CS 106A, Lecture 18
More Classes and Objects; Inheritance

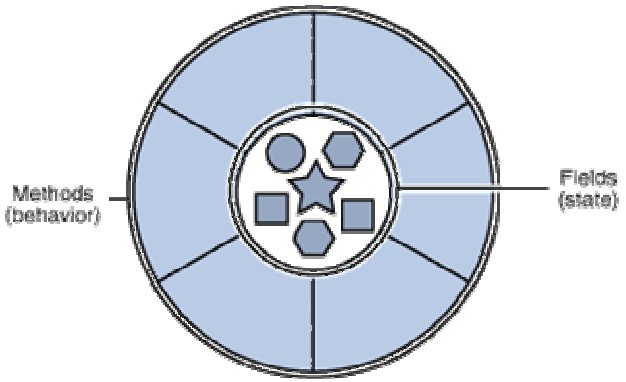
reading:

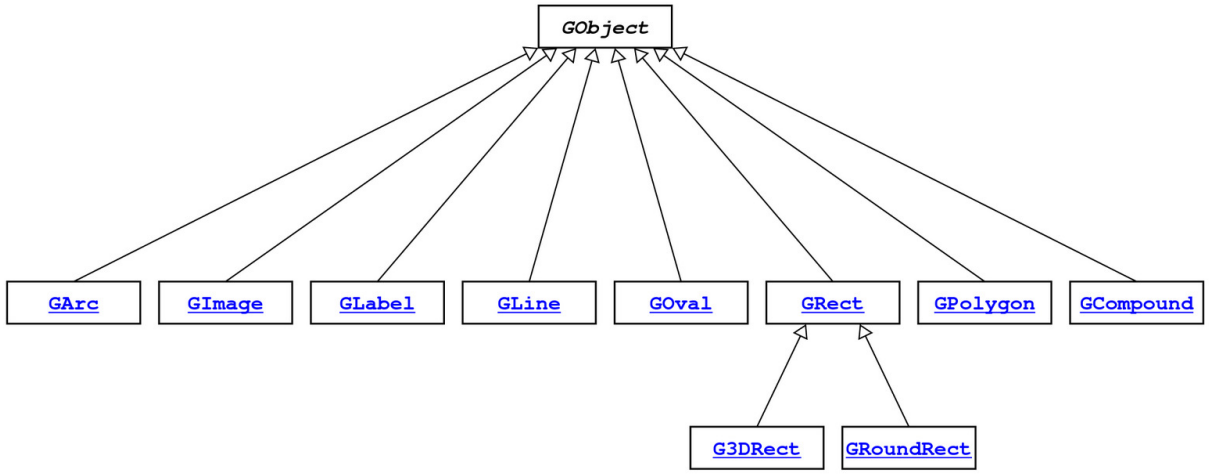
Art & Science of Java, Ch. 6

Lecture Outline

- Today we will revisit **classes and objects**.
  - Classes help us decompose large complex systems so we can solve larger problems in an elegant "object-oriented" way.

- We will also learn about **inheritance**.
  - Inheritance allows one class to be based on another "parent" class.
  - Hierarchies of related classes allow for code reuse.





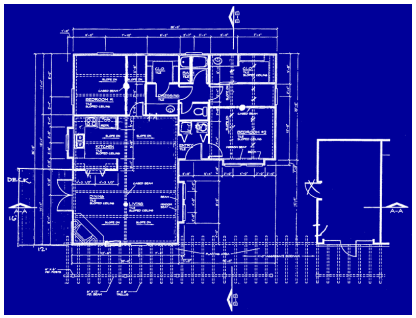
Classes and objects

- **class**: A template for a new type of objects.
- **object**: An entity that contains state and behavior.

