CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

Operators

- Java provides a rich operator environment.
- Most of its operators can be divided into the following four groups:
  - Arithmetic,
  - Bitwise,
  - Relational, and
  - Logical.

Arithmetic Operators

- **The operands of the arithmetic operators must be of a numeric type.**
- **Cannot use them on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.**

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
––	Decrement

The Bitwise Operators

- **Java defines several *bitwise operators* that can be applied to the integer types: long, int, short, char, and byte.**
- **These operators act upon the individual bits of their operands.**

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

```
// Demonstrate the bitwise logical operators.
class BitLogic {
  public static void main(String args[]) {
    String binary[] = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };
    int a = 3; // 0 + 2 + 1 or 0011 in binary
    int b = 6; // 4 + 2 + 0 or 0110 in binary
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b)|(a & ~b);
    int g = ~a & 0x0f;

    System.out.println("        a = " + binary[a]);
    System.out.println("        b = " + binary[b]);
    System.out.println("      a|b = " + binary[c]);
    System.out.println("      a&b = " + binary[d]);
    System.out.println("      a^b = " + binary[e]);
    System.out.println("~a&b|a&~b = " + binary[f]);
    System.out.println("       ~a = " + binary[g]);
  }
}
```

```
 00101010   42
|00001111   15
 ----------
 00101111   47
```

```
 00101010   42
&00001111   15
 ----------
 00001010   10
```

Here is the output from this program:

```
        a = 0011
        b = 0110
      a|b = 0111
      a&b = 0010
      a^b = 0101
~a&b|a&~b = 0101
       ~a = 1100
```

Relational Operators

- **The *relational operators* determine the relationship that one operand has to the other.**
- **Specifically, they determine equality and ordering.**
- **outcome of these operations is a boolean value.**
- **only integer, floating-point, and character operands may be compared to see which is greater or less than the other.**

```
int done;
//...
if(!done)... // Valid in C/C++
if(done)... // but not in Java.
```

```
if(done == 0)... // This is Java-style.
if(done != 0)...
```

- **The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero.**
- **In Java, true and false are nonnumeric values that do not relate to zero or nonzero.**
- **Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.**

Relational Operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Boolean Logical Operators

- **The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.**

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

Boolean Logical Operators

- The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **!false == true**.

```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("        a = " + a);
    System.out.println("        b = " + b);
    System.out.println("      a|b = " + c);
    System.out.println("      a&b = " + d);
    System.out.println("      a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println("       !a = " + g);
  }
}
```

```
        a = true
        b = false
      a|b = true
      a&b = false
      a^b = true
!a&b|a&!b = true
       !a = false
```

Short-Circuit Logical Operators

- Java provides two interesting Boolean operators not found in some other computer languages. conditional-and and the conditional-or.
- These are secondary versions of the Boolean AND and OR operators, and are commonly known as *short-circuit* logical operators.
- OR operator results in **true** when **A** is **true**, no matter what **B** is. Similarly, the AND operator results in **false** when **A** is **false**, no matter what **B** is.
- If you use the **||** and **&&** forms, rather than the **|** and **&** forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.
- This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

```
if (denom != 0 && num / denom > 10)
```

The Assignment Operator

- The *assignment operator* is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:
- *var = expression*;
- Here, the type of *var* must be compatible with the type of *expression*.
- int x, y, z;

  x = y = z = 100; // set x, y, and z to 100

The ? Operator

- Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form:

- expression1 ? expression2 : expression3

- Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same (or compatible) type, which can’t be void.

- Here is an example of the way that the ? is employed:

- ratio = denom == 0 ? 0 : num / denom;

```
// Demonstrate ?.
class Ternary {
  public static void main(String args[]) {
    int i, k;

    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
  }
}
```

Operator Precedence

- Order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence.
- In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left).
- The **[ ]**, **( )**, and **.** can also act like operators. In that capacity, they would have the highest precedence.

Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
–>						
=	op=					
Lowest						

Using Parentheses

- Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:
- a >> b + 3
- This expression first adds 3 to b and then shifts a right by that result. That is, this expression can be rewritten using redundant parentheses like this:
- a >> (b + 3)
- However, if you want to first shift a right by b positions and then add 3 to that result, you will need to parenthesize the expression like this:
- (a >> b) + 3
- In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression. For example, which of the following expressions is easier to read?
- a | 4 + c >> b & 7
- (a | (((4 + c) >> b) & 7))

Control statements

- To cause the flow of execution to advance and branch based on changes to the state of a program.
- Java’s program control statements can be put into the following categories:
  - selection
  - iteration, and
  - jump.

Control statements

- Selection statements:
  - To choose different paths of execution based upon the outcome of an expression or the state of a variable.
- Iteration statements:
  - To enable program execution to repeat one or more
- Jump statements:
  - To allow your program to execute in a nonlinear fashion.

