- Boolean Expressions
- Expressions and Operator Precedence
- Interactive Input to Java Programs
- Coding Style, Errors, Debugging
- Statement Flow Control

---

**Boolean Values**

- Recall `boolean` data type
  - May only have the values `true` or `false`
  - `true` and `false` are constants, just like numerical constants
- Comparison operators result in a boolean value
  - “Equal to” is the operator: `==`
  - “Not equal to” is the operator: `!=`
  - Less than: `<`  Less than or equal to: `<=`
  - Greater than: `>`  Greater than or equal to: `>=`
- Example
  ```java
  int n, m;
  boolean same = (n == m);
  boolean different = (n != m);
  ```
Boolean Operators

- Boolean values can be combined using logic operators to form other boolean values.
- Logical **negation (not)** operator: !
  - Applies to a **single** boolean value.
  - Value of operation is true if operand is false.
  - Value of operation is false if operand is true.
  ```
  boolean opposite = ! answer;
  ```
- Logical **conjunction (and)** operator: &&
  - Applies to **two** boolean operands.
  - Value is true only if both operands are true, otherwise false.
  ```
  boolean both = test1passed && test2passed;
  ```
- Logical **disjunction (or)** operator: ||
  - Applies to **two** boolean operands.
  - Value is false only if both operands are false, otherwise true.
  ```
  boolean atLeastOne = test1passed || test2passed;
  ```

- Logical **exclusion (exclusive or)** operator: ^
  - Applies to **two** boolean operands.
  - Value is true only if exactly one operand is true, otherwise false.
  ```
  boolean justOne = test1passed ^ test2passed;
  ```

Truth table for boolean operators:

|   |   | !a | a&&b | a||b | a^b |
|---|---|----|------|------|-----|
| true | true | false | true | true | false |
| true | false | false | true | true | true |
| false | true | true | false | true | true |
| false | false | false | false | false | false |
Boolean Expressions

- True if a number is even and positive
  - \((\text{num} \mod 2 == 0) \&\& (\text{num} > 0)\)
- True if the three numbers are in ascending order
  - \((\text{num1} <= \text{num2}) \&\& (\text{num2} <= \text{num3})\)
- True if either two or three divide a number, but it cannot be zero
  - \(((\text{num} \mod 2 == 0) || (\text{num} \mod 3 == 0)) \&\& (\text{num} != 0)\)
- True if neither three nor five divide a number
  - \(!(\text{num} \mod 3 == 0) || (\text{num} \mod 5 == 0))\)
- True if only one number divides the other
  - \((\text{num1} \mod \text{num2} == 0) \^ (\text{num2} \mod \text{num1} == 0)\)

.assignment

Assignment Revisited

- The right side of an assignment can be an expression involving one or more operators
  - May even use variable being assigned
  - \(\text{int i} = 7;\)
    - \(\text{i} = 6 * \text{i}; // \text{Now i is 42}\)
- Compound assignment operators are shorthand for an operation and assignment
  - \(\text{price} *= .75; // \text{Same as price} = \text{price} * .75\)
  - \(\text{hours} += 12; // \text{Same as hours} = \text{hours} + 12\)
  - \(\text{a} += \text{b} * \text{c}; // \text{Same as a} = \text{a} + (\text{b} * \text{c})\)
- Compound assignment operators
  - \(+= \ -= \ *= \ /= \ %=\)
Increment and Decrement

- A common operation is to increase (or decrease) a counter by one
  - \( i = i + 1; \)
  - Or, using compound assignment: \( i += 1; \)
- Java has a special increment operator which is equivalent
  - \( ++i; \) // Statement has same effect as \( i = i + 1; \)
  - Similarly \( --i; \) is equivalent to \( i = i - 1; \)
- Increment operator is an expression whose value can be used
  - `int i = 4, a = 3 * ++i;`
    - `a` will have the value 15, and `i` will have the value 5
- Increment and decrement may also be used in postfix form
  - In this case, the increment or decrement is still done, but the value used in the expression is the original value
  - `int i = 4, a = 3 * i++;`
    - `a` will have the value 12, and `i` will have the value 5
- `++` and `--` permit compact notation for complex expressions
  - But used too much, they can make code hard to read

