CS/ENGRD 2110 SPRING 2018

Lecture 5: Local vars; Inside-out rule; constructors

http://courses.cs.cornell.edu/cs2110

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!

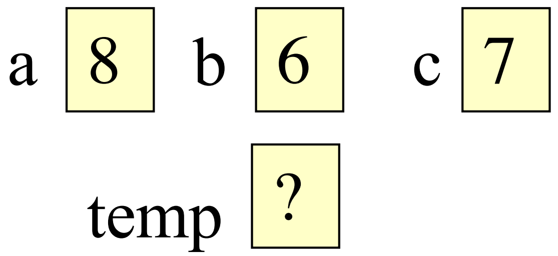
Local variables

middle(8, 6, 7)

```
/** 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);
}
```

Parameter: variable declared in () of method header

Local variable: variable declared in method body



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

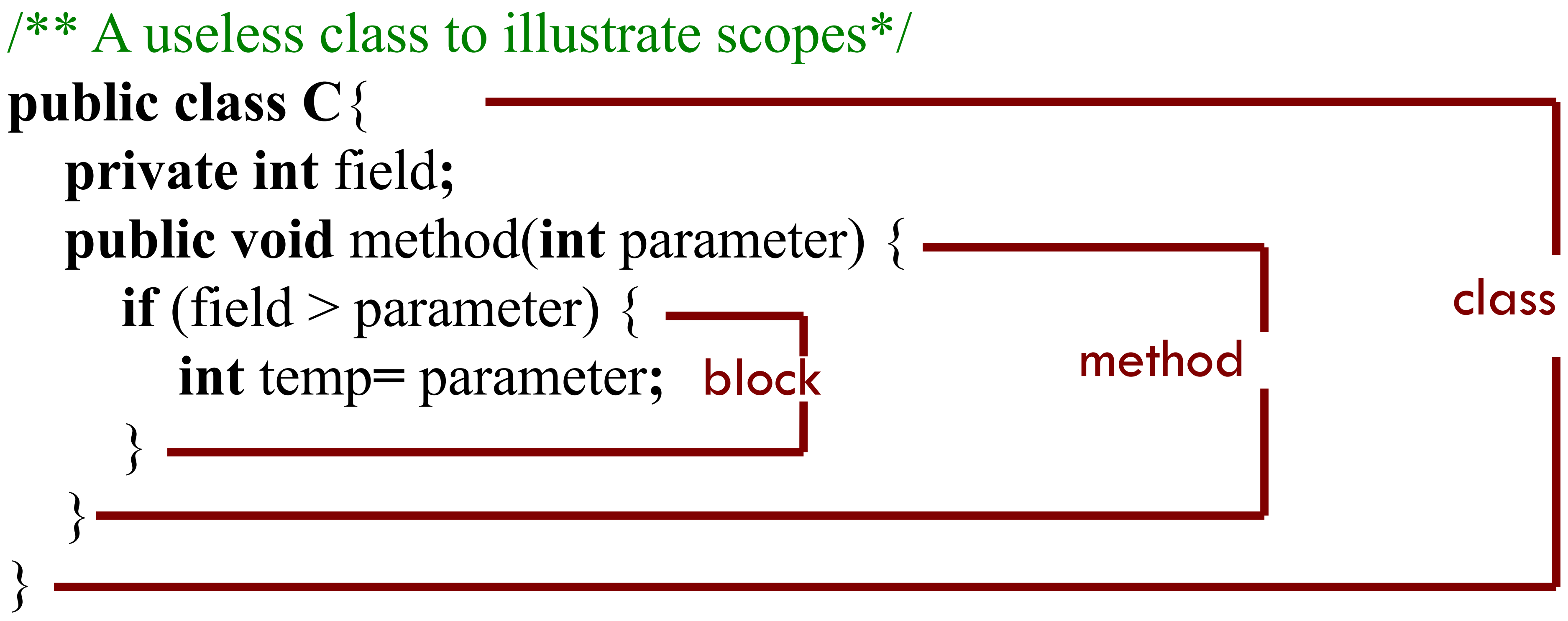
```
/** Return middle value of a, b, c (no ordering assumed) */
public static int middle(int a, int b, int c) {
    if (b > c) {
        int temp= b;
        b= c;
        c= temp;
    }

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

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

block

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.)



