

## Topic 5: Branches and loops in Java

1. The *if* branch statement
2. Relational and Boolean expressions
3. Multiway branches: *switch*
4. Java loop statements
5. Writing correct loops

David Keil      Computer Science I Using Java      5. Control structures      12/09      1

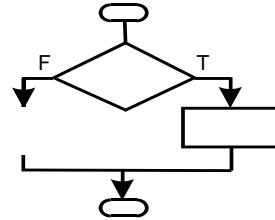
## Objectives

13. Identify declarations, loops, branches, and method calls and definitions in a Java program
14. Translate a flowchart into Java code
15. Solve a loop problem in Java
16. Debug a defective program

## 1. The *if* branch statement

```
age = in.nextInt();  
if (age < 0) True/false expression  
    out.print("Invalid age");
```

- An *if* statement consists of the keyword *if*, a Boolean expression (condition) in parentheses, and *one* statement that is executed if the condition is true



## Check file stream before reading

```
FileReader freadr = new  
    FileReader("x.txt");  
Scanner fin = new Scanner(freadr);  
String line;  
if (fin.hasNextLine())  
    line = infile.getNextLine();
```

- Here, the *FileReader* object *fin* can detect the state of the stream

## if with compound statement

```
int age = in.nextInt();
if (age < 0)
{
    out.print("Invalid age");
    age = in.nextInt();
}
```

*Omitting braces here would change meaning!*

- When multiple statements execute conditional on an *if*, they must be part of a compound statement

## Compound statements and scope of access

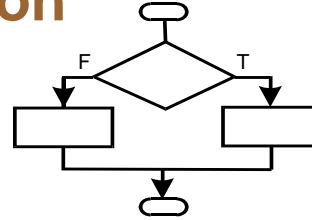
```
int curr_year = 1998, expir_date = 2000;
if (curr_year <= expir_date)
{
    out.print("Amount of purchase: ");
    double amount = 0.0;
    amount = in.nextInt();
}
out.print(amount);
```

*Compound statement*

*Undefined in this scope*

- The *scope of access* of an identifier is the *block* (compound statement) where it is declared
- For code above to work, put output within braces

## *if...else* provides two paths of execution



```
if (quantity >= 5)
    out.print("Discount 10%");
else
    out.print("No discount");
```

- *Exactly one* of the output statements executes

## What is the output, on inputs $a = 3$ , $b = 5$ ?

```
int a,b;
a = in.getNextInt();
b = in.getNextInt();
if (a > 2)
    if (b < 4)
        out.print("OK");
    else
        out.print("a <= 2");
```

- The code above has a logic error. What is it?

## The pairing rule with *else*:

Each *else* is paired with the most recent unpaired *if* in the same scope.

*Example* (misleading code):

```
if (test1)
    if (test2)
        statement1();
else
    statement2();
```

(If *test2* fails, *statement2* executes.)

Good solution: use compound statements

## 2. Relational and Boolean expressions

### Relational operators

equal-to	==	not-equal-to	!=
greater	>	less-than-or-equal	<=
less-than	<	greater-or-equal	>=

- Expressions with relational operators have *Boolean* values
- Each operator has a complement
- *Tip*: don't compare *doubles* or *Strings* for equality

**== is a relational operator; = is not**

```

out.println("Enter your age: ");
int age;
age = in.nextInt(); if (age = 0) Watch out!
    out.println(" Invalid");

```

[agebad.cpp]

- Assignment as an *if* condition is valid syntax but usually logic error
- With assignment of constant, in effect no test is performed

**A Boolean variable (flag) stores a truth value**

*may be replaced by int* *a flag*

- `boolean invalid = (age < 0);`

...

```

if (invalid)
    out.println("Invalid age");

```

- *boolean* is a standard Java data type with a range of values {*false*, *true*} (0, 1)
- Boolean variables hold values for later use

## Logical operators

Operation	Eng.	Logic	Java	Example
Negation	not	$\neg$	!	!(price > cost)
Conjunction	and	$\wedge$	&&	a > b && b > c
Disjunction	or	$\vee$		x == 1    x == 2