Operator Precedence

- How does an expression like `x = a + b * c;` evaluate in Java?
  - Multiplication is done "first", then addition
  - Same rules as you learned in middle school
- Operators in Java have precedence levels
  - Precedence determines grouping, i.e., as if you had used parentheses
- Associativity also affects grouping
  - Associativity is left to right for everything but assignment
  - E.g., addition and subtraction have the same precedence level, but they are left associative, so \( a + b - c \) is the same as \( (a + b) - c \)
  - But \( a += b -= c \) is the same as \( a += (b -= c) \)
### Operator Precedence Table

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --</td>
<td>var++, var-- (Postfix)</td>
</tr>
<tr>
<td>+, -, ++, --</td>
<td>Unary plus, minus, ++var, --var (Prefix)</td>
</tr>
<tr>
<td>(type)</td>
<td>Casting</td>
</tr>
<tr>
<td></td>
<td>Not</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Multiplication, division, remainder</td>
</tr>
<tr>
<td>+, -</td>
<td>Binary addition and subtraction</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>Comparison</td>
</tr>
<tr>
<td>==, !=</td>
<td>Equality, Inequality</td>
</tr>
<tr>
<td>&amp;</td>
<td>Unconditional (bitwise) AND</td>
</tr>
<tr>
<td>^</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td></td>
<td>Unconditional (bitwise) OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Conditional AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+=, -=, *=, /=, %=</td>
<td>Assignment operators</td>
</tr>
</tbody>
</table>

### Evaluation Order

- **In what order do we do the operations in** `a*b + c*d`?
  - Precedence groups the two products, so `a*b` and `c*d` must be computed.
  - Certainly these products must be computed before we can compute the sum.
  - But which is done first? `a*b` or `c*d`?
  - Who cares – we’ll get the same answer either way.

- **But operators can have side effects (like `++`, `--`, assignment)**
  - Is order important in `a++ + (b=a)`?
  - Yes, suppose we start with 2 for `a`, and 3 for `b`. If we do the `a++` first then `a` becomes 3, and `b` becomes 3. If we do `b=a` first, then `b` becomes 2, and `a` becomes 3.

- **Java guarantees left to right evaluation order**
  - But expressions like the previous example are still not a good idea since they are hard to read and understand.
  - Other similar languages (like C or C++) may not guarantee the order of evaluation.
Evaluation Order

What is the order of evaluation in the following expressions?

- \(a + b + c + d + e\)
  - 1. 2. 3. 4.

- \(a + b \ast c - d / e\)
  - 3. 1. 4. 2.

- \(a / (b + c) - d \% e\)
  - 2. 1. 4. 3.

- \(a / (b * (c + (d - e)))\)
  - 4. 3. 2. 1.

- \(a =+ b = c = d = e\)
  - 4. 3. 2. 1.

- \(c = a++ + (b = a)\)
  - 4. 1. 3. 2.

Input to Java Programs

- Interactive input makes most programs more interesting
  - Avoids hard coding of values
  - Program works with different input values without re-compiling

- Ways for a program to get input
  - Characters typed at keyboard
  - Mouse clicks and movements
  - Data read from file
  - Data read from a device, e.g., modem, ipod, etc.

- We will look at character input from keyboard
Input using Scanner

- Easy way for a program to obtain input
  - Does not use a graphical interface
  - Gets "console" input – text typed in a command prompt window
  - Versatile, e.g., input could be redirected to come from a file or from another program
  - Easy to parse input and perform conversions
- Uses **Scanner** class (Java 1.5 and later)
  - **Scanner** object is created
  - Various "next" methods are used to get input
  - Think of input as being stream of words to be handled
- Methods to get input
  - **next()** gets a String (a "word" marked by spaces)
  - **nextInt()** gets an integer (string of digits, converted)
  - **nextDouble()** gets a double (string of digits, ., etc., converted)
  - and more ... (see Scanner API documentation)

ScannerTest.java

Input from Graphical Dialogs

- Generally, we will deal with "command line" programs, i.e., programs that run in a character based command prompt environment
  - Simple execution model – program starts from a shell, gets input from keyboard, sends output to command prompt
  - Command prompt in Windows, shell on Mac
- But even command line programs can pop up dialogs to accept typed input
  - Dialogs are a graphical interface
  - Our programs will not otherwise be graphical
- Graphical dialogs involve a lot of details
  - Creation of new windows
  - Painting of bitmapped images on screen
  - Responding to keyboard input and mouse clicks
- Fortunately, the Java library provides methods to hide all these details and make it easy to use a dialog
Input Dialog

- An input dialog has several attributes:
  - Title, Icon, Prompt, Text box, buttons

