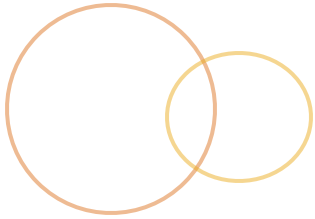
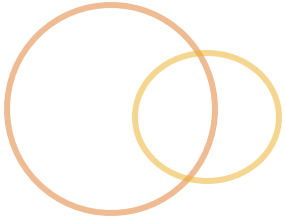
Variables, Operators and Data





Objectives

At the end of this module you should be able to:

- Describe the rules for creating legal variable names in Java
- Describe and use the basic primitive data types in Java
- Use `String` data
- Determine the data type of a literal

Strong Typing in Java

- Java is a strongly typed language
  - Each variable and each expression has a type
  - Can be identified by the compiler at compile time
  - A variable's type cannot be changed
- In loosely typed languages, like JavaScript & VB

Example 2-1: Loose variable typing in JavaScript

```
// JAVASCRIPT:  This is not allowed in JAVA!!!
// Declare a variable "x" with no type
var x
x = "Hi there"    // x is holding string data
x = 1234          // x is now holding numeric data
y = x + "343"     // String or numeric operation??
```

- Strong typing helps prevent errors

Data Types in Java

- There are two types
  - Reference data
  - Primitive data

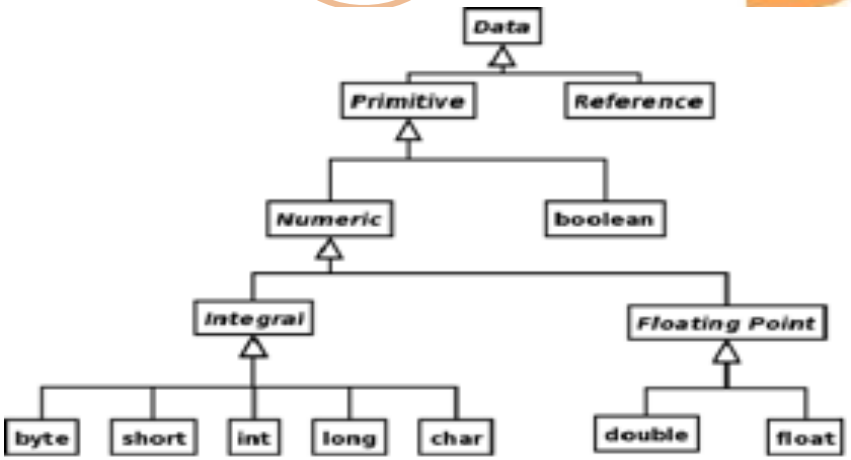


Fig. 3-1: Hierarchy of data types in Java

- The primitive data types resemble the types in C / C++
- Data types in Java are defined by the language specification
  - They are platform independent
  - For example, the data type `int` is *always* four bytes long

Identifiers

- Identifiers are used to describe
  - Classes
  - Variables
  - Method
- Identifier rules are platform independent
  - Must be an arbitrarily long sequence of letters and digits
  - Are case sensitive
  - The first character must be a letter
    - Any valid letter in the Unicode character set
    - The underscore "_" and dollar sign "$" are considered to be letters while symbols like © and + are not
  - They may **not** contain any white space
  - They may **not** be the same as reserved Java keywords

Reserved Keywords

Reserved Java keywords

```
abstract     default  if             private     this
boolean      do   implements     protected   throw
break        double   import         public  throws
byte     else instanceof     return       transient
case     extends  int            short       try
catch        final    interface      static      void
char     finally  long           strictfp    volatile
class        float    native         super       while
const        for      new            switch      true
continue     goto     package        synchronized
false        null
```

Fig. 2-2: **Reserved Java Keywords**

`goto` and `const` are reserved but yet to be used

Variable Names

Example: Valid and invalid identifiers in Java

```
Account_Balance // valid - remember that _ is a letter
$34             // valid - remember that $ is a letter
this      // invalid - same as reserved keyword
π           // valid - Greek letter pi is a Unicode letter
This        // valid - different in case from 'this'
Next.item       // invalid - symbol "." not allowed
23skidoo        // invalid - must start with letter
```

