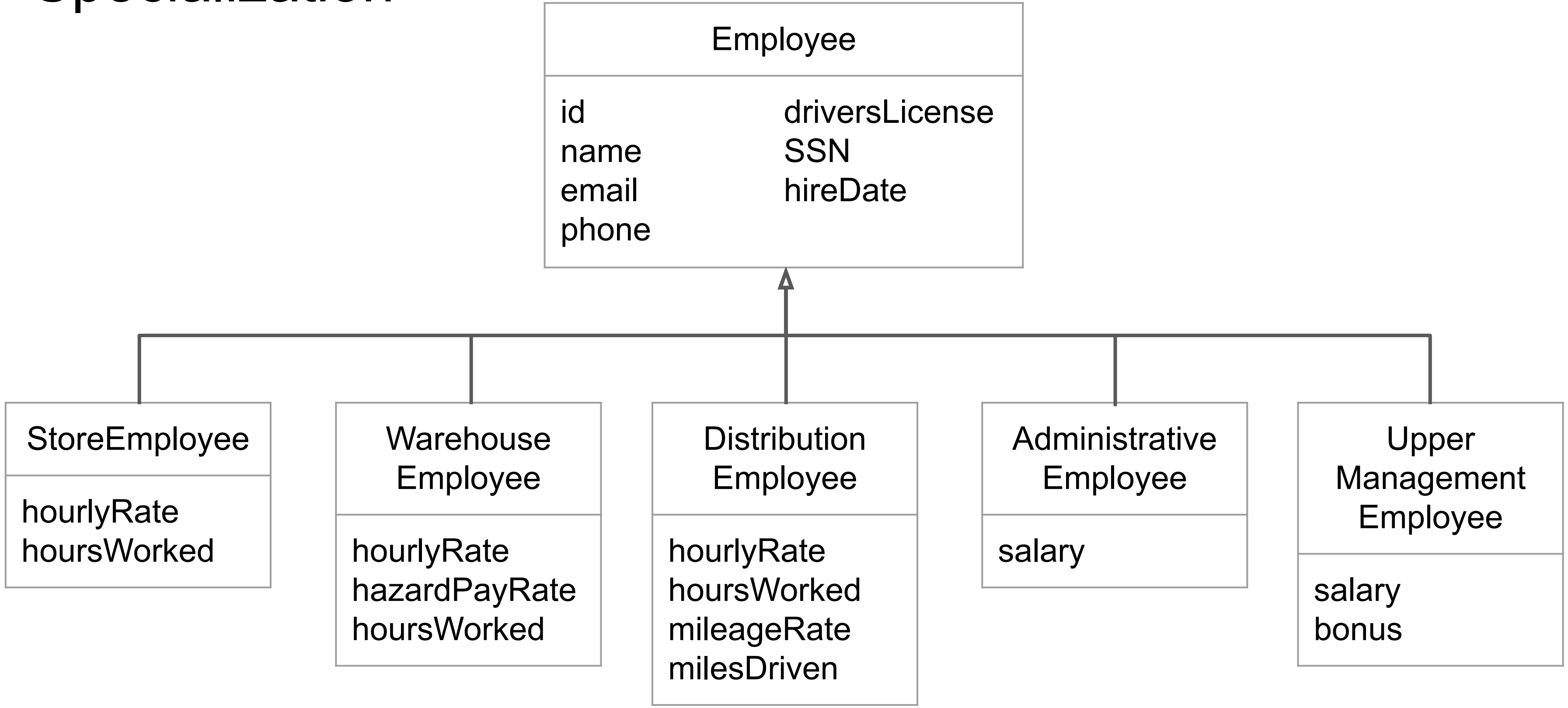


Relationships: Inheritance - Generalization and Specialization

- Specialization - find all individual data/behaviors to create new Classes



Relationships: Inheritance - Generalization and Specialization



Relationships: Inheritance

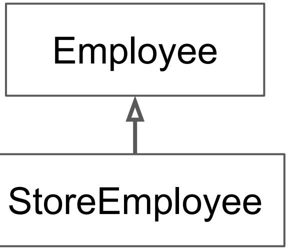
- In order for Inheritance and Polymorphism to work programmatically, we need to connect them somehow (basically, the compiler needs to know that the classes are connected and how they “fit” into each other)

