

CS/ENGRD 2110

SPRING 2017

Lecture 5: Local vars; Inside-out rule; constructors
<http://courses.cs.cornell.edu/cs2110>

Announcements

2

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

3

- 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

4

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 ...

Local variables

middle(8, 6, 7)

5

/ 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;  
}
```

Local variable:
variable
declared in
method body

Parameter: variable
declared in () of
method header

a	8	b	6	c	7
		temp	?		

```
if (a <= b) {  
    return b;  
}
```

```
return Math.min(a, c);  
}
```

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.


Scope of local variables

6

/ 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 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

7

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.)

*/** A useless class to illustrate scopes*/*

```
public class Class{  
    private int field;  
    public void method(int parameter) {  
        if (field > parameter) {  
            int temp= parameter; block  
        }  
    }  
}
```

The diagram illustrates the nested scope structure of the provided code. It uses three levels of nested brackets to represent the scope boundaries:

- class**: The outermost level, spanning from the opening curly brace of `public class Class{` to the closing curly brace of `}` at the bottom.
- method**: The middle level, spanning from the opening curly brace of `public void method(int parameter) {` to the closing curly brace of `}` above the `}` at the bottom.
- block**: The innermost level, spanning from the opening curly brace of `if (field > parameter) {` to the closing curly brace of `}` above the `}` of the method.

Each level is labeled with its corresponding scope name (`class`, `method`, or `block`) in red text to the right of the closing brace that defines its extent.

Principle: declaration placement

8

```
/** 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);
```

```
}
```

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.

Assertions promote understanding

9

```
/** 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);
```

```
}
```

write assert to check our assumptions

Assertion: Asserting that $b \leq c$ at this point. Helps reader understand code below.

Poll time! What 3 numbers are printed?

10



```
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;    here a is the local variable  
    }
```

0, 6, 6

```
    public static void main(String[] args) {  
        int a = 5;  
        ScopeQuiz s = new ScopeQuiz(a);  
        System.out.println(s.a);  
    }  
}
```

Bottom-up/overriding rule

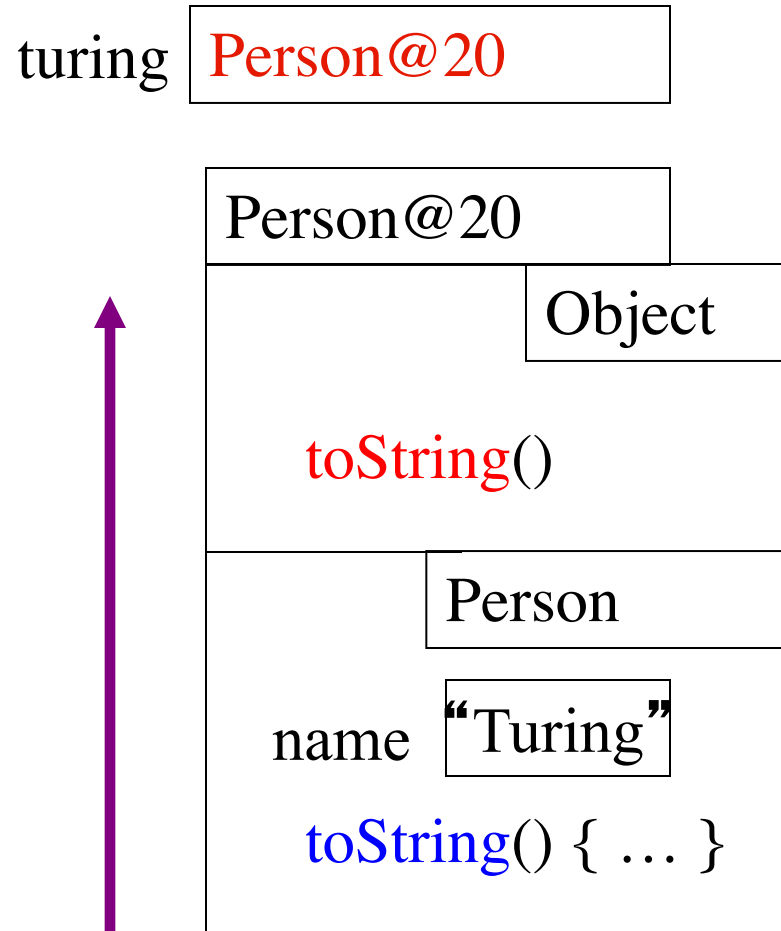
11

Which method `toString()` is called by

`turing.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.



Calling a constructor from a constructor

12

```
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;
    }
    ...
}
```

Want to change body
to call first constructor

Calling a constructor from a constructor

13

