

# CS 106A, Lecture 6

## Scope and Parameters

suggested reading:

*Java Ch. 5.1-5.4*

# Plan For Today

- Announcements
- Recap: Control Flow in Java
- Nested Loops
- Methods in Java
- Scope
- Parameters

# Announcements

- Sections are finalized *today at 5PM*
  - Email [cs198@cs.stanford.edu](mailto:cs198@cs.stanford.edu) if you have a schedule conflict with your current section
  - Fill out Annie's form if you have a partner in mind and want to swap into their section

# Plan For Today

- Announcements
- **Recap: Control Flow in Java**
- Nested Loops
- Methods in Java
- Scope
- Parameters

# Conditions in Java

```
while(condition) {  
    body  
}
```

```
if(condition) {  
    body  
}
```

The condition should be a “boolean” which is either **true** or **false**

# Booleans

1 < 2

true

# Relational Operators

Operator	Meaning	Example	Value
==	equals	$1 + 1 == 2$	true
!=	does not equal	$3.2 != 2.5$	true
<	less than	$10 < 5$	false
>	greater than	$10 > 5$	true
<=	less than or equal to	$126 <= 100$	false
>=	greater than or equal to	$5.0 >= 5.0$	true

\* All have equal precedence

# Practice: Sentinel Loops

- **sentinel**: A value that signals the end of user input.
  - **sentinel loop**: Repeats until a sentinel value is seen.
- Example: Write a program that prompts the user for numbers until the user types -1, then output the sum of the numbers.
  - In this case, -1 is the sentinel value.

Type a number: **10**

Type a number: **20**

Type a number: **30**

Type a number: **-1**

Sum is 60



# Practice: Sentinel Loops

```
// fencepost problem!  
// ask for number - post  
// add number to sum - fence
```

```
int sum = 0;  
int num = readInt("Enter a number: ");  
while (num != -1) {  
    sum += num;  
    num = readInt("Enter a number: ");  
}  
println("Sum is " + sum);
```

# Practice: Sentinel Loops

```
// Solution #2: "break" out of the loop  
// ONLY appropriate to use in fencepost cases
```

```
int sum = 0;  
while (true) {  
    int num = readInt("Enter a number: ");  
    if (num == -1) {  
        break; // immediately exits loop  
    }  
    sum += num;  
}  
println("Sum is " + sum);
```

Colin prefers this solution, but the debate of how to solve the “loop-and-a-half” problem has been raging for >50 years!

# Compound Expressions

In order of precedence:

Operator	Description	Example	Result
!	not	!(2 == 3)	true
&&	and	(2 == 3) && (-1 < 5)	false
	or	(2 == 3)    (-1 < 5)	true

Cannot "chain" tests as in algebra; use && or || instead

```
// assume x is 15
2 <= x <= 10
true <= 10
Error!
```

```
// correct version
2 <= x && x <= 10
true && false
false
```

# Boolean Variables

```
// Store expressions that evaluate to true/false
boolean x = 1 < 2;           // true
boolean y = 5.0 == 4.0;     // false

// Directly set to true/false
boolean isFamilyVisiting = true;
boolean isRaining = false;

// Ask the user a true/false (yes/no) question
boolean playAgain = readBoolean("Play again?", "y", "n");
if (playAgain) {
    ...
}
```

# If/Else If/Else

```
if (condition1) {  
    ...  
} else if (condition2) {           // NEW  
    ...  
} else {  
    ...  
}
```

Runs the first group of statements if ***condition1*** is true; otherwise, runs the second group of statements if ***condition2*** is true; otherwise, runs the third group of statements.

You can have multiple else if clauses together.

# If/Else If/Else

```
int num = readInt("Enter a number: ");  
if (num > 0) {  
    println("Your number is positive");  
} else if (num < 0) {  
    println("Your number is negative");  
} else {  
    println("Your number is 0");  
}
```

# For Loops in Java

This code is run once, just before the for loop starts

Repeats the loop if this condition passes

This code is run each time the code gets to the end of the 'body'

```
for (int i = 0; i < 3; i++) {  
    println("I love CS 106A!");  
}
```

# Using the For Loop Variable

```
// Launch countdown
for(int i = 10; i >= 1; i--) {
    println(i);
}
println("Blast off!");
```

Output:

```
10
9
8
...
Blast off!
```



# Plan For Today

- Announcements
- Recap: Control Flow in Java
- **Nested Loops**
- Methods in Java
- Scope
- Parameters

# Nested loops

- **nested loop:** A loop placed inside another loop.

```
for (int i = 0; i < 5; i++) {  
    for (int j = 0; j < 10; j++) {  
        print("*");  
    }  
    println();    // to end the line  
}
```

- **Output:**

```
*****  
*****  
*****  
*****  
*****
```

- The outer loop repeats 5 times; the inner one 10 times.