Java’s Selection Statements

- To control the flow of your program’s execution based upon conditions known only during run time.
- Java supports two selection statements:
  - if and switch.

Java's Selection Statements

- If statement
  - The if statement is Java's conditional branch statement.
  - Used to route program execution through two different paths.

```
if (condition)
        statement1;
else
        statement2;
```

- Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block).
- The condition is any expression that returns a boolean value.
- The else clause is optional.
- The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed.
- The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.
- If there is no final **else** and all other conditions are **false**, then no action will take place.

Nested If

```
if(condition)
   statement;
else if(condition)
   statement;
else if(condition)
   statement;
.
.
.
else
   statement;
```

Selection statement: If statement

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else {
  waitForMoreData();
  bytesAvailable = n;
}
```

```
boolean dataAvailable;
//...
if (dataAvailable)
  ProcessData();
else
  waitForMoreData();
```

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

Selection statement: Switch

- The switch statement is Java's multiway branch statement.
- It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- A better alternative than a large series of if-else-if statements.
- Expression must resolve to type byte, short, int, char, String or an enumeration.
- Duplicate case values are not allowed. The type of each value must be compatible with the type of expression.

```
switch (expression) {
  case value1:
     // statement sequence
     break;
  case value2:
     // statement sequence
     break;
.
.
.
  case valueN :
     // statement sequence
     break;
  default:
     // default statement sequence
  }
```

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Selection statement: Switch

```
switch(count) {
  case 1:
    switch(target) { // nested switch
      case 0:
        System.out.println("target is zero");
        break;
      case 1: // no conflicts with outer switch
        System.out.println("target is one");
        break;
    }
    break;
  case 2: // ...
```

case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch.
The count variable is compared only with the list of cases at the outer level.
If count is 1, then target is compared with the inner list cases.

Selection statement: Switch

```
class StringSwitch {
  public static void main(String args[]) {

    String str = "two";

    switch(str) {
      case "one":
        System.out.println("one");
        break;
      case "two":
        System.out.println("two");
        break;
      case "three":
        System.out.println("three");
        break;
      default:
        System.out.println("no match");
        break;
    }
  }
}
```

```
// An improved version of the season program.
class Switch {
  public static void main(String args[]) {
    int month = 4;

    String season;

    switch (month) {
      case 12:
      case 1:
      case 2:
        season = "Winter";
        break;
      case 3:
      case 4:
      case 5:
        season = "Spring";
        break;
      case 6:
      case 7:
      case 8:
        season = "Summer";
        break;
      case 9:
      case 10:
      case 11:
        season = "Autumn";
        break;
      default:
        season = "Bogus Month";
    }
    System.out.println("April is in the " + season + ".");
  }
}
```

Selection statement: Switch

- Three important features of the switch statement to note:
- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
- No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.
- A switch statement is usually more efficient than a set of nested ifs.

Selection statement: switch vs if

- To select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-else**s.
- The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression.
- The compiler has no such knowledge of a long list of **if** expressions.

Iteration Statements:

- Java’s iteration statements are
  - for,
  - while, and
  - do-while.
- These statements are commonly call loops.
- A loop repeatedly executes the same set of instructions until a termination condition is met.

Iteration statement: While

- The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.

```
while(condition) {
   // body of loop
}
```

- The condition can be any Boolean expression.
- The body of the loop will be executed as long as the conditional expression is true.
- When condition becomes false, control passes to the next line of code immediately following the loop.
- The curly braces are unnecessary if only a single statement is being repeated.

```
// The target of a loop can be empty.
class NoBody {
  public static void main(String args[]) {
    int i, j;

    i = 100;
    j = 200;

    // find midpoint between i and j
    while(++i < --j); // no body in this loop

    System.out.println("Midpoint is " + i);
  }
}
```

Midpoint is 150

```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1

Iteration statement: do-while

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.

- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

- Java’s loops, condition must be a Boolean expression.

```
do {
  // body of loop
} while (condition);
```

```
// Using a do-while to process a menu selection
class Menu {
  public static void main(String args[])
    throws java.io.IOException {
    char choice;

    do {
      System.out.println("Help on: ");
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. while");
      System.out.println("  4. do-while");
      System.out.println("  5. for\n");
      System.out.println("Choose one:");
      choice = (char) System.in.read();
    } while( choice < '1' || choice > '5');

    System.out.println("\n");
```

Iteration statement: While vs do While

- While loop
  - The conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all.
- Do while loop
  - Execute the body of a loop at least once, even if the conditional expression is false to begin with.
  - To test the termination expression at the end of the loop rather than at the beginning.
  - The do-while loop always executes its body at least once.