```
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!**

Constructing with a Superclass

14

/ Constructor: person “f n” */**

```
public Person(String f, String l) {  
    first= n;  
    last= l;  
}
```

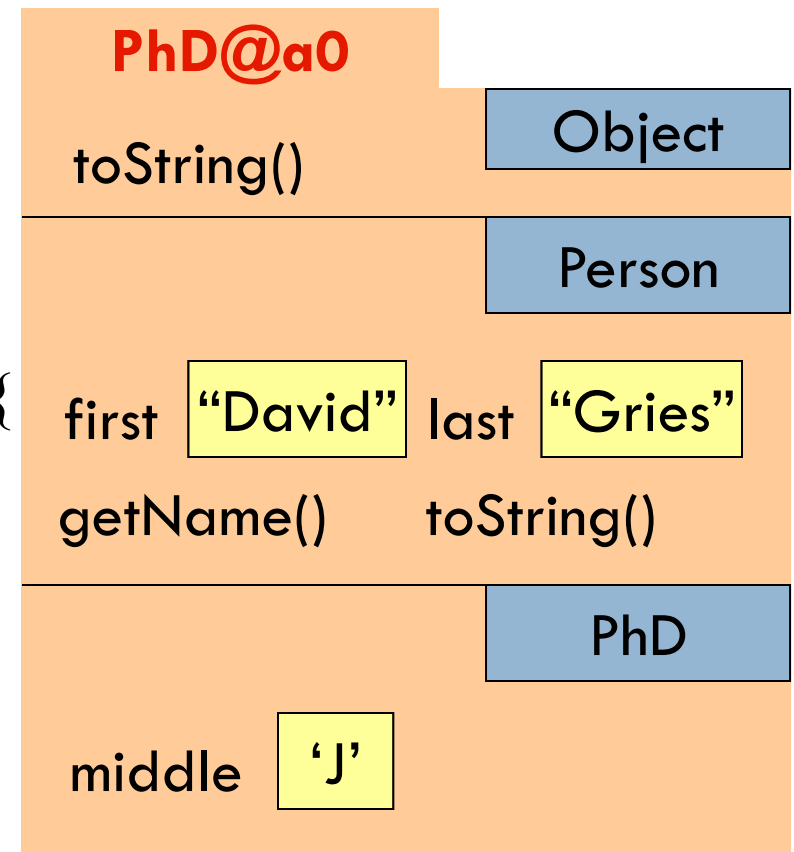
Use **super** (not **Person**) to call superclass constructor.

/ Constructor: PhD “Dr. f m. l”*/**

```
public PhD(String f, char m, String l) {  
    super(f, l);  
    middle= m;  
}
```

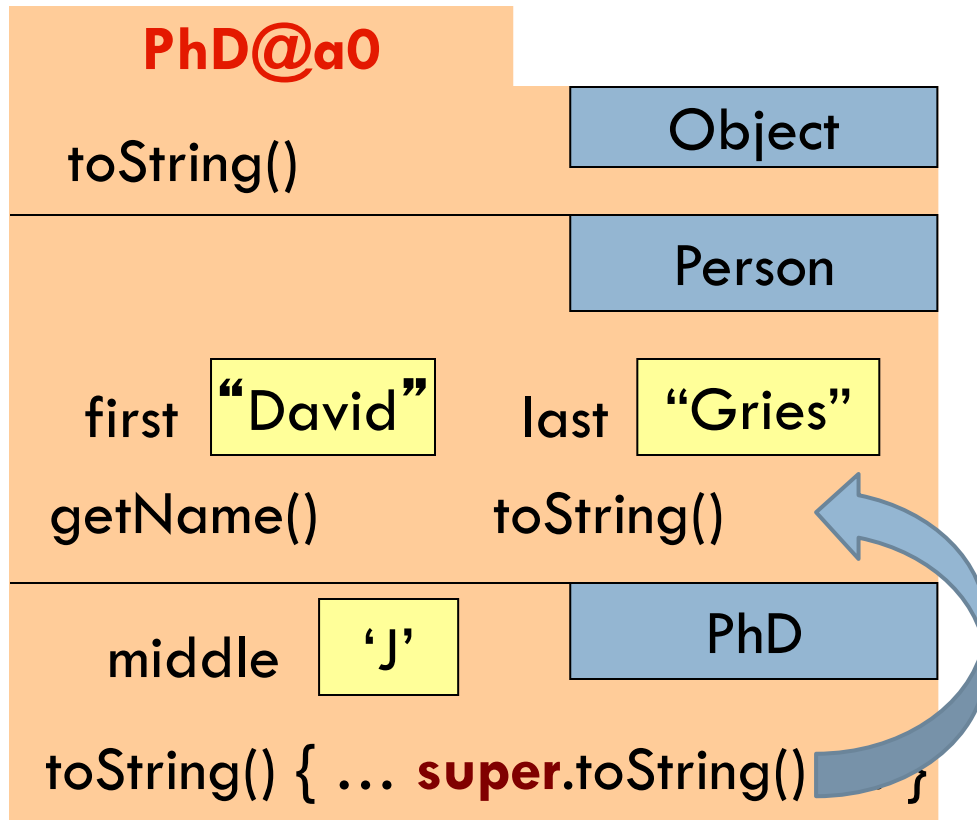
Must be **first statement** in constructor body!

```
new PhD(“David”, ‘J’, “Gries”);
```