# Nested loop question

- Q: What output is produced by the following code?

```
for (int i = 0; i < 5; i++) {  
    for (int j = 0; j < i + 1; j++) {  
        print("*");  
    }  
    println();  
}
```

- A.   
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*
- B.   
\*\*\*\*\*  
\*\*\*\*  
\*\*\*  
\*\*  
\*
- C.**   
\*  
\*\*  
\*\*\*  
\*\*\*\*  
\*\*\*\*\*
- D.   
1  
22  
333  
4444  
55555
- E.   
12345

*(How would you modify the code to produce each output above?)*

# Nested loop question 2

- How would we produce the following output?

```
....1  
...22  
..333  
.4444  
55555
```

# Nested loop question 2

- How would we produce the following output?

```
....1  
...22  
..333  
.4444  
55555
```

- Answer:

```
for (int i = 0; i < 5; i++) {
```

```
}
```

# Nested loop question 2

- How would we produce the following output?

```
....1
...22
..333
.4444
55555
```

- Answer:

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5 - i - 1; j++) {
        print(".");
    }
}
```

```
}
```

# Nested loop question 2

- How would we produce the following output?

```
....1
...22
..333
.4444
55555
```

- Answer:

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5 - i - 1; j++) {
        print(".");
    }
    for (int j = 0; j <= i; j++) {
        print(i + 1);
    }
}
```

# Nested loop question 2

- How would we produce the following output?

```
....1
...22
..333
.4444
55555
```

- Answer:

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5 - i - 1; j++) {
        print(".");
    }
    for (int j = 0; j <= i; j++) {
        print(i + 1);
    }
    println();
}
```



# Plan For Today

- Announcements
- Recap: Control Flow in Java
- Nested Loops
- **Methods in Java**
- Scope
- Parameters

# Defining New Commands in Karel

We can make new commands (or **methods**) for Karel. This lets us *decompose* our program into smaller pieces that are easier to understand.

```
private void name() {  
    statement;  
    statement;  
    ...  
}
```

For example:

```
private void turnRight() {  
    turnLeft();  
    turnLeft();  
    turnLeft();  
}
```

# Methods in Java

We can define new **methods** in Java just like in Karel:

```
private void name() {  
    statement;  
    statement;  
    ...  
}
```

For example:

```
private void printGreeting() {  
    println("Hello world!");  
    println("I hope you have a great day.");  
}
```

# Methods in Java

```
public void run() {  
    int x = 2;  
    printX();  
}
```

```
private void printX() {  
    // ERROR! "Undefined variable x"  
    println("X has the value " + x);  
}
```

# Plan For Today

- Announcements
- Recap: Control Flow in Java
- Nested Loops
- Methods in Java
- **Scope**
- Parameters

# A Variable love story

By Chris Piech

Once upon a time..

# There was a variable named x

```
int x = 5;
```

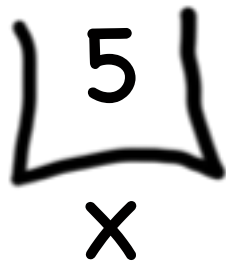
```
if (lookingForLove()) {
```

```
    int y = 5;
```

```
}
```

```
println(x + y);
```

---



A diagram illustrating the state of the variable `x`. It consists of a large, hand-drawn curly brace that encompasses the number `5`. Below the bottom edge of the brace, the letter `x` is written, indicating that the variable `x` holds the value `5`.



# ...x was looking for love!

