

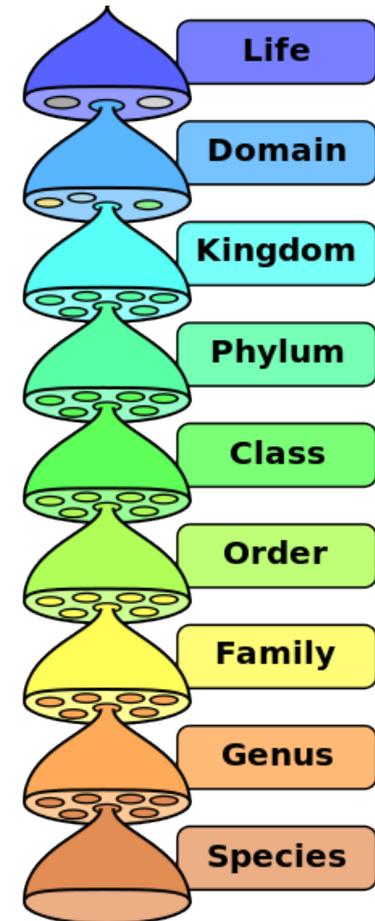
# Inheritance & Super

(interacting with the superclass)

Subset of the Supplement Lesson slides from: Building Java Programs, Chapter 9.1 & 9.2  
by Stuart Reges and Marty Stepp (<http://www.buildingjavaprograms.com/>).

# Why add Hierarchy?

- Why would we want to create a Hierarchy for our code?
- Why does Biology use a Hierarchy to study life?
  - Life: Cellular live (other: non-Cellular)
  - Domain: Eukaryotes (others: Archaea & Bacteria)
  - Kingdom: Animalia (others: Plantae, Fungi, Bacteria, ...)
  - Phylum: Chordate – has a spine
  - Class: Mammalia – has hair, born live, ...
  - Order: Carnivora – eats meat
  - Family: Canidae
  - Genus: Canis
  - Species: *C. lupus*
  - SubSpecies: *C. l. familiaris* – like Bulldogs
- Are there other places where these orders of class and subclasses helps us organize things?



# Interfaces (in context)

- Establish a 'is-a' relationship without code sharing
- One way of addressing the software crisis
- Allow dividing up a complex task
  - all developers code to a common interface
  - implementations can be changing as long as interface is respected
- Free developers from details (ArrayList vs LinkedList? Doesn't matter, they're both Lists!)
- Reuse utilities like Collections.sort
  - Just need to implement the Comparable interface!

# The software crisis

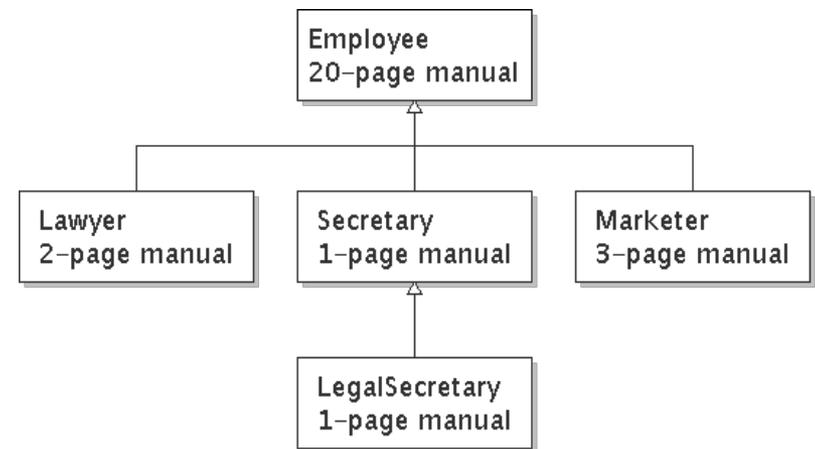
- **software engineering:** The practice of developing, designing, documenting, testing large computer programs.
- Large-scale projects face many issues:
  - getting many programmers to work together
  - getting code finished on time
  - avoiding redundant code
  - finding and fixing bugs
  - maintaining, improving, and reusing existing code
- **code reuse:** The practice of writing program code once and using it in many contexts.

# Law firm HR Analogy

- Our Law firm has a variety of Employees:
  - Lawyers
  - Marketers
  - Secretaries
    - Legal Secretaries
- They all have:
  - vacation & vacation forms
  - salary
  - and various behaviours: suing, advertise, take dictation, & prepare legal documents.
  - training manuals
- So how can we create an HR system to reflect this?...

