A word about PowerPoint. PowerPoint was released by Microsoft in 1990 as a way to euthanize cattle using a method less cruel than hitting them over the head with iron mallets. After PETA successfully argued in court that PowerPoint actually was more cruel than iron mallets, the program was adopted by corporations for slide show presentations.

Conducting a PowerPoint presentation is a lot like smoking a cigar. Only the person doing it likes it. The people around him want to hit him with a chair.

PowerPoint is usually restricted to conference rooms where the doors are locked from the outside. It is, therefore, considered unsuited for large rallies, where people have a means of escape and where the purpose is to energize rather than daze.

- Roger Simon, Politico
Flow of Control

- Java branching statements:
  - if-else statement
  - switch statement
  - The Conditional Operator

- Loops:
  - while
  - for
  - repeat-until

- The type boolean

Reading:
=> Section 1.3
Flow of Control

- Flow of control is the order in which a program performs actions.
  - So far, the order has been sequential.

- A branching statement chooses between two or more possible actions.

- A loop statement repeats an action until a stopping condition occurs.
The *if-else* statement is one type of branching statement.

Syntax:

```java
if (Boolean_Expression)
    Statement_1
else
    Statement_2
```
Example:

```java
int age, numberOfBeers;

System.out.print("Enter age: ");
age = keyboard.nextInt();
if (age >= 21)
    numberOfBeers = 2;
else
    numberOfBeers = 0;
```
import java.util.*;
public class BankBalance {
    public static void main(String[] args) {
        double balance;
        Scanner kb = new Scanner(System.in);
        System.out.print("Enter your checking account balance: ");
        balance = kb.nextDouble();
        System.out.println("Original balance ");
        if (balance >= 0)
            balance = balance + (0.02*balance)/12;
        else
            balance = balance - 8.00;
        System.out.println("After adjusting for one month")
        System.out.println("of interest and penalties,");
        System.out.println("your new balance is ");
    }
}
The else clause is optional; in such a case, if the expression after the `if` is false, no action occurs.

Syntax:

```
if (Boolean_Expression)
    Statement
```
Example:

```java
int age, numberOfBeers;

numberOfBeers = 0; // Initial assumption
System.out.print("Enter Age:");
age = keyboard.nextInt();
if (age >= 21)
    numberOfBeers = 2;
System.out.println("Dont Drink and Drive!");
```
To include multiple statements in a branch, enclose the statements in braces.

```java
int x, total, count;
:
if (x < 10) {
    total = 0;
    count = 0;
}
else {
    System.out.println("today is Friday");
    System.out.println("tomorrow is Saturday");
}
```
To include multiple statements in a branch, enclose the statements in braces.

```c
int x, y;
:
if (x == y) { // Note the equality comparison operator
    x = x + 1;
    y = 0;
}
```
A common mistake:

```java
if (x == y)
    x = x + 1;
    y = 0;
```

Similarly:

```java
if (count < 10) {
    total = 0;
    count = 0;
}
else
    System.out.println(“today is Friday”);
    System.out.println(“tomorrow is Saturday”);
```

“White space” (Indentation, spacing, and tabs) have no impact on program execution!
Introduction to Boolean Expressions

- The condition in an if-statement (or loop) is a boolean expression.

- The value of a boolean expression is either true or false.

- Simple examples:

  ```
  time < limit
  balance <= 0
  ch == 'a'
  x >= y+z*w  // Note the role of precedence x >= y + z * w
  x != y      // Note the inequality operator
  ```
Boolean expressions are composed using individual “terms” which typically use Java Comparison operators.

<table>
<thead>
<tr>
<th>Math Notation</th>
<th>Name</th>
<th>Java Notation</th>
<th>Java Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
<td>==</td>
<td>balance == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>answer == 'y'</td>
</tr>
<tr>
<td>≠</td>
<td>Not equal to</td>
<td>!=</td>
<td>income != tax</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>answer != 'y'</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&gt;</td>
<td>expenses &gt; income</td>
</tr>
<tr>
<td>≥</td>
<td>Greater than or equal to</td>
<td>&gt;=</td>
<td>points &gt;= 60</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;</td>
<td>pressure &lt; max</td>
</tr>
<tr>
<td>≤</td>
<td>Less than or equal to</td>
<td>&lt;=</td>
<td>expenses &lt;= income</td>
</tr>
</tbody>
</table>

Display 3.2
Java Comparison Operators
Primary Logical Operators

- Individual terms can be connected using logical operators:
  - `&&` “and” or “conjunction”
  - `||` “or” or “disjunction”
  - `!` “not” or “negation”

- Truth tables:

  And (`&&`):
  - `true && true = true`
  - `true && false = false`
  - `false && true = false`
  - `false && false = false`

  Or (`||`):
  - `true || true = true`
  - `true || false = true`
  - `false || true = true`
  - `false || false = false`

  Negation (`!`):
  - `!true = false`
  - `!false = true`
Example using the “and” (&&) operator.

```java
if ((score >= 0) && (score <= 100))
    System.out.println("Valid score");
else
    System.out.println("Invalid score");
```

Not allowed:

```java
if (0 <= score <= 100)
...
```
Example using the “or” (||) operator.

Example:

```java
if ((quantity > 5) || (cost < 10))
    ...
```

Example using negation:

- “not” (!) operator.

Example:

```java
If (!(x > 10))
    ...
```
Primary Logical Operators

Arbitrarily complex logical expressions can be composed.

- \( (x < y) \&\& (z \geq w) \)
- \( (x < y) \mid\mid ((x + y) == z) \&\& (w != u) \)
- \( !((u <= v) \&\& (x == y)) \&\& ((x >= w) \mid\mid (z == y)) \)

Note this adds to our rules of precedence.

More examples:

- \( !(x < y) \mid\mid x + y == z \&\& w != u \) // No parenthesis
- \( (!!(x < y)) \mid\mid (((x + y) == z) \&\& (w != u)) \) // Fully parenthasized
- \( !(x < y) \mid\mid ((x + y) == z) \&\& (w != u) \) // Happy medium
- "The sum is:" + x + y

Parentheses can be used to enhance readability.
Recall that *boolean* is a Java primitive type.

Variables can be declared and initialized:

```java
boolean b1, b2;
boolean b3 = true;

b1 = true;
b2 = false;
b3 = b1 && (b2 || !b3);
```

Personal Opinion – boolean variables can be helpful, but are frequently unnecessary and over complicate a program.
Boolean variables frequently appear in the condition of an if-statement:

```java
if (b3 == true)  // rookie "mistake"
    ...

if (b3)         // the more accepted way
    ...

if (b3 == false) // another rookie "mistake"
    ...

if (!b3)        // the more accepted way
    ...
```
An `if-else` statement can contain any sort of statement within it.

In particular, it can contain another `if-else` statement in the:

- “if” part.
- “else” part.
- or both.