About **super**

15

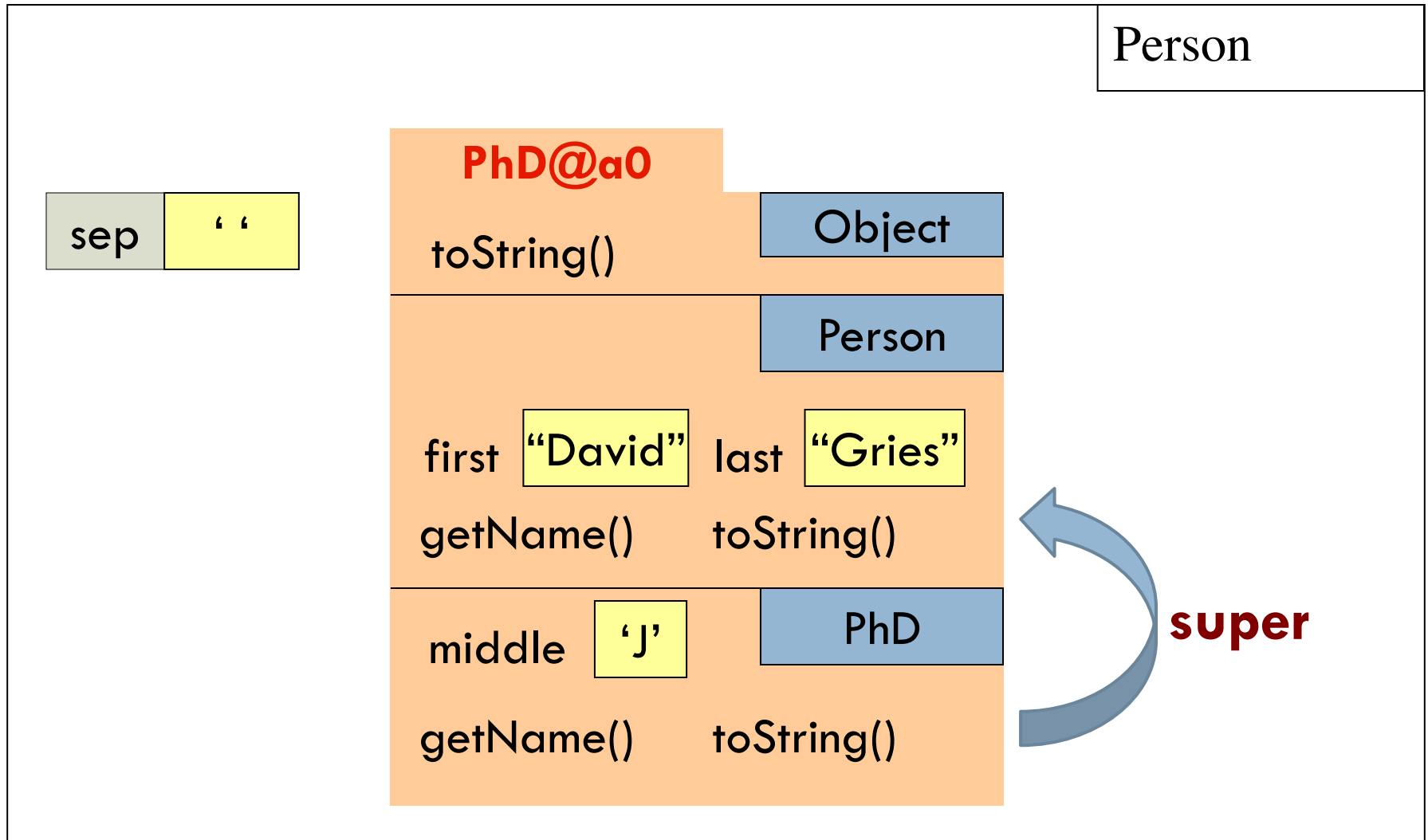


Within a subclass object, **super** refers to the partition above the one that contains **super**.

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

Bottom-Up and Inside-Out

16



Without OO ...

17

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