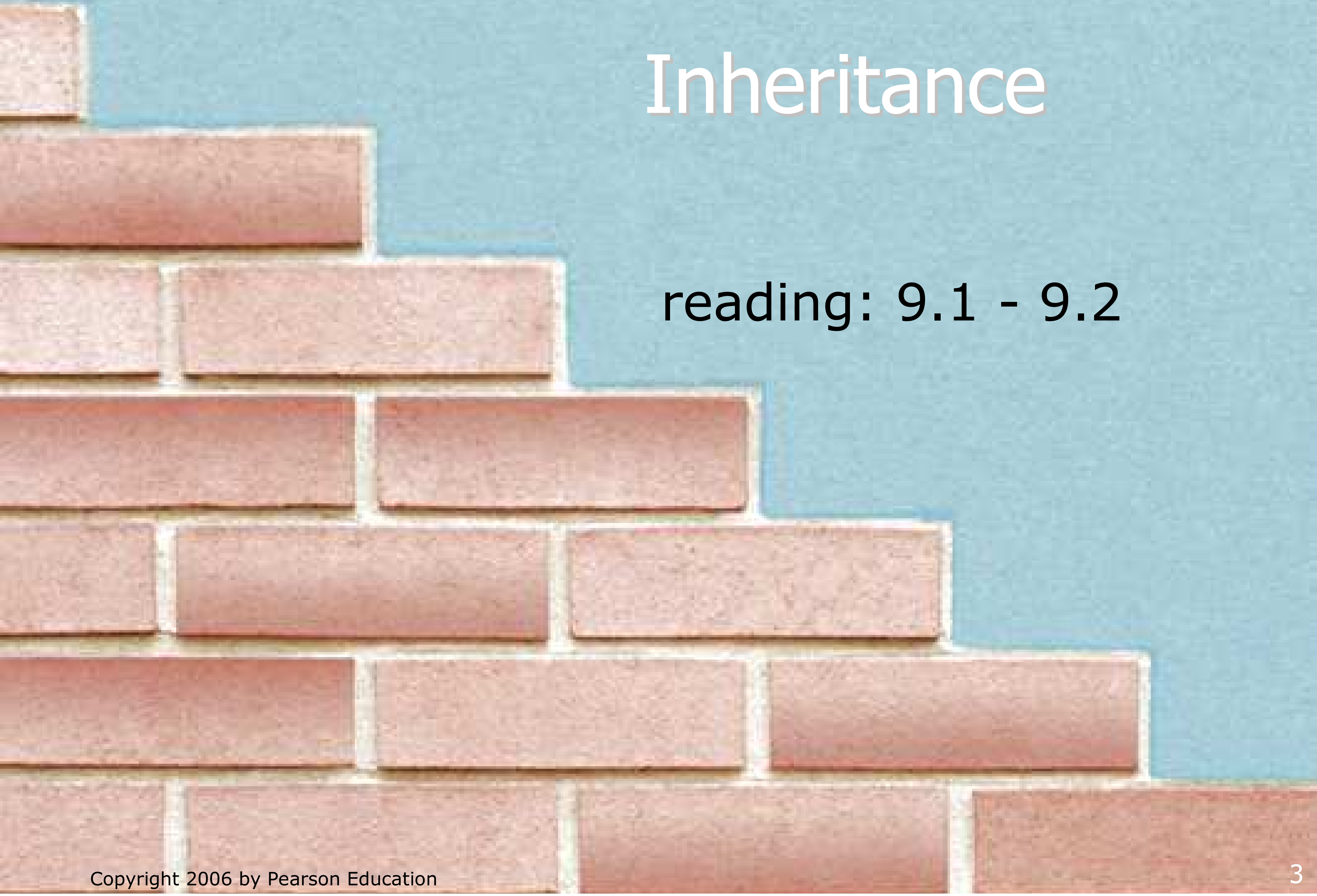
Building Java Programs

Chapter 9: Inheritance and Interfaces

Lecture outline

- background
  - categories of employees
  - relationships and hierarchies
- inheritance programming
  - creating subclasses
  - overriding behavior
  - multiple levels of inheritance
  - interacting with the superclass using `super`