- In Java, we use the reserved word “***extends***”

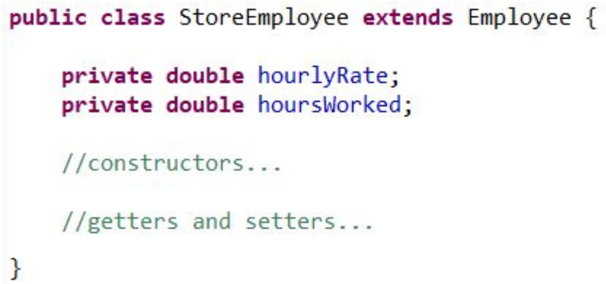
- The general class is often called the “superclass” or “parent class” and the specialized class is often called the “subclass” or “child class”
    - We often say that the subclass **extends** the superclass

Relationships: Inheritance

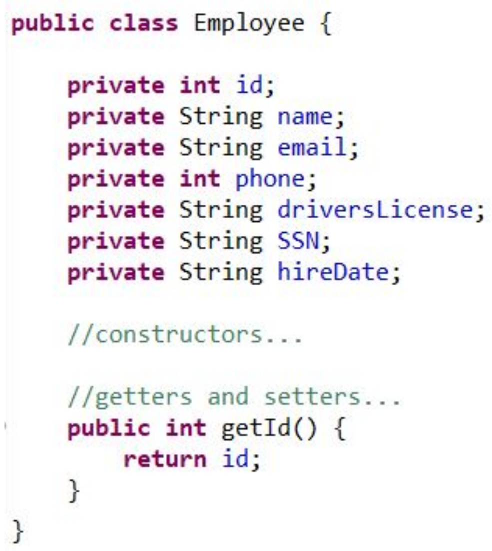
UML for Inheritance:



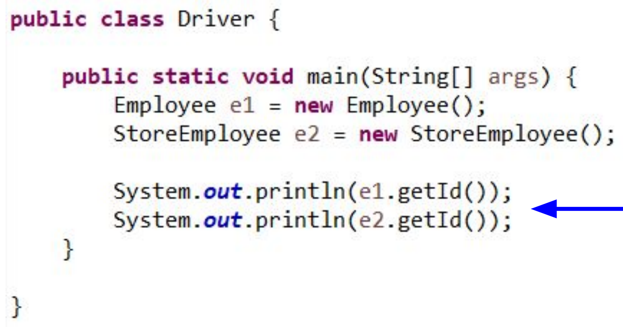
Inherits from Employee



```
public class StoreEmployee extends Employee {

    private double hourlyRate;
    private double hoursWorked;

    //constructors...

    //getters and setters...

}
```



```
public class Employee {

    private int id;
    private String name;
    private String email;
    private int phone;
    private String driversLicense;
    private String SSN;
    private String hireDate;

    //constructors...

    //getters and setters...
    public int getId() {
        return id;
    }
}
```



```
public class Driver {

    public static void main(String[] args) {
        Employee e1 = new Employee();
        StoreEmployee e2 = new StoreEmployee();

        System.out.println(e1.getId());
        System.out.println(e2.getId());
    }

}
```

