Programming and Software Development
COMP90041

Lecture 3

Control

Dr. Thomas Christy
thomas.christy@unimelb.edu.au

Review

- Hello World!
  - `Main`
- Input
  - `import`
  - `Scanner`
  - `args[ ]`
- Output
  - `print, println, printf`

Handling Command Line Inputs Netbeans

- To converts string to int:
  `Integer.parseInt(`*`string`*`)`

```
// Print double the command line integer
public class Hello3 {
    public static void main(String[] args) {
        System.out.println("Twice your number is "
                + 2 * Integer.parseInt(args[0]));
    }
}
```

Program Use

```
 frege% java Hello3 4
Twice your number is 8
```

Pitfall: Mixing with nextLine

- `nextLine()` reads up to and including newline
- Others do not read after the next word
- After `next`, `nextInt`, or `nextDouble`, `nextLine` just reads rest of current line (maybe nothing!)
- To read a number on one line followed by the next whole line:

```
int num = keyboard.nextInt();
keyboard.nextLine(); // throw away rest of line
String line = keyboard.nextLine();
```

- Ideally, avoid mixing `nextLine` with the others

Control

- Java's control statements allow you to control execution of code
- Conditional statements determine which statements to execute, possibly bypassing some
- Loop statements repeat some statements some number of times, under programmer control
- Programmer writes the program; user runs it
- It's up to the programmer to control the program based on the situation, including user actions

IF

- `if` statement decides whether or not to execute a statement based on a `boolean` expression
- Form:
  `if (expr) Statement`
- Executes the *Statement* if the *expr* is `true`, otherwise it does nothing
- The parenthesis are required
- The *expr* must be boolean
  - Use `!= 0` to test an int
- *E.g.*, negate `x` if it's negative:

```
if (x < 0) x = -x;
```

Compound Statements

- Most often, you need to execute multiple statements if the condition is true
- A compound statement turns multiple statements into a single statement that can be used in an `if`
- Also used in the other constucts in this lecture
- Form: { $Statement_1$; $\cdots$ $Statement_n$; }
- Don't follow the brace with semicolon
- This is a single statement that executes $Statement_1$; $\cdots$ $Statement_n$; in turn

```
if (x < 0) {
    x = -x;
    System.out.println(x + " is negative!");
}
```

Compound Statements

- Most often, you need to execute multiple statements if the condition is true
- A compound statement turns multiple statements into a single statement that can be used in an `if`
- Also used in the other constucts in this lecture
- Form: { $Statement_1$; $\cdots$ $Statement_n$; }
- Don't follow the brace with semicolon
- This is a single statement that executes $Statement_1$; $\cdots$ $Statement_n$; in turn

```
if (x < 0) {
    x = -x;
    System.out.println(x + " is negative!");Is it?
}
```

Compound Statements

- Most often, you need to execute multiple statements if the condition is true
- A compound statement turns multiple statements into a single statement that can be used in an `if`
- Also used in the other constucts in this lecture
- Form: { $Statement_1$; $\cdots$ $Statement_n$; }
- Don't follow the brace with semicolon
- This is a single statement that executes $Statement_1$; $\cdots$ $Statement_n$; in turn

```
if (x < 0) {
    System.out.println(x + " is negative!");
    x = -x;
}
```

Best Practice

- What's wrong with this?

```
if (x < 0)
    System.out.println(x + " is negative!");
    x = -x;
```

Best Practice

- What's wrong with this?

```
if (x < 0)
    System.out.println(x + " is negative!");
    x = -x;
```

- Best practice: always use braces, even for only one statement

```
if (x < 0) {
    x = -x;
}
```

Best Practice

- What's wrong with this?

```
if (x < 0)
    System.out.println(x + " is negative!");
    x = -x;
```

- Best practice: always use braces, even for only one statement

```
if (x < 0) {
    x = -x;
}
```

- Possible exception: whole `if` statement on one line
  - Unlikely to try to fit another statement on the same line

```
if (x < 0) x = -x;
```

If / Else

- Form:
  `if (expr)` $Statement_1$ `else` $Statement_2$
- Executes $Statement_1$ if the *expr* is `true`, else executes $Statement_2$
- Always executes exactly one of the statements
- Also best practice to surround $Statement_1$ and $Statement_2$ with braces

