Flow of Control of Program Statements

- Java provides two types of program flow of control statements:
  - decision statements, or **conditional statements**: decide whether or not to execute a particular statement
  - repetition statements, or **loop statements**: perform a statement over and over repetitively

- With conditional and loop statements, we can write substantial programs

Basic if Conditional Statement

**Executes statement if test is true**

```java
if (test)
    statement
```

- **Example:**
  ```java
double gpa = console.nextDouble();
if (gpa >= 4.0)
    System.out.println("Welcome to Yale University!");
```

Basic Test: Relational Operators

- A basic test is to compare two values using a **relational operator**:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
<td><code>1 + 1 == 2</code></td>
<td>true</td>
</tr>
<tr>
<td>!=</td>
<td>does not equal</td>
<td><code>3.2 != 2.5</code></td>
<td>true</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td><code>10 &lt; 5</code></td>
<td>false</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td><code>10 &gt; 5</code></td>
<td>true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td><code>126 &lt;= 100</code></td>
<td>false</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td><code>5.0 &gt;= 5.0</code></td>
<td>true</td>
</tr>
</tbody>
</table>

- Note the difference between the equality operator (`==`) and the assignment operator (`=`)
Lesson: Comparing Real Numbers

- You should rarely use the equality operator (==) when comparing two real numbers
  - in many situations, you might consider two real numbers to be "close enough", e.g.,

    ```java
    final double EPSILON = 0.00001;
    ... if (Math.abs (f1 - f2) < EPSILON)
    System.out.println ("Essentially equal.");
    ```

Comparing Two Integers

- With if/else, to compare two integers a and b, and distinguish two cases:
  - a > b, a <= b

  ```java
  public static void compare(int a, int b) {
    if (a > b)
      System.out.println(a + " > " + b);
    else
      System.out.println(a + " <= " + b);
  }
  ```

Beyond basic if: Comparing two Integers

- If we had only if, to compare two integers a and b, and distinguish the two cases:
  - a > b, a <= b

  ```java
  public static void compare(int a, int b) {
    if (a > b)
      System.out.println(a + " > " + b);
    else if (a <= b)
      System.out.println(a + " <= " + b);
  }
  ```

  # comparisons: 2

Nested if/else

- Chooses among outcomes using many tests
- The statement executed as a result of an if-else statement could be another if statement
  - these are called nested if/else statements

  ```java
  if (case1)
    // do case 1
  else if (case 2) // not case 1, test case 2
    // do case 2
  else if (case 3) // not cases 1 and 2, test case 3
    // do case 3
  else // not cases 1, 2, 3
    // default case
  ```

# Comparison: 1
Comparing two Integers: Sequential if

- Write a method to compare two integers a and b, and distinguish three cases:
  - a > b, a < b, a == b

```java
public static void compare(int a, int b) {
  if (a > b) {
    System.out.println(a + " > " + b);
  } else if (a < b) {
    System.out.println(a + " < " + b);
  } else {  // a == b
    System.out.println(a + " == " + b);
  }
}
```

Q: How many comparisons?

Solution II: Nested Comparison

```java
public static void compare(int a, int b) {
  if (a > b) {
    System.out.println(a + " > " + b);
  } else if (a < b) {
    System.out.println(a + " < " + b);
  } else {  // a == b
    System.out.println(a + " == " + b);
  }
}
```

Q: How many comparisons?

Benefit of nested comparison: reduce # comparisons

Fix: Using nested comparison to achieve mutual exclusion.

Grading Curve

- How is this code for grading based on percentiles?

```java
Scanner console = new Scanner(System.in);
System.out.print("What percentile? ");
int percent = console.nextInt();
if (percent >= 90) {  // A
  System.out.println("You got an A!");
} else if (percent >= 80) {  // B
  System.out.println("You got a B!");
} else if (percent >= 70) {  // C
  System.out.println("You got a C!");
} else if (percent >= 60) {  // D
  System.out.println("You got a D!");
} else {  // F
  System.out.println("You got an F!");
}
```

Nested if/else/if

- If a nested if/else ends with else, exactly one case be chosen.
- If ends with if, the code might not choose any case.

```java
if (test) {
  statement(s);
} else if (test) {
  statement(s);
} else if (test) {
  statement(s);
}
```

