Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής & Τηλεπικοινωνιών

Προηγμένες Μέθοδοι Προγραμματισμού

ΠΜΣ 2022-23 (M135.CS1E, M135.CS23B, M135.IC1E, παλαιό: M117)

Θεμελιώδεις έννοιες (επανάληψη)

Δρ. Κώστας Σαΐδης (saiko@di.uoa.gr)

Θεμέλιοι προγραμματιστικοί λίθοι

- Θεμελιώδεις αφαιρέσεις
- Threads
- Memory (Stack and Heap)

Η Θεμελιώδης Αφαίρεση

Ποια είναι;

Η συνάρτηση

- Με διάφορα ονόματα και παραλλαγές, ανάλογα με τη γλώσσα
- Function, Procedure, Method, Subroutine, Message (etc.)
- Divide and conquer
- Black box abstraction

Η συνάρτηση και ο κώδικας (γενικά)

- Ένα "κομμάτι κώδικα" που λαμβάνει κάποιες εισόδους, εκτελεί έναν ή περισσότερους υπολογισμούς και "επιστρέφει / παράγει" 0 ή περισσότερες εξόδους
- `<return_type> functionName(<type1> arg1Name, <type 2> arg2Name)`
- `functionName(arg1Name: <type1>, arg2Name: <type 2>) : <return_type>`
- Pure function (function as in math): για την ίδια είσοδο, παράγεται πάντα το ίδιο αποτέλεσμα, χωρίς "παρενέργειες (side-effects)"

Η μνήμη και τα δεδομένα (γενικά)

- Ο "χώρος" στον οποίο αποθηκεύονται τα δεδομένα (όσο τρέχει το πρόγραμμα - προσωρινά).
- Δεδομένα: Μεταβλητές, δομές δεδομένων, πεδία αντικειμένων, κλπ.

Variables

- Addressable/identifiable "memory slots" that (can) hold a piece of data (bits and bytes).
- Their type (symbolic name) determines the "actual meaning / form" of the data (what these bits and bytes "represent").

Το σύστημα τύπων (type system)

- Ένα σύστημα κανόνων που προσπαθεί να αποδώσει "νόημα" στο πρόγραμμα (στις "συναρτήσεις" του και στα "δεδομένα" του, αν/όπου υπάρχει τέτοια διάκριση).
- Στόχος (type checking): να εννοιολογήσει και να κατηγοριοποιήσει τα στοιχεία του προγράμματος, ώστε να διασφαλίσει ιδιότητες και να εγγυηθεί συμπεριφορές, αποτρέποντας λάθη.
- Type safety
- Memory safety

Type safety (type system soundness)

- Well-typed programs cannot "go wrong" (R. Millner).
- All expressions accepted by the type system must evaluate to a value of the appropriate type (rather than produce a value of some other, unrelated type or crash with a type error) (Wikipedia).

Type safety (contd.)

- Progress and Preservation (Wright and Felleisen)
  - A well-typed program never gets "stuck": every expression is either already a value or can be reduced towards a value in some well-defined way.
  - After each evaluation step, the type of each expression remains the same (that is, its type is preserved).

Memory safety

- Protect against various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers.
- Requires runtime checks.
- Thread safety is relevant but also different (another lecture).

Dynamic vs static typing

Statically typed languages:

- Resolve the types of variables at compile-time (statically).
- The type of a variable cannot change at runtime (e.g. Java, C#).

Dynamically typed languages:

- Resolve the types of variables at runtime (dynamically, during execution).
- The type of a variable can change at runtime (e.g. Ruby, C, Javascript).

Δηλαδή;

- Τι σημαίνει το "the type of a variable can/cannot change";
- Να μπορείς να αλλάξεις τη "μορφή" του memory slot (π.χ. είχα δεσμεύσει έναν "χώρο" για να κρατάω "string" και τώρα, στον ίδιο "χώρο" (στην ίδια διεύθυνση), θέλω να κρατάω "integer").
- Δεν αλλάζει το ίδιο το value, ο "χώρος" για να αποθηκευθεί αυτό αλλάζει.

Strong vs weak typing

Strongly typed languages:

- Guarantee type conformance.
- You cannot convert the value of a variable to an incompatible/wrong type (e.g. Java, Ruby).
- Usually implies memory-safety, too.

Weakly typed languages:

- Leaky/broken type abstactions.
- For example, C does not prevent you from shooting yourself in various painful ways.
- While Javascript has some broken primitives/arithmetics/coercions.
- Caution though: C is not memory-safe, Javascript is.

