CSE 143
Lecture 6

More ArrayIntList; Inheritance

reading: 15.1 - 15.2; 9.1, 9.3 - 9.4

slides created by Marty Stepp
http://www.cs.washington.edu/143/
Let's add the following features to ArrayIntList:

- a constant for the default list capacity
- better encapsulation and protection of implementation details
- a better way to print list objects
Class constants

public static final type name = value;

• **class constant**: a global, unchangeable value in a class
  – used to store and give names to important values used in code
  – documents an important value; easier to find and change later

• classes will often store constants related to that type
  – Math.PI
  – Integer.MAX_VALUE, Integer.MIN_VALUE
  – Color.GREEN

// default array length for new ArrayIntLists
public static final int DEFAULT_CAPACITY = 10;
"Helper" methods

• Currently our list class has a few useful "helper" methods:
  
  - public void checkResize()
  - public void checkIndex(int index, int min, int max)

• We wrote them to help us implement other required methods.

• We don't want clients to call these methods; they are internal.
  – How can we stop clients from calling them?
A private method

private type name(type name, ..., type name) {
    statement(s);
}

• a private method can be seen/called only by its own class
  – encapsulated, similar to fields
  – your object can call the method on itself, but clients cannot call it
  – useful for "helper" methods that clients shouldn't directly touch

private void checkIndex(int index, int min, int max) {
    if (index < min || index > max) {
        throw new IndexOutOfBoundsException(index);
    }
}
• Currently our list class has a `print` method:

```java
// client code
ArrayIntList list = new ArrayIntList();
...
list.print();
```

– Why is this a bad idea? What would be better?
The `toString` method

- Tells Java how to convert an object into a `String`
  
  ```java
  ArrayIntList list = new ArrayIntList();
  System.out.println("list is " + list);
  ```

- Syntax:
  
  ```java
  public String toString() {
      code that returns a suitable String;
  }
  ```

- Every class has a `toString`, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) number:
    ```java
    ArrayIntList@9e8c34
    ```
// Returns a String representation of the list.
public String toString() {
    if (size == 0) {
        return "[]";
    } else {
        String result = "[" + elementData[0];
        for (int i = 1; i < size; i++) {
            result += ", " + elementData[i];
        }
        result += "]";
        return result;
    }
}
Exercise

• Write a class called StutterIntList.
  – Its constructor accepts an integer \textit{stretch} parameter.
  – Every time an integer is added, the list will actually add \textit{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
```
• **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
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
    }
}

• Lawyers, Secretaries, etc. have similar behavior to the above.
  • How to implement those classes without redundancy?
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
Overriding methods

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

```java
public class Lawyer extends Employee {
    // overrides getSalary method in Employee class;
    // give Lawyers a $5K raise
    public double getSalary() {
        return 45000.00;
    }
}
```
• Subclasses can call overridden methods with `super`

    `super.method(parameters)`

  - Example:

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

  - This version makes sure that Lawyers always make $5K more than Employees, even if the Employee's salary changes.
Calling super constructor

```
super(parameters);
```

- Example:

```java
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name); // calls Employee constructor
    }
    ...
}
```

- `super` allows a subclass constructor to call a superclass one.
- The `super` call must be the first statement in the constructor.
- Constructors are not inherited; If you extend a class, you must write all the constructors you want your subclass to have.
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

```java
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 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;
    ...
}