# Law firm employee analogy

- common rules: hours, vacation, benefits, regulations ...
  - all employees attend a common orientation to learn general company rules
  - each employee receives a 20-page manual of common rules
- each subdivision also has specific rules:
  - employee receives a smaller (1-3 page) manual of these rules
  - smaller manual adds some new rules and also changes some rules from the large manual

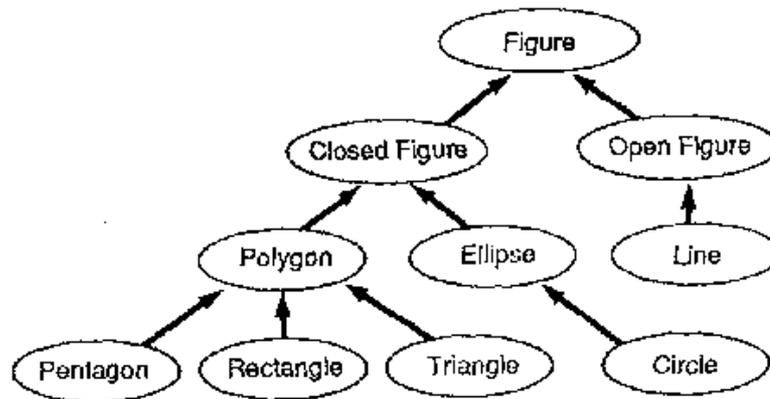


# Separating behavior

- 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: Only one update if a common rule changes.
  - locality: Quick discovery of all rules specific to lawyers.
- Some key ideas from this example:
  - General rules are useful (the 20-page manual).
  - Specific rules that may override general ones are also useful.

# 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.



# Employee regulations

- Consider the following employee regulations:
  - Employees work 40 hours / 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.

# An Employee class

```
// 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. (Secretaries can take dictation.)

# Redundant Secretary class

```
// 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

- `takeDictation` is the only unique behavior in `Secretary`.
- We'd like to be able to say:

*// A class to represent secretaries.*

```
public class Secretary {
```

```
    copy all the contents from the 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
- One class can *extend* another, absorbing its data/behavior.
  - **superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass

# Inheritance syntax

```
public class name extends superclass {
```

– 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 client code (seen later)

# 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

- Consider the following lawyer 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.
- Problem: We want lawyers to inherit *most* behavior from employee, but we want to replace parts with new behavior.

# Overriding methods

- **override:** To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

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

- Exercise: Complete the `Lawyer` class.
  - (3 weeks vacation, pink vacation form, can sue)

# 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: Complete the `Marketer` class. Marketers make \$10,000 extra (\$50,000 total) and know how to advertise.

# 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

- Multiple levels of inheritance in a hierarchy are allowed.
  - Example: A legal secretary is the same as a regular secretary but makes more money (\$45,000) and can file legal briefs.

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

- Exercise: Complete the `LegalSecretary` class.

# 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

Subset of the Supplement Lesson slides from: Building Java Programs, Chapter 9.2  
by Stuart Reges and Marty Stepp (<http://www.buildingjavaprograms.com/>).

# Changes to common behavior

- Let's return to our previous company/employee example.
- Imagine a company-wide change affecting all employees.

Example: Everyone is given a \$10,000 raise due to inflation.

- 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

```
// A class to represent employees (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
    }

    ...
}
```

– Are we finished?

- The `Employee` subclasses are still incorrect.
  - They have overridden `getSalary` to return other values.

# 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;  
    }  
    ...  
}
```

- Problem: The subclasses' salaries are based on the Employee salary, but the `getSalary` code does not reflect this.

# Calling overridden methods

- Subclasses can call overridden methods with `super`

```
super.method (parameters)
```

## – Example:

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

- Exercise: Modify `Lawyer` and `Marketer` to use `super`.

# 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 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 are not inherited.
  - Subclasses don't inherit the `Employee(int)` constructor.
  - Subclasses receive a default constructor that contains:

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

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

# Calling superclass constructor

```
super (parameters) ;
```

## – Example:

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

– The `super` call must be the first statement in the 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.
  - Secretaries' years of employment are not tracked.
  - They do not earn extra vacation for years worked.

# 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);
    }
}
```

- Since `Secretary` doesn't require any parameters to its constructor, `LegalSecretary` compiles without a constructor.
  - Its default constructor calls the `Secretary()` constructor.

# Inheritance and fields

- Try to give lawyers \$5000 for each year at the company:

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

- Does not work; 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 subclasses.
  - One reason: So that subclassing can't break encapsulation.
  - How can we get around this limitation?

# Improved Employee code

Add an accessor for any field needed by the subclass.

```
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?
- Redesign our `Employee` class 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

- Secretary can selectively override `getSeniorityBonus`; when `getVacationDays` runs, it will use the new version.
  - Choosing a method at runtime 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);
    }
}
```