In such a case the `if` statement is referred to as *nested*, and as having an *inner* and *outer* `if` statements.
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1
    else
        Statement_2
else
    if (Boolean_Expression_3)
        Statement_3
    else
        Statement_4
Nested Statements, cont.

if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1
    else
        Statement_2
else
    if (Boolean_Expression_3)
        Statement_3
    else
        Statement_4
System.out.print("Enter temp:");
int temperature = keyboard.nextInt();
System.out.print("Is it sunny?");
char sunny = keyboard.nextChar(); // user enters ‘y’ or ‘n’

if (temperature > 90)
    if (sunny == ‘y’)
        System.out.println("Beach");
    else
        System.out.println("Movie");
else
    if (sunny == ‘y’)
        System.out.println("Tennis");
    else
        System.out.println("Stay home");
System.out.print(“Enter temp:”);
int temperature = keyboard.nextInt();
System.out.print(“Is it sunny?”);
char sunny = keyboard.nextChar(); // user enters ‘y’ or ‘n’

if (temperature > 90) // int temperature
  if (sunny == ‘y’) // char sunny
    System.out.println(“Beach”);
  else
    System.out.println(“Movie”);
else
  if (sunny == ‘y’)
    System.out.println(“Tennis”);
  else
    System.out.println(“Stay home”);
The inner `if` statement, outer `if` statement, or both, may contain only an `if` part; consequently there are many forms of nesting.

```plaintext
if (Boolean_Expression_1)
    Statement_1
else
    if (Boolean_Expression_3)
        Statement_2
    else
        Statement_3
```
Nested Statements, cont.

```java
if (grade >= 70)
    System.out.println("pass");
else
    if (grade >= 60)
        System.out.println("barely pass");
    else
        System.out.println("fail");
```
Nested Statements, cont.

```java
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1
    else
        Statement_2
else
    Statement_3
```
if (grade < 70)
    if (grade < 60)
        System.out.println("fail");
    else
        System.out.println("barely pass");
else
    System.out.println("pass");
Nested Statements, cont.

- Note that this one is somewhat unique…why?

```python
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1
    else
        Statement_2
else
    if (Boolean_Expression_3)
        Statement_3
```
if (grade >= 60)
    if (grade >= 70)
        System.out.println("pass");
    else
        System.out.println("barely pass");
else
    if (grade < 50)
        System.out.println("wicked fail");
Why is the following example confusing?

```java
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1
    else
        if (Boolean_Expression_3)
            Statement_2
        else
            Statement_3
```
Nested Statements, cont.

- Which is it?

```java
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
        Statement_1
    else
        if (Boolean_Expression_3)
            Statement_2
        else
            Statement_3
```
Each else is paired with the nearest preceding unmatched if.

So, the compiler parses (i.e., interprets) both of the following the same:

First form
if (a > b)
  if (c > d)
    e = f;
else
  g = h;

Second form (the correct interpretation)
if (a > b)
  if (c > d)
    e = f;
else
  g = h;

Second form is preferable; indentation should reflect the nesting of if-else statements.
Nested Statements, cont.

- Indentation can communicate which \texttt{if} goes with which \texttt{else}.

- Indentation does \texttt{not} determine which \texttt{else} goes with which \texttt{if}.

- Braces can be used to group statements and over-ride the above rule.
For example, however, are different:

<table>
<thead>
<tr>
<th>First form</th>
<th>Second form</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &gt; b)</td>
<td>if (a &gt; b)</td>
</tr>
<tr>
<td>{</td>
<td>if (c &gt; d)</td>
</tr>
<tr>
<td>if (c &gt; d)</td>
<td>e = f;</td>
</tr>
<tr>
<td>e = f;</td>
<td>else</td>
</tr>
<tr>
<td>}</td>
<td>g = h;</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>g = h;</td>
<td></td>
</tr>
</tbody>
</table>

Note how the braces force the `else` to be associated with the outer `if`.
The following pattern of nested if-else statements is common:

```java
if (Boolean_Expression_1)
    Statement_1
else
    if (Boolean_Expression_2)
        Statement_2
    else
        if (Boolean_Expression_3)
            Statement_3
        else
            if (Boolean_Expression_4)
                Statement_4
            else
                Default_Statement
```
int score;
System.out.print(“Enter score:”);
score = keyboard.nextInt();

if (score >= 90)
    System.out.print(“A”);
else
    if (score >= 80)
        System.out.print(“B”);
    else
        if (score >= 70)
            System.out.print(“C”);
        else
            if (score >= 60)
                System.out.print(“D”);
            else
                System.out.print(“F”);
It is common practice to format this case as follows:

```java
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
else if (Boolean_Expression_3)
    Statement_3
else if (Boolean_Expression_4)
    Statement_4
else
    Default_Statement
```
int score;
System.out.print(“Enter score:”);
score = keyboard.nextInt();

if (score >= 90)
    System.out.print(“A”);
else if (score >= 80)
    System.out.print(“B”);
else if (score >= 70)
    System.out.print(“C”);
else if (score >= 60)
    System.out.print(“D”);
else
    System.out.print(“F”);
Logically, the preceding is equivalent to:

```java
char grade;

if (score >= 90)
    grade = 'A';

if ((score >= 80) && (score < 90))
    grade = 'B';

if ((score >= 70) && (score < 80))
    grade = 'C';

if ((score >= 60) && (score < 70))
    grade = 'D';

if (score < 60)
    grade = 'F';

System.out.println(grade);
```
Is the following equivalent?

```java
if (score >= 90)
    grade = 'A';

if (score >= 80)
    grade = 'B';

if (score >= 70)
    grade = 'C';

if (score >= 60)
    grade = 'D';

if (score < 60)
    grade = 'F';

System.out.println(grade);
```
A leap year happens every 4 years, for example 2016 is a leap year. Exception: Century years are NOT leap years UNLESS they can be evenly divided by 400. (For example, 1700, 1800, and 1900 were not leap years, but 1600 and 2000, which are divisible by 400, were.)

Write a program that reads a year from the command line. Print a message if (and only if) that year is a leap year.

```java
public class LeapYear{
    public static void main (String[] args){
        int year = Integer.parseInt(args[0]);

        if (year % 100 == 0) {
            if (year % 400 == 0) {
                System.out.println("Leap year.");
            } else {
                System.out.println("Not a leap year.");
            }
        } else if (year % 4 == 0) {
            System.out.println("Leap Year.");
        } else {
            System.out.println("Not a leap year.");
        }
    }
}
```
Suppose we have int variables, created and initialized as follows:

```java
int x, y, z, max;
System.out.print("Enter value:");
x = keyboard.nextInt();
System.out.print("Enter value:");
y = keyboard.nextInt();
System.out.print("Enter value:");
z = keyboard.nextInt();
```

Consider the following code for determining the maximum of the three variables x, y and z.
Version #1 (yuk!):

```c
if (x >= y)
    if (x >= z)
        max = x;
    else
        max = z;
else
    if (y >= z)
        max = y;
    else
        max = z;
```
Version #2 (reformatted version #1):

```plaintext
if (x >= y)
    if (x >= z)
        max = x;
    else
        max = z;
else if (y >= z)
    max = y;
else
    max = z;
```
Version #3:

max = x;

if (y >= max)
    max = y;

if (z >= max)
    max = z;
Branch Statement Examples

- Version #4:

```java
if (x >= y && x >= z )
    max = x;
else if (y >= x && y >= z)   // This test could be simplified
    max = y;
else
    max = z;
```

- How would you extend each of the previous solutions to 4 values?
Now suppose we want to determine the median.