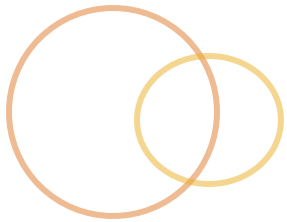
Declaring a Variable

- Variables are declared in Java with the syntax

```
type name [= initial_value];
```

- It is good programming practice to *initialize variables when they are declared*
- A variable can also be initialized after it is declared
- Java will prevent the use of uninitialized variables in code

Declaring a Variable

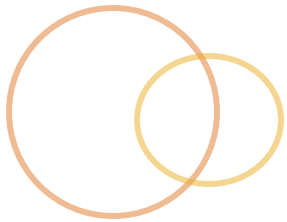


Example 2-3: Initializing variables

```
String best = "Best"; // Preferred - initialized at declaration
String okToo;     // Declared - not initialized
int x;         // Declared - not initialized

okToo = "value";      // Now okToo is initialized.
x = x + 1;         // ERROR! Use of an uninitialized variable
```

Declaring a Variable



Example 2-4: Initializing multiple variables

```
boolean a,b;                // a and b are both of type boolean
boolean c = true, d = false; // initialization for both c and d
boolean e,f = true;         // WARNING! Only f is initialized!
```

Example 2-5: Variables in memory

```
int var1;
boolean var2 = true;
var1 = 9
```

Positioning Variable Declarations

- The basic principal in Java, and in OOP in general, is to declare a variable at its point of first usage

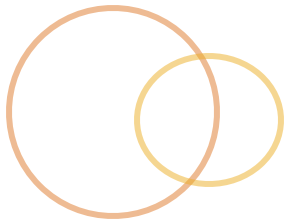
Example 2-6: Positioning variable declarations in code

```
class Test {
  public static void main(String [] args) {
    int sum = 0;
    for (int counter = 0; counter < 10; counter++)
      sum = sum + counter;
     String message = "The sum is ";
     System.out.println(message + sum);
    }
  }
}
```

Boolean Data Types

- `boolean` data is either `true` or `false`
- In C and C++, `boolean` values are numeric data
  - i.e. Zero is false, any non-zero is true
- In Java, `boolean` variables are not numeric
- Can have only the values `true` or `false`

Boolean Data Types

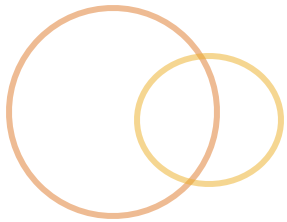


Example 2-7: boolean data in C+ and Java

```
// In C++ you do can this:
int x = 43;
// non-zero x is taken as a true
if (x) {
  System.out.println("x is "+x);
}
// In Java, the above code does not compile.  You need a
// boolean variable or expression.

boolean test = (x==43);
// boolean variable is OK
if (test) {
  System.out.println("x is "+x);
}
// OK because result of test is boolean
if (x == 43) {
  System.out.println("x is "+x);
}
```

Integral Data Types



- All integral values are signed - there are no unsigned values
- Integral values do not contain a decimal point

Type	Bytes	Minimum Value	Maximum Value
`byte`	1	-128	127
`short`	2	-32768	32767
`int`	4	-2147483648	214748364
`long`	8	-9223372036854775808	9223372036854775807

Fig 2-4: **Integral Data Types**

Integral Literals

- A sequence of digits without a decimal point is assumed to be integral data
  - literals are of type `int` unless there is an `L` - upper or lowercase - immediately following the digits
  - In this case, the literal is taken to be a `long`
  - `7836` and `-98` are `int`
  - `881L` and `-9112l` are `long`
- Literals are interpreted in Base 10 unless
  - The literal starts with a `0`, it is interpreted as Base 8
  - The literal starts with a `0x` or `0X`, it is interpreted as Base 16