- Example:

```java
if (place == 1) {
  System.out.println("Gold medal!");
} else if (place == 2) {
  System.out.println("Silver medal!");
} else if (place == 3) {
  System.out.println("Bronze medal!");
}
```
Nested if Structures

Exactly 1 case (mutually exclusive)
if (test) {
  statement(s);
} else if (test) {
  statement(s);
} else {
  statement(s);
}

0 or 1 case (mutually exclusive)
if (test) {
  statement(s);
} else if (test) {
  statement(s);
} else if (test) {
  statement(s);
}

0, 1, or many cases (independent tests; may not be exclusive)
if (test) {
  statement(s);
} if (test) {
  statement(s);
} if (test) {
  statement(s);
} if (test) {
  statement(s);
}

Matching Nested if Statements

- Nested if statements may have a matching problem

```java
if (temperature < 50)
  System.out.println("Cool!");
else
  System.out.println("Hot!");
```

Nested if w/ Ambiguity

Give a value of temperature to produce different results

- Rule: an else clause is matched to the last unmatched if (no matter what the indentation implies)
- If you find such statements confusing, avoid writing such structures and always use block statements ({}) to make the structure clear.
Summary: Basic if/else

```
<op1> <relational operator > <op2>
```

```java
if (case1)
// do case 1
else if (case 2) // not case 1, test case 2
// do case 2
else if (case 3) // not cases 1 and 2, test case 3
// do case 3
else // not cases 1, 2, 3
// default case
```

Example

- Design a method to compute the number of days in a month?

DaysInMonth1.java

Refinement I

- There can be many test cases producing the same result:
  - case 1: result 1
  - case 2: result 2
  - case 3: result 1
  - case 4: result 1
- No need to enumerate the cases one-by-one
  => merge the test cases together
  => use a logical OR to combine the cases

```java
if (m == 1 || m == 3 || m == 5 ||
    m == 7 || m == 8 || m == 10 || m == 12)
    numDays = 31;
```

Refinement II

- We may need to check complex logical conditions
  - a leap year condition: “… most years that are evenly divisible by 4 are leap years; However, there are some exceptions to this rule: Years that are evenly divisible by 100 are not leap years, unless they are also evenly divisible by 400”

```
year % 400 == 0 or
year % 4 == 0 and
year % 100 != 0
```

Testing Leap Year

- We may need to check multiple conditions

```
y % 400 == 0 || (y % 100 != 0 && y % 4 == 0)
```

Categories of Loops

- **definite loop**: Executes a constant number of times.
  - The for loops we have seen are mostly definite loops.
- **indefinite loop**: One where the number of times its body repeats is unknown in advance by the programmer, e.g.,
  - Prompt the user until they type a non-negative number.
  - Print random numbers until a prime number is printed.
  - Repeat until the user has typed “q” to quit.

Java Repetition /Loop Statements

- Java has three kinds of loop statements:
  - the *for* loop
  - the *while* loop
  - the *do-while* loop, and
- They have the same expressive power
- Which loop statements to use will depend on the situation, and we should choose the one which is more intuitive

```
Java while Loop

- **while loop**: Repeatedly executes its body as long as a logical test is true.
  ```java
  while (<test>) {
      <statement(s>);
  }
  ```
  while is a keyword
- Example:
  ```java
  int num = 1; // initialization
  while (num <= 200) { // test
      System.out.print(num + " ");
      num = num * 2; // update
  }
  // output: 1 2 4 8 16 32 64 128
  ```
The while Statement

- If the condition of a while statement is false initially, the statement is never executed.
- Therefore, the body of a while loop will execute zero or more times.

Example: while Loop

