Variables, expressions, constants

Initially Assigned Variables

- Static variables
- Instance variables of class instances
- Instance variables of initially assigned struct variables
- Array elements
- Value parameters
- Reference parameters
- Variables declared in a catch clause or a foreach statement

Initially Unassigned Variables

- Instance variables of initially unassigned struct variables
- Output parameters
  - Including the `this` variable of struct instance constructors
- Local variables
  - Except those declared in a `catch` clause or a `foreach` statement

Guidelines for Initializing Variables

- When the problems can happen?
  - The variable has never been assigned a value
  - The value in the variable is outdated
  - Part of the variable has been assigned a value and a part has not
    - E.g. `Student` class has initialized name, but faculty number is left unassigned
- Developing effective techniques for avoiding initialization problems can save a lot of time

Variable Initialization

- Initialize all variables before their first usage
  - Local variables should be manually initialized
  - Declare and define each variable close to where it is user or better at its declaration
  - Special attention deserve the iterables and counters
  - Check the need for re-initialization
  - Check input parameters for validity
  - Ensure objects cannot get into partially initialized state – Name and ID

Variable Initialization

- Don’t define unused variables
- Don’t use variables with hidden purpose
  - int mode = 1; // read ; 2 -> write
- Assign the result of a method to a variable
  - return 60 * 60 * wagePerHour;

Visibility of Variables

- Variables' visibility is explicitly set restriction regarding the access to the variable
  - `public`, `protected`, `internal`, `default`, `private`
- Always try to reduce the variables scope and visibility
  - This reduces potential coupling
  - Avoid public fields (exceptions: `readonly` / `const` / `static final`)
  - Don't access fields directly

Span of Variables

- Variable span
  - The of lines of code (LOC) between variable usages
  - Average span can be calculated for all usages
  - Variable span should be kept as low as possible
- Rules for usage
  - Define variables at their first usage, not earlier
  - Initialize variables as late as possible
  - Try to keep together lines using the same variable
- Always define and initialize variables just before their first use!

Variable Live Time

- Variable live time
  - The number of lines of code (LOC) between the first and the last variable usage
  - should be kept as low as possible
- Rules for minimizing span:
  - Define variables at their first usage
  - Initialize variables just before their first use
  - Try to keep together lines using the same variable

Example

```
1   int count;
2   int[] students = new int[100];
3   for (int i = 0; i < students.length; i++)
4   {
5     students[i] = i;
6   }
7   count = 0;
8   for (int i = 0; i < students.length/ 2; i++)
9   {
10    students[i] = students[i] * students[i];
11  }
12  for (int i = 0; i < students.length; i++)
13  {
14    if (students[i] % 42 == 0)
15    {
16      count++;
17    }
18  }
19  System.out.println(count);
```

live time = 19

span = (5+8+2) / 3 = 5

Advantages of short span/time

- You can grasp the code easier
- Reduces the chance of initialization errors and any other errors
- Increases readability

Best Practices

- Initialize variables used in a loop immediately before the loop
- Don’t assign a value to a variable until just before the value is used
  - Never follow the old C / Pascal style of declaring variables in the beginning of each method

Best Practices

- Begin with the most restricted visibility
  - Expand the visibility only when necessary
- Group related statements together

Variables Usage

- Variables should have single purpose
  - Never use a single variable for multiple purposes!
  - Economizing memory is not an excuse
- Can you choose a good name for variable that is used for several purposes?
  - Example: variable used to count students or to keep the average of their grades
  - Proposed name: `studentsCountOrAvgGrade`

Variables Naming

- The name should describe the object clearly and accurately, which the variable represents
  - Bad names: i18n, __hkcd, rcf, a1, a20
- Address the problem, which the variable solves – "what" instead of "how”
  - Bad names: myArray, customerFile, customerHashTable

Naming Rules

- Naming depends on the scope and visibility
  - – Bigger scope, visibility, longer lifetime → longer and more descriptive name: customerWallet
  - – Variables with smaller scope and shorter lifetime can be shorter: i and j
- The enclosing type gives a context for naming:
  - – Class Account { User holder; }