Integral Literals

Example 2-8 Integral literals

```
63 // an int in base 10
-63    // a negative int in base 10
63L    // a long in base 10
063    // an int in base 8 (equal to 51 in base 10)
063l   // a long in base 8
-063l  // a negative long in base 8

091    // illegal!  Cannot have the digit 9 in base 8!

0x33   // an int in base 16 (equivalent to 51 in base 10)
0X33L  // a long in base 16
0xFF   // an int in base 16
0xff   // same as the previous line - case does not matter.
0xg1   // illegal! Can only have a-f as base 16 digits.
-0xFF  // an negative int in base 16
```

Floating Point Data Types

- Floating point are numerical values with fractional parts.
- Two kinds of floating point numbers
- `float`: 4 bytes long
  - Largest `float` is 3.4028234 E +38
  - About 6 or 7 significant digits
- `double`: is 8 bytes long
  - Largest `double` in magnitude is 1.7976931348623157 E +308
  - About 15 significant digits

Infinities, Negative Zeros and Non-numbers

Example 2-9: Test class for positive infinity

```
class InfinityTest {
  public static void main(String [] args) {
       // Set up bigd as a large double
       double bigd = 1e306;

   // loop - we should see bigd overflow about the third iteration
       for (int i=1; (i<100) && (bigd<Double.POSITIVE_INFINITY); i++) {
     System.out.println("Iteration="+ i +": bigd="+bigd);
     bigd = bigd * 10.0;
   }//end for loop

  }//end main

}//end class
```

Floating Point Literals

Example 2-11: Floating point literals

```
38.0       // double
38.0f      // float
38.98D     // double
1.78e23    // double
1.78e23f   // float

-789.983   // double
-1.89e-17F // float
```

Character Data

- A kind of integral data
  - The type name is `char`
  - Takes on values from 0 to 65535
- Java supports the Unicode standard - each character is stored as a two-byte representation
- ASCII is a subset of Unicode - Java handles the ASCII/Unicode conversions behind the scenes

Character Literals

- Character literals usually represent a single Unicode character in single quotes
- Character literals can also be the numeric code for Unicode characters
  - Unicode escape sequence notation
  - '`\udddd`' where `dddd` is the hexidecimal representation of the Unicode character
- Certain common non-printable characters, as well as the single and double quote and backslash, have special escape sequences that are recommended for use instead of the corresponding Unicode escape sequence

Character Literals

Example 2-12: Character literals

```
'a'        '7'     'ξ'     '©'     ' '
'\''       '\\'        '\u0F34'    '\u0004'    '\ffd1'
'_'
 ‾'

'ab'       // Not a char literal - two characters between quotes
'\ug189'   // Not a char literal - illegal Unicode code.
```

***Fig 2-5* Unicode Escape Sequences**

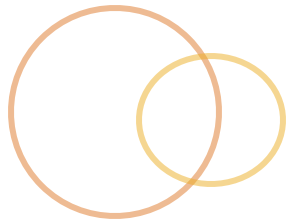
```
   '\b'            /* \u0008: backspace BS */
   '\t'            /* \u0009: horizontal tab HT */
   '\n'            /* \u000a: linefeed LF */
   '\f'            /* \u000c: form feed FF */
   '\r'            /* \u000d: carriage return CR */
   '\"'            /* \u0022: double quote " */
   '\''            /* \u0027: single quote ' */
   '\\'            /* \u005c: backslash \ */
```

Strings

- There is no string primitive data type in Java
  - `String`s are actually a reference data type that is implemented in the Java SE APIs
  - Java allows `String` data to be used syntactically as if it were a primitive data type
  - Intended to make working with character strings more "programmer friendly”

- `String` literals are sequences of Unicode Characters
  - Enclosed in double quotes
  - `char` escape sequences are valid for `String`

Chars and Strings



Example 2-13: Using Character data

```
char a = 'a';     // single character
char b = 'b';     // single character
char nl = '\n';   // using the non-printable escape code for newline
char x = '\u7878';    // using the Unicode escape sequence
a + b;         // the result is an int.
```

Example 2-14: Strings

```
String s = "This is a string";
s = "This is a string with a backspace \b in it";
s = "This is a string with a \" double quote inside";
String t = s;
t = "";    // this is the empty string - still a string though
t = 'a';   // illegal! 'a' is not a string.
```