Version #1:

```
if (x >= y)
    if (x <= z)
        mid = x;
    else
        if (y >= z)
            mid = y;
        else
            mid = z;
else
    if (y <= z)
        mid = y;
    else
        if (x >= z)
            mid = x;
        else
            mid = z;
```
Version #2: (reformatted version of #1)

```c
if (x >= y)
    if (x <= z)
        mid = x;
    else if (y >= z)
        mid = y;
    else
        mid = z;
else if (y <= z)
    mid = y;
else if (x >= z)
    mid = x;
else
    mid = z;
```
Version #3:

```java
if (x <= y && x >= z)
    mid = x;
else if (x <= z && x >= y)
    mid = x;
else if (y <= x && y >= z)
    mid = y;
else if (y <= z && y >= x)
    mid = y;
else
    mid = z;
```
Branch Statement Examples

Version #4:

```java
if ((x <= y && x >= z) || (x <= z && x >= y))
    mid = x;
else if ((y <= x && y >= z) || (y <= z && y >= x))
    mid = y;
else
    mid = z;
```
Version #5:

\[
\text{mid} = x; \\
\text{if } ((y <= x && y >= z) || (y <= z && y >= x)) \\
\quad \text{mid} = y; \\
\text{else if } ((z <= x && z >= y) || (z <= y && z >= x)) \\
\quad \text{mid} = z;
\]
Branch Statement Examples

- Version #6:

```c
if (y >= x) ;  
   // swap x and y

if (z >= x) ;  
   // swap x and z

if (z >= y) ;  
   // swap y and z

mid = y;
```

- Question - how do we swap the contents of two variables?
1. A party at the Kardashian’s house is said to be successful if the number of attendees is between 40 and 60, inclusively. If, however, the party is on the weekend, then as long as there is at least 40 people, then the party is successful. Give a segment of Java code that will input an integer that represents the number of attendees at a party, plus an integer that represents the day of the week, i.e., 0=Sun, 1=Mon, 2=Tue,…,6=Sat. After inputting the two values, your code should output whether or not the party is successful.

2. Suppose you and your date attend a party at the Kardashian’s where attendees are each ranked on a scale of 0 to 10 on the stylishness of their cloths. Give a segment of Java code that will input your ranking and your date’s ranking, and then output if either of you is very stylish (8 or more), i.e., “yes” or “no.” If, however, either of you is ranked with a style of 2 or less, then the answer is “no” regardless of the other person’s ranking (even if it is 8 or more). In all other cases, the result is “maybe.”

3. The Kardashians spend most of their day outside by the pool, but only if the temperature is just right. Specifically, they will be by the pool if the temperature is between 60 and 90, inclusively. Unless it is summer, in which case the upper limit is 100 rather than 90. Give a segment of Java code that will input an integer value that represents the temperature, and an integer that indicates whether or not it is summer, i.e., 0=no and 1=yes, and then output whether or not it is a day for the Kardashians to hang out by the pool or not, i.e., “pool time” or “not pool time.”

4. Kim Kardashian is pulled over in her car by a police officer. If her speed is 60 or less, then she is not given a ticket. If her speed is between 61 and 80, inclusively, then she is given a ticket for $100. If her speed is 81 or more, inclusively, then she is given a ticket for $200. On the other hand, if it is her birthday, then her speed can be 5 miles per hour higher before the fine is determined. Give a segment of Java code that will input Kim’s speed, and an indication of whether or not it is her birthday (as an integer, with 0=no and 1= yes), and outputs the amount of her fine.

5. Give a segment of Java code that will prompt the user for two integers and output their sum. However, if the sum is between 10 and 19, inclusively, then the code should output 20 instead.
6. Give a segment of Java code that will input an integer representing the day of the week, i.e., 0=Sun, 1=Mon, 2=Tue,….6=Sat, and a Boolean value indicating if Kim Kardashian is on vacation. Your code should output a string indicating the time her alarm clock should be set to. On weekdays her alarm should be set to “7:00” and on weekends it should be “10:00.” However, if Kim is on vacation, then on weekdays it should be set to “10:00” and on weekends it should be set to “off.”

7. Give a segment of Java code that will input two integer values a and b. The program should output “success” if either of the numbers is 6, or if their sum or difference is 6. Note that the function Math.abs(num) computes the absolute value of a number.

8. Give a segment of Java code that will input an integer value x and a Boolean value “mode.” The code should output “success” if x is in the range 10..20, inclusively, and “fail” otherwise. However, if the value of “mode” is true, then it should output “success” if x is outside the range 10..20 and “fail” otherwise.

9. A number is said to be “special” if it is a multiple of 11, or 1 greater than some multiple of 11. Give a segment of Java code that will input a positive integer and then determine if that number is special (output “yes” or “no”). Hint: use the mod (%) operator.

10. A number is said to be “cool” if it is a multiple of 3 or 5, but not both. Give a segment of Java code that will input an integer and then determine if that number is cool (output “yes” or “no”). Hint: use the mod (%) operator.

11. Suppose a lottery ticket has three integers a, b and c on it. If all three integers are different then the ticket wins nothing. If all three are the same then the ticket wins $100. Finally, if exactly two integers are the same, then the ticket wins $50. Give a segment of Java code that will input three integers from a lottery ticket and determine how much the ticket wins.

12. Give a segment of Java code that will input three integers a, b and c, and then determine if any two added together equals the third (output “yes” or “no”).
FIT vs. George Mason University:
http://redalertpolitics.com/2016/02/19/george-mason-students-cant-name-ronald-reagan-joe-biden/
The `==` operator is used for determining if two integers or characters have the same value:

```java
int a = keyboard.nextInt();
if (a == 3)
    ...
```

Recall that only a finite number of real numbers can be represented in any fixed number, e.g., 32, of bits.

For real numbers, this results in:

- Round-off error – results from inexact representation of real numbers.
- Error propagation – results from applying arithmetic operations to values that have been approximated.
Because of round-off error and error propagation, the `==` operator is not appropriate for determining if two real number values are equal.