Optimal Name Length

- Optimal length – 10 to 16 symbols
  - Too long – numberOfPeopleAttendingCleanCodeCourse
  - Too short – n
  - Correct - cleanCodeAttendeesCount

Naming Data Types

- Naming counters – readersCount, rowsCount, studentsCount
- Naming variables for state – accountState, memoryState
- Naming temporary variables
  - k, aa, tmp, var2
  - index, value, count

Naming Data Types

- Name Boolean variables with names implying "Yes/ No” /   “True/False” answers – isReadable, used, available, ready, valid
- Booleans variables should bring "truth" in their name
  - notReadable, notAvailable
  - isReadable, available

Naming Data Types

- Naming enumeration types
  - Use build in enumeration types – enums – DaysOfWeek.MONDAY, DaysOfWeek.Tuesday
  - Or use appropriate prefixes (e.g. in PHP/JS) – weekDayMonday, weekDayTuesday
- Naming constants – use capital letters – BUFFER_SIZE
- Follow language style guides

Naming Convention

- Some programmers resist to follow standards and conventions
  - But why?
- Conventions benefits
  - Transfer knowledge across projects
  - Helps to learn code more quickly on a new project
  - Avoid calling the same thing by two different names

Naming Convention

- When should we use a naming convention?
  - Multiple developers are working on the same project
  - The source code is reviewed by other programmers
  - The project is large
  - The project will be long-lived
- You always benefit from having some kind of naming convention!

Language-Specific Conventions

- C# and Java / JavaScript conventions
  - i and j are integer indexes
  - Constants are in ALL_CAPS separated by underscores (sometimes PascalCase in C#)
  - Variable and method names use uppercase in C# and lowercase in JS and Java for the first word
  - The underscore _ is not used within names - Except for names in all caps

Standard Prefixes

- Hungarian notation – not used
- Semantic prefixes (ex. `btnSave`)
  - Better use `buttonSave`
- Do not miss letters to make name shorter
- Abbreviate names in consistent way throughout the code
- Create names, which can be pronounced (not like `btnDfltSvRzlts`)
- Avoid combinations, which form another word or different meaning (ex. `preFixStore`)

Names to Avoid

- Document short names in the code
- Remember, names are designed for the people, who will read the code
  - Not for those who write it
- Avoid variables with similar names, but different purpose it – StudentStatus, StudentCurrentStatus
- Avoid names, that sounds similar – tree, trie, try
- Avoid digits in names

Names to Avoid

- Avoid words, which can be easily mistaken – adsl, adcl, adctrl, actrl, acld
- Avoid using non-English words
- Avoid using standard types and keywords in the variable names – int, list, dictionary, map
- Do not use names, which has nothing common with variables content
- Avoid names, that contains hard-readable symbols / syllables, (prefer being searchable) - e.g. Brikstronst

Avoid Complex Expressions

- Never use complex expressions in the code!
  - Incorrect example: arr[xCoord[findMin(i)-n-i]] [yCoord[findMin(j)-n-j]]
- Complex expressions are evil because:
  - hard to read code,
  - hard to understand code,
  - hard to debug,
  - hard to modify
  - hard to maintain

Avoid Magic Numbers and Strings

- What is magic number or value?
  - Magic numbers / values are all literals different than `0`, `1`, `-1`, `null` and `""` (empty string)
- Avoid using magic numbers / values
  - They are hard to maintain - in case of change, you need to modify all occurrences of the magic number / constant
  - Their meaning is not obvious - what the number `1024` means?
- 3.1415926 * a * b

Constants

- C# - compile-time, run-time
- JS – no constants

When to Use Constants?

- When we need to use numbers or other values and their logical meaning and value are not obvious
- File names – CONFIG_FILE_NAME
- Math constants – E, PI
- Bounds, Limits and ranges – BUFFER_SIZE

When to Avoid Constants?

- Sometime it is better to keep the magic values instead of using a constant
  - Error messages and exception descriptions
  - SQL commands for database operations
  - Titles of GUI elements (labels, buttons, menus, dialogs, etc.)
- For internationalization purposes use resources, not constants
  - Resources are special files embedded in the assembly / JAR file, accessible at runtime

THANK YOU