Arrays

- An array is a data structure that holds multiple values of the same type
  - The values of an array are called the array *elements*
  - They are accessed by index or their numerical position from the start of the array
  - In Java, all arrays are zero-based which means that the index of the first position is 0
  - Initialized arrays have an intrinsic attribute describing the size - `length`
- The easiest way declare and initialize an array:

```
data_type [] array_name = { list, of, initial,
  values };
```

Creating Arrays Example

Example 2-21: Creating arrays

```
class ArrayTest {
  public static void main(String [] args) {
    int [] bob = {9,78,-3,0,89 };
    String [] a = {"black", "brown", "white",
                   "green", "blue", "brown"};
  }
}
```

The Array a

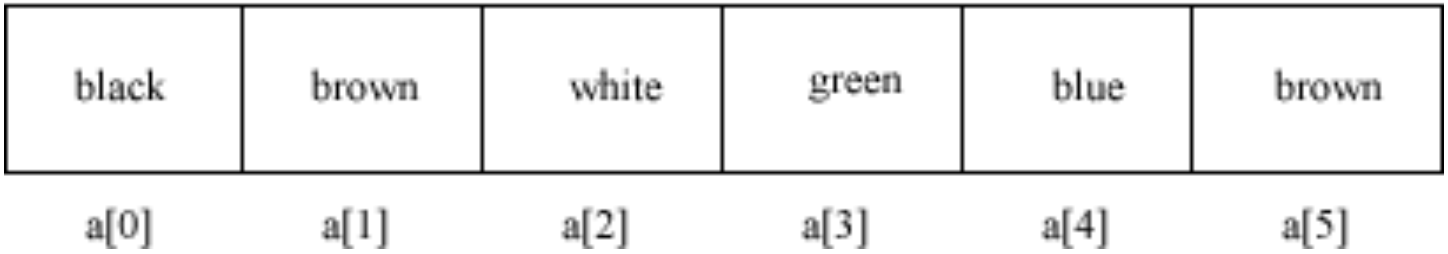


Fig. 4-1: **Array from Example 2-21**

Using Arrays Example

Example 2-22: Using arrays

```
class ArrayTest2 {
  public static void main(String [] args) {
    int [] bob = {9,78,-3,0,89 };
    String [] a = {"black", "brown", "white", "green",
                                          "blue", "brown"};
    int index = 0;
    while (index < a.length) {
      System.out.println("a["+index+"] ->"+a[index]);
      index++;
    }
    index = 0;
    while (index < bob.length) {
      if (index == 3 || index == 2) bob[index]=9999;
      System.out.println("bob["+index+"] ->"+bob[index]);
      index++;
    }
  }
}
```

Summary

We covered

- The rules for creating legal variable names in Java
- Describe and use the basic primitive data types in Java
- Use `String` data
- Determining the data type of a literal

Exercises

- *Exercise 2-1: Working with Variables*
  - *In this exercise you will*
    - *work with variables*
    - *determine legal identifiers*
- *Exercise 2-2: Working with Data Types*
  - *In this exercise you will*
    - *work with data types*
    - *investigate legal values*

Variables, Operators and Data

Part II

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

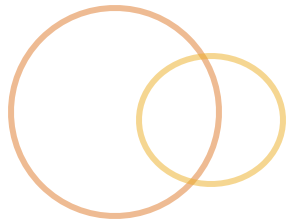
Operators and Expressions (cont.)

- 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.