- It is common to use `<` and some appropriate tolerance instead:

  ```java
  // The "boss" gives you the specific value
  double EPSILON = 0.000001;
  :
  if (Math.abs(b - c) < EPSILON)
  :
  
  - where `b` and `c` are of a floating point type, i.e., `double` or `float`
The **switch** statement is another multi-way branch statement.
- More restricted than **if-else** statements
- Branching **must** be based on an integer, char or String expression.
Consider the following nested if-statement:

```java
int x, y, z;

System.out.print("Enter value:");
x = keyboard.nextInt();

if (x == 0) {
    System.out.println("zero");
    x = x * 10;
}
else if (x == 1) {
    System.out.println("one");
    y = 0;
    z = x * 20;
}
else if (x == 5) {
    System.out.println("five");
    y = 10;
    x = x * 15;
}
else {
    System.out.println("default case");
    x = 30;
}
```
An equivalent switch statement:

```java
switch (x) {
    case 0:
        System.out.println("zero");
        x = x * 10;
        break;
    case 1:
        System.out.println("one");
        y = 0;
        z = x * 20;
        break;
    case 5:
        System.out.println("five");
        y = 10;
        x = x * 15;
        break;
    default:
        System.out.println("default case");
        x = 30;
        break;
}
```
Note that, in this case, order doesn’t matter (sometimes it does):

```java
switch (x) {
    case 5:
        System.out.println("five");
        y = 10;
        x = x * 15;
        break;
    case 1:
        System.out.println("one");
        y = 0;
        z = x * 20;
        break;
    case 0:
        System.out.println("zero");
        x = x * 10;
        break;
    default:
        System.out.println("default case");
        x = 30;
        break;
}
```
An INEQUIVALENT switch statement:

```java
switch (x) {
    case 0:
        System.out.println("zero");
        x = x * 10;
        break;
    case 1:
        System.out.println("one");
        y = 0;
        z = x * 20;
        break;
    case 5:
        System.out.println("five");
        y = 10;
        x = x * 15;
        break;
    default:
        System.out.println("default case");
        x = 30;
        break;
}
```
Another nested if-statement:

```java
int x, y, z;

System.out.print("Enter value:");
x = keyboard.nextInt();

if (x == 0) {
    System.out.println("zero");
    x = x * 10;
} else if ((x == 1) || (x == 2)) {
    System.out.println("one or two");
    y = 0;
    z = x * 20;
} else {
    System.out.println("anything else");
    x = 30;
}
```
An equivalent switch statement:

```java
switch (x) {
    case 0:
        System.out.println("zero");
        x = x * 10;
        break;
    case 1:
    case 2:
        System.out.println("one or two");
        y = 0;
        z = x * 20;
        break;
    default:
        System.out.println("anything else");
        x = 30;
        break;
}
```
Yet another example:

```java
if (x == 0) {
    System.out.println("zero");
    x = x * 10;
} else if (x == 1) {
    y = 0;
    z = x * 20;
} else if (x == 2) {
    System.out.println("one");
    y = 0;
    z = x * 20;
} else {
    System.out.println("anything else");
    x = 30;
}
```
An equivalent switch statement:

```java
switch (x) {
    case 0:
        System.out.println("zero");
        x = x * 10;
        break;
    case 2:
        System.out.println("one");
    case 1:
        y = 0;
        z = x * 20;
        break;
    default:
        System.out.println("any other case");
        x = 30;
        break;
}
```
The `switch` statement begins with the keyword `switch` followed by an integer, `char` or `String` expression in parentheses called the `controlling expression`.

A list of cases follows, enclosed in braces.

Each case consists of the keyword `case` followed by a case label, which can be:
- a literal (integer or string) called the `case label`
- a colon
- a list of statements.

The action for each case `typically` ends with the word `break`, which prevents the consideration of other cases.
The list of cases is “searched” for a case label matching the controlling expression.

The action associated with a matching case label is executed.

If no match is found, the case labeled default is executed.
  ➢ The default case is optional.
An if-statement be replaced by a switch statement if and only if:

- The controlling expression evaluates to an integer, char or String.
- Each case must be a comparison for equality.
- Each case must compare the expression to a constant (or literal).

By the way, a switch statement can be nested as well, although we won’t consider any examples just yet.
The switch Statement with Strings

```java
System.out.print("Enter day of week : ");
String weekDay = kb.next();
switch (weekDay) {
    case "monday" : System.out.println("Its going to be a long week");
        break;
    case "tuesday" : System.out.println("Not much better than monday");
        break;
    case "wednesday" : System.out.println("Half way there!");
        break;
    case "thursday" : System.out.println("One more day to go!");
        break;
    case "friday" : System.out.println("Happy hour!");
        break;
    case "saturday" : System.out.println("Chill time!");
        break;
    case "sunday" : System.out.println("Crap! tomorrow is monday!");
        break;
    default : System.out.println("Bad input");
        break;
}
```
char grade;
System.out.print("Enter letter grade:");
grade = kb.nextChar();
switch(grade)
{
    case 'A':
    case 'B':
    case 'C':
    case 'D':
        System.out.println("Pass");
        break;
    case 'W':
        System.out.println("Withdraw");
        break;
    case 'I':
        System.out.println("Incomplete");
        break;
    default:
        System.out.println("Fail");
}
The conditional operator…is weird…it is:

- a way to branch
- an operator, and hence has a value (in contrast to if or switch statements)

Basic syntax (simplified):

```plaintext
<boolean expression>  ?  <value#1>  :  <value#2>
```
Consider the following:

```java
int x = keyboard.nextInt();
String S1;

if (x >= 0)
    S1 = "non-negative";
else
    S1 = "negative";

System.out.println("The number is " + S1);
```

The above is equivalent to:

```java
S1 = ((x >= 0) ? "non-negative" : "negative");
System.out.println("The number is " + S1);
```
Similarly:

```c
int x, y;
  :
if (x == 50)
  y = x * 10;
else
  y = x * 20;
```

Is equivalent to:

```c
y = ((x == 50) ? (x * 10) : (x * 20));
```
And lastely:

```java
if (x < 0)
    y = 25;
else
    y = 20;
```

Is equivalent to (silly):

```java
if ((x < 0) ? true : false)
    y = 25;
else
    y = 20;
```
Erroneous Conditional Operators

Suppose:

```java
int x, y;
String S1, S2, S3;

System.out.print("Enter an integer:");
x = keyboard.nextInt();
System.out.print("Enter a word:");
S1 = keyboard.next();
S2 = "dog";
```

What is wrong with each of the following:

```java
S2 = ((x == 0) : "zero" ? "non-zero");
y = ((x < 0) ? "negative" : "non-negative");
y = ((S1 == true) ? (x*10) : (x*20));
S3 = ((true == false) ? S1 : S2);
```
Conditional expressions can also be nested:

```c
int x,y,z,w;
if (x < 0)
    w = 20;
else
    if (y == z)
        w = 25;
    else
        w = 65;
```

Is equivalent to:

```c
w = ((x < 0) ? 20 : ((y == z) ? 25 : 65));
```
Loop Statements

- A loop repeats a statement or a group of statements.
  - This group of statements is called the body of the loop.

- Example:
  - Computing a class exam average.
  - Computing quiz or exam averages for each student.
Every loop has a *control structure*, which determines how many times the group of statements is repeated.

Loop control *typically* has three components:

1. Initialization
2. Condition for termination (or continuation)
3. Updating the condition
Loop Statements

- Three different types of Java loops:
  - while
  - for
  - do-while

- Each type of loop will manage loop control in a different manner.

- Every loop, regardless of type, will have a body.

- It will be important to learn and understand which type of loop is preferable in which types of situations.
  - This is particularly true for do-while loops!!!!!!!!!!!!!!!!!!!!
  - When in doubt, don’t use a do-while loop (or ask the instructor or GSAs)
A while loop is controlled by a boolean expression:

- If the expression is true then the loop continues, i.e., the body is repeated.
- If the expression is false then the loop is terminated.

Example #1:

```java
int i, n;
System.out.print(“Enter N:”);
n = keyboard.nextInt();
i = 1;
while (i <= n)
{
    System.out.println(“Hello World!”);
    i = i + 1;
}
```
The loop body could be a single statement, or multiple statements:

```plaintext
while (Boolean_Expression)  
    Body_Statement

while (Boolean_Expression)  
{
    First_Statement
    Second_Statement
    ...
}```
Example #2:
- Input a positive integer $n \geq 0$
- Output the value of $i$ and $2^i$, for all $0 \leq i \leq n$

```java
int i, v, n;
System.out.print("Enter n:");
n = keyboard.nextInt();
i = 0;
v = 1;
while (i <= n)
{
    System.out.println(i + "    " + v);
    v = v * 2;
    i = i + 1;
}
```

Enter n: 4

<table>
<thead>
<tr>
<th>i</th>
<th>2^i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>
// Loops don't always increment by 1 on each iteration
public static void main(String[] pars) {
    int i, n;

    Scanner keyboard = new Scanner(System.in);

    System.out.print("Enter n:");
    n = keyboard.nextInt();
    i = 0;
    while (i <= n) {
        System.out.println(i);
        i = i + 2;
    }
}
// Loops don't always increment on each iteration
public static void main(String[] pars) {
    int n;

    Scanner keyboard = new Scanner(System.in);

    System.out.print("Enter n:");
    n = keyboard.nextInt();
    while (n >= 1) {
        System.out.println(n);
        n = n - 1;
    }
}
Example: Bug Infestation

- **Suppose:**
  - Roach volume: 0.002 cubic feet
  - Reproductive rate is: 95% per week

- **Given:**
  - Starting roach population
  - Total house volume

- **Find:**
  - Number of weeks to exceed the capacity of the house
  - Number and volume of roaches at that point

- Assume that roaches do not die, shrink or leave.
The following program is an example of a **computer simulation**.

As you have probably already come to know, most notably through computer games, simulations are very common.

Simulations, by their very nature, tend to **approximate** the real world.

- Simulations usually make assumptions that simplify or improve the efficiency of a computation.
- For example, the assumption that roaches do not die, shrink or leave.
- Similarly, population is a real number in the following program.
- In general, games are very simplified, inaccurate simulations.

In other words, simulations typically do not model the real world exactly.
Enter total house volume (cubic feet): 1000
Enter initial number of roaches: 20

Initial population: 20
House volume: 1000.0 cubic feet

The house will be filled in 16 weeks.
There will be 874145 roaches.
They will fill 1748.2912063155743 cubic feet.
First, some declarations and initial input:

```java
public class bugTest {

    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        final double GROWTH_RATE = 0.95;
        final double ONE_BUG_VOLUME = 0.002;

        double houseVolume, population, totalBugVolume;
        int startPopulation;

        System.out.print("Enter total house volume (cubic feet): ");
        houseVolume = keyboard.nextDouble();
        System.out.println();
        System.out.print("Enter initial number of roaches: ");
        startPopulation = keyboard.nextInt();
        System.out.println();
    }
}
```
Now for the main loop:

```plaintext
population = startPopulation;
totalBugVolume = population * ONE_BUG_VOLUME;

while (totalBugVolume < houseVolume) {
    population = population + (GROWTH_RATE * population);
    totalBugVolume = population * ONE_BUG_VOLUME;
}
```
In order to count the number of weeks, one more variable is added:

```c
int countWeeks = 0;
population = startPopulation;
totalBugVolume = population * ONE_BUG_VOLUME;

while (totalBugVolume < houseVolume) {
    population = population + (GROWTH_RATE * population);
    totalBugVolume = population * ONE_BUG_VOLUME;
    countWeeks = countWeeks + 1;
}
```
Finally, the output:

```java
System.out.println("Initial population:" + startPopulation);
System.out.println("House volume:" + houseVolume + " cubic feet");
System.out.println();
System.out.println("The house will be filled in " + countWeeks + " weeks.");
System.out.println("There will be " + (int)population + " roaches.");
System.out.println("They will fill " + totalBugVolume + " cubic feet.");
```

Simple modifications:

- Output the population and total volume on each loop iteration.
- Make the growth rate, bug volume, or both, part of the input.
- What happens if the growth rate is negative (try it)!?

A more substantial modification:

- What if there are also frogs in the house?
- Maybe frogs also have a growth rate, eat some number of roaches per day.
- Maybe there are lots of roaches initially, but only a couple of frogs, growth rate of roaches reproduce more quickly than frogs, (you make the assumptions), etc.
- How soon before the frogs overtake (and eliminate) the roaches? Or do they overtake them at all?
- Prompt the user for a positive integer \( n \), followed by \( n \) integers, output the sum of the integers and the average.

- Prompt the user for a list of grades, terminated by a -1, compute and output a grade average. Assume all grades are positive values.

- Prompt the user for a positive integer \( n \), output the sequence of integers 1, 2, 3,...,\( n \), in that order.

- Prompt the user for a positive integer \( n \), output the sequence of integers \( n \), \( n-1 \), \( n-2 \),...,1, in that order.

- Prompt the user for a positive integer \( n \), output the value of \( i^2 \), for all \( i \) between 1 and \( n \).

- Prompt the user for a positive integer \( n \), output the sum of \( i^2 + 5i + 13 \) as \( i \) goes from 1 to \( n \).
Prompt the user for a positive integer n, add up all the even integers between 1 and n, and add up all of the odd integers between 1 and n, and then output the two sums (hint: if x is even, then x%2 = 0).