```
// finds the first factor of 91
int n = 91;
int factor = 2;
while (n % factor != 0) {
    factor++;
}
System.out.println("First factor is " + factor);
```

Programming Pattern: Sequential Scan

- Visit each element in a set, and update state variables.
  - Initialize states (e.g., sumSoFar, minSoFar, maxSoFar)
  - For each element:
    - Process element (e.g., update state such as sumSoFar, minSoFar, maxSoFar)

Programming Pattern: Early Exit

- Break out a loop before seeing all elements.
  - For each element:
    - Process element (e.g., checking if it is what we are looking for, if it is, terminate the loop)
- Typical in a search problem:
  - Find what you are looking for, i.e., an “evidence”
Sequential Scan and User Input

- Using repeat statements to process a stream of user input (e.g., command or data) is a quite common task.
- We first focus on collecting statistics from a stream of user data to, e.g.,
  - a sequence of grades
  - a sequence of words

- A programming human user interface decision issue: How does your program know how many data items?

User Input Control

- **Two input styles**
  - **Out of band** control protocol
    - User first specifies the number of data items
  - **In-band** control protocol
    - User finishes input by entering a **sentinel** value
      - e.g., -1 to signal the end of input grades
    - Why in-band sentinel: flexibility.
    - Implication: a data item just read can be either a real data item or the signaling sentinel

Input Style

- **sentinel**: A value that signals the end of user input.
  - **sentinel loop**: Repeats until a sentinel value is seen.

- **Example**: Write a program that prompts the user for text until the user types nothing, then output the total number of characters typed.
  - (In this case, the empty string is the sentinel value.)

Sentinel Values

- Type a line (or nothing to exit): hello
  - hello
  - Type a line (or nothing to exit): this is a line
  - You typed a total of 19 characters.
Solution

```java
int sum = 0;
String response = "dummy"; // "dummy" value, anything but ""
while (!response.equals("")) { // if not equal to empty string
    System.out.print("Type a line (or nothing to exit): ");
    response = StdIn.readLine(); // read a string from std input
    sum += response.length(); // the length of the string read
}
System.out.println("You typed a total of " + sum + " characters.");
```

Changing the Sentinel Value

- **Changing the sentinel's value to "quit"**

```java
int sum = 0;
String response = "dummy"; // "dummy" value, anything but "quit"
while (!response.equals("quit")) { // if not equal to "quit" string
    System.out.print("Type a line (or "quit" to exit): ");
    response = StdIn.readLine();
    sum += response.length();
}
System.out.println("You typed a total of " + sum + " characters.");
```

- **This solution produces the wrong output (should still be 19 characters). Why?**

A "Simpler" Problem...

- **Revisit the countDown method that prints from a given maximum (>=1) to 1, separated by commas.**

For example, the call:
```
countDown(5)
```

should print:
```
5, 4, 3, 2, 1
```

Previous “Solution”

- **public static void countDown(int max) {**
  ```java
  for (int i = max; i >= 1; i--) {
      System.out.print(i + " , ");
  }
  System.out.println(); // to end the line of output
  ```
- **Output from countDown(5):**
  ```java
  5, 4, 3, 2, 1
  ```

- **public static void countDown(int max) {**
  ```java
  for (int i = max; i >= 1; i--) {
      System.out.print(" , "+ i);
  }
  System.out.println(); // to end the line of output
  ```
- **Output from countDown(5):**
  ```java
  5, 4, 3, 2, 1
  ```
Fence Post Analogy

- We print \( n \) numbers but need only \( n - 1 \) commas.
- Similar to building a fence with wires separated by posts:
  - If we use a flawed algorithm that repeatedly places a post + wire, the last post will have an extra dangling wire.

```java
for (length of fence) {
    place a post.
    place some wire.
}
```

Fencepost Loop

- Add a statement outside the loop to place the initial "post."
  - Also called a fencepost loop or a "loop-and-a-half" solution.

```java
place a post.
for (length of fence - 1) {
    place some wire.
    place a post.
}
```

Many programmers feel “heart-broken” to see this "redundancy" not removed by loop.

Fencepost Method Solution

```java
public static void countDown(int max) {
    System.out.print(max);    // first post
    for (int i = max-1; i >= 1; i--) {
        System.out.print(i + ", " + i); // wire + post
    }
    System.out.println();      // to end the line
}
```

- Alternate solution: Either first or last "post" can be taken out:

```java
public static void countDown(int max) {
    for (int i = max; i >= 2; i--) {
        System.out.print(i + ", "); // post + wire
    }
    System.out.println(1);      // last post
}
```

Back to Counting w/ Sentinel