StoreEmployee has access to all of Employee's data/behavior

Relationships: Inheritance - super

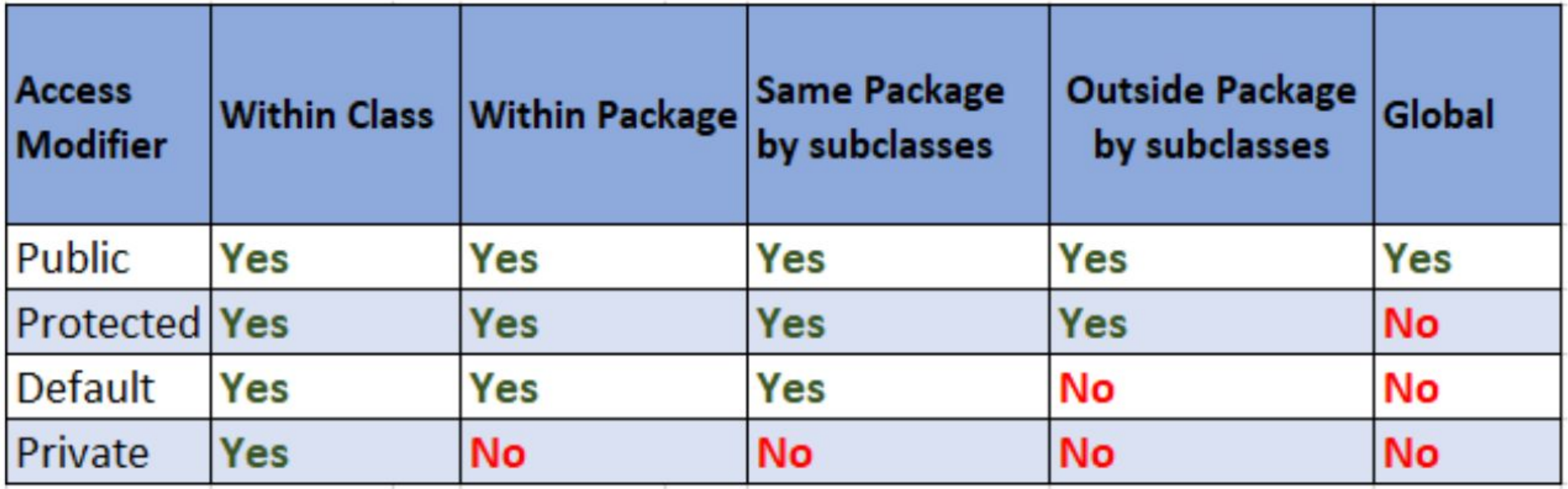
- **super** is a reserved word that allows a subclass (child) to directly reference its superclass (parent)

- Most often used in constructors, when we *extend* a class, we inherit all of its data (though we may not be able to directly access it; revisit visibility), so when we make constructors for our subclass, we must also make parameters for the necessary data in our superclass

- In order to pass this data to the superclass, we call can call “super()” as a method (it will act/access a constructor of the superclass, depending on the parameters that are input)
    - We have to access the superclass constructor specifically in order for Java to set aside the proper amount of memory for the subclass Object

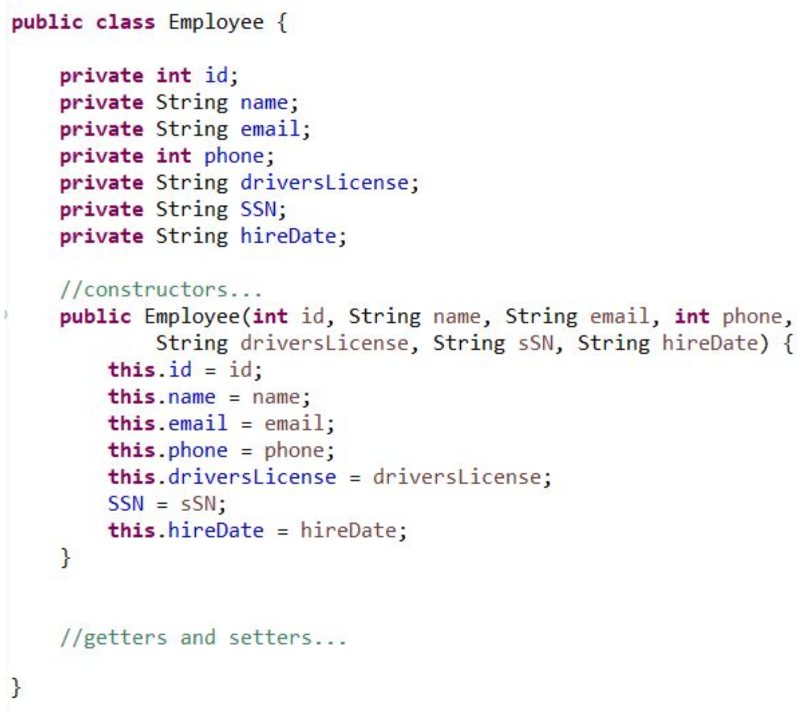
Relationships: Inheritance - Scope/Visibility Revisited

- Visibility still applies like normal in Inheritance. This means that if a superclass's attributes are set to *private*, even though the subclass "inherits" it, the subclass CANNOT access it
  - You must either make the attributes *protected/default/public* or access them through *protected/default/public* superclass methods