Prompt the user for a list of positive integers terminated by -1. The program should output whether or not the list is in non-decreasing (i.e., “increasing”) order.
A `for` loop typically iterates over an integer range:

```java
int i;
for (i = 1; i <= 4; i++)           // Note the increment operator
    System.out.println(i);
System.out.println("Done");
```

Equivalent to:

```java
int i;
i = 1;
while (i <= 4) {
    System.out.println(i);
    i++;
}
System.out.println("Done");
```
A for loop can decrement as well as increment:

```java
int i;
for (i = 3; i >= 0; i--)
    System.out.println(i);
```

The increment, or decrement, can be by any amount:

```java
int i;
for (i = 1; i <= 10; i = i + 2)
    System.out.println(i);
```
The range can start or end anywhere:

```java
int i;
for (i = 34; i <= 83; i++)
    System.out.println(i);
```

The body might never get executed:

```java
int i;
for (i = 10; i <= 5; i++)
    System.out.println(i);
```
Typically variables are used as the upper or lower bound:

```java
int i, n;
    n = keyboard.nextInt();
    for (i = 1; i <= n; i++)
        System.out.print(i);
```

The body can contain multiple statements:

```java
int i;
    for (i = 1; i <= 10; i++)
        { System.out.print("hello ");
        System.out.println("world!");
        }
```
Typically the loop control variable is declared in the for loop:

```java
for (int i = 1; i <= 10; i++) {
    System.out.print("hello ");
    System.out.println("world! for the "+i+"th time");
}
```

In such a case the *scope* of the loop control variable is limited to the loop body; the following will not compile:

```java
for (int i = 1; i <= 10; i++) {
    System.out.print("hello ");
    System.out.println("world! for the "+i+"th time");
}
System.out.println(i);
```
More generally, both the initialization and the boolean expression can be more complicated expressions:

```java
int u, v, k, j;

u = 5;
v = 10;
k = 20;
j = 40;
for (int i = u*v; i <= k*j-5; i++)
    System.out.println("hello");
```

How many iterations does the above loop perform?
Also called a do-while loop (repeat-until loop)

Similar to a while statement, except that the loop body is executed at least once.

Syntax:

```java
    do
    {
        Body_Statement
    }
    while (Boolean_Expression);
```

Recall that you should not use the do-while loop in this class unless you ask, or are specifically directed to do so.
Equivalent while statement:

\[
\text{Statement(s)}_S1 \\
\text{while } (\text{Boolean\_Condition}) \\
\text{Statement(s)}_S1
\]
Menu driven programs are good examples for do-while loops:

```java
int option;

    do {
        System.out.println("Select an option");
        System.out.println("1: This class is cool");
        System.out.println("2: This class is NOT cool");
        System.out.print("Enter Choice:");
        option = kb.nextInt();

        if (option == 1)
            System.out.println("You pass");
        else if (option == 2)
            System.out.println("You fail");
        else if (option != 3)
            System.out.println("Invalid entry");
        System.out.println();
    } while (option != 3);
```

Select an option
1: This class is cool
2: This class is NOT cool
3: Quit
Enter Choice: 1
You pass

Select an option
1: This class is cool
2: This class is NOT cool
3: Quit
Enter Choice: 2
You fail

Select an option
1: This class is cool
2: This class is NOT cool
3: Quit
Enter Choice: 5
Invalid Entry

Select an option
1: This class is cool
2: This class is NOT cool
3: Quit
Enter Choice: 3
do-while Loop

- See http://www.cs.fit.edu/~pbernhar/teaching/cse1001/dowhile.txt
Some general terminology:

- **while** loops and **for** loops are commonly referred to as *pre-test* loops.
- **do-while** loops are referred to as *post-test* loops.

For the obvious reasons…
Choosing a Loop Type

- In most cases pre-test loops are preferable.

- The choice of loop type is a “judgment call.”

- If you know how many times a loop will be iterated and, in particular, if the loop is going to iterate over a fixed range of “enumerated” values, i.e., integers or chars - \texttt{for} loop

- If you don’t know how many times the loop will be iterated, but it is:
  - Zero or more times - \texttt{while} loop
  - One or more times - \texttt{do-while}
Techniques for Loop Termination

- Asking the user before each iteration is common:

  ```java
  String CMD;
  int x, y;

  System.out.print("Enter Command:");
  CMD = kb.next();
  while (!CMD.equals("quit")) {
      System.out.print("Enter two integers:");
      x = kb.nextInt();
      y = kb.nextInt();
      if (CMD.equals("add"))
          System.out.println("Sum is:" + (x+y));
      else if (CMD.equals("subtract"))
          System.out.println("Difference is:" + (x-y));
      else
          System.out.println("Invalid Command");

      System.out.println();
      System.out.print("Enter Command:");
      CMD = kb.next();
  }
  ```

Enter Command: subtract
Enter two integers: 15 7
Difference is: 8

Enter Command: add
Enter two integers: 10 9
Sum is: 19

Enter Command: dog
Invalid Command

Enter Command: quit
A **sentinel value** is frequently used to signal the end of the list:

- A negative number after a list of (nonnegative) scores would be suitable.

    90  0  10  -1

    System.out.println("Enter a list of positive ints followed by -1:");
    int next = keyboard.nextInt();
    while (next >= 0)
    {
        // Process_The_Int
        next = keyboard.nextInt();
    }
Nested Loops

- The body of a loop can contain any kind of statements, including:
  - Other loops
  - if-else statements
  - Variable declarations

- This nesting can be arbitrarily deep and complex.
Nested Loops

Example:

```java
int line, star;       // Normally declared in the for-loops

for (line = 1; line <= 3; line++)
{
    for (star = 1; star <= 4; star++)
        System.out.print('*');
    System.out.println();
}
```

```
****
****
****
```
Some Terminology:

```java
for (line = 1; line <= 3; line++)
{
    for (star = 1; star <= 4; star++)
    {
        System.out.print('*');
        System.out.println();
    }
}
```

- **body of outer loop**
- **body of inner loop**
Nested Loops

Consider a program that inputs a list of $n$ student names and, for each student, $m$ quiz scores. For each student, the program should output each student’s name, along with a quiz average.

Enter number of students: 2
Enter number of quizzes: 4

Enter name: Smith
Enter quiz scores: 70 90 80 100
Quiz average for Smith: 85.0

Enter name: Jones
Enter quiz score: 95 90 85 72
Quiz average for Jones: 85.5
Assume the following variable declarations.

```java
int n, m, quizScore, quizSum;
double quizAvg;
String name;

Scanner keyboard = new Scanner(System.in);
```
// Prompt for and input the # of students (n) and the # of quizzes (m)

for (int i = 1; i <= n; i++) {
    // Prompt for and input the name

    // Prompt for the quiz scores, input and add them all up

    // Compute and output the average
}
System.out.print("Enter number of students:");
n = keyboard.nextInt();
System.out.print("Enter number of quizzes:");
m = keyboard.nextInt();

for (int i = 1; i <= n; i++) {
    // Prompt for and input the name

    // Prompt for the quiz scores, input and add them all up

    // Compute and output the average
}
System.out.print("Enter number of students:");
n = keyboard.nextInt();
System.out.print("Enter number of quizzes:");
m = keyboard.nextInt();

for (int i = 1; i <= n; i++) {
    System.out.print("Enter name:");
    name = keyboard.next();

