Unit 3, Part 4

Indefinite Loops and Boolean Expressions

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Definite Loops

- The loops that we've seen thus far have been *definite loops*.
  - we know exactly how many iterations will be performed before the loop even begins
- In an *indefinite loop*, the number of iterations is either:
  - not as obvious
  - impossible to determine before the loop begins

Sample Problem: Finding Multiples

- Problem: Print all multiples of a number (call it `num`) that are less than 100.
  - output for `num` = 9:

```
9   18   27   36   45   54   63   72   81   90   99
```

- Pseudocode for one possible algorithm:

```
mult = num
repeat as long as mult < 100:
    print mult + "  "
    mult = mult + num
print a newline
```

Sample Problem: Finding Multiples (cont.)

- Pseudocode:

```
mult = num
repeat as long as mult < 100:
    print mult + "  "
    mult = mult + num
print a newline
```

- Here's how we would write this in Java:

```
int mult = num;
while (mult < 100) {
    System.out.print(mult + "  ");
    mult = mult + num;
}
System.out.println();
```

while Loops

- In general, a while loop has the form

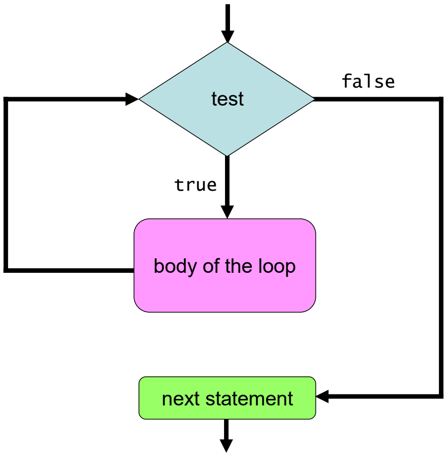
```
while (test) {
    one or more statements
}
```

- As with for loops, the statements in the block of a while loop are known as the *body* of the loop.

Evaluating a while Loop

Steps:

1. evaluate the test
2. if it's false, skip the statements in the body
3. if it's true, execute the statements in the body, and go back to step 1



Tracing a `while` Loop

- Let's trace through our code when `num` has the value `15`:

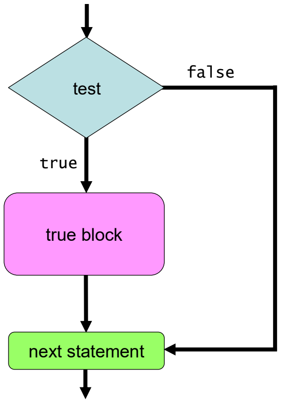
```
int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    mult = mult + num;
}
```

	output thus far	mult
before entering the loop		15
after the first iteration	15	30
after the second iteration	15 30	45
after the third iteration	15 30 45	60
after the fourth iteration	15 30 45 60	75
after the fifth iteration	15 30 45 60 75	90
after the sixth iteration	15 30 45 60 75 90	105

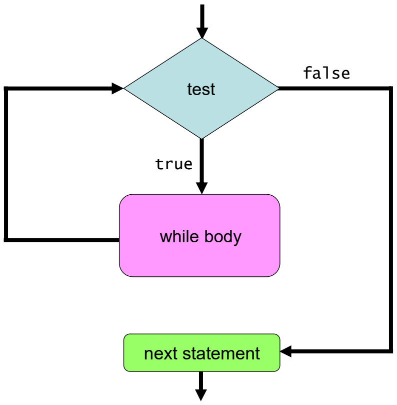
and now `(mult < 100)` is `false`, so we exit the loop

Comparing `if` and `while`

if statement



while statement



- The true block of an if statement is evaluated at most once.
- The body of a while statement can be evaluated multiple times, provided the test remains true.

Typical while Loop Structure

- Typical structure:

```
initialization statement(s)
while (test) {
    other statements
    update statement(s)
}
```

- In our example:

```
int mult = num;                     // initialization
while (mult < 100) {
    System.out.print(mult + "  ");
    mult = mult + num;              // update
}
```

Comparing for and while loops

- while loop (typical structure):

```
initialization
while (test) {
    other statements
    update
}
```

- for loop:

```
for (initialization; test; update) {
    one or more statements
}
```

Infinite Loops

- Let's say that we change the condition for our `while` loop:

```
int mult = num;
while (mult != 100) {     // replaced < with !=
    System.out.print(mult + " ");
    mult = mult + num;
}
```

- When `num` is 15, the condition will always be true.
  - why?
  - an *infinite loop* – the program will hang (or repeatedly output something), and needs to be stopped manually
  - what class of error is this (syntax or logic)?
- It's generally better to use <, <=, >, >= in a loop condition, rather than == or `!=`

Infinite Loops (cont.)