```java
int sum = 0;
String response = "dummy"; // "dummy" value, anything but "quit"
while (!response.equals("quit")) {
    System.out.println("Type a line (or " + "quit" + " to exit): ");
    response = StdIn.readLine();
    sum += response.length();
}
System.out.println("You typed a total of " + sum + " characters.");
```

Type a line (or "quit" to exit): hello
Type a line (or "quit" to exit): this is a line
Type a line (or "quit" to exit): quit
You typed a total of 23 characters.
The Problem with our Code

- Our code uses a pattern like this:
  ```java
  sum = 0;
  while (input is not the sentinel) {
    prompt for input; read input.
    add input length to the sum.
  }
  ```

- On the last pass, the sentinel’s length (4) is added to the sum:
  ```java
  prompt for input; read input ("quit").
  add input length (4) to the sum.
  ```

- This is a fencepost problem.
  - Must read \( N \) lines, but only sum the lengths of the first \( N-1 \).

A Fencepost Solution

```java
sum = 0;
prompt for input; read input. // place a "post"
while (input is not the sentinel) {
  add input length to the sum. // place a "wire"
prompt for input; read input. // place a "post"
}
```

- Sentinel loops often utilize a fencepost "loop-and-a-half" style solution by pulling some code out of the loop.

Fencepost Sentinel Loop

```java
public static final String SENTINEL = "quit";
public static final String PROMPT = "Type a line (or " + SENTINEL + " to exit):
public static Count() {
  int sum = 0;
  // pull one prompt/read ("post") out of the loop
  String response = getString(PROMPT, console);
  while ( !response.equals(SENTINEL) ) {
    sum += response.length(); // wire
    response = getString(PROMPT, console); // post
  }
  System.out.println("You typed a total of " + sum + " characters.");
}
public static String getString(String prompt, Scanner console) {
  System.out.print(prompt);
  return StdIn.readLine();
}
```

Summary

- place a post.
- for (length of fence) {
  place a post.
  place some wire.
}
- for (length of fence - 1) {
  place a post.
  place some wire.
}
Another Approach

for (length of fence) {
    place a post.
    place some wire.
}

for (length of fence) {
    place a post.
    if (last post)
        place some wire.
}

Detect if it is the last post in the loop, if it is, do not place the wire

Sentinel Loop Internal Test

int sum = 0;
int grade = 0;
while (grade != -1) {
    System.out.print("Enter a number (-1 to quit): ");
    grade = StdIn.readInt();
    if (grade != -1) {
        // detect the last post
        sum = sum + grade;
    }
}
System.out.println("The total was "+ sum);

Duplicated code
Duplicate execution
Do we need both tests?

Summary: Solutions

Scanner console = new Scanner(System.in);
int sum = 0;
int grade = 0;
while (true) {
    System.out.print("Enter a number (-1 to quit): ");
    grade = StdIn.readInt();
    if (grade == -1) {
        // detect the last post
        break;
    }
    sum = sum + grade;
}
System.out.println("The total was "+ sum);

int sum = 0;
int grade = 0;
while (true) {
    System.out.print("Enter a number (-1 to quit): ");
    grade = StdIn.readInt();
    if (grade == -1) {
        // detect the last post
        break;
    }
    sum = sum + grade;
}
System.out.println("The total was "+ sum);

int sum = 0;
System.out.print("Enter a number (-1 to quit): ");
int grade = console.nextInt();
while (grade != -1) {
    sum = sum + grade;
    System.out.print("Enter a number (-1 to quit): ");
    grade = StdIn.readInt();
}
Comments

- Choice between “sentinel loop with break” vs “non-break” is typically a personal choice.

- For many programmers, the “non-break” version is preferred, because:
  - the high-level structure (“topical sentence”) indicates that the loop is controlled by a sentinel.
  - the high-level structure (“topical sentence”) of the “break” version indicates that the loop is an infinite loop.

Summary: for Loop

- for loop is easy to read, since it provides visual aid to recognize the four components:
  - if the initialization, increment, or condition too complicated, a for loop may not read as well as a while or do/while loop.
  - typically used in sequential iteration.

Summary: while, do/while Loop

- The choice between do/while and while depends on the logic of the program:
  - first check condition or first execute statement.

```c
while (condition) {
    statement list;
}
do {
    statement list;
} while (condition);
```