```
int x = 5;
```

```
if (lookingForLove()) {
```

```
    int y = 5;
```

```
}
```

```
println(x + y);
```

x was definitely  
looking for love

---

5  
x

# And met y.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

---


5  
x

5  
y


# And met y.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

---

 A hand-drawn diagram showing the number 5 inside a bracket shape, with the letter x written below it.

x

 A hand-drawn diagram showing the number 5 inside a bracket shape, with the letter y written below it.

y

Hi, I'm y

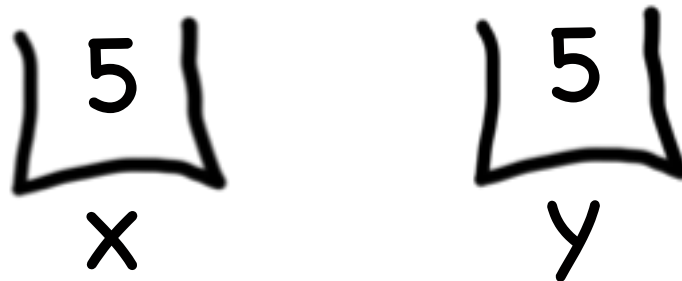
“Wow!”

# And met y.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

---

Wow

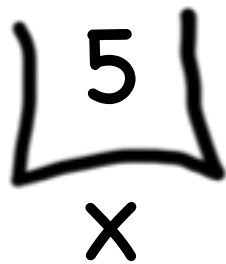


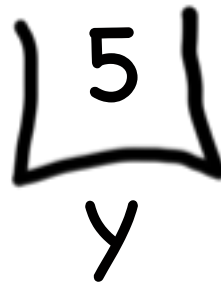
x                      y

# And met y.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

---

  
5  
x

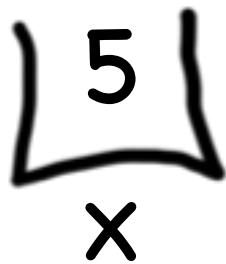
  
5  
y

We have so much  
in common

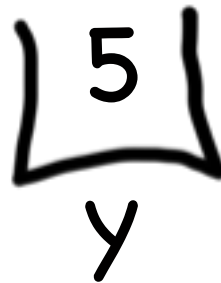
# And met y.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

---



A hand-drawn diagram showing a variable `x` containing the value `5`. The number `5` is inside a hand-drawn container shape, with the letter `x` written below it.



A hand-drawn diagram showing a variable `y` containing the value `5`. The number `5` is inside a hand-drawn container shape, with the letter `y` written below it.

We both have  
value 5!

# And met y.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

5  
x

5  
y


Maybe sometime  
we can...




# And met y.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

---

  
x

  
y

println together?

It was a beautiful match...

...but then tragedy struck.

# Tragedy Strikes

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

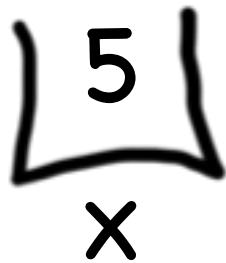
---

5  
x

5  
y

# Tragedy Strikes

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```



Nooooooooooooooooooooo!

You see...

when a program exits a code block,  
all variables declared inside that block go away!

# Since $y$ is inside the if-block...

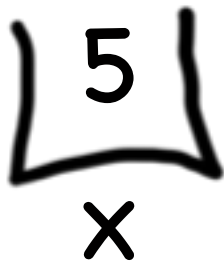
```
int x = 5;
```

```
if (lookingForLove()) {
```

```
    int y = 5;
```

```
}
```

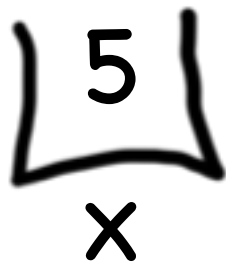
```
println(x + y);
```





...it goes away here...

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}  
println(x + y);
```

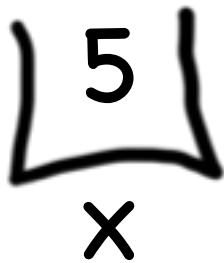


...and doesn't exist here.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
}
```

```
println(x + y);
```

**Error.**  
**Undefined**  
**variable y.**



The End

Sad times 😞

# Variable Scope

- The **scope** of a variable refers to the section of code where a variable can be accessed.
- **Scope starts** where the variable is declared.
- **Scope ends** at the termination of the code block in which the variable was declared.
  
- A **code block** is a chunk of code between { } braces

# Variable Scope

Variables have a lifetime (called scope):

```
public void run() {  
    double v = 8;  
    if (condition) {  
        v = 4;  
        ... some code  
    }  
    ... some other code  
}
```

# Variable Scope

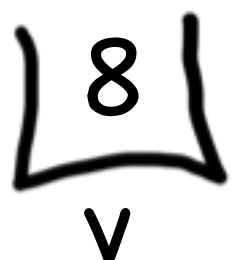
Variables have a lifetime (called scope):

```
public void run() {  
    double v = 8;  
    if (condition) {  
        v = 4;  
        ... some code  
    }  
    ... some other code  
}
```

# Variable Scope

Variables have a lifetime (called scope):

```
public void run() {  
    double v = 8; ← Comes to life here  
    if (condition) {  
        v = 4;  
        ... some code  
    }  
    ... some other code  
}
```



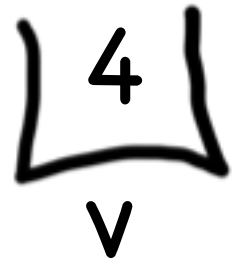


# Variable Scope

Variables have a lifetime (called scope):

```
public void run() {  
    double v = 8;  
    if (condition) {  
        v = 4;  
        ... some code  
    }  
    ... some other code  
}
```

This is the **inner most** code block in which it was declared....

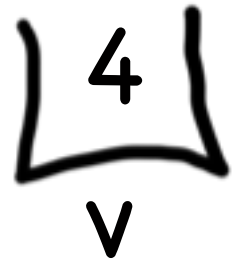


# Variable Scope

Variables have a lifetime (called scope):