```
/** 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

```
/** 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

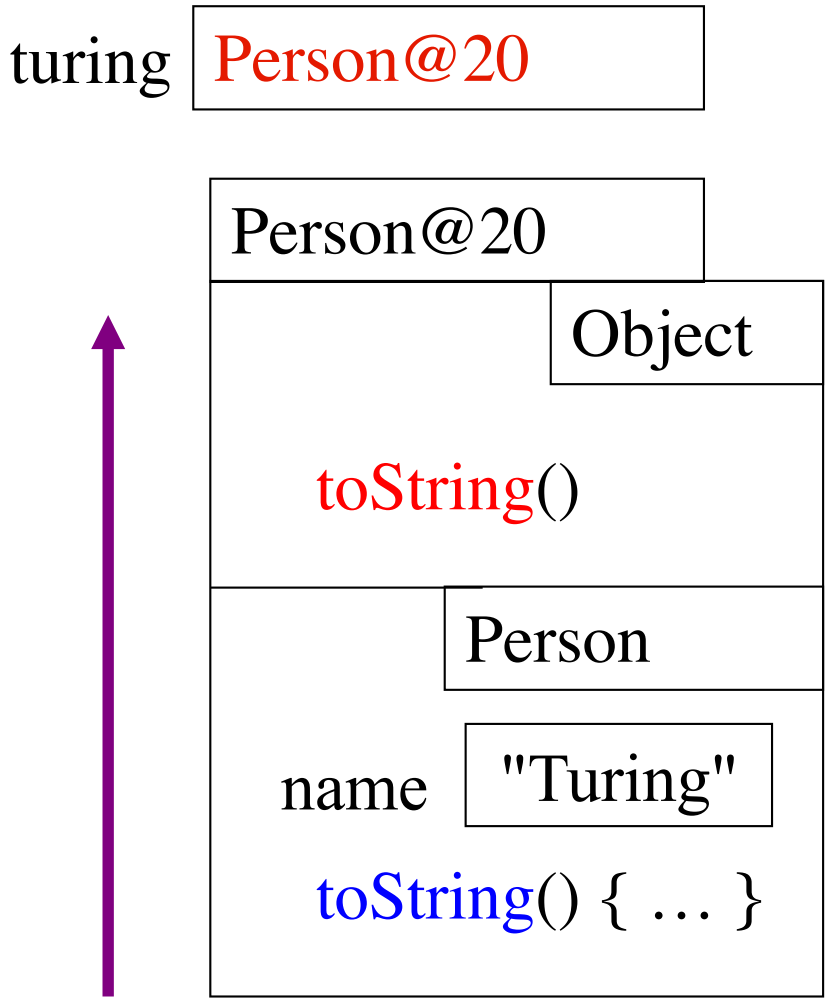
Principle: Declare a local variable as close to its first use as possible.

