Fast Track to Java

Customized for Starbucks

Delivered by DevelopIntelligence

Control Structures and Data

Objectives

At the end of this section you should be able to

- Describe what operators and expressions are
- Describe how operators are used to create expressions
- Describe the operators in Java, the kinds of data they operate on, and the types of expressions they produce
- Describe the difference between narrowing and widening operators
- Use the cast operator correctly

Operators and Expressions

- An expression in Java is something that evaluates to a result
  - A variable is an expression because it evaluates to a result - the value of the data it contains
  - A literal is also an expression
- An operator is used to combine two expressions to produce a new expression
- Think of variables as nouns and operators as verbs
  - Combine nouns and verbs to create phrases
  - Phrases correspond to expressions

Operators and Expressions (cont.)

- Every expression has a type, just like a variable
- The type of an expression is determined by the type associated with the data results when we evaluate the expression
- For example, a ***`boolean`*** expression is one that results in a ***`boolean`*** result while a ***`String`*** expression is one that results in a ***`String`***

Operators

- Operators are of three valences in programming languages
  - Unary operators - operate on a single expression
  - Binary operators - combine two expressions
  - Ternary operators - combine three expressions
- Most operators fall in the binary operator category
- There is only one ternary operator in Java
- Java does not allow operator overloading like in C++ / C#

Operators - Example

- There is no relationship between the type of operator (category)
- and the expression type.

Operators in Java

```
1 + 4   // arithmetic operator "+" operating on two ints
1.0 * 4.1   // arithmetic operator "*" operating on two doubles
true && false // logical operator operating on two booleans
true + false  // illegal! - you can't do arithmetic on booleans

// Error in the following line, even though almost all the
// operators in the expression are arithmetic, the final result
// is a boolean, and cannot be assigned to x.
int x = (((34 + 12)/13) * (89-16)/(13 *2)) > 0;
```

Arithmetic Operators

- These operators are the standard **`+ - * / %`** operators
  - Java also has the increment and decrement operators
  - Defined for both the integral and floating point types
- Mixed Mode Arithmetic
  - All arithmetic operators work on either two integral operands or two floating point operands
  - Smaller operand types are promoted to the larger type, and to *at least* **`int`**
  - If one operand is integral and one operand is floating point then the integral operand is converted to a floating point number before the operation takes place

Mixed Mode Arithmetic

- Converting from integral to floating point values might produce a loss of precision
  - binary representation of floating point values does not reliably match “exact” decimals

Mixed Mode Arithmetic

```
1 + 4       // result is integral 5
1.1 + 4.2   // result is 5.3 (or 5.3000000000000001 sometimes)
1.0 + 4.0   // result is 5.0 – still floating point
1.0 + 4 // result is 5.0 – one operand is floating point
```

Division and Modulus

Division in Java

```
17 / 3         // result is 5 – integral division
17 % 3         // result is 2 – remainder of 17 / 3
17.0 / 3.0 // result 5.2 – floating point division
17.3 / 3   // result is 5.766666666666667
              // -- floating point division
17.0 % 3   // result is 2.0 – floating point modulus
14.5 % 3.32// result is 1.2200000000000006
              // whatever that means.
```

Increment and Decrement

- Java provides increment and decrement operators: ++ and –
- Postfix: <VAR>++ and <VAR>--
  - The value of the variable is used in the expression first, then modified
- Prefix - ++<VAR> and --<VAR>
  - The value of the variable is incremented or decremented first, then modified

```
x++ is equivalent to x = x + 1
++x is equivalent to x = x + 1
x-- is equivalent to x = x - 1
--x is equivalent to x = x - 1
```