Iteration statement:for

- There are two forms of the for loop.
  - for loop
  - for each loop

```
for(initialization; condition; increment){
// body of the loop
}
```

Iteration statement: for

```
for(initialization; condition; increment){
// body of the loop
}
```

- First, the initialization portion of the loop is executed only once when the loop starts.
- Next, Boolean expression condition is evaluated.
  - If this expression is true, then the body of the loop is executed.
  - If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed.
- The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

```
// Demonstrate the for loop.
class ForTick {
  public static void main(String args[]) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

```
// Declare a loop control variable inside the for.
class ForTick {
  public static void main(String args[]) {

    // here, n is declared inside of the for loop
    for(int n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

```
boolean done = false;

for(int i=1; !done; i++) {
  // ...
  if(interrupted()) done = true;
}
```

```
for( ; ; ) {
  // ...
}
```

An infinite loop .This loop will run forever because there is no condition under which it will terminate.

To allow two or more variables to control a for loop, Java permits you to include multiple statements in both the initialization and iteration portions of the for. Each statement is separated from the next by a comma.

```
// Using the comma.
class Comma {
  public static void main(String args[]) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

a = 1
b = 4
a = 2
b = 3

```
// Parts of the for loop can be empty.
class ForVar {
  public static void main(String args[]) {
    int i;
    boolean done = false;

    i = 0;
    for( ; !done; ) {
      System.out.println("i is " + i);
      if(i == 10) done = true;
      i++;
    }
  }
}
```

Iteration statement: For-Each version of the for loop

- A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

```
//The general form of the for-each
for(type itr-var : collection)
statement-block
```

  - type specifies the type
  - itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

```
// Use a for-each style for loop.
class ForEach {
  public static void main(String args[]) {
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;

    // use for-each style for to display and sum the values
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
    }

    System.out.println("Summation: " + sum);
  }
}
```

```
// Use break with a for-each style for.
class ForEach2 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // use for to display and sum the values
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
      if(x == 5) break; // stop the loop when 5 is obtained
    }

    System.out.println("Summation of first 5 elements: " + sum);
  }
}
```

for-each style loop.
Its iteration variable is “read-only” as it relates to the underlying array.
An assignment to the iteration variable has no effect on the underlying array.
i,.e can’t change the contents of the array by assigning the iteration variable a new value.

```
// The for-each loop is essentially read-only.
class NoChange {
  public static void main(String args[]) {
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    for(int x: nums) {
      System.out.print(x + " ");
      x = x * 10; // no effect on nums
    }

    System.out.println();

    for(int x : nums)
      System.out.print(x + " ");

    System.out.println();
  }
}
```

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[][] = new int[3][5];

    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j = 0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // use for-each for to display and sum the values
    for(int x[] : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

```
// Use type inference in a for loop.
class TypeInferenceInFor {
  public static void main(String args[]) {

    // Use type inference with the loop control variable.
    System.out.print("Values of x: ");
    for(var x = 2.5; x < 100.0; x = x * 2)
      System.out.print(x + " ");

    System.out.println();

    // Use type inference with the iteration variable.
    int[] nums = { 1, 2, 3, 4, 5, 6};
    System.out.print("Values in nums array: ");
    for(var v : nums)
      System.out.print(v + " ");

    System.out.println();
  }
}
```

Values of x: 2.5 5.0 10.0 20.0 40.0 80.0
Values in nums array: 1 2 3 4 5 6

Jump Statements

- Java supports three jump statements:
  - break,
  - continue, and
  - return.
- These statements transfer control to another part of your program.

Break

- Force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Using break as labels

```
// Using break as a civilized form of goto.
class Break {
  public static void main(String args[]) {
    boolean t = true;

    first: {
      second: {
        third: {
          System.out.println("Before the break.");
          if(t) break second; // break out of second block
          System.out.println("This won't execute");
        }
        System.out.println("This won't execute");
      }
      System.out.println("This is after second block.");
    }
  }
}
```

```
Before the break.
This is after second block.
```

```
// Using break to exit from nested loops
class BreakLoop4 {
  public static void main(String args[]) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
      }
      System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

```
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.
```

Using break as labels

```
// This program contains an error.
class BreakErr {
  public static void main(String args[]) {

    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break one; // WRONG
      System.out.print(j + " ");
    }
  }
}
```

Using continue

- In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression.
- For all three loops, any intermediate code is bypassed.

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

```
0 1
2 3
4 5
6 7
8 9
```

```
// Using continue with a label.
class ContinueLabel {
  public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
         for(int j=0; j<10; j++) {
           if(j > i) {
             System.out.println();
             continue outer;
           }
           System.out.print(" " + (i * j));
         }
       }
       System.out.println();
  }
}
```

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

jump statement:Return

- The return statement is used to explicitly return from a method.
- That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a

```
// Demonstrate return.
class Return {
  public static void main(String args[]) {
    boolean t = true;

    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
  }
}
```

Before the return.