- Another common source of infinite loops is forgetting the update statement:

```
int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    // update should go here
}
```

A Need for Error-Checking

- Let's return to our original version:

```
int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    mult = mult + num;
}
```

- This could still end up in an infinite loop! How?

Using a Loop When Error-Checking

- We need to check that the user enters a positive integer.
- If the number is <= 0, ask the user to try again.
- Here's one way of doing it using a `while` loop:

```
Scanner console = new Scanner(System.in);
System.out.print("Enter a positive integer: ");
int num = console.nextInt();
while (num <= 0) {
    System.out.print("Enter a positive integer: ");
    num = console.nextInt();
}
```

- Note that we end up duplicating code.

Error-Checking Using a do-while Loop

- Java has a second type of loop statement that allows us to eliminate the duplicated code in this case:

```
Scanner console = new Scanner(System.in);
int num;
do {
    System.out.print("Enter a positive integer: ");
    num = console.nextInt();
} while (num <= 0);
```

- The code in the body of a `do-while` loop is always executed at least once.

do-while Loops

- In general, a `do-while` statement has the form

```
do {
     one or more statements
} while (test);
```

- Note the need for a semi-colon after the condition.

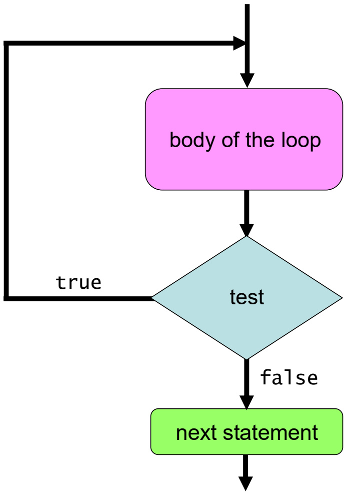
- We do *not* need a semi-colon after the condition in a `while` loop.
  - beware of using one – it can actually create an infinite loop!

Evaluating a `do-while` Loop

Steps:

1. execute the statements in the body
2. evaluate the test
3. if it's true, go back to step 1

(if it's false, continue to the next statement)



Formulating Loop Conditions

- We often need to repeat actions *until* a condition is met.
  - example: keep reading a value *until* the value is positive
  - such conditions are *termination* conditions – they indicate when the repetition should stop
- However, loops in Java repeat actions *while* a condition is met.
  - they use *continuation* conditions
- As a result, you may need to convert a termination condition into a continuation condition.

Which Type of Loop Should You Use?

- Use a `for` loop when the number of repetitions is known in advance – i.e., for a definite loop.
- Otherwise, use a `while` loop or `do-while` loop:
  - use a `while` loop if the body of the loop may not be executed at all
    - i.e., if the condition may be `false` at the start of the loop
  - use a `do-while` loop if:
    - the body will always be executed at least once
    - doing so will allow you to avoid duplicating code

Find the Error…

- Where is the syntax error below?

```
Scanner console = new Scanner(System.in);

do {
    System.out.print("Enter a positive integer: ");
    int num = console.nextInt();
} while (num <= 0);

System.out.println("\nThe multiples of " + num +
  " less than 100 are:");

int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    mult = mult + num;
}

System.out.println();
```

Practice with `while` loops

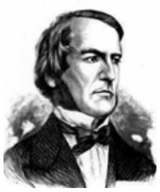
- What does the following loop output?

```
int a = 10;
while (a > 2) {
    a = a - 2;
    System.out.println(a * 2);
}
```

	a > 2	a	output
before loop			
1st iteration			
2nd iteration			
3rd iteration			
4th iteration			

`boolean` Data Type

- A condition like `mult < 100` has one of two values: `true` or `false`
- In Java, these two values are represented using the `boolean` data type.
  - one of the primitive data types (like `int`, `double`, and `char`)
  - `true` and `false` are its two literal values
- This type is named after the 19th-century mathematician George Boole, who developed the system of logic called *boolean algebra*.



`boolean` Expressions

- We have seen a number of constructs that use a "test".
  - loops
  - if statements
- A more precise term for a "test" is a *boolean expression*.
- A boolean expression is any expression that evaluates to `true` or `false`.
  - examples:

```
num > 0
false
firstChar == 'P'
score != 20
```

`boolean` Expressions (cont.)