Increment and Decrement Example

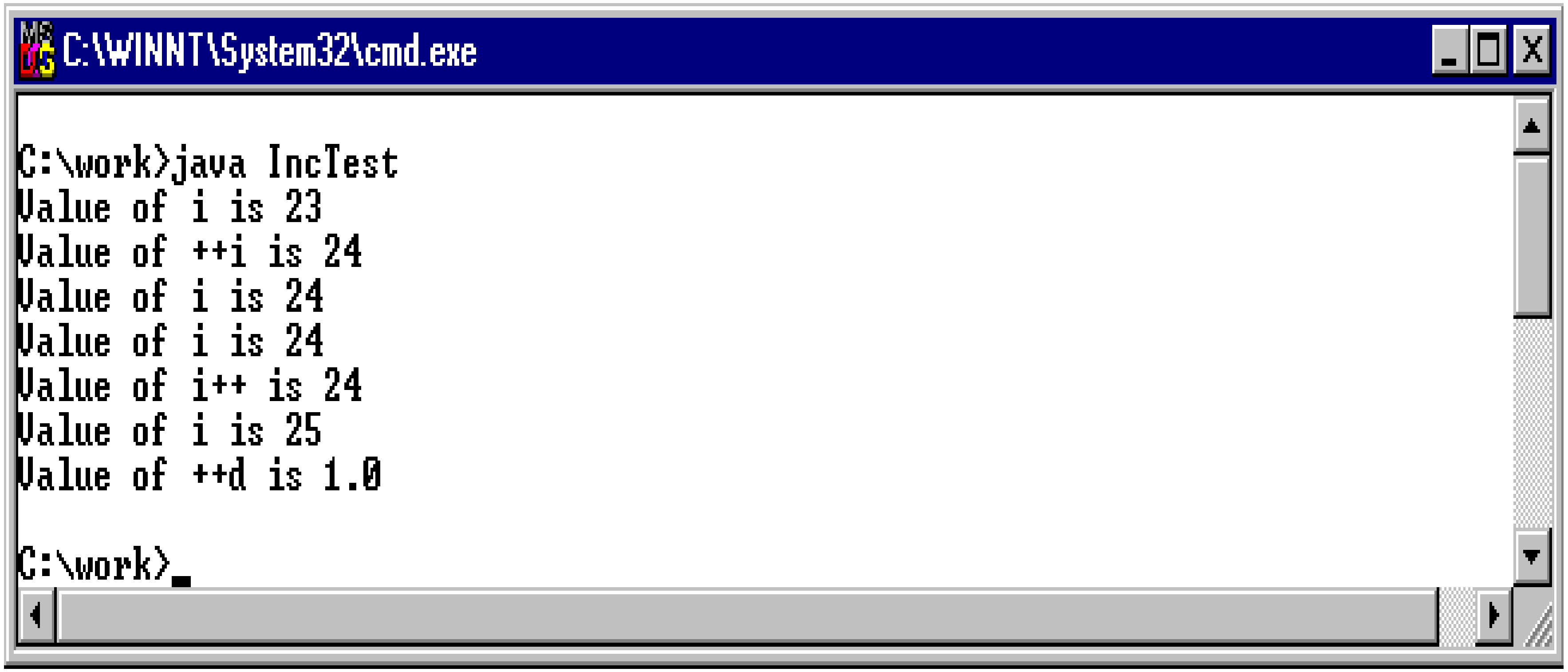
Increment and Decrement

```
int i = 23;
double d = 0.0;

// Print out i
System.out.println("Value of i is "+ i);
// Print out ++i - i is printed out after being incremented
System.out.println("Value of ++i is "+ (++i));
// Print out i again - it has been incremented.
System.out.println("Value of i is "+ i);

// Print out i
System.out.println("Value of i is "+ i);
// Print out i++ - i is printed out before being incremented
System.out.println("Value of i++ is "+ (i++));
// Print out i again - it has been incremented.
System.out.println("Value of i is "+ i);
// Just to see that it works with floating points
System.out.println("Value of ++d is "+ (++d));
```

Increment and Decrement Example Output



```
C:\work>java IncTest
Value of i is 23
Value of ++i is 24
Value of i is 24
Value of i is 24
Value of i++ is 24
Value of i is 25
Value of ++d is 1.0

C:\work>_
```

Output of the IncTest

Comparison Operators With Numerics

- Comparison operators are defined for numeric and character data
- The result of all comparisons is either true or false

Comparison Operators in Java

`==`	Equality
`<`	Less than
`>`	Greater Than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`!=`	Not equal

Comparison Operators With Other Types

- Only the `==` and `!=` operators are defined for `boolean` types
- Not all comparison operators work for `String` types
  - Remember, a `String` is not a primitive data type
  - Only the == and != work with `String` types, but not quite as you might expect
  - You will learn more about how these operators work in a later module