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.

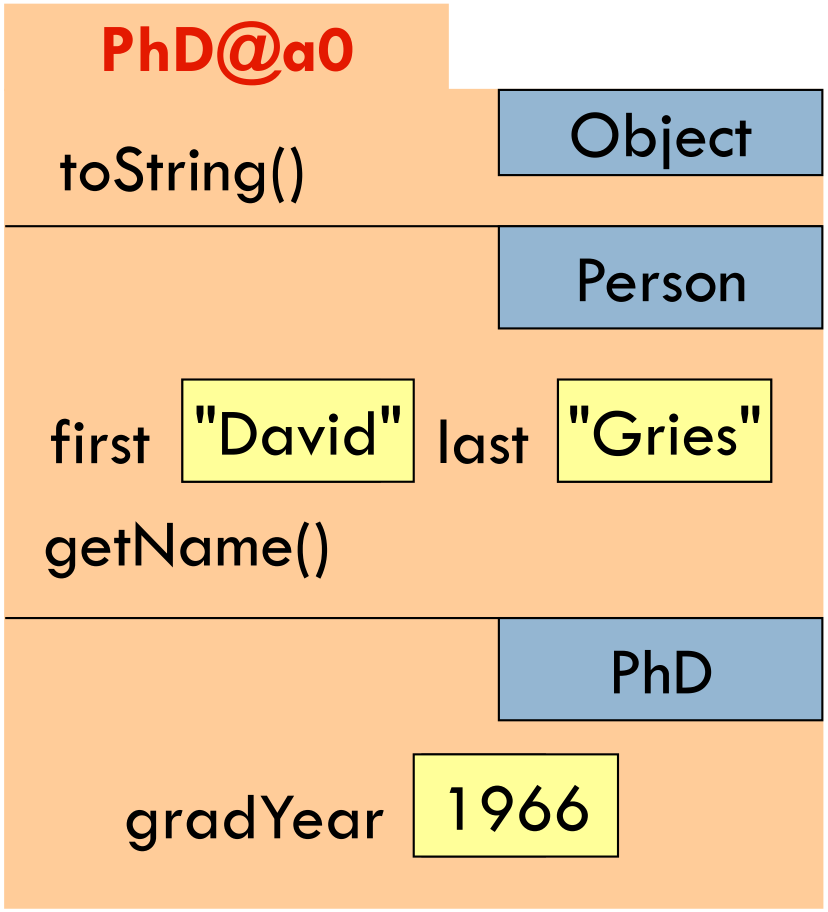


Constructing with a Superclass

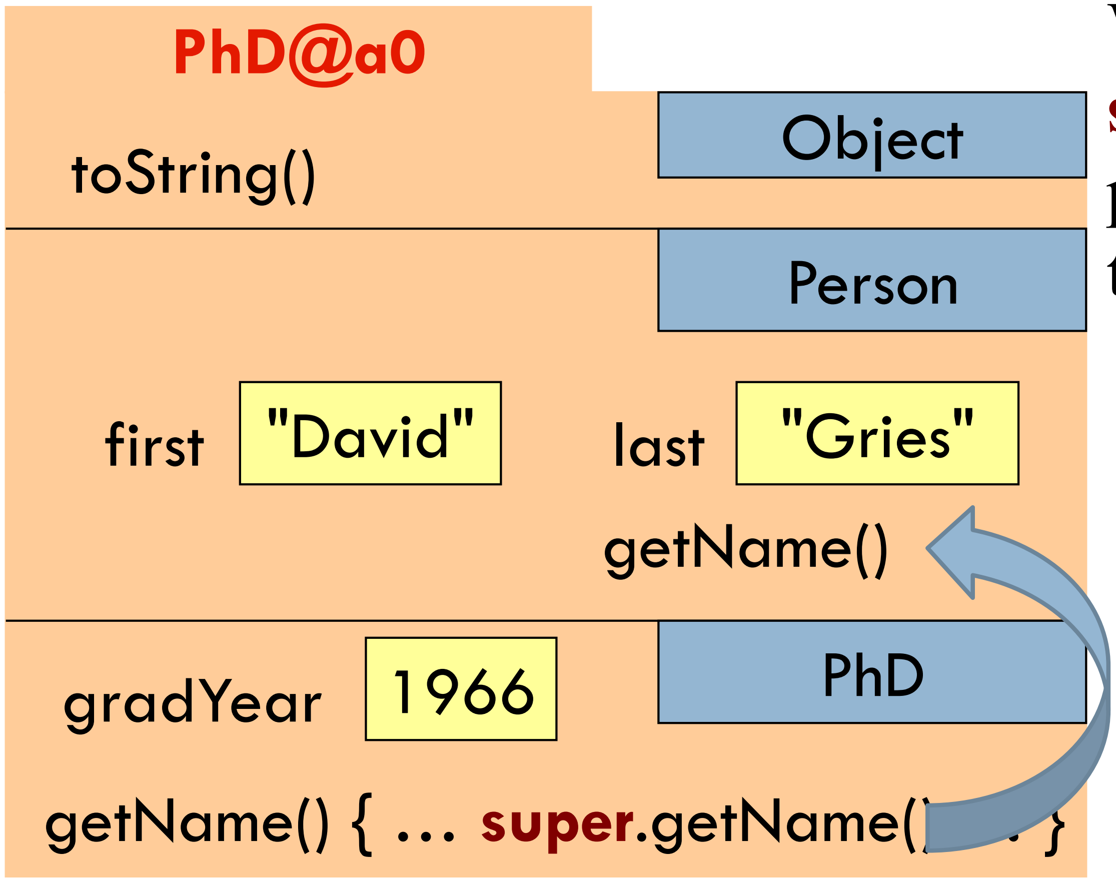
```
/** Constructor: person “f n” */
public Person(String f, String l) {
    first= n;
    last= l;
}