Inheritance

reading: 9.1 - 9.2

The software crisis

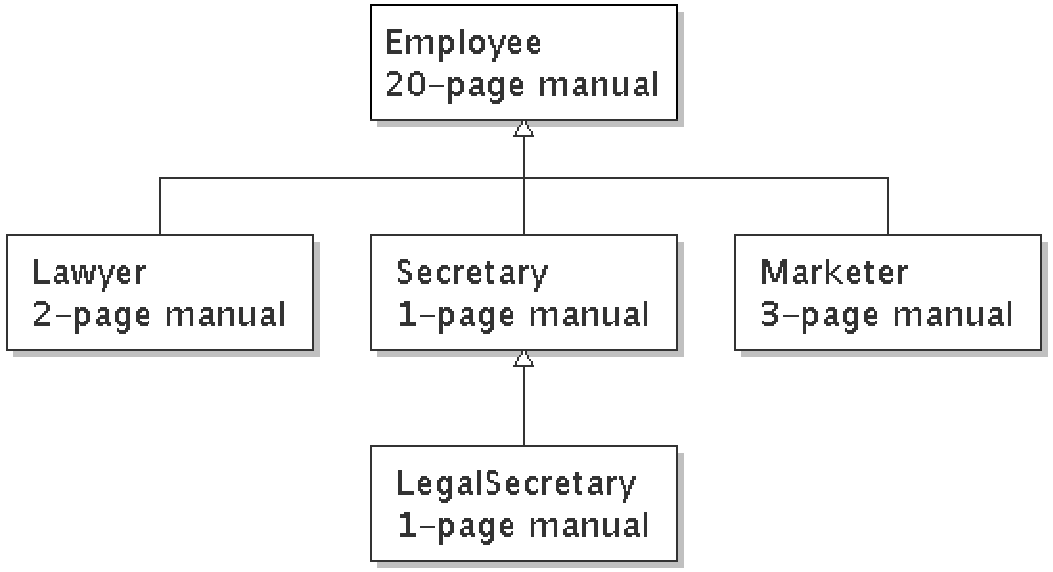
- **software engineering**: The practice of conceptualizing, designing, developing, documenting, and testing large-scale computer programs.

- Large-scale projects face many issues:
  - getting many programmers to work together
  - getting code finished on time
  - avoiding redundant code
  - finding and fixing bugs
  - maintaining, improving, and reusing existing code

- **code reuse**: The practice of writing program code once and using it in many contexts.

Law firm employee analogy

- common rules: hours, vacation time, benefits, regulations, ...
  - all employees attend common orientation to learn general rules
  - each employee receives 20-page manual of the common rules
- each subdivision also has specific rules:
  - employee attends a subdivision-specific orientation to learn them
  - employee receives a smaller (1-3 page) manual of these rules
  - smaller manual adds some rules and also changes some rules from the large manual ("use the pink form instead of yellow form"...)



Separating behavior

- Why not just have a 22 page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?

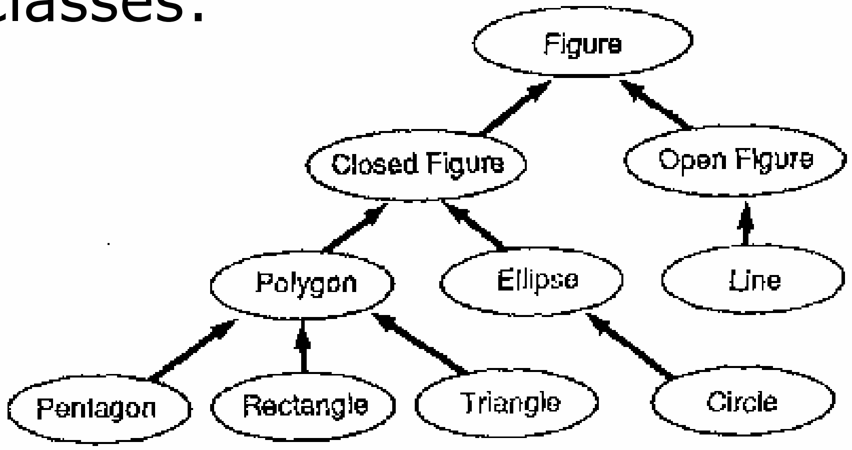
- Some advantages of the separate manuals:
  - maintenance: If a common rule changes, we'll need to update only the common manual.
  - locality: A person can look at the lawyer manual and quickly discover all rules that are specific to lawyers.

- Some key ideas from this example:
  - It's useful to be able to describe general rules that will apply to many groups (the 20-page manual).
  - It's also useful for a group to specify a smaller set of rules for itself, including being able to replace rules from the overall set.

Is-a relationships, hierarchies

- **is-a relationship**: A hierarchical connection where one category can be treated as a specialized version of another.
  - every marketer is an employee
  - every legal secretary is a secretary

- **inheritance hierarchy**: A set of classes connected by is-a relationships that can share common code.
  - Often drawn as a downward tree of connected boxes or ovals representing classes:



Employee regulations

- Consider the following employee regulations:
  - Employees work 40 hours per week.
  - Employees make $40,000 per year, except legal secretaries who make $5,000 extra per year ($45,000 total), and marketers who make $10,000 extra per year ($50,000 total).
  - Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
  - Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.

- Each type of employee has some unique behavior:
  - Lawyers know how to sue.
  - Marketers know how to advertise.
  - Secretaries know how to take dictation.
  - Legal secretaries know how to prepare legal documents.