iPod blueprint (class)

state:
current song
volume
battery life

behavior:
power on/off
change station/song
change volume
choose random song



constructs

iPod (object) #1

state:
song = "10000 Miles"
volume = 17
battery life = 2.5 hrs

behavior:
power on/off
change station/song
change volume
choose random song

iPod (object) #2

state:
song = "Letting You"
volume = 9
battery life = 3.41 hrs

behavior:
power on/off
change station/song
change volume
choose random song

iPod (object) #3

state:
song = "Discipline"
volume = 24
battery life = 1.8 hrs

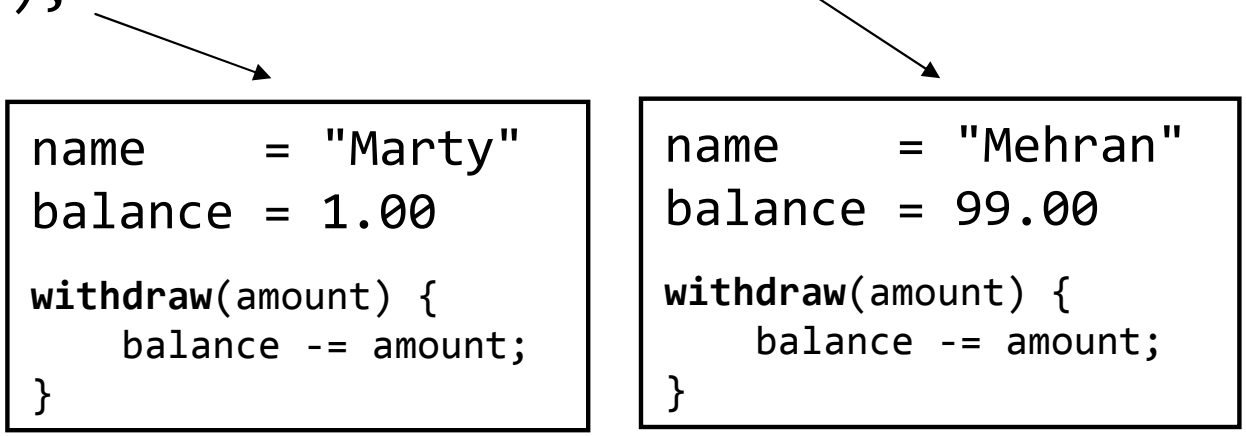
behavior:
power on/off
change station/song
change volume
choose random song

Class at a glance

```
// class - a template/factory for a new type of objects
public class BankAccount {
    // fields - data stored in each object
    private String name;
    private double balance;

    // constructor - initializes new objects
    public BankAccount(String nm, double bal) {
        name = nm;
        balance = bal;
    }

    // methods - behavior in each object
    public void withdraw(double amount) {
        if (amount > 0.00 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

Client code

```
// client - a program that uses your class of objects
public class WellsFargo extends ConsoleProgram {
    public void run() {
        BankAccount ba1 = new BankAccount("Marty",   1.00);
        BankAccount ba2 = new BankAccount("Mehran", 99.00);
        ba2.withdraw(10.00);
        ba1.withdraw(0.50);
        ...
    }
}
```



- When you call an object's method:
  - It executes that object's copy of the code from the class.
  - If that code refers to fields, it means that object's copy of those fields.

Using constructors

```
BankAccount ba1 =
    new BankAccount("Marty", 1.00);
```

ba1

```
name    = "Marty"
balance = 1.00

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

```
BankAccount ba2 =
    new BankAccount("Mehran", 9999.00);
```

ba2

```
name    = "Mehran"
balance = 9999.00

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

- When you call a **constructor** (with new):
  - Java creates a new object of that class.
  - The constructor runs, letting you set the fields of that new object.
  - The newly created object is returned to your program.

Variable names/scope

- Usually illegal to have 2 variables in same scope with same name:

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String nm) {
        name = nm;
    }
}
```

  - The parameter to `setName` is named `nm` to be distinct from the object's field `name` .