/** Constructor: PhD with a year. */
public PhD(String f, String l, int y) {
    super(f, l);
    gradYear= y;
}

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

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

Must be first statement in constructor body!



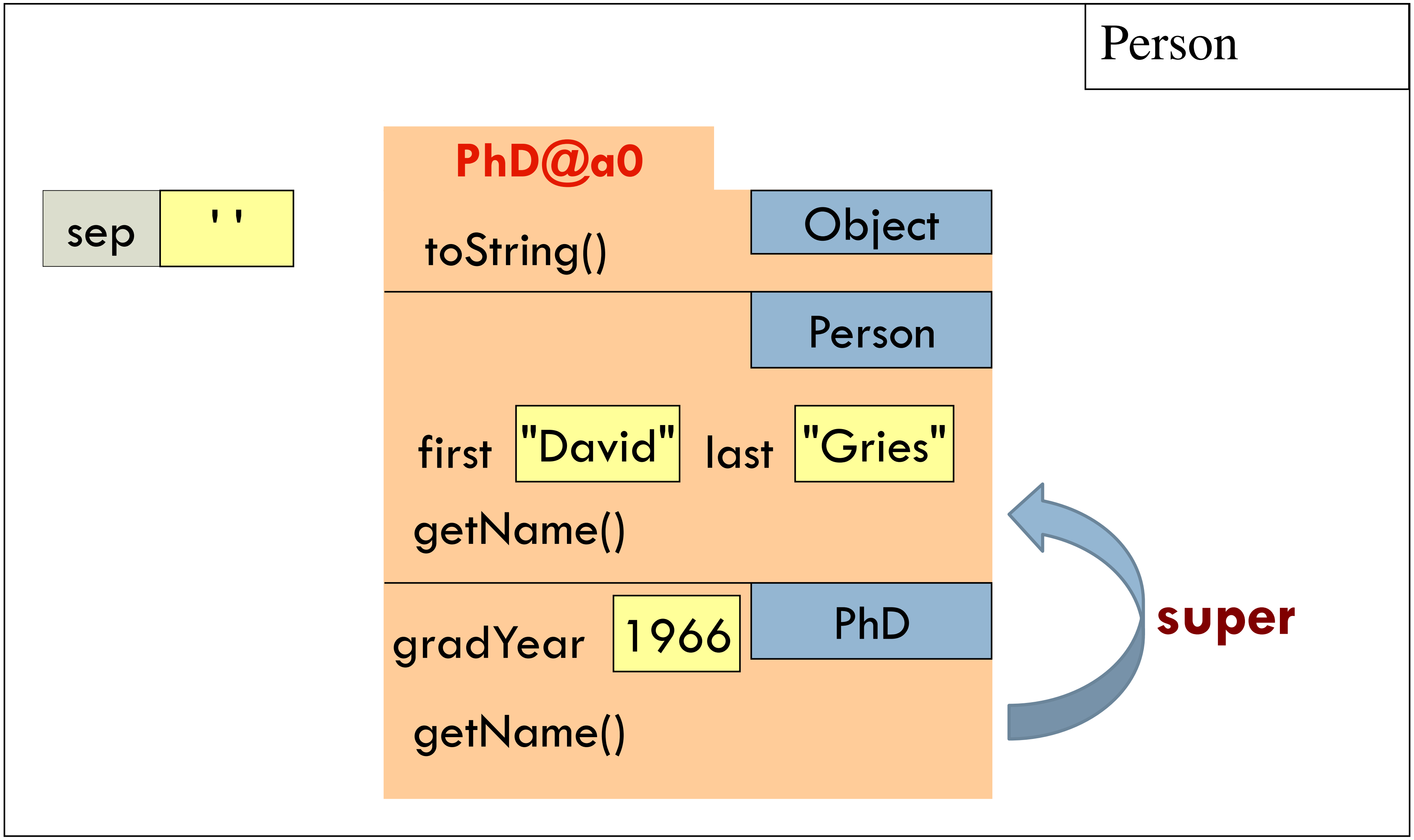
About **super**



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.

Bottom-Up and Inside-Out



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