- What form of input can we expect from an input dialog?
  - We type in characters from the keyboard, so the input is a bunch of characters
  - Thus, we get a String from the dialog
  - If the cancel button was clicked, then we get nothing, which is different from not typing anything and clicking OK

- The input dialog method returns a String
  - The String is the characters that were typed
  - null is returned if cancel was chosen

How to code an Input Dialog

```java
String response = JOptionPane.showInputDialog( null,
    "Enter a temperature:",
    "Temperature to convert",
    JOptionPane.QUESTION_MESSAGE);
```

InputDialog.java
Converting Strings

- Sometimes we want to interpret the input as a numerical value instead of a string
  - But the user can only type characters (although some characters may be digits)
  - And the input dialog can only return a String
- Method to convert a String to an integer
  - Method `Integer.parseInt`
  - Takes a String, returns an integer
  - Analyzes characters, and returns corresponding numerical value
  - Not at all the same as casting
- Method to convert a String to a double
  - Method `Double.parseDouble`

Case Study: Payment Calculator

- Problem: compute monthly payments for a loan
- Useful formula for payment:

\[
L \times \left( r + \frac{r}{(1 + r)^n - 1} \right)
\]

- \(L\) is original loan amount, \(n\) is number of payments, \(r\) is interest rate for payment period
- Strategy for program to solve problem
  - Use `Scanner` to get loan amount, annual interest, length in years
  - Calculate number of monthly payments (12 months per year)
  - Calculate monthly interest rate
  - Use formula to calculate monthly payment
  - Output result
  - Use `Math.pow` to handle exponent in formula
Formatting Output

- System.out.print automatically formats integers and doubles
  - Precision of doubles may not be what we want
  - Format objects can be created for locale specific output and other customizations
  - This model generalizes to other output types (e.g., dates, money)
- New in Java 5: System.out.printf provides simple formatting
  - Uses a control string containing %d, %f, etc. with precision specifications
  - Additional parameter expressions must be provided to match all the % controls
  - Will generate exceptions if missing expressions or mis-matched types
  - Can also use to pad with leading zeroes, blanks
- LoanPayment2.java

Coding Style

- How a program looks is important
  - Easier to read and understand
  - Easier to spot errors
  - Reflects organized design and thinking
- Include appropriate comments
- Use indentation to show structure
  - Blocks and nested blocks
  - Use white space to increase readability
  - Align closing brace with block beginning
- Naming conventions
  - Use meaningful class, variable, and method names
  - Capitalize class names
- Designing and coding a program is a creative and artistic activity
Programming Errors

- **Compiler Errors**
  - Error messages from the compiler are syntax or semantic errors
  - The compilation fails, and no executable program is produced
  - Examples of syntax errors:
    - missing semi-colons, braces, parentheses
    - improper use of keywords, illegal use of operators
  - Examples of semantic errors
    - Variable used but not defined
    - Assignment of double to int without cast
    - Missing import causes undefined class
  - One error (e.g., a missing brace) may lead to many others
    - Sometimes it's best to fix first few, then compile again

- **Runtime Errors**
  - Program compiles successfully, but throws exception when run
  - Some illegal condition has occurred that means the program cannot continue to safely execute
  - Examples:
    - Division between two variables where denominator value is zero
    - Attempt to use parseInt on a String that is not all digits
    - No static main method in class being executed
  - Fix runtime errors by coding in logic to avoid those situations

- **Logic Errors**
  - Program compiles and runs, but produces incorrect results
  - Code is okay, but does not do what you intended
  - Fix by changing code to correspond to what you want to do
Debugging

- IDEs (Integrated Development Environments) provide sophisticated debugging
  - Display values of variables
  - Trace execution
  - Set breakpoints to stop at critical places

- For small programs
  - Use print statements to display crucial values
  - Use print statements for tracing (e.g., does program get here?)
  - Comment out blocks of code to isolate problems
  - Edit, compile, and test programs incrementally
    - Start with small amount of working code and add to it

if Statements

- The code of a program is a sequence of statements, executed in order
- The if control flow construct allows a statement or block of statements to be executed or not, according to a condition

- Syntax of if statement
  - `if` followed by a boolean expression enclosed in parentheses
  - A block of statements, optionally with braces, to be executed only if the condition is true

```plaintext
keyword if [condition] statements
```

```plaintext
a boolean expression
```

```plaintext
parentheses required
```

```plaintext
braces required if more than one statement
```
Flow chart of if statement

```
n = 17; m = 3;

if (n % m == 0) {
    System.out.println(m + " divides " + n);
}
```

Divides.java

**if-else Statements**

- Sometimes we want to select between two choices of code to execute.
- The **if-else** control flow construct selects between two statements (or blocks) executing one or the other (but not both), according to a condition.