- Nested *ifs* may express conjunction too:
 

```
if (age > 0)
    if (age < 120)
        out.print("Valid age");
```
- The above is equivalent to
 

```
if (age > 0 && age < 120)
    out.print("Valid age");
```

## Using logical operators

```
out.print("Enter 3 integers: ");
int a = in.nextInt(), |
    b = in.nextInt(),
    c = in.nextInt();
if (a == b && b == c)
    out.print("They're the same");
out.print("Enter your age: ");
int age;
age = in.nextInt();
boolean impossible = (age < 0 || age > 120);
if (! impossible)
    out.print("Thank you");
```

*Boolean variable*

[logops.cpp]

## Truth tables

- The logical operators in Java have the same results as the basic logic gates used in computer hardware

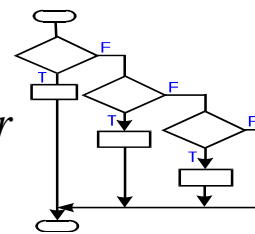
$a$	$!a$
1	0
0	1

$a$	$b$	$a    b$
0	0	0
0	1	1
1	0	1
1	1	1

$a$	$b$	$a \&\& b$
0	0	0
0	1	0
1	0	0
1	1	1

## 3. Multiway branches: *switch*

- Appropriate for mutually exclusive cases
- Test for exact matches, not ranges of values
- Selector should be an *int* or *char*
- Case labels may be stacked
- Don't forget to use *break*!
- default* triggers processing when no *case* label matches the selector




## Using *switch* with case labels


Nested *if*:

```
if (age == 1) out.print("Baby");
else if (age == 2) out.print("Toddler");
else if (age == 3) out.print("Preschool");
else out.print("Other");
```

*switch*:  Selector

```
switch (age)
{
     Case labels
    case 1: out.print("Baby"); break;
    case 2: out.print("Toddler"); break;
    case 3: out.print("Preschool"); break;
    default: out.print("Other");
}
```

## Stacking case labels

```
out.print("Grade: ");
char grade = in.getNextChar();
switch(toupper(grade))
{
    case 'A':
    case 'B':  Stacked to accept
                either 'A' or 'B'
        out.print("Honors"); break;
    case 'C':
    case 'D':
        out.print("Passing"); break;
    case 'E':
        out.print("Failing"); break;
    default:
        out.print("Invalid input");
}
```

[honors.cpp]

## Menus

- A *menu* is part of a user interface that presents options to the user
- A *switch* statement after the user choose a menu option can specify a response to each case (possible choice)

```
int option = in.nextInt();
switch(option)
{
    case 1: ... break;
    case 2: ... break;
    ...
    default: out.print("Invalid option");
}
```

