Announcements

1. Writing tests to check that the code works when the precondition is satisfied is not optional.
2. Writing assertions to verify the precondition is satisfied is not optional, and if you do so incorrectly you will lose points.
3. Writing tests to verify that you have done (2) correctly is optional. Piazza note @129 tells you how.
4. Watch the loop invariant tutorials before next week’s recitation. They are linked from the Lecture Notes page.

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-commnets …
- 4.4 Use returns to simplify method structure
- 4.6 Declare local variables close to first use …

Local variables

/** Return middle value of a, b, c (no ordering assumed) */
public static int middle(int a, int b, int c) {
    if (b > c) {
        int temp= b;
        b= c;
        c= temp;
    }
    if (a <= b) {
        return b;
    }
    return Math.min(a, c);
}

Scope of local variables

/** Return middle value of a, b, c (no ordering assumed) */
public static int middle(int a, int b, int c) {
    if (b > c) {
        int temp= b;
        b= c;
        c= temp;
    }
    if (a <= b) {
        return b;
    }
    return Math.min(a, c);
}
Scope In General: Inside-out rule

**Inside-out rule**: Code in a construct can reference names declared in that construct, as well as names that appear in enclosing constructs. (If name is declared twice, the closer one prevails.)

```java
/** A useless class to illustrate scopes*/
public class Class{
    private int field;
    public void method(int parameter) {
        if (field > parameter) {
            int temp = parameter;
        }
    }
}
```

Principle: declaration placement

```java
/** Return middle value of a, b, c (no ordering assumed) */
public static int middle(int a, int b, int c) {
    int temp;
    if (b > c) {
        temp = b;
        b = c;
        c = temp;
    } // b <= c
    if (a <= b) {
        return b;
    } // a and c are both greater than b
    return Math.min(a, c);
}
```

Poll time! What 3 numbers are printed?

```java
public class ScopeQuiz {
    private int a;
    public ScopeQuiz(int b) {
        System.out.println(a);
        int a = b + 1;
        this.a = a;
        System.out.println(a);
        a = a + 1;
    }
    public static void main(String[] args) {
        int a = 5;
        ScopeQuiz s = new ScopeQuiz(a);
        System.out.println(s.a);
    }
}
```

Calling a constructor from a constructor

```java
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) { hr = h; min = m; assert ...; }
    /** Constructor: instance with m minutes ... */
    public Time(int m) { hr = m / 60; min = m % 60; }
    ...
}
```

Assertions promote understanding

```java
/** Return middle value of a, b, c (no ordering assumed) */
public static int middle(int a, int b, int c) {
    if (b > c) {
        int temp = b;
        b = c;
        c = temp;
    } // b <= c
    if (a <= b) {
        return b;
    } // a and c are both greater than b
    return Math.min(a, c);
}
```

Poll time! What 3 numbers are printed?

```java
public class ScopeQuiz {
    private int a;
    public ScopeQuiz(int b) {
        System.out.println(a);
        int a = b + 1;
        this.a = a;
        System.out.println(a);
        a = a + 1;
    }
    public static void main(String[] args) {
        int a = 5;
        ScopeQuiz s = new ScopeQuiz(a);
        System.out.println(s.a);
    }
}
```

Calling a constructor from a constructor

```java
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) { hr = h; min = m; assert ...; }
    /** Constructor: instance with m minutes ... */
    public Time(int m) { hr = m / 60; min = m % 60; }
    ...
}
```
Calling a constructor from a 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) { hr = h; min = m; assert ...; }
/** Constructor: instance with m minutes ... */
public Time(int m) {
this(m / 60, m % 60);
} 
Use this (not Time) to call another constructor in the class. Must be first statement in constructor body!

About super

Within a subclass object, super refers to the partition above the one that contains super.

Because of keyword super, the call toString here refers to the Person partition.

Without OO ...

Without OO, you would write a long involved method:

public double getName(Person p) {
  if (p is a PhD)
    { ... } 
  else if (p is a GradStudent)
    { ... }
  else if (p prefers anonymity)
    { ... }
  else ...
}

OO eliminates need for many of these long, convoluted methods, which are hard to maintain. Instead, each subclass has its own getName. Results in many overriding method implementations, each of which is usually very short

Constructing with a Superclass

/** Constructor: person “f n” */
public Person(String f, String l) {
  first= n;
  last= l;
}
/** Constructor: PhD “Dr. f m. l” */
public PhD(String f, char m, String l) {
  super(f, l);
  middle= m;
} 
new PhD(“David”, ’J’, “Gries”);

PhD@a0

Use super (not Person) to call superclass constructor.

Must be first statement in constructor body!

Bottom-Up and Inside-Out

PhD@a0

OO eliminates need for many of these long, convoluted methods, which are hard to maintain. Instead, each subclass has its own getName. Results in many overriding method implementations, each of which is usually very short