Cautions with Comparison Operators

- It is possible for a loss of precision to occur when working with floating point numbers
  - This is unavoidable
  - Sometimes using floating point numbers in comparisons can produce counterintuitive results

Relational operators and floating point numbers

```
double d = 1.1 + 4.2; // As we saw, could be 5.3000000000000001
d == 5.3             // Because of representation, this is false

double d1 = 1e300; // d1 is a very large number
d1 < (d1 + 1)      // because of representation, this is false
d1 == (d1 + 1)     // but this is true
```

Logical Operators

- The operators &, |, ^, and ! all work according to the usual rules of boolean operations
- The two operators && and || are called short circuit operators

Evaluation of Logical Operators
Assume x and y are boolean expressions .

```
x & y   true if both x and y are true, false otherwise
x | y   false if both x and y are false, true otherwise.
x ^ y   true if either x or y is true but not both
!x      false if x is true, true is x is false

x && y same as &, but if x is false, y is not evaluated
x || y same as | but if x is true, y is not evaluated.
```

Short Circuit Evaluations

- In a short circuit evaluation, we stop evaluating the expression as soon as we know what the outcome will be
- This avoids unnecessary processing if the first part of the evaluation is ***`false`***

Short circuit evaluation

```
int x = 0;
boolean test = false & (1 == ++x);
// x is incremented and is now 1
test = false && (2 == ++x);
// second operand is not evaluated!
// x still has value 1
```

Assignment Operators

Java allows C/C++ style assignment operator notation

Operator Assignment

```
x = x * 34;  //can be written as   x *= 34;
x = x / 2;   //can be written as   x /= 2;
x = x + y;   //can be written as   x += y;
X = x - y;      //can be written as   x -= y;
```

String Operators

- Operators in general are not defined for the String type data
- String catenation can be performed using
  - +
  - +=
- Only work when at least one operand is a String
- Other operand is converted to a String
- The two Strings are then concatenated into a new String
- Any kind of data can be converted to a String

String Operators Example

String Catenation

```
String message = "String data ";
int i = 34;
float f = 89.13F;
boolean b = true;
char c = '*';

message + i -> "String data 34"
f + message -> "89.13String data "
message + b -> "String data true"
message + c -> "String Data *"
f + " " + i -> "89.13 34"
b + " " + c + " " + i ->  "true * 34"

// implicit string conversion
(i + "") + f  -> "3489.13"
i + f -> 123.13
```

Operator Precedence and Associativity

Operator	Associates
`[] . () function_call`	Left to right
`! ~ ++ -- cast new - +(unary form)`	Right to left
`* / %`	Left to right
`+ - (binary form)`	Left to right
`<< >> >>>`	Left to right
`< <= > >=`	Left to right
`== !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Left to right
`= (op=)`	Right to left

Operator Precedence and Associatively Example

Operator Precedence

Since `*` has a higher precedence than `+`

```
2 * 4 + 3 -> 8 + 3 -> 11
3 + 2 * 4 -> 3 + 8 -> 11
```

But we can change the order of operations with `()` to make the `+` be evaluated first

```
2 * (4 + 3) -> 2 * 7 -> 14
(3 + 2) * 4 -> 5 * 4 -> 20
```

The `&&` operator associates from left to right.
In the following assume `x` is 3, and `y` is 5

```
(x == 3) && (y == 5) && ( x == y) -> true && ( x == y ) -> false
```

which we can override with `()`

```
(x == 3) && ((y == 5) && ( x == y)) -> ( x == y) && false -> false
```

On the other hand assignment associates from right to left. Assume `y` is 5

```
x = y = 4 -> x = 4 -> 4
```

Widening Conversions

- Java will always do a widening conversion
- Converting a numeric data type to a wider version of the same type
- The numeric value is preserved exactly without any loss in precision
- Converting any integral data type to a floating point type is allowed
- There is no loss of magnitude, but there can be a loss of precision

Widening Conversions Example I

Widening Conversions

```
long longVar;
int  intVar;
short shortVar;
byte  byteVar;
char  charVar;
float floatVar;
double doubleVar;

byteVar = 120;
shortVar = byteVar;
intVar = shortVar;
longVar = intVar;
System.out.println("LongVar is "+ longVar); // value is 120L
```