## Syntax for branches

*branch-statement:*

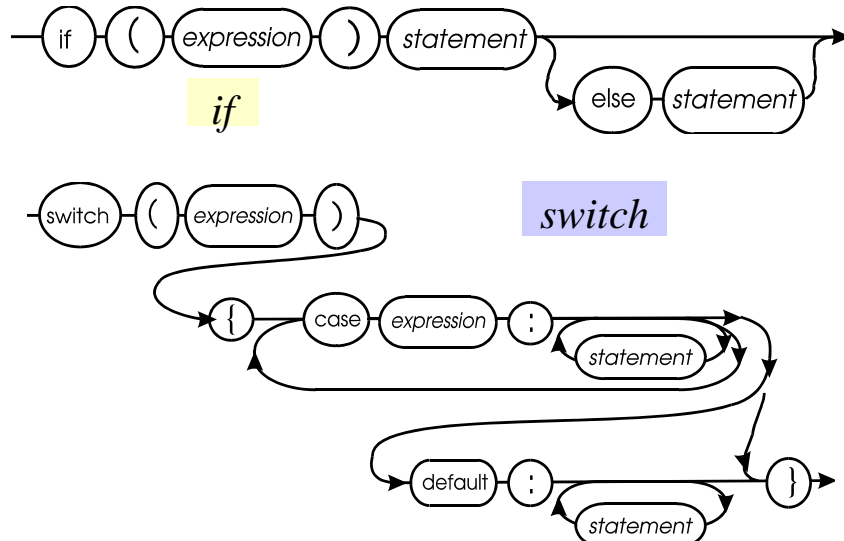
**if** ( *expr* ) *statement*

**if** ( *expr* ) *statement* **else** *statement*

**switch** ( *expr* ) *compound-statement*

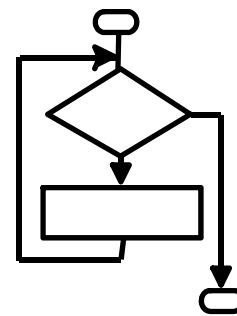
- *Statement* may be assignment, compound statement, *if* statement, loop statement, method call, etc.
- The subordinate *statement* in a *switch* is normally compound, with case labels, alternative statements, *breaks*

## Syntax diagrams for branches



## 4. Java loop statements

- Loops are based mathematically on the notion of *induction*
- Algorithm design is chiefly concerned with the design of loops that satisfy a specification
- A central goal in this course is for *all* students to successfully design and implement loops to solve simple problems



## The math of loops: Peano's axioms\*

1. 0 is a natural number
2. Every natural number  $n$  has a unique successor,  $n'$ , also a natural number
3. All natural numbers follow (1) or (2)

- *Significance:* These axioms, or assumptions, provide a formal logical basis to work with counting numbers.
- Computation is a formal way to manipulate numbers and objects representable by them.

\*1.  $0 \in \mathbb{N}$ ; 2.  $(\forall n \in \mathbb{N}) n' \in \mathbb{N}$ ; 3.  $(\forall n \in \mathbb{N}) n = 0 \vee \exists m \in \mathbb{N} \text{ s.t. } n = m'$

## Predecessors and addition

- 1 is shorthand for  $0'$  (successor of 0), 2 for  $0''$ , etc.
- Every natural number but 0 has a unique predecessor,  $pred(n)$ . Where  $m' = n$ ,  $m$  is the predecessor of  $n$ .
- $(a + b)$  is shorthand for
 
$$\begin{cases} a & \text{if } b = 0 \\ a' + pred(b) & \text{otherwise} \end{cases}$$
- **Addition is an example of a function that is computable using a loop**
- *Significance:* Any finite repetitive process can be specified by inductive methods.

### Example: $\Sigma$ (Sigma, summation)

- The summation operator  $\Sigma$  lets us add a series of numbers

- Case: 
$$\sum_{k=1}^n k = \begin{cases} 1 & \text{if } n = 1 \\ n + \sum_{k=1}^{n-1} k & \text{otherwise} \end{cases}$$

- E.g.: 
$$\sum_{k=1}^3 k = 1 + 2 + 3 = 6$$

- Generalizing to any function  $f$ :

$$\sum_{k=1}^n f(k) = \begin{cases} f(1) & \text{if } n = 1 \\ f(n) + \sum_{k=1}^{n-1} f(k) & \text{otherwise} \end{cases}$$

- Examples: if  $f(x) = 2$ ; if  $f(x) = x$ , etc.

### What triggers loop exit?

- A loop may terminate (a) after a *count*, or (b) after a certain *sentinel* value is input
- Other kinds of exit condition are possible
- Counted loop:

Do 50 times:  
draw a hyphen

- Sentinel controlled loop:

Repeat  
input a number  
until number is 0

*Sentinel value*

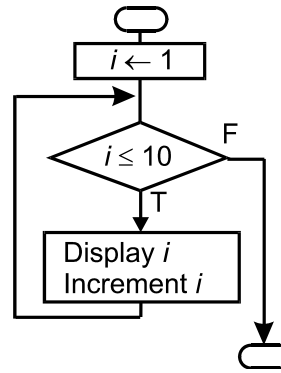
## Top-tested loops: *while*

A loop to count from 1 to 10:

```
int i = 1;
while (i <= 10)
{
    out.print(i + " ");
    ++i;
}
```

*Output:*

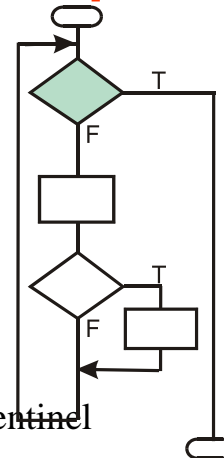
1 2 3 4 5 6 7 8 9 10



## A sentinel-controlled loop

```
final int sentinel = 0;
int input = -1, total = 0;
while (input != sentinel)
{
    out.print("Enter int " + sentinel
        + " to exit): ");
    input = in.nextInt();
    if (input != sentinel)
        total += input;
}
out.print("Total: " + total);
```