**Syntax of if-else statement**

```
if (condition) {
    statements;
} else {
    statements;
}
```

Keyword: `if` - code to execute only if condition is true.
Keyword: `else` - code to execute only if condition is false.
Flow chart of if-else statement

```java
n = 17; m = 3;

if (m > n) {
    biggest = m;
} else {
    biggest = n;
}
```

Bigger.java

Nested Statements

- A block of statements can appear in any place where you could code a single statement
- An if construct and an if-else construct are statements themselves
  - E.g., an if can control an if-else statement
  - Braces may be important to get the correct control logic
  - Each else must be matched with an if
  - An else matches to the "closest" if (respecting braces)
- When the else clause is an if or if-else, we use the "else if" style
  ```java
  if (m > n)
      System.out.println(m + " is bigger");
  else if (n > m)
      System.out.println(n + " is bigger");
  else
      System.out.println("they are the same");
  ```
Conditional Evaluation

- All evaluation is left-to-right, including logical operators `&&` and `||`
- But these operators also have the property of "short circuit" evaluation
  - For a logical AND, if the first term is false, then there is no reason to evaluate the second term, since the expression would still be false
  - For a logical OR, if the first term is true, then there is no reason to evaluate the second term, since the expression would still be true
- This is called **conditional** evaluation
  - More efficient – irrelevant code is not executed
  - Makes it easy to do defensive checks
    - Example: `if (n == 0 || m%n != 0) // m not divisible by n`
  - Or if there are side effects
    - Example: `if (i < s.length() && s.charAt(i++) == ' ')`

Conditional Operator

- An if-else statement in Java selects one of two **statements** to **execute**
  - Only one of the statements is executed
    - Simple example:
      ```java
      if (a > b) max = a;
      else max = b;
      ```
  - The conditional operator selects one of two **expressions** to **evaluate**
    - The value of the conditional operator is the selected expression value
    - Only one of the expressions is evaluated
    - Operator consists of two symbols, `?` and `:` - to separate 3 operands
    - This is the only operator in Java that takes 3 operands
    - Example
      ```java
      max = (a > b) ? a : b; // first expression, evaluated if condition is true
      ```
Loops

- Many problem solutions involve the repetition of a task
  - Calculate average by adding numbers one after the other to a sum, then divide by the count of numbers
  - Taking turns in a game
  - Finding an answer by approximation – make repeated "better" guesses
  - Put a list of numbers or names in order
- Programming languages have looping control flow constructs
  - Fundamental model is to repeatedly test and perform task, stopping when test condition is no longer true
- Java has three forms of loops
  - while, do-while, for

While Loop

- The while loop has a test condition and a body
  - Test condition is a boolean expression
  - Body is a single statement, or a block of statements
  - If the condition is true, the body is executed
    - Then the condition is checked again
  - Syntax of while statement
    - while (condition) { statements; }
Flow chart of while loop

```
howmany = 3; start = 5;
number = 7; sum = 18; count = 4;

while (count <= howmany) {
    number += 1;
    sum += number;
    count += 1;
}
System.out.println("Sum from " + start + \" to \" + number + \" is \" + sum);
```

RangeSum.java

Do-While Loop

- Similar to a while loop, but the condition is checked after the body of statements
  - Useful when we want to execute at least once
  - If the condition is true, the body is executed again
    - Then the condition is checked again
- Syntax of do-while statement
  ```
do {
    statements;
} while (condition);
```
  - Keywords: do, while
  - Code to execute before checking condition
  - If condition is true, statements are executed again
  - Semicolon required
Flow chart of do-while loop

```java
// Flow chart of do-while loop

// Guess.java

import java.util.Scanner;

public class Guess {
    public static void main(String[] args) {
        int target = 3;
        int guess = 3;
        do {
            System.out.print("Guess:");
            guess = scan.nextInt();
        } while (guess != target);
        System.out.println("Right!");
    }
}
```

For Loop

- Loops are often used to execute a block of statements a fixed number of times
  - We can do this with a while loop by initializing a counter and using a condition to check the counter, and incrementing the counter as the loop progresses
  - This is a common pattern:
    - Initialize some "counting" variable
    - Check that the counter is within the range
    - Execute the statements
    - Increment the counter
    - If the counter is within the range, repeat
  - The for loop construct makes this pattern easy to code
    - Could use while loop, but a for loop may make the logic more obvious
    - A for loop helps distinguish between the mechanics of the loop and the work that is to be done repeatedly
For Loop

