References to text and JavaSummary.pptx

- Local variable: variable declared in a method body  
  B.10–B.11 slide 45
- Inside-out rule, bottom-up/overriding rule C.15 slide 31-32 and consequences thereof slide 45
- Use of this B.10 slide 23-24 and super C.15 slide 28, 33
- Constructors in a subclass C.9–C.10 slide 24-29
- First statement of a constructor body must be a call on another constructor —if not Java puts in super(); C.10 slide 29
Homework

Visit course website, click on Resources and then on Code Style Guidelines. Study

4.2 Keep methods short
4.3 Use statement-comments …
4.4 Use returns to simplify method structure
4.6 Declare local variables close to first use …
/** Return middle value of b, c, d (no ordering assumed) */

public static int middle(int b, int c, int d) {
    if (b > c) {
        int temp = b;
        b = c;
        c = temp;
    }
    // { b <= c }
    if (d <= b) {
        return b;
    }
    // { b < d and b <= c }
    return Math.min(c, d);
}

Local variable: variable declared in method body

Parameter: variable declared in () of method header

All parameters and local variables are created when a call is executed, before the method body is executed. They are destroyed when method body terminates.
/** Return middle value of b, c, d (no ordering assumed) */

public static int middle(int b, int c, int d) {
    if (b > c) {
        int temp = b;
        b = c;
        c = temp;
    }

    // { b <= c }

    if (d <= b) {
        return b;
    }

    // { b < d and b <= c }

    return Math.min(c, d);
}
/** Return middle value of b, c, d (no ordering assumed) */

public static int middle(int b, int c, int d) {
    int temp;
    if (b > c) {
        temp = b;
        b = c;
        c = temp;
    }
    // { b <= c }
    if (d <= b) {
        return b;
    }
    // { b < d and b <= c }
    return Math.min(c, d);
}

Not good! No need for reader to know about temp except when reading the then-part of the if-statement

Principle: Declare a local variable as close to its first use as possible.
/** Return middle value of b, c, d (no ordering assumed) */
public static int middle(int b, int c, int d) {
    if (b > c) {
        int temp = b;
        b = c;
        c = temp;
    }
    // { b <= c }
    if (d <= b) {
        return b;
    }
    // { b < d and b <= c }
    return Math.min(c, d);
}
Which method `toString()` is called by `c.toString()`?

**Overriding rule or bottom-up rule:**
To find out which is used, start at the bottom of the object and search upward until a matching one is found.
**Inside-out rule**: Code in a construct can reference any names declared in that construct, as well as names that appear in enclosing constructs. (If name is declared twice, the closer one prevails.)

Person's objects and static components

```
Person@a0

n

getNAndPop() {
    return n + PersonPop;
}

Person@a1

n

getNAndPop() {
    return n + PersonPop;
}
```
Parameters participate in inside-out rule

Parameter n “blocks” reference to field n. (n is a “shadowed” variable)
A solution: use **this**

Memorize: Within an object, **this** evaluates to the name of the object.

In object Person@a0, **this** evaluates to Person@a0

In object Person@a1, **this** evaluates to Person@a1

Person@a0.n is this variable
Within a subclass object, \texttt{super} refers to the partition above the one that contains \texttt{super}.

Because of the keyword \texttt{super}, this calls \texttt{toString} in the Object partition.
public class Time {
    private int hr;  //hour of day, 0..23
    private int min; // minute of hour, 0..59

    /** Constructor: instance with h hours and m minutes */
    public Time(int h, int m) { …}

    /** Constructor: instance with m minutes … */
    public Time(int m) {
        hr = m / 60;
        min = m % 60;
    }

    … Want to change body to call first constructor
}
public class Time
    private int hr;  //hour of day, 0..23
    private int min; // minute of hour, 0..59

    /** Constructor: instance with h hours and m minutes … */
    public Time(int h, int m) {
        this(m / 60, m % 60);
    }

    /** Constructor: instance with m minutes … */
    public Time(int m) {
        this(m / 60, m % 60);
    }

Use this (Instead of Time) to call another constructor in the class.
Must be first statement in constructor body!
Initialize superclass fields first

Class Employee contains info that is common to all employees — name, start date, salary, etc. `getCompensation` gives the salary.

Executives also get a bonus. `getCompensation` is overridden to take this into account.

Could have other subclasses for part-timers, temporary workers, consultants, etc., each with a different `getCompensation`.

---

**Executive@a0**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td><code>toString()</code> …</td>
</tr>
<tr>
<td>Employee</td>
<td><code>toString()</code>, <code>getCompensation()</code></td>
</tr>
<tr>
<td>Executive</td>
<td><code>toString()</code>, <code>getCompensation()</code>, <code>getBonus()</code></td>
</tr>
</tbody>
</table>

- **Employee(String, int)**
  - **ToString:**
    - `toString()`
  - **getCompensation:**
    - `getCompensation()`
  - **Salary:** 50,000
  - **Name:** “G”
  - **Start Date:** 1969

- **Executive**
  - **Salary:** 50,000
  - **Bonus:** 10,000

Without OO …

Without OO, you would write a long involved method:

```java
public double getCompensation(...) {
    if (worker is an executive)
        { ... }
    else if (worker is part time)
        { ... }
    else if (worker is temporary)
        { ... }
    else ...
```

OO eliminates need for many of these long, convoluted methods, which are hard to maintain.

Instead, each subclass has its own `getCompensation`.

End up with many more methods, which are usually very short.
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s) {
    name= n;
    start= d;
    salary= s;
}

Principle: initialize superclass fields first

Employee(String, int, double)
**Principle: initialize superclass fields first**

```java
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s)

/** Constructor: executive with name n, year hired d, salary of $50,000, bonus b */
public Executive(String n, int d, double b)
```

**Principle:** In subclass constructor, fill in the superclass fields first

How to do that if they are private?

**Call constructor in superclass**

![Diagram: Employee and Executive constructors with fields and arrows indicating calling the constructor in superclass.](Diagram.png)
** Principle: initialize superclass fields first **

```java
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s)

/** Constructor: executive with name n, year hired d, salary of $50,000, bonus b */
public Executive(String n, int d, double b) {
    super Employee(n, d, 50000);
    bonus= b;
}
```

To call a superclass constructor, use `super( ... )`

---

Diagram:

- Executive
  - salary
  - name
  - start
  - bonus
- Employee
  - salary
  - name
  - start
- Executive (String, int, double)
/** Constructor: an instance with ... */
public C (…) {
    super();
    S0;
    S1;
    ...
}

Java syntax: First statement of any constructor you write must be a call on another constructor
this( ... ); or super( ... );

If you don’t put one in, Java silently inserts this one:

super();