- Danger with sentinels: users can confuse sentinel value with non-sentinel
- What if *year = 99* were sentinel in a file of student records?



```

/* Readfile.java: Displays integers from
   file 'Readfile.txt'.  D. Keil 7/08 */
import java.util.Scanner;
import java.io.FileReader;
import java.io.FileNotFoundException;
public class Readfile
{
    public static void main(String[] args)
        throws FileNotFoundException
    {
        System.out.println("Reading Readfile.txt");
        FileReader reader = new FileReader("Readfile.txt");
        Scanner fin = new Scanner(reader);
        while (fin.hasNextInt())
        {
            int x = fin.nextInt();
            System.out.print(x + " ");
        }
        fin.close();
    }
}

```

## A file-reading loop

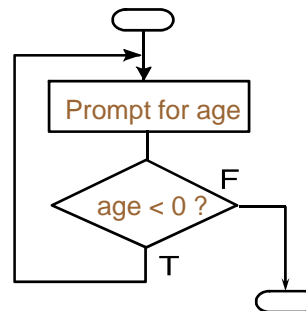
## Bottom-tested (*do...while*) loops

*Input validation:*

```

int age;
do {
    out.print("Age? ");
    age = in.nextInt();
} while (age < 0);

```



- The body of a bottom-tested loop will always execute at least once
- The loop above has a general exit condition
- For “repeat... until”, use a bottom-tested loop

## The *for* loop statement

*initialization*      *exit test*      *update*

```
for (int i = 1; i <= 10; ++i)
{
    out.print(i + " ");
}
Output: 1 2 3 4 5 6 7 8 9 10
```

- The *for* statement is a convenience; it is equivalent to a *while* loop, but exit test and update may be built into the header
- Syntax is flexible; e.g. *for (;)* is valid
- *for* is top tested

## Loop statements are interchangeable

```
int counter = 1;
do {
    out.print(counter + " ");
    ++counter;
} while (counter <= 10);
```

```
for (int i = 1; i <= 10; ++i)
{
    out.print(i + " ");
}
```

```
int i = 1;
while (i <= 10)
{
    out.print(i + " ");
    ++i;
}
```

[*docount.cpp*, *count2.cpp*,  
*count10.cpp*]

## Loops and strings

- String manipulation is a major application for loops
- The parameter of *charAt* is called an *index* or *subscript* to the string

```
// Finds first character after  
// first space in 'name'  
String name = in.nextLine();  
for (int i=0; i < name.length; i++)  
    if (name.charAt(i) == ' ')  
        out.println(name.charAt(i+1));
```

## Which statement to use?

- When the loop body should execute in every case, consider using *do...while*  
*Example:* User input and response to it
- When the loop body should sometimes *not* execute, use *while*

*Example:*

```
while (! infile.eof())...
```

## Loop syntax

*loop-statement:*

```
while ( expression ) statement
do statement while ( expression ) ;
for ( expr ; expr ; expr ) statement
```

## Loop semantics

- The *expression* in the *while* and *do...while* statements is the exit test.
- The *expressions* in the *for* statement are for initialization, exit test, and updating.

## Extracting substrings from strings

*Example:* to read a line of text and extract the third word

```
String s = in.nextLine();
// Find 2nd and 3rd spaces:
int num_spcs = 0, spc_1_loc = 0,
    spc_2_loc = 0;
for (int i=1; i < s.length(); i++)
    if (s.charAt(i) == ' ')
    {
        num_spcs++;
        if (num_spcs == 2)
            spc_2_loc = i;
        if (num_spcs == 3)
            spc_3_loc = i;
    }
int word_len = spc_3_loc - spc_2_loc;
String third_word = s.substring(spc_2_loc, word_len);
```

## 5. Writing correct loops

- Loop control and termination
- Errors and correctness
- Testing
- Tracing and debugging