General employee code

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;      // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;           // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";     // use the yellow form
    }
}
```

- Exercise: Implement class `Secretary`, based on the previous employee regulations.

Redundant secretary code

```
// A redundant class to represent secretaries.
public class Secretary {
    public int getHours() {
        return 40;          // works 40 hours / week
    }

    public double getSalary() {
        return 40000.0;     // $40,000.00 / year
    }

    public int getVacationDays() {
        return 10;          // 2 weeks' paid vacation
    }

    public String getVacationForm() {
        return "yellow";    // use the yellow form
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Desire for code-sharing

- The `takeDictation` method is the only unique behavior in the `Secretary` class.

- We'd like to be able to say the following:

```
// A class to represent secretaries.
public class Secretary {
    <copy all the contents from Employee class.>

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Inheritance

- **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes

- We say that one class can *extend* another by absorbing its state and behavior.
  - **superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
    - The subclass receives a copy of every field and method from its superclass.

Inheritance syntax

- Creating a subclass, general syntax:

  `public class` ***<name>*** `extends` ***<superclass name>*** `{`

  - Example:

```
public class Secretary extends Employee {
    ....
}
```

- By extending `Employee`, each `Secretary` object now:
  - receives a `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` method automatically
  - can be treated as an `Employee` by any other code (seen later)

    (e.g. a `Secretary` could be stored in a variable of type `Employee` or stored as an element of an `Employee[]`)

Improved secretary code

```
// A class to represent secretaries.
public class Secretary extends Employee {
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

- Now we only write the parts unique to each type.
  - `Secretary` inherits `getHours`, `getSalary`, `getVacationDays`, and `getVacationForm` methods from `Employee`.
  - `Secretary` adds the `takeDictation` method.

Implementing Lawyer

Let's implement a `Lawyer` class.

- Consider the following employee regulations:
  - Lawyers who get an extra week of paid vacation (a total of 3).
  - Lawyers use a pink form when applying for vacation leave.
  - Lawyers have some unique behavior: they know how to sue.
- The problem: We want lawyers to inherit *most* of the behavior of the general employee, but we want to replace certain parts with new behavior.

Overriding methods

- **override**: To write a new version of a method in a subclass that replaces the superclass's version.
  - There is no special syntax for overriding.
    To override a superclass method, just write a new version of it in the subclass. This will replace the inherited version.
  - Example:

```
public class Lawyer extends Employee {
    // overrides getVacationForm method in Employee class
    public String getVacationForm() {
        return "pink";
    }

    ...
}
```

  - Exercise: Complete the `Lawyer` class.

Complete Lawyer class

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacationDays() {
        return 15;           // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

- Exercise: Now complete the `Marketer` class. Marketers make $10,000 extra ($50,000 total) and know how to advertise.

Complete Marketer class

```
// A class to represent marketers.
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return 50000.0;      // $50,000.00 / year
    }
}
```

Levels of inheritance

- Deep hierarchies can be created by multiple levels of subclassing.
  - Example: The legal secretary is the same as a regular secretary except for making more money ($45,000) and being able to file legal briefs.

```
public class LegalSecretary extends Secretary {
    ...
}
```

  - Exercise: Complete the `LegalSecretary` class.

Complete LegalSecretary class

```
// A class to represent legal secretaries.
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }

    public double getSalary() {
        return 45000.0;      // $45,000.00 / year
    }
}
```

Interacting with the superclass: the `super` keyword

reading: 9.3

Changes to common behavior

- Imagine that a company-wide change occurs that affects all employees.

  Example: Because of inflation, everyone is given a $10,000 raise.

  - The base employee salary is now $50,000.
  - Legal secretaries now make $55,000.
  - Marketers now make $60,000.

- We must modify our code to reflect this policy change.

Modifying the superclass

- This modified `Employee` class handles the new raise:

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;           // works 40 hours / week
    }

    public double getSalary() {
        return 50000.0;      // $50,000.00 / year
    }

    ...
}
```

  - What problem now exists in the code?

- The `Employee` subclasses are now incorrect.
  - They have overridden the `getSalary` method to return other values such as 45,000 and 50,000 that need to be changed.

An unsatisfactory solution

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 55000.0;
    }
    ...
}

public class Marketer extends Employee {
    public double getSalary() {
        return 60000.0;
    }
    ...
}
```

- The employee subtypes' salaries are tied to the overall base employee salary, but the subclasses' `getSalary` code does not reflect this relationship.

Calling overridden methods

- A subclass can call an overridden method with the `super` keyword.