Code Layout

- Always use indentation to show code structure
  - More indented code is part of less indented code
  - Indent one level per nesting level of braces
  - Not required by Java, but demanded by human readers
- One common layout:

```
if (x < 0)
{
    System.out.println("negative");
}
else
{
    System.out.println("non-negative");
}
```

Code Layout

- A more compact layout:

```
if (x < 0) {
    System.out.println("negative");
} else {
    System.out.println("non-negative");
}
```

- Amount to indent for each level:
  - 1 is too little; more than 8 too much
  - 4 is popular
- Beware of tabs: they mean different levels of indentation to different programs
  - 8 columns is standard
  - Best to avoid tabs altogether

Else If

- Java has no special form for handling a chain of conditions
  - Just nest one `if-else` in the `else` part of another

```
if (x < 0) {
    System.out.println("negative");
} else if (x == 0) {
    System.out.println("zero");
} else {
    System.out.println("positive");
}
```

- Nest `if` and `if-else` within one another to any depth
  - Braces also makes this easier to read

Ternary Operator

- Java also has an if-else expression:

$$expr_1 \ ? \ expr_2 \ : \ expr_3$$

  - If $expr_1$ is `true` value is $expr_2$
  - If $expr_1$ is `false` value is $expr_3$

- This:

```
lesser = x < y ? x : y;
```

does exactly the same as this:

```
if (x < y) {
    lesser = x;
} else {
    lesser = y;
}
```

Switch

- `switch` statement chooses one of several cases based on an `int`, `short`, `byte`, or `char` value
- As of Java 7, it can also be a `String`: more useful
- Form:

```
switch (expr) {
  case value1:
    statements...
    break;
  :
  case valuen:
    statements...
    break;
}
```

Switch

- Execution begins by evaluating the expression
- It then looks for a `case` with matching *`value`*
- If it finds one, it begins executing with the next statement
- It stops executing when it reaches a `break` or the end of the `switch`
- Cases can be put in any order

Default

- As a special case, can use `default` in place of one `case` *`value`*
- If no `case` *`value`* matches, the code after the `default:` is executed, up to the next `break;`
- If no `case` *`value`* matches and there is no `default:`, `switch` statement finishes without executing any of the statements

Pitfall: Missing break

- If there is no `break` before the next `case` label, Java keeps executing until the next break
- **Very** easy to forget a `break`
- Best practice: even put `break` at end of last case
  - You may later add a new case after the last one
- If you leave out a `break` on purpose, put in a comment saying why
  - So whoever reads code (including you, later) knows it was omitted on purpose
- Exception: same code for multiple cases: just put common `case` labels together, followed by code

Example: Spell Out Morse Code

```
switch (ch) {
case '.':
    System.out.print("dot ");
    break;
case '-':
case '_':
    System.out.print("dash ");
    break;
case ' ':
    System.out.println(); // start new line
    break;
default:
    System.out.println("\nbad character '" + ch + "'");
    break;
}
```

While

- Form:
  `while (expr) Statement`
- If *expr* is `true` then:
  - Execute the *Statement*, then
  - Then go back and check *expr* again
  - Keep executing *Statement* as long as *expr* is true
- Stops when *expr* is `false` at top of loop
- Use to execute *Statement* an unlimited number of times, as long as *expr* is true
- Only useful if *Statement* can change value of *expr*
- Best practice again: put *Statement* in braces unless whole `while` fits on one line

While Example

```
public class whileExample {
    public static void main(String[] args) {
        int i = 1;
        int limit = 10;
        int sum = 0;
        while (i <= limit) {
            sum += i;
            ++i;
        }
        System.out.println("The sum is " + sum);
    }
}
```

Generated output

```
The sum is 55
```

Quick Poll

- bit.ly/tchristy
- qp.unimelb.edu.au/tchristy

link

Quick Poll:

- What will this print?

```
int x=3, y=0;
while (x >= 0) {
    y++;
    x--;
}
System.out.println(y);
```

A. 0
B. 1
C. 2
D. 3
E. 4

What will this print?

```
int x=3, y=0;
while (x >= 0) {
    y++;
    x--;
}
System.out.println(y);
```

A. 0
B. 1
C. 2
D. 3
E. 4

Quick Poll:

