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. Look at JavaHyperText entry for JUnit testing, to see how to test whether an assert statement is correct.

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 …

Assignment 1

Due on September 6 (tomorrow!).

Form a group before submitting (or lose points). One partner has to invite the other on CMS, and the other has to accept.

Finish early!

References to JavaHyperText

- local variable
- scope
- this
- shadowing a variable
- inside-out rule
- super
- constructor, constructor call; constructor, default; constructor call, default

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**

```java
/** Return middle value of a, b, c (no ordering assumed) */
public static int middle(int a, int b, int c) {
    int temp; // A useless class to illustrate scopes
    if (b > c) {
        temp = b;
        b = c;
        c = temp;
    }
    if (a <= b) {
        return b;
    }
    return Math.min(a, c);
}
```

**Scope In General: Inside-out rule**

```java
/** A useless class to illustrate scopes*/
public class C {
    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; // Declare a local variable as close to its first use as possible.
    if (b > c) {
        temp = b;
        b = c;
        c = temp;
    }
    if (a <= b) {
        return b;
    }
    return Math.min(a, c);
}
```

**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
import java.util.Random;

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);
    }
}
```

**Bottom-up/overriding rule**

```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);
    }
}
```

---

**Scope of local variables**

- **Scope of local variable** (where it can be used): from its declaration to the end of the block in which it is declared.

**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.

**Principle: declaration placement**

- **Declare a local variable as close to its first use as possible.**

**Assertions promote understanding**

- **Assertion**: Asserting that b <= c at this point. Helps reader understand code below.

**Poll time! What 3 numbers are printed?**

- **Original output**: A: 5, 6, 6  B: 0, 6, 6  C: 6, 6, 6  D: 0, 6, 0

**Bottom-up/overriding rule**

- **Which method toString() is called by turing.toString()**?
  - The **overriding rule**, a.k.a. the **bottom-up rule**: To find out which method is used, start at the bottom of the object and search upward until a matching one is found.
Calling a constructor from a constructor

```java
public class Person {
    private String firstName;
    private String lastName; // minute of hour, 0..59
    /** Create a person with the given names. */
    public Person(String f, String l) {
        assert ...;
        firstName = f; lastName = l;
    }
    /** Create a person with the given full name. */
    public Person(String fullName) {
        firstName = ...; lastName = ...;
    }
}
```

Want to change body to call first constructor

Calling a constructor from a constructor

```java
public class Person {
    private String firstName;
    private String lastName; // minute of hour, 0..59
    /** Create a person with the given names. */
    public Person(String f, String l) {
        assert ...;
        firstName = f; lastName = l;
    }
    /** Create a person with the given full name. */
    public Person(String fullName) {
        firstName = ...; lastName = ...
    }
}
```

Use this (not Person) to call another constructor in the class.

Must be first statement in constructor body!

Constructing with a Superclass

```java
/** Constructor: person “f n” */
public Person(String f, String l) {
    first= n;
    last= l;
}
/** Constructor: PhD with a year. */
public PhD(String f, String l, int y) {
    super(f, l);
    gradYear= y;
}
new PhD("David", "Gries", 1966);
```

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:

```java
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.