- Calling an overridden method, syntax:

  `super .` ***<method name>*** `(` ***<parameter(s)>*** `)`

  - Example:

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.0;
    }
    ...
}
```

  - Exercise: Modify the `Lawyer` and `Marketer` classes to also use the `super` keyword.

Improved subclasses

```
public class Lawyer extends Employee {
    public String getVacationForm() {
        return "pink";
    }

    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}

public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

Inheritance and constructors

- Imagine that we want to give employees more vacation days the longer they've been with the company.
  - For each year worked, we'll award 2 additional vacation days.
  - When an Employee object is constructed, we'll pass in the number of years the person has been with the company.
  - This will require us to modify our `Employee` class and add some new state and behavior.
  - Exercise: Make the necessary modifications to the `Employee` class.

Modified Employee class

```
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getHours() {
        return 40;
    }

    public double getSalary() {
        return 50000.0;
    }

    public int getVacationDays() {
        return 10 + 2 * years;
    }

    public String getVacationForm() {
        return "yellow";
    }
}
```

Problem with constructors

- Now that we've added the constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
       ^
```

  - The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.
  - The long explanation: (next slide)

The detailed explanation

- Constructors aren't inherited.
  - The `Employee` subclasses don't inherit the `public Employee(int years)` constructor.
  - Since our subclasses don't have constructors, they receive a default parameterless constructor that contains the following:

```
public Lawyer() {
    super();        // calls public Employee() constructor
}
```

- But our `public Employee(int years)` replaces the default `Employee` constructor.
  - Therefore all the subclasses' default constructors are now trying to call a non-existent default superclass constructor.

Calling superclass constructor

- Syntax for calling superclass's constructor:

  `super(` ***<parameter(s)>*** `);`

  - Example:

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);   // call Employee constructor
    }
    ...
}
```

  - The call to the superclass constructor must be the first statement in the subclass constructor.

  - Exercise: Make a similar modification to the `Marketer` class.

Modified Marketer class

```
// A class to represent marketers.
public class Marketer extends Employee {
    public Marketer(int years) {
        super(years);
    }

    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

- Exercise: Modify the `Secretary` subclass to make it compile:
  - Secretaries' years of employment are not tracked and they do not earn extra vacation for them.
  - `Secretary` objects are also constructed without a `years` parameter.

Modified Secretary class

```
// A class to represent secretaries.
public class Secretary extends Employee {
    public Secretary() {
        super(0);
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

  - Note that since the `Secretary` doesn't require any parameters to its constructor, the `LegalSecretary` now compiles without a constructor (its default constructor calls the parameterless `Secretary` constructor).

- This isn't the best solution; it isn't that Secretaries work for 0 years, it's that they don't receive a bonus. How can we fix it?

Inheritance and fields

- Suppose that we want to give lawyers a $5000 raise for each year they've been with the company.

- The following modification doesn't work:

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
    }

    public double getSalary() {
        return super.getSalary() + 5000 * years;
    }
    ...
}
```

- The error is the following:

```
Lawyer.java:7: years has private access in Employee
        return super.getSalary() + 5000 * years;
                                          ^
```

Private access limitations

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
    }

    public double getSalary() {
        return super.getSalary() + 5000 * years;
    }
    ...
}
```

- The error is the following:

```
Lawyer.java:7: years has private access in Employee
        return super.getSalary() + 5000 * years;
                                          ^
```

- Private fields cannot be directly accessed from other classes, not even subclasses.
  - One reason for this is to prevent malicious programmers from using subclassing to circumvent encapsulation.
  - How can we get around this limitation?

Improved Employee code

Add an accessor for any field needed by the superclass.

```
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getYears() {
        return years;
    }
    ...
}
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
    }

    public double getSalary() {
        return super.getSalary() + 5000 * getYears();
    }
    ...
}
```

Revisiting Secretary

- The `Secretary` class currently has a poor solution.
  - We set all Secretaries to 0 years because they do not get a vacation bonus for their service.
  - If we call `getYears` on a `Secretary` object, we'll always get 0.
  - This isn't a good solution; what if we wanted to give some other reward to *all* employees based on years of service?

- Let's redesign our `Employee` class a bit to allow for a better solution.

Improved Employee code

Let's separate the standard 10 vacation days from those that are awarded based on seniority.

```
public class Employee {
    private int years;

    public Employee(int initialYears) {
        years = initialYears;
    }

    public int getVacationDays() {
        return 10 + getSeniorityBonus();
    }

    // vacation days given for each year in the company
    public int getSeniorityBonus() {
        return 2 * years;
    }
    ...
}
```

- How does this help us improve the `Secretary`?

Improved Secretary code

The `Secretary` can selectively override the `getSeniorityBonus` method, so that when it runs its `getVacationDays` method, it will use this new version as part of the computation.

- Choosing a method at runtime like this is called *dynamic binding*.

```
public class Secretary extends Employee {
    public Secretary(int years) {
        super(years);
    }

    // Secretaries don't get a bonus for their years of service.
    public int getSeniorityBonus() {
        return 0;
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```