Local variables

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

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

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

---

**Bottom-up/overriding rule**

Which method `toString()` is called by

```
c.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.

---

**Inside-out rule**

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

```
Person@0
  n
getNAndPop() {  return n + PersonPop;
}
```

Person’s objects and static components

---

**Parameters participate in inside-out rule**

Parameters participate in inside-out rule.

```
Person@0
  n
setN(String name) {
    n = name;
  }
```

Parameter `n` “blocks” reference to field `n`. (n is a “shadowed” variable)

---

**A solution: use this**

Memorize: Within an object, `this` evaluates to the name of the object.

In object `Person@a0`, `this` evaluates to `Person@a0`

```
Person@0
  n
setN(String n) {
  this.n = n;
}
```

In object `Person@a1`, `this` evaluates to `Person@a1`

```
Person@1
  n
setN(String n) {
  this.n = n;
}
```

---

**About super**

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

```
PhD@0
  toString()
```

Because of the keyword `super`, this calls `toString` in the `Object` partition.
**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) { ... }

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

Want to change body to call first 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) { ... }

    /** Constructor: instance with m minutes */
    public Time(int m) {
        this(m / 60, m % 60);
    }
}
```

Use `this` (instead of `Time`) to call another constructor in the class.
Must be first statement in constructor body!

---

**Principle: Initialize superclass fields first**

Class `Employee` contains info that is common to all employees — name, start date, salary, etc.
`getCompensation` gives the salary

Executives also get a bonus.
`getCompensation` is overridden to take this into account

Could have other subclasses for part-timers, temporary workers, consultants, etc., each with a different `getCompensation`

**Without OO …**

Without OO, you would write a long involved method:

```java
public double getCompensation(...) {
    if (worker is an executive) {
        ...}
    else if (worker is part time) {
        ...}
    else if (worker is temporary) {
        ...}
    else ...
}
```

OO eliminates need for many of these long, convoluted methods, which are hard to maintain.

Instead, each subclass has its own `getCompensation`.
End up with many more methods, which are usually very short

---

**Principle:** Initialize superclass fields first

```java
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s) {
    name= n;
    start= d;
    salary=  s;
}
```

/** Constructor: executive with name n, year hired d, salary of $50,000, bonus b */
public Executive(String n, int d, double s)

Principle: In subclass constructor, fill in the superclass fields first
How to do that if they are private?

Call constructor in superclass
**Principle: initialize superclass fields first**

```java
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s)
/** Constructor: executive with name n, year hired d, salary of $50,000, bonus b */
public Executive(String n, int d, double b) {
    super Employee(n, d, 50000);
    bonus = b;
}

To call a superclass constructor, use super(...)
```

---

**Java syntax:** First statement of any constructor you write must be a call on another constructor this(…); or super(…);