```
public void run() {  
    double v = 8;  
    if (condition) {  
        v = 4;  
        ... some code  
    }  
    ... some other code  
}
```

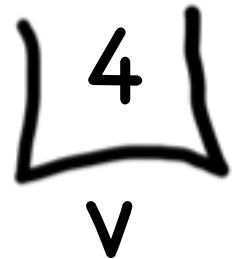
Still alive here...



# Variable Scope

Variables have a lifetime (called scope):

```
public void run() {  
    double v = 8;  
    if (condition) {  
        v = 4;  
        ... some code  
    }  
    ... some other code  
}
```



It goes away here (at the end of its code block)

# Variable Scope

Variables have a lifetime (called scope):

```
public void run() {  
    double v = 8;  
    if (condition) {  
        v = 4;  
        ... some code  
    }  
    ... some other code  
}
```



It goes away here (at the end of its code block)



# Variable Scope

Variables have a lifetime (called scope):

```
public void run() {  
    ... some code  
    if (condition) {  
        int w = 4;  
        ... some code  
    }  
    ... some other code  
}
```

w is created here

w goes away here (at the end of its code block)

The diagram illustrates the scope of a variable 'w'. A blue arrow points from the text 'w is created here' to the declaration 'int w = 4;'. Another blue arrow points from the text 'w goes away here (at the end of its code block)' to the closing brace of the 'if' block.

# Variable Scope

Variables have a lifetime (called scope):

```
public void run() {  
    ... some code  
    if (condition) {  
        int w = 4;  
        ... some code  
    }  
    ... some other code  
}
```



This is the scope of **w**

# Variable Scope

You *cannot* have two variables with the same name in the *same scope*.

```
for (int i = 1; i <= 100; i++) {  
    int i = 2;           // ERROR  
    print("/");  
}
```



# Variable Scope

You *cannot* have two variables with the same name in the *same scope*.

```
for (int i = 1; i <= 100; i++) {  
    while (...) {  
        int i = 5;           // ERROR  
    }  
}
```

# Variable Scope

You *can* have two variables with the same name in *separate scopes*.

```
public void run() {  
    for (int i = 0; i < 5; i++) { // i ok here  
        int w = 2; // w ok here  
    }  
  
    for (int i = 0; i < 2; i++) { // i ok here  
        int w = 3; // w ok here  
    }  
}
```

# Variable Scope

You *can* have two variables with the same name in *separate scopes*.

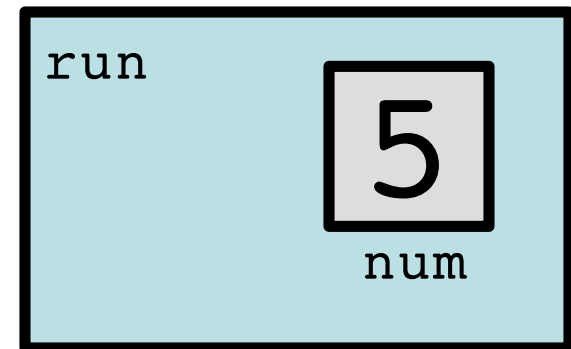
```
public void run() {  
    int num = 5;  
    cow();  
    println(num);           // prints 5  
}  
  
private void cow() {  
    int num = 10;  
    println(num);         // prints 10  
}
```

# Variable Scope

You *can* have two variables with the same name in *different scopes*.

```
public void run() {  
    int num = 5;  
    cow();  
    println(num);  
}
```

```
private void cow() {  
    int num = 10;  
    println(num);  
}
```

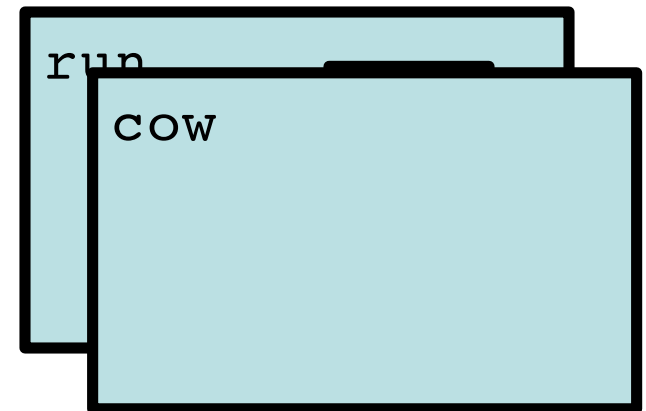


# Variable Scope

You *can* have two variables with the same name in *different scopes*.