The keyword this

- **`this`** : A reference to the implicit parameter.
  - *implicit parameter:* object on which a method is called

- Syntax for using `this` :
  - To refer to a field:

    `this.`***`field`***
  - To call a method:

    `this.`***`method`***`(`***`parameters`***`);`
  - To call a constructor from another constructor:

    `this(`***`parameters`***`);`

Variable shadowing

- An instance method parameter name can match a field name:

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String name) {
        this.name = name;
    }
}
```

  - Field name is *shadowed* by the parameter with the same name.
  - Any code inside `setName` that refers to `name` will use the parameter, not the field. To refer to the field, say `this.name` .

Printing objects

- By default, Java doesn't know how to print objects.

```
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1);   // ba1 is BankAccount@9e8c34


// better, but cumbersome to write
println("ba1 is " + ba1.getName() + " with $"
        + ba1.getBalance());   // ba1 is Marty with $1.25


// desired behavior
println("b1 is " + ba1);   // ba1 is Marty with $1.25
```

The toString method

tells Java how to convert an object into a string

```
public String toString() {
    code that returns a String
    representing this object;
}
```

- Method name, return, and parameters must match exactly.

– Example:

```
// Returns a string representing this account.
public String toString() {
    return name + " has $" + balance;
}
```

Using toString

```
// client code
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1);

// the above code is really calling the following:
println("ba1 is " + ba1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - Default: class's name @ object's memory address (in base 16)

```
BankAccount@9e8c34
```

Array as field

```
private type[] name;        // declare
...
name = new type[length];    // initialize (in constructor)
```

– Example:

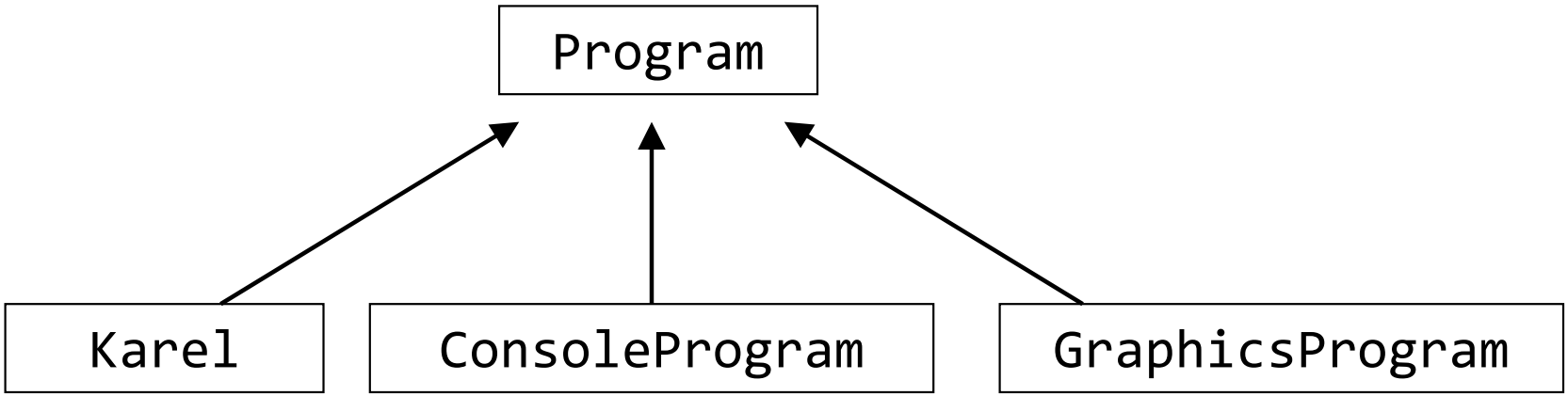
```
// Represents a hand in the card game Uno.
public class UnoHand {
    private String[] cards;

    public UnoHand() {
        cards = new String[7];
        ...
    }
}
```

