CSE 143
Lecture 6

Inheritance; binary search

reading: 9.1, 9.3 - 9.4; 13.1

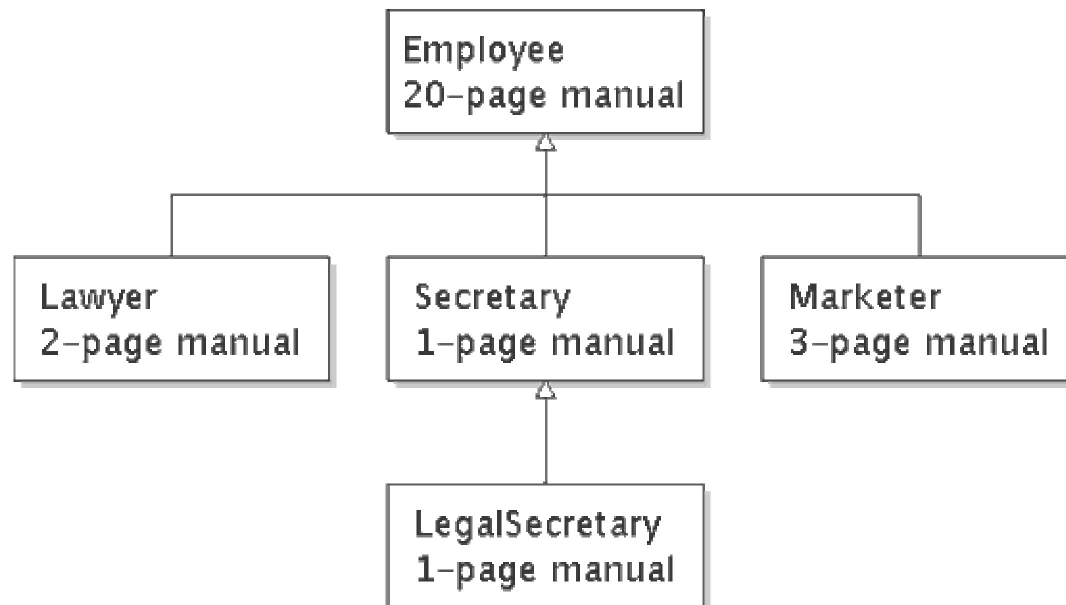
slides created by Marty Stepp
http://www.cs.washington.edu/143/

Inheritance basics

reading: 9.1, 9.3 - 9.4

Inheritance

- **inheritance**: Forming new classes based on existing ones.
  - a way to share/**reuse code** between two or more classes
  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
    - gets a copy of every field and method from superclass



Inheritance syntax

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

– Example:

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

- By extending `Employee`, each `Lawyer` object now:
  - receives a copy of each method from `Employee` automatically
  - can be treated as an `Employee` by client code
- Lawyer can also replace ("override") behavior from Employee.

The `super` keyword

- A subclass can call its parent's method/constructor:

```
super.method(parameters)      // method
super(parameters);            // constructor
```

  - Example:

```
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
    }

    // give Lawyers a $5K raise (better)
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;
    }
}
```

Exercise

- Write a class called `StutterIntList`.
  - Its constructor accepts an integer *stretch* parameter.
  - Every time an integer is added, the list will actually add *stretch* number of copies of that integer.

- Example usage:

```
StutterIntList list = new StutterIntList(3);
list.add(7);       // [7, 7, 7]
list.add(-1);      // [7, 7, 7, -1, -1, -1]
list.add(2, 5);    // [7, 7, 5, 5, 5, 7, -1, -1, -1]
list.remove(4);    // [7, 7, 5, 5, 7, -1, -1, -1]
System.out.println(list.getStretch());   // 3
```

Exercise solution

```
public class StutterIntList extends ArrayIntList {
    private int stretch;

    public StutterIntList(int stretchFactor) {
        super();
        stretch = stretchFactor;
    }

    public StutterIntList(int stretchFactor, int capacity) {
        super(capacity);
        stretch = stretchFactor;
    }

    public void add(int value) {
        for (int i = 1; i <= stretch; i++) {
            super.add(value);
        }
    }

    public void add(int index, int value) {
        for (int i = 1; i <= stretch; i++) {
            super.add(index, value);
        }
    }

    public int getStretch() {
        return stretch;
    }
}
```

Subclasses and fields

```
public class Employee {
    private double salary;
    ...
}

public class Lawyer extends Employee {
    ...
    public void giveRaise(double amount) {
        salary += amount;    // error; salary is private
    }
}
```

- Inherited private fields/methods cannot be directly accessed by subclasses. *(The subclass has the field, but it can't touch it.)*
  - How can we allow a subclass to access/modify these fields?

Protected fields/methods

```
protected type name;    // field

protected type name(type name, ..., type name) {
    statement(s);           // method
}
```

- a **protected field** or **method** can be seen/called only by:
  - the class itself, and its subclasses
  - also by other classes in the same "package" (discussed later)
  - useful for allowing selective access to inner class implementation

```
public class Employee {
    protected double salary;
    ...
}
```

  - Exercise: Add a method `count` to the `StutterIntList` that returns the number of occurrences of a given value.

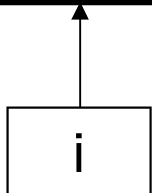
Binary Search

reading: 13.1

Sequential search

- **sequential search**: Locates a target value in an array / list by examining each element from start to finish.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



  - Notice that the array is sorted. Could we take advantage of this?

Binary search (13.1)

- **binary search**: Locates a target value in a *sorted* array / list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min mid max

Arrays.binarySearch

```
// searches an entire sorted array for a given value
// returns its index if found;  a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, value)

// searches given portion of a sorted array for a given value
// examines minIndex (inclusive) through maxIndex (exclusive)
// returns its index if found;  a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, minIndex, maxIndex, value)
```

- The `binarySearch` method in the `Arrays` class searches an array very efficiently if the array is sorted.
  - You can search the entire array, or just a range of indexes (useful for "unfilled" arrays such as the one in `ArrayIntList`)

Using binarySearch

```
// index     0  1  2  3   4   5   6   7   8   9  10  11  12  13  14  15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};

int index  = Arrays.binarySearch(a, 0, 16, 42);   // index1 is 10
int index2 = Arrays.binarySearch(a, 0, 16, 21);   // index2 is -7
```

- `binarySearch` returns the index where the value is found
- if the value is *not* found, `binarySearch` returns:

  `-(insertionPoint + 1)`

  - where `insertionPoint` is the index where the element *would* have been, if it had been in the array in sorted order.
  - To insert the value into the array, negate `insertionPoint` + 1

  ```
  int indexToInsert21 = -(index2 + 1);  // 6
  ```