```
public void run() {  
    int num = 5;  
    cow();  
    println(num);  
}
```

```
private void cow() {  
    int num = 10;  
    println(num);  
}
```

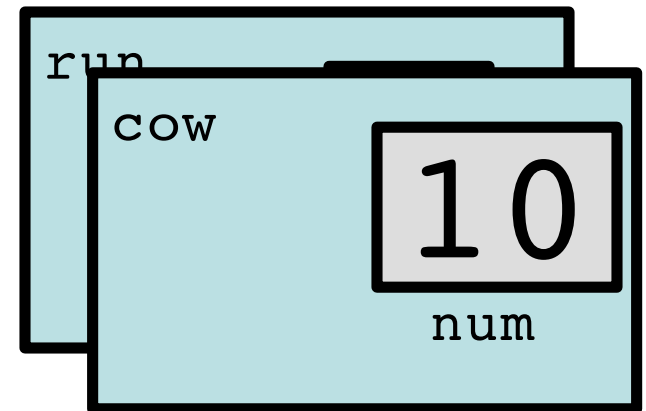


# Variable Scope

You *can* have two variables with the same name in *different scopes*.

```
public void run() {  
    int num = 5;  
    cow();  
    println(num);  
}
```

```
private void cow() {  
    int num = 10;  
    println(num);  
}
```

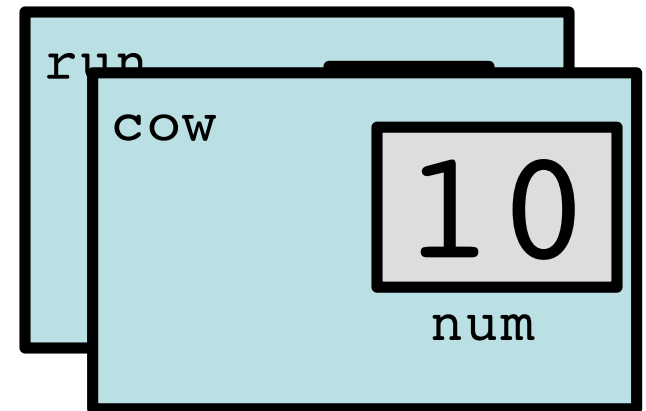


# Variable Scope

You *can* have two variables with the same name in *different scopes*.

```
public void run() {  
    int num = 5;  
    cow();  
    println(num);  
}
```

```
private void cow() {  
    int num = 10;  
    println(num);  
}
```

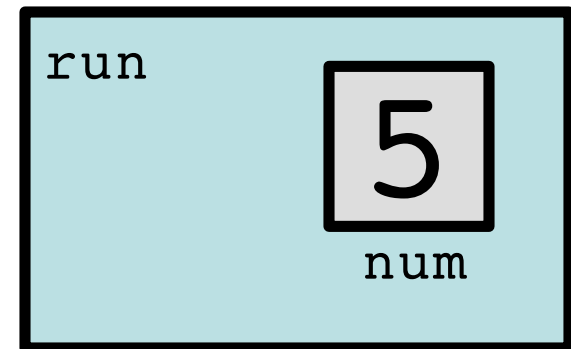


# Variable Scope

You *can* have two variables with the same name in *different scopes*.

```
public void run() {  
    int num = 5;  
    cow();  
    println(num);  
}
```

```
private void cow() {  
    int num = 10;  
    println(num);  
}
```





# Revisiting Sentinel Loops

```
// sum must be declared outside of the loop!  
// Otherwise, it will be redeclared many times  
// num must be declared outside of the loop!  
// Otherwise, the loop condition makes no sense  
int sum = 0;  
int num = readInt("Enter a number: ");  
while (num != -1) {  
    sum += num;  
    num = readInt("Enter a number: ");  
}  
println("Sum is " + sum);
```

# Revisiting Sentinel Loops

```
// Here, num goes out of scope at the end of
// each loop iteration. At that point, we have
// already used its value and can discard it.
```

```
int sum = 0;
while (true) {
    int num = readInt("Enter a number: ");
    if (num == -1) {
        break;    // immediately exits loop
    }
    sum += num;
}
println("Sum is " + sum);
```

# A Variable love story

Chapter 2  
By Chris

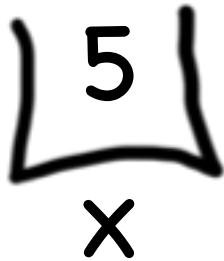
The programmer fixed the bug

# There was a variable named x.

```
int x = 5;
```

```
if (lookingForLove()) {  
    int y = 5;  
    println(x + y);  
}
```

---



A diagram illustrating the state of the variable `x`. It consists of a large, hand-drawn bracket shape that is wider at the top and tapers towards the bottom. Inside the top part of the bracket is the number `5`. Below the bottom tip of the bracket is the letter `x`. This diagram represents the memory state where the variable `x` holds the value `5`.

# ...x was looking for love!