    // Prompt for the quiz scores, input and add them all up

    // Compute and output the average
}
System.out.print("Enter number of students:");
n = keyboard.nextInt();
System.out.print("Enter number of quizzes:");
m = keyboard.nextInt();

for (int i = 1; i <= n; i++) {
    System.out.print("Enter name:");
    name = keyboard.next();

    System.out.print("Enter quiz scores:");
    quizSum = 0;
    for (int j = 1; j <= m; j++) {
        quizScore = keyboard.nextInt();
        quizSum = quizSum + quizScore;
    }

    // Compute and output the average
}
System.out.print("Enter number of students:");
n = keyboard.nextInt();
System.out.print("Enter number of quizzes:");
m = keyboard.nextInt();

for (int i = 1; i <= n; i++) {
    System.out.print("Enter name:");
    name = keyboard.next();

    System.out.print("Enter quiz scores:");
    quizSum = 0;
    for (int j = 1; j <= m; j++) {
        quizScore = keyboard.nextInt();
        quizSum = quizSum + quizScore;
    }

    quizAvg = quizSum/(double)m;
    System.out.println("Quiz average for " + name + ":" + quizAvg);
}
Common Error #1:

```java
System.out.print("Enter number of students:");
n = keyboard.nextInt();
System.out.print("Enter number of quizzes:");
m = keyboard.nextInt();

quizSum = 0;
for (int i = 1; i <= n; i++) {
    System.out.print("Enter name:");
    name = keyboard.next();
    System.out.print("Enter quiz scores:");
    for (int j = 1; j <= m; j++) {
        quizScore = keyboard.nextInt();
        quizSum = quizSum + quizScore;
    }
    quizAvg = quizSum/(double)m;
    System.out.println("Quiz average for " + name + ":" + quizAvg);
}
```
Common Error #2 ("off by one"):

```java
System.out.print("Enter number of students:");
n = keyboard.nextInt();
System.out.print("Enter number of quizzes:");
m = keyboard.nextInt();

for (int i = 1; i < n; i++) {
    System.out.print("Enter name:");
    name = keyboard.next();

    quizSum = 0;
    System.out.print("Enter quiz scores:");
    for (int j = 1; j <= m; j++) {
        quizScore = keyboard.nextInt();
        quizSum = quizSum + quizScore;
    }

    quizAvg = quizSum/(double)m;
    System.out.println("Quiz average for "+ name + ":" + quizAvg);
}
```
Nested Loops

- Common Error #3 ("off by one"):

```java
System.out.print("Enter number of students:");
n = keyboard.nextInt();
System.out.print("Enter number of quizzes:");
m = keyboard.nextInt();

for (int i = 0; i <= n; i++) {
    System.out.print("Enter name:");
    name = keyboard.next();

    quizSum = 0;
    System.out.print("Enter quiz scores:");
    for (int j = 1; j <= m; j++) {
        quizScore = keyboard.nextInt();
        quizSum = quizSum + quizScore;
    }

    quizAvg = quizSum/(double)m;
    System.out.println("Quiz average for " + name + ":" + quizAvg);
}
```
Now suppose there are \( m \) quiz scores and \( k \) exam scores for each student, and that output should include an exam average as well.

Enter number of students: 2
Enter number of quizzes: 4
Enter number of exams: 3

Enter name: Smith
Enter quiz scores: 70 90 80 100
Enter exam scores: 65 73 91
Quiz average for Smith: 85.0
Exam average for Smith: 76.3

Enter name: Jones
Enter quiz scores: 95 90 85 72
Enter exam scores: 100 87 93
Quiz average for Jones: 85.5
Exam average for Jones: 93.3
// Prompt for and input the # of students (n), the # of quizzes (m),
// and the # of exams (k)

for (int i = 1; i <= n; i++) {
    // Prompt for and input the name

    // Prompt for the quiz scores, input and add them all up

    // Prompt for the exam scores, input and add them all up

    // Compute and output the average
}

See http://www.cs.fit.edu/~pbernhar/teaching/cse1001/nested.txt
Exercises:

- Could some variables be eliminated from the previous program(s)?
- What if we move some of the statements around; Is the program still correct?
- Modify the previous program so that the number of quizzes and the number of exams is different for each student, and terminated by a sentinel value, i.e., -1 (see next page).
For example:

Enter number of students: 2

Enter name: Smith
Enter quiz scores: 70 90 80 100 -1
Enter exam scores: 65 73 91 -1
Quiz average for Smith: 85.0
Exam average for Smith: 76.3

Enter name: Jones
Enter quiz scores: 95 90 -1
Enter exam scores: 100 87 93 87 65 -1
Quiz average for Jones: 92.5
Exam average for Jones: 86.4
How would you print out the following (for a generic value of n)?

1 2 3 4 5
2 3 4 5
3 4 5
4 5
5

Note that 2-dimensional processing typically requires doubly-nested loops.
First try this:

1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5

Next, modify the code to give the previous output.

Lesson => start simple, and then enhance; more generally if you are trying to solve a problem that’s too hard, try something similar, yet simpler, then refine your solution.
### Nested Loops

- How about these?

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

5 4 3 2 1
5 4 3 2
5 4 3
5 4
5

5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
```
Each of the patterns could be created by a doubly-nested loop:

1. With an outer loop that \textit{increments} from 1 to \( n \).
2. With an outer loop that \textit{decrements} from \( n \) down to 1.
3. Using either for-loops, while-loops, or a combination.

Try 1 & 2 above for all of the patterns, try 3 for at least one.
Loops are the source of many programming bugs.

Common loop bugs:
- Unintended infinite loops
- Off-by-one errors
- Empty loop bodies
- Testing equality of floating-point numbers
A loop which repeats without ever ending is called an *infinite loop*.

When does this happen?
- When the controlling boolean expression never becomes *false*
- Usually an error, but not always, e.g., event-driven programs, servers.

The behavior of a program containing an infinite loop is often times somewhat unexpected.
- Produces non-stop output.
- Produces no output at all; program seems “dead.”
Examples:

```java
System.out.print("Enter N:");
n = keyboard.nextInt();
sum = 0;
while (true)
    count = 1;
x = 0;
while (count <= n) {
    System.out.print("Enter value:");
    x = keyboard.nextInt();
    sum = sum + x;
}
```

For the roach infestation example, a negative growth rate causes `totalBugVolume always to be less than houseVolume`. 
Off-by-One Errors

- The loop body is repeated one too many times or one too few times.

- Examples:
  - `<` is used when `<=` should be used, or visa-versa.
  - Using the index of the position of last character of a string instead of the length of the string (or vice versa).
  - Starting at 0 instead of 1 (or vice versa).

- Easy to overlook and very frequent.