Widening Conversions Example II

Widening Conversions -- integral to floating point

```
long longVar;
float floatVar;
double doubleVar;

longVar = Long.MAX_VALUE;
floatVar = longVar;
doubleVar = longVar;
System.out.println("longVar is "+ longVar);
System.out.println("floatVar is "+ floatVar);
System.out.println("doubleVar is "+ doubleVar);

// Output is
longVar is 9223372036854775807
floatVar is 9.223372E18
doubleVar is 9.223372036854776E18
```

Narrowing Conversions and Casting

- Narrowing conversions are the opposite of widening conversions
  - Converting a data type to a smaller version of the same type
  - Converting from a floating point data type to an integral data type
- Java does not perform narrowing conversions automatically
- The data must be cast to the narrower type
  - A cast operator is represented as a new data type name in parentheses placed before the variable or expression to be cast
  - variable2 = (new_type) variable;

Narrowing Conversions and Casting Example

Narrowing conversions

```
byte byteVar = (byte)255;
short shortVar = (short)214748360;
int intVar = (int) 1e20F;
int intVar2 =(int)Float.NaN;
float floatVar = (float)-1e300;
float floatVar2 = (float)1e-100;

System.out.println("(byte)255 -> " + byteVar);
System.out.println("(short)214748360-> "+ shortVar);
System.out.println("(int)1e20f -> " + intVar);
System.out.println("(int)NaN -> " + intVar2);
System.out.println("(float)-1e300 -> " + floatVar);
System.out.println("(float)1e-100 -> " + floatVar2);

// produces the output
(byte)255 -> -1
(short)214748360-> -13112
(int)1e20f -> 2147483647
(int)NaN -> 0
(float)-1e300 -> -Infinity
(float)1e-100 -> 0.0
```

String Conversions

- Any primitive data type can be converted to a ***`String`***
  - By implicit ***`String`*** conversion
  - For each primitive data type, we can use a ***`toString()`*** function found in "wrapper" classes to convert the value to a ***`String`***
- We can also convert from a ***`String`*** to various data types
  - This is more complicated because not every string of characters can be converted to a particular primitive data type
  - We will deal with handling this problem later

String Conversions Example

Converting to and from strings

```
String s = "123";
String t;

int k = 123;
float f = 19.801F;

//Integer and Float are “wrapper” classes
t = Integer.toString(k);
t = Float.toString(f);

// This line will not compile, you can't convert from a
// String this way
k = (int)s;
```

Forbidden Conversions

According to the Java language specification:

- No conversion from any reference type to any primitive type
- Except for the ***`String`*** conversions, no permitted conversion from any primitive type to any reference types
- No permitted conversion to ***`boolean`*** types
- No permitted conversion from ***`boolean`*** other to a ***`String`*** conversion

Summary

We Covered

- What operators and expressions are and how operators are used to create expressions
- The operators in Java, the kinds of data they operate on and what types of expressions they produce
- The difference between narrowing and widening operators
- Using the cast operator correctly

Lab 2

- Write a program that counts in a loop so that the output starts at 5,and then counts by 3 to 20. Do this using a for loop, and with a while loop.
- Using methods:
  - Write a program that declares and creates two arrays of integers, of different sizes. Initialize both arrays with random integers between 0 and 5000.
  - Create a method called **addToAll** that takes an integer array and an int as arguments. The method should add the **int** argument to each element of the array argument.
  - Remember to make your method **static**.
  - Call the method with each of your arrays, and print out their contents after the call to make sure the method works.

Lab 2 contd.

- Using Arrays:
  - Write a program that declares an array of Strings. Declare a numeric value "language", and two constants, ENGLISH, and SPANISH with the values 0, and 1, one respectively.
  - Use the value of "language" to control a switch/case statement that initializes the elements of the array with the names of the numbers, from "zero" to "ten", or in Spanish (“uno”, “dos”, “tres”, “quatro”, “cinco”, “seis”, “siete”, “ocho”, “nueve”, “diez”)
  - Iterate over the loop printing output that looks like: *The name of 5 is five*