Inheritance

Inheritance

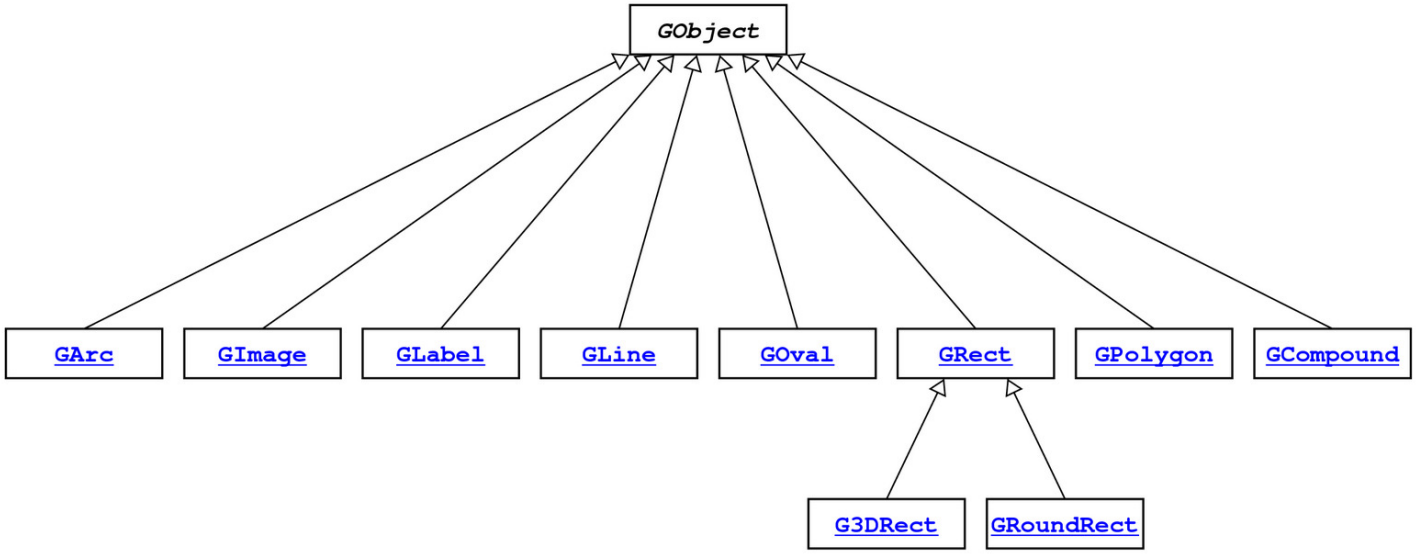
- **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes and share code between them

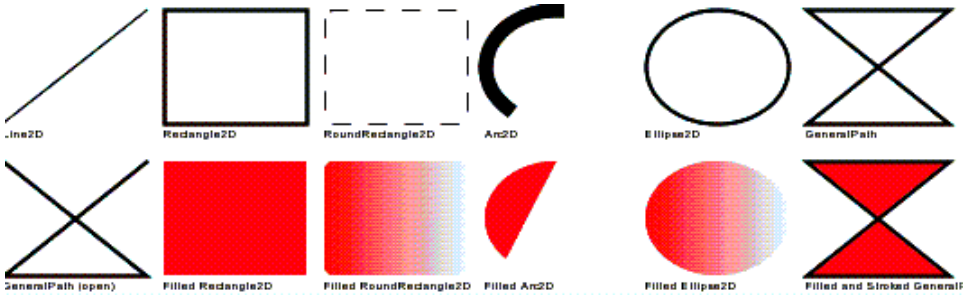
- One class can *extend* another, absorbing its data/behavior.
  - **superclass**: The parent class that is being extended.
  - **subclass**: Child class that extends superclass and inherits its behavior.
    - Subclass gets a copy of every field and method from superclass



GObject inh. hierarchy

- The Stanford library contains an inheritance hierarchy of graphical objects based on a common superclass named **`GObject`**.
  - `GArc`
  - `GCompound`
  - `GImage`
  - `GLabel`
  - `GLine`
  - `GOval`
  - `GPolygon`
  - `GRect`
  - `G3DRect`
  - `GRoundRect`





