Control flow, Conditions, Loops

Straight-Line Code

- When statements' order matters
  - getData();
  - groupData();
  - print();
- Make dependencies obvious
- Name methods according to dependencies
- Use method parameters
  - Data = getData();
  - groupedData = groupData(data);
  - printGroupedData(groupedData);
- Document the control flow if needed

Straight-Line Code

- When statements' order does not matter
  - Make code read from top to bottom like a newspaper
  - Group related statements together
  - Make clear boundaries for dependencies
    - Use blank lines to separate dependencies
    - Use separate method

Straight-Line Code

```
MailFooter createMailFooter(Mail mail) { … }
MailHeader createMailHeader(Mail mail) { … }
Mail createMail(Mail mail) {
        Mail mail = new Mail ();
        mail.header = createMailHeader(mail);
        mail.footer = createMailFooter(mail);
        mail.content = createMailContent(mail);

        return mail;
}
MailContent createMailContent(Mail mail) { … }
```

Straight-Line Code

- The most important thing to consider when organizing straight-line code is
  - Ordering dependencies
- Dependencies should be made obvious
  - Through the use of good routine names, parameter lists and comments
- If code doesn't have order dependencies
  - Keep related statements together

Using conditional statements

Using Conditional Statements

- Always use { and } for the conditional statements body, even when it is a single line:
- Why omitting the brackets could be harmful?
  - misleading code + misleading formatting

Using Conditional Statements

- Always put the normal (expected) condition first after the `if` clause
- Start from most common cases first, then go to the unusual ones

```
Response response = getHttpResponse(request);
if (response.code == Code.OK) {
        ....
} else if (response.code == Code.NotFound) {
        ...
}
```

Using Conditional Statements

- Avoid comparing to `true` or `false`:

```
if (containsErrors(response) == true) { … }
```

- Always consider the else case
  - If needed, document why the else isn't necessary

```
if (containsErrors(response) == true) {
        …
} else {
        // document why we're not doing anything in
                this case or throw an exception
}
```

Using Conditional Statements

- Avoid double negation

```
if (containsNoError(response)) { … }
```

- Write `if` clause with a meaningful statement

```
if (containsNoError(response))
;
else { … }
```

- Use meaningful boolean expressions, which read like a sentence

Using Conditional Statements

- Be aware of copy/paste problems in `if-else` bodies

```
if (someCondition) {
    Student s = getStudent(arg2);
    s.sendMail();
    s.sendSMS();
} else {
    Student s = getStudent(arg1);
    s.sendMail();
    s.sendSMS();
}
```

Use Simple Conditions

- Do not use complex `if` conditions
  - You can always simplify them by introducing boolean variables or boolean methods
  - Complex boolean expressions can be harmful
  - How you will find the problem if you get `ArrayIndexOutOfBoundsException`?

```
if (x > 0 && y < 0 && y > width − 1 && x < height - 1
&& arr[x+1][y] < 0 && arr[x-1][y-1] < 0)
```

  - Instead:
    - name the condition, have separate lines (to put break-points if needed)
    - Use OOP

Use Decision Tables

- Sometimes a decision table can be used for simplicity

```
HashTable table = new HashTable();
table.add("A", …);
table.add("B", …);

Student student = table[studentKey];
student.attendCourse();
```

Positive Boolean Expressions

- Starting with a positive expression improves the readability

  if (containsErrors) instead of if(!containsErrors)

- Use De Morgan's laws for negative checks

Use Parentheses for Simplification

- Avoid complex boolean conditions without parenthesis

```
if (x > 0 && x < y || x == y * 3)
```

- Using parenthesis helps readability as well as ensure correctness
- Too many parenthesis have to be avoided as well
  - Consider separate Boolean methods or variables in those cases