```
int x = 5;
```

```
if (lookingForLove()) {
```

```
    int y = 5;
```

```
    println(x + y);
```

```
}
```

x was definitely  
looking for love

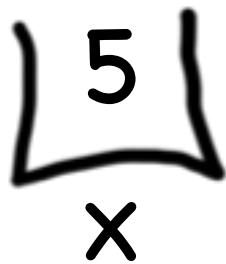
---

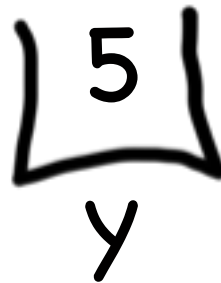
5  
x

# And met y.

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
    println(x + y);  
}
```

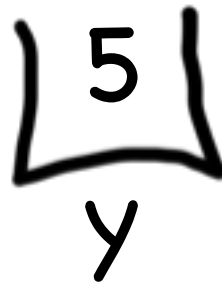
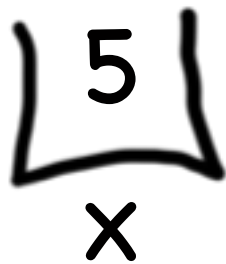
---

A hand-drawn diagram showing the number 5 inside a bracket shape, with the letter x written below it.

A hand-drawn diagram showing the number 5 inside a bracket shape, with the letter y written below it.

# Since they were both "in scope"...

```
int x = 5;  
if (lookingForLove()) {  
    int y = 5;  
    println(x + y);  
}
```





...they lived happily ever after.  
The end.

# Plan For Today

- Announcements
- Recap: Control Flow in Java
- Nested Loops
- Methods in Java
- Scope
- **Parameters**

# Variable Scope

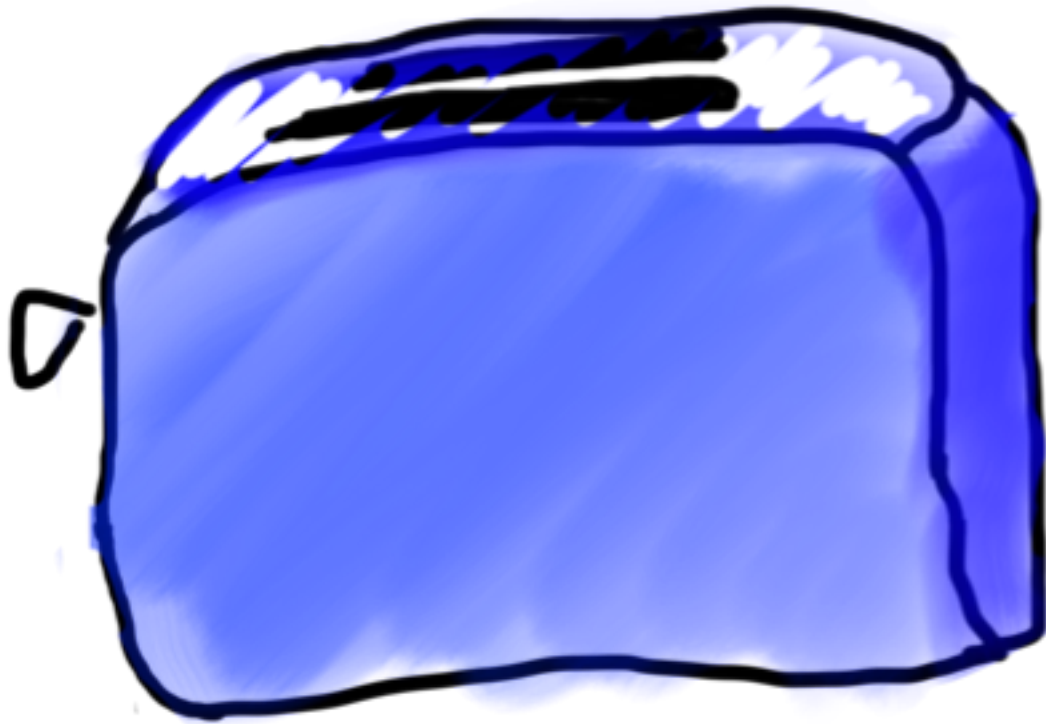
```
public void run() {  
    int x = 2;  
    printX();  
}
```

```
private void printX() {  
    // ERROR! "Undefined variable x"  
    println("X has the value " + x);  
}
```

# Parameters

Parameters let you provide a method some information when you are calling it.