```java
// Print "hello world" n times
int i = 0;
while (i <= n)
{
    System.out.println("hello world");
    i++;
}
```
Some errors are very subtle (not just with loops); what is printed by:

```java
int product = 1, number;
for (number = 1; number <= 10; number++) {
    product = product * number;
}
System.out.println(product);
```

How about:

```java
int product = 1, number = 1;
while (number <= 10) {
    product = product * number;
    number++;
} 
System.out.println(product);
```
Run-Time Tracing

- Run-Time Tracing means watching the variables change while the program is running.

- One way to trace is to place temporary output statements in a program to print out the values of variables of interest.

- Where should such output statements be placed?

- Suggestion - any point where a variable is modified (before or after).
  - Right after those values are input (see below)
  - At the bottom or top of a loop
  - Immediately before or after a computation along with an assignment
Another, and usually better way to trace a program is to use an automated debugger.

Keep in mind that many errors, although difficult to find, are SIMPLE programming errors!
- Similarly, never assume a conspiracy when stupidity will suffice…
- Programmers, like gamers, tend to get paranoid.

Note that run-time tracing is not the same as manual tracing, which we have also covered in class.
Sometimes tracing must be done manually…

Examples:

- [http://my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/quiz7.doc](http://my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/quiz7.doc)
- [http://my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/quiz10.doc](http://my.fit.edu/~pbernhar/Teaching/SoftwareDevelopment1/quiz10.doc)
Sometimes only part of a boolean expression needs to be evaluated to determine the value of the entire expression.

- If the first operand of || is true
  - entire expression is true
- If the first operand of && is false
  - entire expression is false

In either case there is no need to evaluate the second operand.

This is called short-circuit or lazy evaluation.

Java performs short-circuit evaluation for || and && (but not all languages do).
Short-circuit evaluation is not only efficient, sometimes it is essential!

A run-time error can result, for example, from an attempt to divide by 0.

```java
if (sum/number > 5) // What if number contains 0?
```

The following avoids the run-time error:

```java
if (number != 0)
    if (sum/number > 5)
```

And so does:

```java
if ((number != 0) && (sum/number > 5)) // Now what happens?
```
- A `break` statement can be used to end a loop immediately.

- The `break` statement ends only the **innermost** loop that contains the `break` statement.

- `break` statements can make loops more difficult to understand.
  - Could result in multiple exit points.

- Always try to end a loop at only one place--makes debugging easier.
Because of the complications they introduce, break statements in loops should be avoided.

Some contend break statements should never be used in a loop.

Most agree that they should be used at most sparingly.
End of Chapter – Additional slides to follow for the curious…
The “or” in Java is called *inclusive* - the expression is true when *either* or *both* of the sub-expressions is true.

Another type of “or” is *exclusive*:
- True if one or the other, but *not* both is true.

Either work or play, but not both:
- $(\text{work} \text{ || play}) \&\& (\neg \text{work} \text{ || } \neg \text{play})$
- $(\text{work} \text{ || play}) \&\& \neg(\text{work} \&\& \text{play})$

The exclusive-or operator in Java is $^\text{^}$
- $(\text{work} ^ \text{play})$
- not a logical operator in most languages
This is where it gets weird… a for-loop can contain multiple statements in its initialization section.

```java
int n, p;
for (n = 1, p = 1; n < 10; n++)
    p = p * n;
```

Only one boolean expression is allowed, but it can be arbitrarily complex, i.e., consist of `&&`, `||`, and `!`.

Multiple update actions are also allowed.

```java
int i, z, n;
n = kb.nextInt();
for (i = 1, z = n*n; i < n; i++, z = z - i)
    ...
```
An entire loop body could be placed in the update section.

```java
int i, x = 0;
int n = nextInt();
for (i = 1; i <= n; i++)
{
    System.out.println("value i:" + i);
    x = x + i;
}
```

The above is equivalent to:

```java
int n = nextInt();
for (x = 0, i = 1; i <= n; System.out.println("value i:" + i), x = x + i, i++)
```

Note the empty body, and the (required) semicolon.
In fact, a loop control variable is not even necessary.

```java
for (System.out.println("hello"),
    System.out.print("Enter boolean:");
    kb.nextBoolean();
    System.out.println("Enter Boolean:");)
```

Such “abuses” of the for-loop are not recommended.
As discussed previously, `==` is not reliable for floating-point numbers (which are approximate quantities).

- Can cause infinite loops
- Use `<=` or `>=` rather than `==` or `!=`.

See [http://www.cs.fit.edu/~pbernhar/teaching/cse1001/floatTest.txt](http://www.cs.fit.edu/~pbernhar/teaching/cse1001/floatTest.txt)
System.out.print("Enter temp:");
int temperature = keyboard.nextInt();
System.out.print("Is it sunny?");
boolean sunny = keyboard.nextBoolean(); // user enters true or false

if (temperature > 90) // int temperature
    if (sunny) // boolean sunny
        System.out.println("Beach");
    else
        System.out.println("Movie");
else
    if (sunny)
        System.out.println("Tennis");
    else
        System.out.println("Volleyball");
Precedence Rules

- Parentheses should be used to indicate the order of operations.

- When parentheses are omitted, the order of operation is determined by precedence rules.

- Operations with higher precedence are performed before operations with lower precedence.

- Operations with equal precedence are done left-to-right (except for unary operations which are done right-to-left).
Precedence Rules, cont.

Highest Precedence

First: the unary operators +, −, ++, −−, and!
Second: the binary arithmetic operators *, /, %
Third: the binary arithmetic operators +, −
Fourth: the boolean operators <, >, <=, >=
Fifth: the boolean operators ==, !=
Sixth: the boolean operator &
Seventh: the boolean operator |
Eighth: the boolean operator &&
Ninth: the boolean operator ||

Lowest Precedence

Display 3.16
Precedence Rules

Comparison operators:
<, >, <=, >=
==, !=

Logical operators:
&
|
&&
||
In what order are the operations performed?

\[\text{score} < \frac{\text{min}}{2} - 10 \quad || \quad \text{score} > 90\]

\[\text{score} < \left(\frac{\text{min}}{2}\right) - 10 \quad || \quad \text{score} > 90\]

\[\text{score} < \left(\left(\frac{\text{min}}{2}\right) - 10\right) \quad || \quad \text{score} > 90\]

\[(\text{score} < \left(\left(\frac{\text{min}}{2}\right) - 10\right)) \quad || \quad \text{score} > 90\]

\[(\text{score} < \left(\left(\frac{\text{min}}{2}\right) - 10\right)) \quad || \quad \text{score} > 90\]
Be careful!

```java
int x, y, count = 1;
  :
++count;
if ((x < y) && (count < 10) )
  :
System.out.println(count);

// Not the same as:
int x, y, count = 1;
  :
if ((x < y) && (++count < 10) )
  :
System.out.println(count);
```

- Complete evaluation can be achieved by substituting `&` for `&&` or `|` for `||`. 