Boolean Expression Evaluation

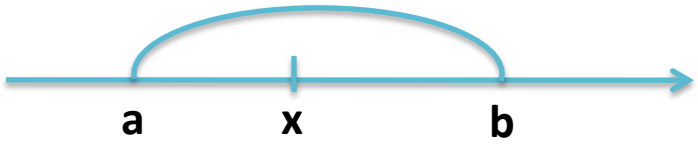
- Most languages evaluate from left to right
  - Stop evaluation as soon as some of the boolean operands is satisfied

    if (falseCondition ~~&& other~~)

    if (trueCondition ~~|| other~~)
- Useful when checking for `null`
- There are languages that does not follow this “short-circuit” rule

Numeric Expressions as Operands

- Write numeric boolean expressions as they are presented on a number line
  - Contained in an interval
    - if (x > a && b > x)
    - if (a < x && x < b)
  - Outside of an interval
    - if (a > x || x > b)
    - if (x < a || b < x)



Avoid Deep Nesting of Blocks

- Deep nesting of conditional statements and loops makes the code unclear
  - More than 2-3 levels is too deep
  - Deeply nested code is complex and hard to read and understand
  - Usually you can extract portions of the code in separate methods
    - This simplifies the logic of the code
    - Using good method name makes the code self-documenting

Deep Nesting – Example

```
if (maxElem != MAX_VAL)
{
    if (arr[i] < arr[i - 1])
    {
        if (arr[i - 1] < arr[i - 2])
        {
            if (arr[i - 2] < arr[i - 3])
            {
                maxElem = arr[i - 3];
            }
            else
            {
                maxElem = arr[i - 2];
            }
        }
        else {}
}
```

Deep Nesting – Example

```
private static int max(int i, int j) {
    return (i < j) ? j : i;
}

private static int max(int i, int j, int k) {
    if (i < j) {
        int maxElem = max(j, k);
        return maxElem;
    } else {
        int maxElem = max(i, k);
        return maxElem;
    }
}
```

Deep Nesting – Example

```
private static int findMax(int[] arr, int i) {
  if (arr[i] < arr[i + 1]) {
    int maxElem = max(arr[i + 1], arr[i + 2], arr[i + 3]);
    return maxElem;
  } else {
    int maxElem = max(arr[i], arr[i + 2], arr[i + 3]);
    return maxElem;
  }
}
```

Using Case Statement

- Choose the most effective ordering of cases
  - Put the normal (usual) case first
  - Order cases by frequency
  - Put the most unusual (exceptional) case last
  - Order cases alphabetically or numerically
- Keep the actions of each case simple
  - Extract complex logic in separate methods
- Use the default clause in a `case` statement or the last `else` in a chain of `if-else` to trap errors

Incorrect Case Statement

```
void processNextChar(char ch) {
  switch (parseState) {
    case InTag:
      if (ch == ">”) {
        System.out.println("Found tag: {0}", tag);
        text = "";
        parseState = ParseState.OutOfTag;
      } else {
        tag = tag + ch;
      }
      break;
    case OutOfTag:
```

Case Statement

```
void processNextChar(char ch) {
  switch (parseState) {
    case InTag:
      ProcessCharacterInTag(ch);
      break;
    case OutOfTag:    …
    default: throw new InvalidOperationException(
        "Invalid parse state: " + parseState);
  }
}
```

Case – Best Practices

- Avoid using fallthroughs
- When you do use them, document them well

```
switch (..) {
        case 1:
        case 2:
                doSth();
                // fallthrough
        case 12:
                doSthElse();
```

Control Statements

- For simple if-else-s, pay attention to the order of the `if` and `else` clauses
  - Make sure the nominal case is clear
- For if-then-else chains and case statements, choose the most readable order
- Optimize boolean statements to improve readability
- Use the default clause in a case statement or the last else in a chain of if-s to trap errors

Using Loops

Using Loops

- Choosing the correct type of loop:
  - Use `for` loop to repeat some block of code a certain number of times
  - Use `foreach` loop to process each element of an array or a collection
  - Use `while` / `do-while` loop when you don't know how many times a block should be repeated
- Avoid deep nesting of loops
  - You can extract the loop body in a new method