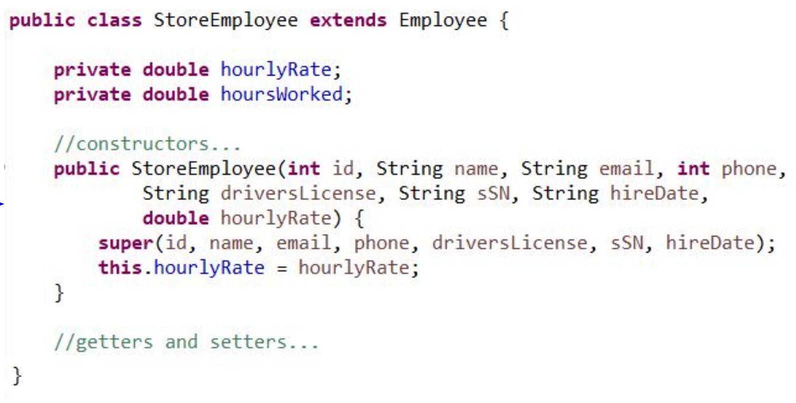
Access Modifier	Within Class	Within Package	Same Package by subclasses	Outside Package by subclasses	Global
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

Relationships: Inheritance - super



```java
public class Employee {

    private int id;
    private String name;
    private String email;
    private int phone;
    private String driversLicense;
    private String SSN;
    private String hireDate;

    //constructors...
    public Employee(int id, String name, String email, int phone,
            String driversLicense, String sSN, String hireDate) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.phone = phone;
        this.driversLicense = driversLicense;
        SSN = sSN;
        this.hireDate = hireDate;
    }


    //getters and setters...

}
```

The constructor of StoreEmployee must require both its own data and the data of its superclass/parent. It must call a superclass/parent constructor using “super” and passing in the necessary data.



```java
public class StoreEmployee extends Employee {

    private double hourlyRate;
    private double hoursWorked;

    //constructors...
    public StoreEmployee(int id, String name, String email, int phone,
            String driversLicense, String sSN, String hireDate,
            double hourlyRate) {
        super(id, name, email, phone, driversLicense, sSN, hireDate);
        this.hourlyRate = hourlyRate;
    }

    //getters and setters...
}
```

Relationships: Inheritance

- In Java, if a Class does not inherit from any other Class it inherits, by default, from java.lang.Object

- In Java, a class can ONLY inherit from ONE Class (i.e., can only have 1 parent)

Overriding

- Java allows methods to be overridden
- Redefine a previously defined method in a superclass/ancestor
- Only subclasses can override methods of classes they inherit from
- To override, you must give the EXACT SAME method declaration as the previously defined method
- Using the overridden version of the method (i.e., using/calling the method through the class that overrode it) will execute the newly defined behaviour/instructions (the old version of the method will NOT be executed in any way)
  - The original method can still be used and is accessed from the superclass by using “super”

Annotations

- A form of metadata that provides information about a program that is not part of the program itself
- Similar to comments in that they do NOT affect the program’s code/instructions/execution in any way
- Not similar to comments in that there are specific words/phrases that must be used (you can’t just write anything for annotations, as each one has a specific meaning)
- Uses the “@” symbol in front of it to indicate that it is an annotation
- Example:
  - @Override

Special Methods

- main()

  -
    ```
    public static void main(String[] args){
        //set of instructions
    }
    ```
  - When running a program, we need to start somewhere; the “main” method is where we start executing instructions

- toString()

  -
    ```
    public String toString(){
        //set of instructions
    }
    ```
  - Inherited from java.lang.Object
  - How do we know what/how to print for any given object?
    - We don’t…
    - So “toString” allows the programmer to dictate how the object should be printed out: what is the important information?, what is the proper format?, etc.

Special Methods

- equals()
  - **public boolean equals(Object obj){**
    //set of instructions
    }
  - Inherited from java.lang.Object
  - We CAN'T use the equality operator (==) to determine if two objects are equal (equality will determine if they are the same object, that is, if two variables reference the same object)
    - Just like the String class
  - How do we know any two given objects are equivalent?
    - We don't…
    - So "equals" allows the programmer to dictate what makes two objects equivalent