# Methods = Toasters



# Methods = Toasters



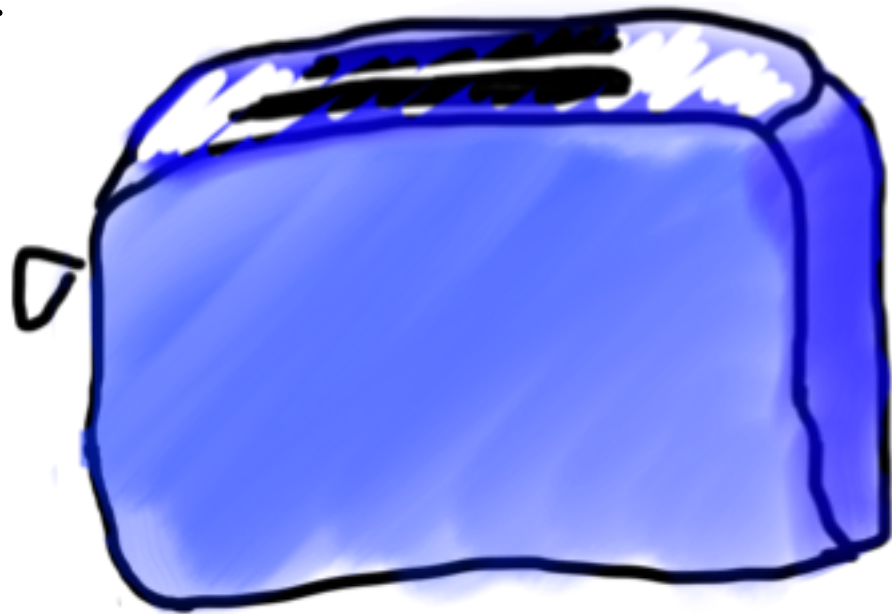
parameter



# Methods = Toasters



parameter



# Methods = Toasters



parameter

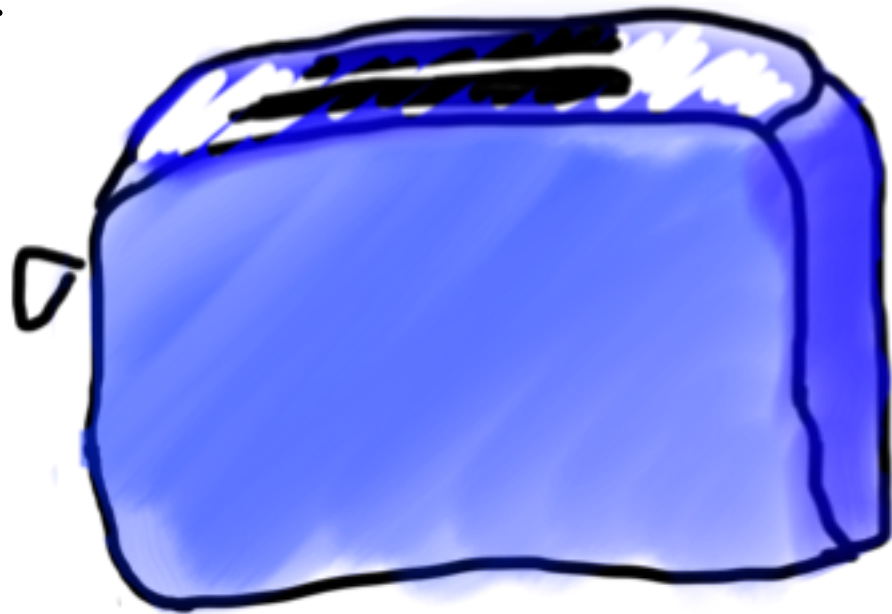




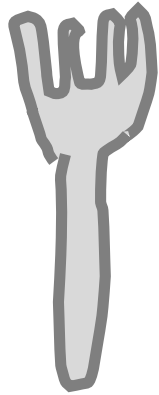
# Methods = Toasters



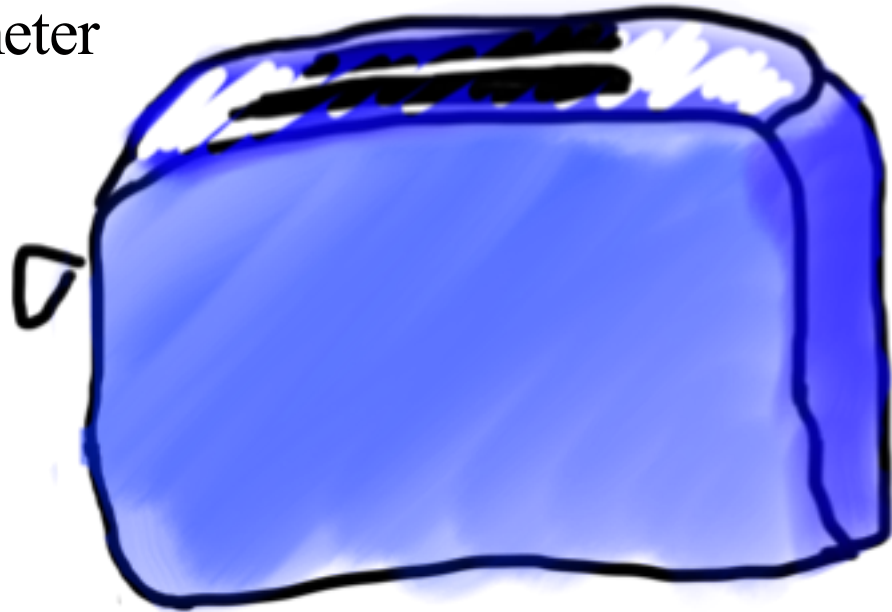
parameter



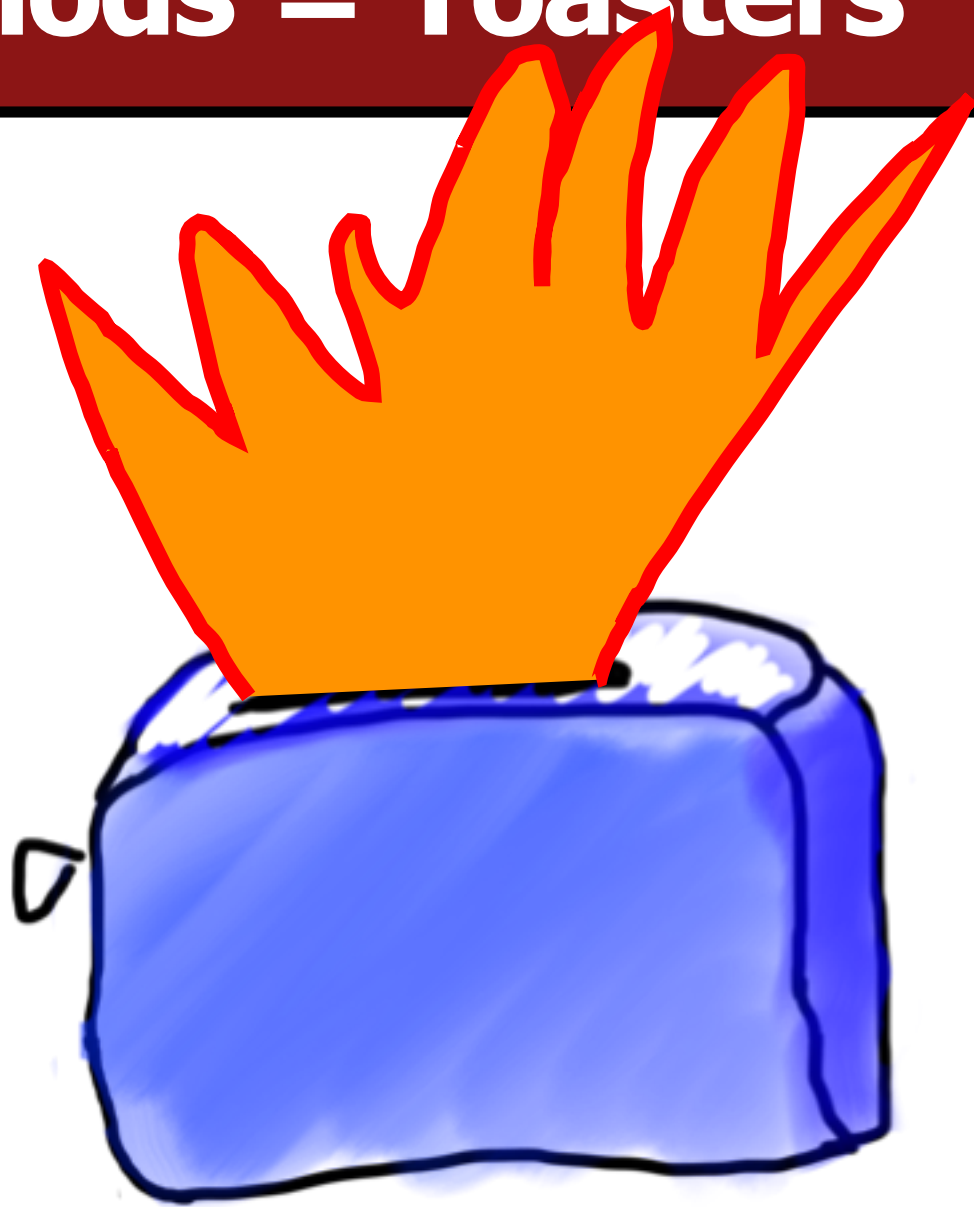
# Methods = Toasters



Invalid parameter



# Methods = Toasters



# Drawing boxes

- Consider the task of printing the following boxes:

```
*****  
*           *  
*           *  
*****
```

```
*****  
*           *  
*           *  
*           *  
*           *  
*****
```

- The code to draw each box will be very similar.
  - Would variables help? Would constants help?

# Wouldn't it be nice if...

```
drawBox(10, 4);
```

Continued next time...