Loops: Best Practices

- Keep loops simple
  - This helps readers of your code
- Treat the inside of the loop as it were a routine
  - Don't make the reader look inside the loop to understand the loop control
- Think of a loop as a black box:

```
while (!inputFile.endOfFile() && !hasErrors) {}
```

Loops: Best Practices

- Keep loop's housekeeping at the start or at the end of the loop block – i++ at the start (or end 0 not in the middle of the loop)
- Use meaningful variable names to make loops readable – sometimes i and j are not enough

Loops: Best Practices

- Avoid empty loops
- Be aware of your language (loop) semantics
  - C# – access to modified closure
  - Is loop limit clause evaluated every time or it is cached?

Loops: Tips on for-Loop

- Don’t explicitly change the index value to force the loop to stop
  - Use `while`-loop with `break` instead
- Put only the controlling statements in the loop header

```
for (i = 0, sum = 0; i < length; sum += arr[i], i++)
```

Loops: Tips on for-Loop

- Avoid code that depends on the loop index's final value

```
for (i = 0; i < length; i++)
{
  if (array[i].id == key)
  {
    break;
  }
}

// Lots of code
…
return (i >= length);
```

Loops: `break` and `continue`

- Use `continue` for tests at the top of a loop to avoid nested `if`-s
- Avoid loops with lots of `break`-s scattered trough it
- Use `break` and `continue` only with caution

How Long Should a Loop Be?

- Try to make the loops short enough to view it all at once (one screen)
- Use methods to shorten the loop body
- Make long loops especially clear
- Avoid deep nesting in loops

Use other control flow structures

The `return` Statement

- Use `return` when it enhances readability
- Use `return` to avoid deep nesting

```
void parseString(String str) {
  if (str != null) {
    // Lots of code
  }
}
```

- Avoid multiple `return`-s in long methods

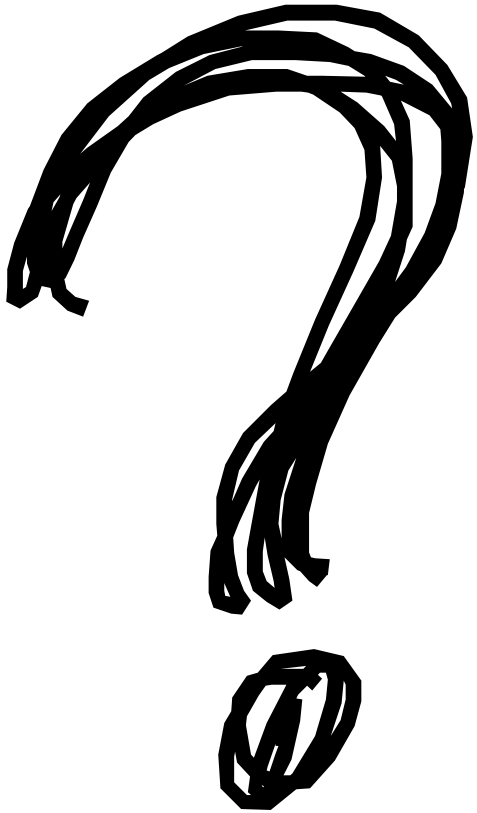
Recursion

- Useful when you want to walk a tree / graph-like structures - GUIs
- Be aware of infinite recursion or indirect recursion

Recursion Tips

- Ensure that recursion has end
- Verify that recursion is not very high-cost
  - Check the occupied system resources
  - You can always use stack classes and iteration
- Don’t use recursion when there is better linear (iteration based) solution, e.g.
  - Factorials
  - Fibonacci numbers
- Some languages optimize tail-call recursions


СВ. КЛИМЕНТ ОХРИДСКИ



THANK YOU