- Recall this line from our ticket-price program:

```
if (choice.equals("orchestra")) …
```

a boolean expression, because it evaluates to `true` *or* `false`

  - if we look at the `String` class in the Java API, we see that the `equals` method has this header:

```
public boolean equals(...)
```

it returns either `true` *or* `false`

Forming More Complex Conditions

- We often need to make a decision based on more than one condition – or based on the opposite of a condition.
  - examples in pseudocode:

```
if the number is even AND it is greater than 100…
if it is NOT the case that your grade is > 80…
```

- Java provides three *logical operators* for this purpose:

operator	name	example
&&	and	`age >= 18 && age <= 35`
||	or	`age < 3 || age > 65`
!	not	`!(grade > 80)`

Truth Tables

- The logical operators operate on boolean expressions.
  - let a and b represent two such expressions
- We can define the logical operators using *truth tables*.

truth table for && (and)

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

truth table for || (or)

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

truth table for ! (not)

a	!a
false	true
true	false

Truth Tables (cont.)

- Example: evaluate the following expression:

```
(20 >= 0) && (30 % 4 == 1)
```

- First, evaluate each of the operands:

```
(20 >= 0) && (30 % 4 == 1)
    true    &&    false
```

- Then, consult the appropriate row of the truth table:

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

- Thus, `(20 >= 0) && (30 % 4 == 1)` evaluates to `false`

Practice with Boolean Expressions

- Let's say that we wanted to express the following English condition in Java:

  "num is not equal to either 0 or 1"

- Which of the following boolean expression(s) would work?

  a) `num != 0 || 1`

  b) `num != 0 || num != 1`

  c) `!(num == 0 || num == 1)`

- Is there a different boolean expression that would work here?

boolean Variables

- We can declare variables of type boolean, and assign the values of boolean expressions to them:

```
int num = 10;
boolean isPos = (num > 0);
boolean isDone = false;
```

  - these statements give us the following picture in memory:

```
isPos  | true |     isDone | false |
```

- Using a boolean variable can make your code more readable:

```
if (value % 2 == 0) {
    ...
```

⬇

```
boolean isEven = (value % 2 == 0);
if (isEven == true) {
    ...
```

boolean Variables (cont.)

- Instead of doing this:

```
boolean isEven = (num % 2 == 0);
if (isEven == true) {
    ...
```

  you could just do this:

```
boolean isEven = (num % 2 == 0);
if (isEven) {
    ...
```

  The extra comparison isn't necessary!

- Similarly, instead of writing:

```
if (isEven == false) {
    ...
```

  you could just write this:

```
if (!isEven) {
    ...
```

Input Using a Sentinel

- Example problem: averaging an arbitrary number of grades.
- Instead of having the user tell us the number of grades in advance, we can let the user indicate that there are no more grades by entering a special *sentinal value*.
- When we encounter the sentinel, we break out of the loop
  - example interaction:

```
Enter grade (-1 to end): 10
Enter grade (-1 to end): 8
Enter grade (-1 to end): 9
Enter grade (-1 to end): 5
Enter grade (-1 to end): -1
The average is: 8.0
```

Input Using a Sentinel (cont.)

- Here's one way to do this:

```
Scanner console = new Scanner(System.in);
int total = 0;
int numGrades = 0;

System.out.print("Enter grade (or -1 to quit): ");
int grade = console.nextInt();
while (grade != -1) {
    total += grade;
    numGrades++;
    System.out.print("Enter grade (or -1 to quit): ");
    grade = console.nextInt();
}

if (numGrades > 0) {
    System.out.print("The average is ");
    System.out.println((double)total/numGrades);
}
```

Input Using a Sentinel and a Boolean Flag

- Here's another way, using what is known as a *boolean flag*, which is a variable that keeps track of some condition:

```
Scanner console = new Scanner(System.in);
int total = 0;
int numGrades = 0;
boolean done = false;

while (!done) {
    System.out.print("Enter grade (or -1 to quit): ");
    int grade = console.nextInt();
    if (grade == -1) {
        done = true;
    } else {
        total += grade;
        numGrades++;
    }
}

if (numGrades > 0) {
    ...
```

Input Using a Sentinel and a break Statement

- Here's another way, using what is known as a break statement, which "breaks out" of the loop:

```
Scanner console = new Scanner(System.in);
int total = 0;
int numGrades = 0;

while (true) {
    System.out.print("Enter grade (or -1 to quit): ");
    int grade = console.nextInt();
    if (grade == -1) {
        break;
    }
    total += grade;
    numGrades++;
}

// after the break statement, the flow of control
// resumes here...
if (numGrades > 0) {
    ...
```