GObject members

- **GObject** defines the state and behavior common to all shapes:

```
contains(x, y)
getColor(), setColor(color)
getHeight(), getWidth(), getLocation(), setLocation(x, y)
getX(), getY(), setX(x), setY(y), move(dx, dy)
setVisible(visible), sendForward(), sendBackward()
toString()
```

- The subclasses add state and behavior unique to them:

GLabel

```
get/setFont
get/setLabel

...
```

GLine

```
get/setStartPoint
get/setEndPoint

...
```

GPolygon

```
addEdge
addVertex
get/setFillColor
..
```

Inheritance syntax

```
public class Name extends Superclass {
```

- Example:

```
public class CheckingAccount extends BankAccount {
    ...
}
```

- By extending BankAccount, each `CheckingAccount` object now:
  - has a name and balance field
  - has a `deposit`, `withdraw` method

Law firm employees

- Consider a law firm that employs lawyers, secretaries, legal secretaries, marketers, etc.

- The company has the following employee policies:
  - *hours:* Employees work **40 hours** / week.
  - *salary:* Employees make **$40,000 per year**, except legal secretaries **$45,000**; marketers **$50,000**.
  - *vacation:* Employees have **2 weeks** of paid vacation leave per year, except lawyers get **3 weeks**.
  - *forms:* Employees use a **yellow form** to apply for leave, except lawyers use a **pink form**.

- Also, 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**.

Employee class

```
// A class to represent employees in general.
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
    }
}
```

Secretary class

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

- Now we write only the parts that are unique to each type.
  - Inherit the other methods.
  - Add 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

- **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 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 String sue() {
        return "I'll see you in court!";
    }
}
```

- *Exercise:* Complete the `Marketer` class.
  - They 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 String advertise() {
        return "Act now while supplies last!";
    }

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

Multiple levels

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

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

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

Design for change

- 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.

The super keyword

- 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;
    }
    ...

public class Lawyer extends Employee {
    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }
    ...
```

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

```
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";
    }
}
```

A new error

- 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)

Long explanation

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

```
// implicit constructor (if you don't write one)
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.

Call superclass c'tor

```
super(parameters);
```

– Example:

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

– The `super` call must be the *first* statement in the constructor.

– Exercise: Modify the `Secretary` subclass.
  - Secretaries' years of employment are not tracked.
  - They do not earn extra vacation for years worked.

Modified Secretary

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

    public String takeDictation(String text) {
        return "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?

Accessors

- Add a "getter" method 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 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; it's not really true that they worked 0 years.
    - What if we need to know how many years the secretary worked?
    - What if we want to give a reward based on years of service?
    - Etc.

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

Improved Employee

```
// A class to represent employees in general.
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;
    }
    ...
}
```

- Separate the 10 vacation days from the seniority vacation bonus.
  - How does this help us improve the `Secretary`?

Improved Secretary

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

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

    ...
}
```

- `Secretary` can selectively override `getSeniorityBonus`; when `getVacationDays` runs, it will use the new version.
  - Choosing a method at runtime is called *dynamic binding*.

Overflow (extra) slides

Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.

```
public class BankAccount {
    private double balance;
    private String name;

    public BankAccount(String name) {
        this.name = name;
        balance = 0.00;
    }

    public BankAccount(String name, double bal) {
        this.name = name;
        balance = bal;
    }
    ...
}
```

Multiple constructors

- One constructor can call another using `this` :

```
public class BankAccount {
    private double balance;
    private String name;

    public BankAccount(String name) {
        this(name, 0.00); // call other constructor
    }

    public BankAccount(String name, double bal) {
        this.name = name;
        balance = bal;
    }
    ...
}
```

The class Object

- The class **`Object`** forms the root of the overall inheritance tree of all Java classes.
  - Every class is implicitly a descendent of `Object`.

- The `Object` class defines several methods that become part of every class you write.

  e.g. `public String` **`toString`**`()`