Example 2-15: 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
  - Mixed mode arithmetic means that 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 may produce a
- loss of precision that can show up at odd times. This is due to
- rounding of floating point values

Example 2-16: 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

Example 2-17: 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 uses the C/C++ increment and decrement operator
  - ++
  - --
- The are two forms
  - Postfix - <VAR>++ and <VAR>--
  - Prefix - ++<VAR> and --<VAR>
- In the postfix form the value of the variable is used in the expression first, then incremented or decremented
- In the prefix for the value of the variable is incremented or decremented first, then used in the expression

Increment and decrement operators in Java

```
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

Example 2-18: 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

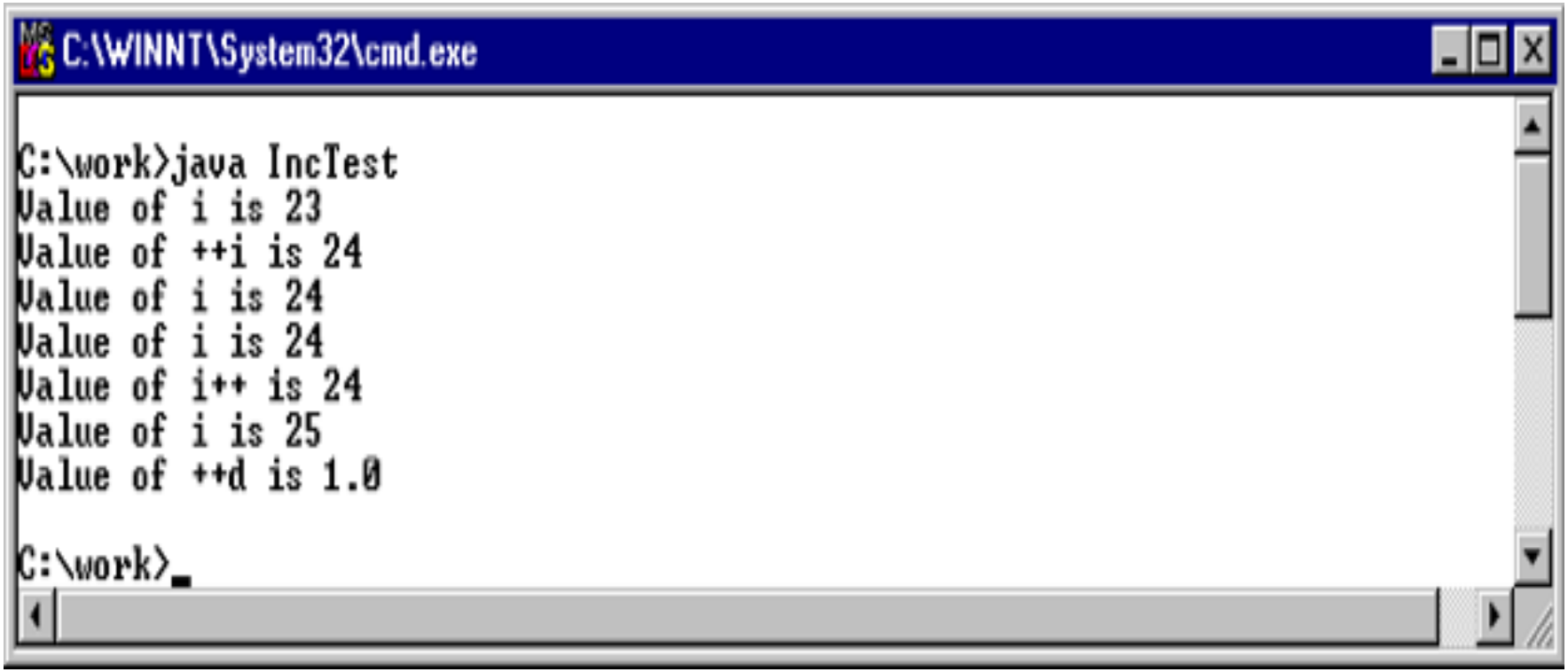


Fig. 3-5: **Output of the IncTest in Example 2-18**

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

Example 2-19: Relational operators and floating point numbers

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

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

Example 2-20: 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 does allow C/C++ style assignment operator notation

Example 2-21: 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
- All other operands are 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

Example 2-22: 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

Fig 2-6: **Operator precedence in Java**

Operator Precedence and Associatively Example

Example 2-23: 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

Example 2-24: 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

Example 2-25: 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

Example 2-26: 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

Example 2-27: 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

- *In this Lab you will*
  - *investigate the use of operators*
  - *solve problems with existing code*

- *In this Lab you will*
  - *work with casting*
  - *determine how to fix illegal conversion*