- What will this print?

```
int x=3, y=0;
while (x >= 0) {
    y++;
    x--;
}
System.out.println(y);
```

A. 0
B. 1
C. 2
D. 3
E. 4

Do While

- Form:
  `do` *Statement* `while (`*expr*`)`
- First execute *Statement*
- Then, if *expr* is `true`, go back and do it again
  - Keep executing *Statement* as long as *expr* is true
- Stops when *expr* is `false` at bottom of loop
- Use when you must execute *Statement* before testing *expr*
- Only useful if *Statement* can change value of *expr*
- Best practice again: put *Statement* in braces unless whole `while` fits on one line

Do While Example

```
public class dowhileExample {
    public static void main(String[] args) {
        int i = 1;
        int limit = 10;
        int sum = 0;
        do {
            sum += i;
            ++i;
        } while (i <= limit);
        System.out.println("The sum is " + sum);
    }
}
```

Generated output

```
The sum is 55
```

So, What's The Difference?

- `while` executes *`Statement`* zero or more times
- `do while` executes *`Statement`* one or more times
- Use `while` if you need to check a condition every time before executing the *`Statement`*
- Use `do while` if you need to execute the *`Statement`* before evaluating the *`expr`* every time
- Changing `limit` to 0 in the `while` example will print a sum of 0
- Changing `limit` to 0 in the `do while` example will print a sum of 1! That's wrong!
- `while` is more commonly used

For

- `for` is like `while` with initialisation and increment
- Form:
  `for (init ; test ; update) Statement`
- *init* is for variable initialisations, *e.g.*, `x = 0`
- *test* is a boolean expression to decide whether to execute `Statement`
- *update* is executed after each iteration
- Useful to execute a specific number of iterations
- Equivalent to:

```
init;
while(test) {
    Statement;
    update;
}
```

For Example

```
public class forExample {
    public static void main(String[] args) {
        int limit = 10;
        int sum = 0;
        for (int i = 0; i <= limit; ++i) {
            sum += i;
        }
        System.out.println("The sum is " + sum);
    }
}
```

Generated output

```
The sum is 55
```

For

- Any of *init*, *test* and *update* parts can be omitted
  - Infinite loop if *test* is omitted, but see below
- Variables declared in *init* part are scoped to the `for`: not available after the loop
- But you can declare variable before loop, and just initialise it in the *init* part
- Can include multiple initialisations and updates by separating them with commas
  - But if you put a declaration in the *init* part, you can only specify one type (not so useful)
- Only one *test* part is allowed, but can use `&&` and `||` to define it

Break and Continue

- Inside a `for`, `while` or `do while` loop, a `break` terminates the (innermost) loop immediately
- This is useful inside an `if` inside a loop
- A `continue` statement immediately returns to the top of the innermost loop and continues from there
- Can immediately exit whole program with `System.exit(0);` statement
  - Use 0 to indicate "success" and > 0 to indicate error
  - Will see a better way to handle errors later...

Pitfall: Common Loop Errors

- Infinite loop: loop never terminates
  - Forget to update the counter
  - Use wrong test
- Best practice: use < or <= (or > or >=) in loop test, not == or !=
- Off-by-one (fence post) error: one too many or few iterations
  - Start or end too low or too high
  - Use < instead of <= or vice-versa
- For `n` iterations, do one of:
  - `for (i=0 ; i<n ; ++i)` or
  - `for (i=1 ; i<=n ; ++i)`

Assert

- Use `assert(test)` to sanity-check your code
- Often program errors go undetected for a long time
- Very difficult to trace symptom back to cause
- Worst thing a program can do is not crash, but run normally producing wrong results
- `assert` stops the program if something is wrong
- *E.g.*, if at some point `x` must always be positive, add this statement at that point:

```
assert x > 0;
```

- Assertions not normally checked
  - Turn on checking by running program with:
    `java -{enableassertions ProgramName`

- Logging with Boolean?

Summary

- Use `if` or `if else` or `switch` to conditionally execute a statement
- Enclose multiple statements in {braces} to treat as a single statement
- Remember: end each `switch` case with a `break`
- Use `while` or `do while` or `for` loops to repeat a statement
- Use `break` to terminate loop immediately
- Use `contine` to restart a loop immediately

THE UNIVERSITY OF

MELBOURNE