Type inference

- The ability to automatically decide upon the type of a variable at compile-time (without requiring the programmer to explicitly declare it).

```
List<String> list = new ArrayList<>();
var list = new ArrayList<String>();
```

The memory of running programs

- The stack
- The heap

The stack

- Also named as Call stack, Execution stack, Program stack, Runtime stack
- A LIFO data structure managed automatically by the (high-level) programming language
- Primary role: to keep track of the specific point to which each a function call should return control after execution.
- Function calls are nested: a function calls another function, that calls another function, etc.
- A function can also call itself (recursion).

The stack frames

- The elements of the stack, also named activation records (function activations/invocations)
- Each frame usually holds:
  - The parameters passed to the function call (if any)
  - The local variables of the function
  - The return address of the caller (the frame pointer)

Stack overflow

- Programming languages apply a limit to the size of the stack (e.g. 10.000 framses)
- If the program exceeds the limit a Stack Overflow error is thrown
- Most probably, an infinite loop / non-terminated recursion has occurred

The stack / call / error trace

- I suppose you' ve seen one, right?
- Stack unwinding: pop one or more frames off the stack to resume execution elsewhere in the program
- Exception handling:
  - The stack frame holds entries specifying exception handlers (non-local control structures that may exist in an outer/caller function)
  - When an exception is thrown (in the body of the currently active function), the programming language unwinds the stack until a suitable handler (trap) is found that can handle (catch) the specific exception

The primitive types

- Many languages have a small set of primitive types
- Such as char, int, bool, etc, that represent "simple forms" of data
- In order to be able to easily store such data in the stack frame (in the space kept for holding the local variables)

The reference types

- Let's recall our OOP discussion and the `new` operator that creates instances of classes
- The classes are usually called reference types and the instances/objects are also called references
- Where are theses instances/objects held?

The heap

- The space where dynamic, not-in-the-stack, memory allocation occurs
- Addressable/identifiable through pointers
- Manual memory management
- Automatic memory management

Manual memory management

- The programmer is responsible for allocating and de-allocating the required portions of the heap space required to hold the data of the program
- Hard, error-prone, security issues with buffer overflow/underflow

Automatic memory management

- The programming language is responsible for allocating and de-allocating the required portions of the heap space required to hold the data of the program
- Memory safety guarantees with additional runtime checks
- The `new` operator performs the allocation (and hides the underlying pointers from the programmer)
- The programming language's runtime employs Garbage collection techiques for performing the de-allocation, recycling unused memory to be reused by the same program
  (we will discuss GC in more detail in another lecture)

Out of memory errors

- Once again, the programming language enforces a limit on the size of the heap space (in automatically-managed memory)
- If your program exceeds this limit, an OutOfMemory error is thrown (not enough space exists for storing a new instance of some class/type)

Stack vs. Heap

- Stack is faster (as fast as it gets, usually utilizing the CPU registers)
- Heap is bigger (main memory + swap space on disk(s), using Virtual Memory techniques)
- Both StackOverflow and OutOfMemory errors are usually non-recoverable (causing abnormal program termination)

Wait, who's actually running the program?

- Who is executing the code?
- Who is invoking the functions?

The threads

- Also called lightweight processes.
- Similar to processes, threads have their own execution environment, comprising a stack, registers and program counter. Depending on the OS, they may also have a thread-local storage.
- Creating a new thread requires fewer resources than creating a new process.

The threads

- Threads exist/live in a process, sharing the same memory space (heap).
- Every process has at least one thread, executing the main execution path (the one followed by the main method).
- Threads are independent execution paths within the process, executing
  simultaneously and asynchronously to each other.

Multi-threaded vs single-threaded

- Some languages are designed to be single-threaded (e.g. Javascript), some to be multi-threaded (e.g. Java)
- In the first case, the runtime environment may support threads in a transparent fashion for the programmer (e.g. Browser)
- In the second case, the programmer is responsible for spawning and managing any additional threads, yet the language can provide a lot of helpful utilities

Simple threading example in Java

```
public class MyRunnable implements Runnable {
  public void run ( ) {
    while ( true ) {
      println ("Hello from another thread") ;
      try { Thread.sleep (1000) ; }
      catch( InterruptedException ie ) { println ("Interrupted") ; }
    }
  }

  public static void main(String[] args) {
    Thread t = new Thread(new MyRunnable());
    t.start();
    // continue with other tasks
  }
}
```

The JVM Main Thread

- Every java application starts with one thread, called the main thread.
- Behind the scenes, the JVM spawns additional ‘‘internal’’, housekeeping
  threads – e.g., for garbage collection or for signal handling. Such threads
  are not ‘‘visible’’ to the developer.
- The main thread is responsible for running the main method:

```
public static void main(String[] args)
```

Keep in mind

- Since a common heap space is shared by multiple threads, we can easily shoot ourselves on the foot (and corrupt data or miscalculate something), if we don't synchronize the memory access operations.
- We will discuss synchronization in a another lecture.