## Boolean variables for loop control

```
final int sentinel = 0;
boolean done = false;
int total = 0;

while (! done)
{
    // <total> stores sum of input so far
    int input;
    out.print("Enter a number, 0 to exit: ");
    input = in.nextInt();
    if (input != sentinel)
        total += input;
    else
        done = true;
}
out.print(total);
```

*Boolean loop-control variable (flag)*

## Convergence of exit-test value assures termination

```
Input  $n$   
 $count \leftarrow 0$   
while  $n > 0$   
   $n \leftarrow \lfloor n / 2 \rfloor$   
   $count \leftarrow count + 1$ 
```

- The value  $n$  converges on 0
- This guarantees that the loop will exit

## An uncontrolled infinite loop

- The hardest-to-find infinite loop is one that *may* exit sometimes:

```
int power;  
while (power < 1000)  
{  
  out.print(power + " ");  
  power = power * 2;  
}
```

HANGS  
sometimes

- To exit, a loop must change a value that is tested in the exit condition:

```
int count = 0;  
while (count < 100)  
  out.print(count + " ");  
  ++count;
```

HANGS

## Interactive computing uses controlled infinite loops

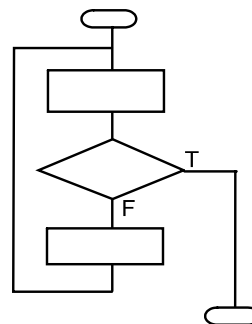
- Applications and operating environments today *normally* execute event-driven infinite loops:
 

```
do {
    event = get_user_input();
    respond_to(event);
} while (event != Quit);
```
- Non-algorithmic interactive computation occurs when input and output are interleaved with termination determined by input
- *Objects* (topic 6) may respond infinitely to messages if never sent a “destroy” message
- *Reference*: P. Wegner, *CACM*, 5/97

## Middle-tested loops (not recommended)

*Example:*

```
while(true)
{
    int input =
        in.nextInt();
    if (input == 0)
        break;
    out.print(input);
}
```



- *Unstructured code* is hard to understand, hence hard to debug

## Kinds of errors

- *Syntax*: violation of grammar rule; caught by compiler
- *Specification*: Program solves the wrong problem
- *Logic*: Programmer's chief concern; program produces incorrect or unpredictable output
- *Runtime*: Cause abnormal termination due to invalid operations, illegal memory access, etc. Preventable.

## Common pitfalls with loops

- Declaring inside a loop a value updated by the loop
- Iterating one too few times
- Iterating one too many times
- Impossible exit conditions
  - value tested not changed in loop body
  - value changed may fail to move toward exit value
- Exit condition that is never met
- [want examples]

## Invariants help verify correctness

```

sum ← 0
For i ← 1 to 5
  Input termi
  sum ← sum + termi

```

- Invariant: *sum* stores  $\sum_{k=1}^i term_k$
- Postcondition of this pseudocode:  
*sum* stores  $\sum_{k=1}^5 term_k$

## What are the invariants?

```

int input, total = 0;
input = in.nextInt();
while (input > 0)
{
  input =
  in.nextInt();
  total += input;
}
int i = 0, count = 0;
String input = in.nextLine();
while (i < s.length())
{
  if (input.charAt(i) == ' ')
    count++;
  ++i;
}

```

## Exercise all paths in testing a module

- To test a program containing an *if* statement, you must use test data that will cause a *true* value and a *false* value to appear as the result of the *if* condition
- With nested *ifs*, fully exercising your code (getting test results that reflect execution of each statement in program) may require 4 tests, 8, etc.

## Tracing a loop

- When a loop produces bad results, *tracing hidden values* helps in debugging
- **Trace statement** below shows garbage values

```
int count, input, total;
input = in.nextInt();
while (input > 0)
{
    out.print("input=" + input +
            " total=" + total);
    input = in.nextInt();
    total += input;
}
[trace.java]
```

### References

Cay Horstmann. *Big Java*, 3<sup>rd</sup> ed. Wiley, 2008, Chs. 5-6.

R. Johnson and D. Keil. *Introduction to Computer Programming Using Turbo Pascal*. West, 1995.

D. Keil. Menus. Classroom handout.

D. Keil. String manipulation and search. Classroom handout.