Local variables

```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;
    }
    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;
    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;
        } else {
            method block
        }
    }
}
```

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;
    }
    if (a <= b) {
        return b;
    }
    return Math.min(a, c);
}
```

Announcements

1. A1 is due today
   - If you are working with a partner, you must form a group on CMS and submit one solution!
2. A2 is out. Remember to get started early!
3. Next week’s recitation is on testing. No tutorial/quiz this week!
**Bottom-up/overriding rule**

Which method `toString()` is called by

```
  Person turing = new Person("Turing", "Smith");
  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.

---

**About `super`**

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

```
PhD davidGries = new PhD("David", "Gries", 1966);
```

Because of keyword `super`, the call to `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 l” */
public Person(String f, String l) {
  first= f;
  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);
```

Must be first statement in constructor body!

---

**Bottom-Up and Inside-Out**

```
PhD@o0 toString()
```

```
first "David" last "Gries"
```

```
gradYear 1966
```

```
Object
```

```
Person
```

```
toString()
```

```
getName
```

```
null
```

```
"David"
```

```
"Gries"
```

```
0
```

```
null
```

```
** Constructor: person “f l” */
public Person(String f, String l) {
  first= f;
  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);
```

---

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.