The for loop has four parts:
- Initialization, which is only done once
- Condition, which is checked for each iteration
- Body which is executed if the condition is true
- Iterator (usually an increment), which is executed after the body, each time the body is executed

After body and iterator are executed, condition is checked again
Syntax of for statement

```
for (expression evaluated once, may be a declaration
    condition controlling the loop
    increment done after the statements to get to next iteration
) {
    code to execute only if condition is true
    body, which is executed if the condition is true
    statement
}
```

```
for (int count = 1; count <= howmany;++count) {
    sum += ++number;
}
System.out.println("Sum from "+ start + " to " + number + " is " + sum);
``

Flow chart of for loop

```
howmany = 3; start = 5;
number = 7; sum = 18; count = 4;
for (int count = 1; count <= howmany; ++count) {
    sum += ++number;
}System.out.println("Sum from " + start + " to " + number + " is " + sum);
```

RangeSum2.java
For Loop

- Each of the control parts of a for loop may be omitted
- If the initialization part is left out, there is no initial code to execute
  - Variables have values from previous statements
- If the condition is left out, the loop test is always true
  - An "infinite" loop, so there must be other code to terminate
- If the increment is left out, there is no increment to execute
  - There should be other code to advance the loop
  - In this case, we probably would use a while loop instead
- Multiple variables can be initialized and incremented
  - Can be useful, but may make loop hard to understand

More on Loops

- If condition never becomes false, the loop will execute "forever"
  - An infinite loop will continue until some program fault is reached
  - The program may need to be externally terminated
  - Make sure when coding a loop that you provide logic to advance the condition toward termination
  - Logic may be decrement, dividing by two, multiplying by 3, ...
    - Logic must guarantee that condition is eventually false
- Loops are a type of statement
  - A loop can be selected by an if or if-else
  - Loop bodies may contain if, if-else, or other loop statements
  - So loops may be nested
Break and Continue

- The **break** statement causes the early termination of a loop
  - Control flows immediately to the statement which is right *after* the loop
  - Syntax is: `break;`

- The **continue** statement jumps to the next iteration of the loop
  - Control flows immediately *back* to the loop control, skipping the rest of the statements of the loop
  - In a for loop, control skips to the increment, then to the condition
  - Syntax is: `continue;`

---

Flow of break and continue

```
int sum = 0;
for (int count = 0; count < 10; ++count) {
    System.out.print("Enter a number: ");
    if (!input.hasNext()) {
        System.out.println("End of input");
        break;
    } else if (!input.hasNextInt()) {
        String s = input.next();
        System.out.println("not a number");
        continue;
    } else {
        int number = input.nextInt();
        sum += number * number;
    }
}
System.out.println("sum is "+ sum);
```

SquareSum.java
Switch Statements

- If choice among multiple alternatives is based on an integral expression, a switch construct can be used
  - Controlling value could be byte, int, char
  - Choices are listed explicitly (no comparisons like < or >)
  - Keyword default is used for value not explicitly matched
- Switch acts like a table transferring control to the matching statement(s)
- Syntax of switch statement

```
switch (expression) {
    case value1:
        statements;
        break;
    case value2:
        statements;
        break;
    default:
        statements;
        break;
}
```

- keyword `switch` is required
- keyword `case` used
- value1: a constant integral value
- colon is required
- code to execute only if expression matches this explicit value
- matched only if no case matches
- keyword `break` to jump to end of case list

Flow control of switch statement

```
v = 0.0;
switch(type) {
    case 0:
        v = len * w * h;
        break;
    case 1:
        v = 5 * len * w * h;
        break;
    case 2:
        v = (1.0/3) * len * w * h;
        break;
    default:
        System.err.println ...
        System.out.println("Volume is "+v);
}
```

- switch transfers into case list
- case 0 transfers control
- v = len * w * h;
- break;
- case 1 transfers control
- v = 5 * len * w * h;
- break;
- case 2 transfers control
- v = (1.0/3) * len * w * h;
- break;
- default transfers control
- System.err.println ...
- System.out.println("Volume is "+v);

- len = 3; w = 4; h = 5;
- type = 1;
- v = 30.0;

- normal sequential execution

- break transfers out of case list

- Volume.java