Introduction

Object-Oriented Programming

236703

Summer 2018
Course Staff

• Lecturer: Prof Yossi Gil (yogi@cs)
• TA:
  – Guy Suday (TA in charge) (guy-suday@cs),
• Supporting staff:
  – Eric Kiel (eric.oop.course@gmail)
Slide Credits

• Original slides by Prof. Yossi Gil
  – Created in the late 20th century
• Retouched by Keren Lenz
  – ... and probably others over the years
• Latest retouches by Tal Cohen
  – Any mistakes, blame me.
This Course

• ... is about abstraction mechanisms
  – not object oriented programming
  – not object oriented design

• A comparative approach
  – not about a specific language

• Learn language features that make writing high quality code easier
  • This course will make you better programmers!
Motivation

• Permanent software change principle
  – Software that does not change dies
  – Changes: bug fixes, feature additions, porting...
  – How can we make changes a little bit safer?

• Déjà vu principle
  – two similar code segments will lead to failure rising from unsynchronized changes

• The solution: Encapsulation and Abstraction to the rescue...
Encapsulation

• Rationale: If a program element, e, is accessible by many other elements, then changing e will affect all its dependents

• Encapsulation: A mechanism that limits the access to certain elements of the program
  – Representation of the data can be changed
  – Algorithms can be changed
Abstraction

• The process of:
  – identifying similarities between code fragments
  – eliminating these similarities by making a single instance of the repeated code
    • Possibly parameterized

• Allows a piece of code to ignore irrelevant details
Baseline program: No Abstraction

```cpp
int a[100];
for (i = 0 ; i < 100; i++)
    cout << a[i];
```

- The number **100** appears twice
- What happens if we change the array size and forget to modify the array bound?
#define N 100

int a[N];

for (i = 0 ; i < N; i++)
    cout << a[i];
Abstracting constants: enum definition

```cpp
enum {N = 100};

int a[N];
for (i = 0 ; i < N; i++)
    cout << a[i];
```
Abstracting array size: macro definition

#define SIZEOF(array)  
(sizeof(array)/sizeof(array[0]))

int a[100];
for (i = 0 ; i < SIZEOF(a); i++)
    cout << a[i];
Array size in Java

```java
int[] a = new int[100];
for (int i = 0; i < a.length; i++)
    System.out.print(a[i]);
```
Abstracting array iteration: enhanced for loop

```java
int[] a = new a[100];
for (int ai : a)
    System.out.print(ai);
```
Abstraction mechanisms improve over time...

Generally speaking, levels of abstraction improved as programming languages evolved.

- First HLL: FORTRAN (1957)
- Types: Algol (1958)
- Modules: Algol 68 (1968)
- Classes: Simula (1965)
- Generics: ML (1973)
The OOP Manifesto

1. Everything is an *object*.

2. Computation is performed by objects communicating with each other by sending and receiving *messages*.

3. Each object has its own memory (*state*), which consists of other objects.

4. Every object is an instance of a *class*.

5. The class is the repository for *behavior*. All instances of the same class can perform the same actions.

6. Classes are organized into a singly rooted tree, called the *inheritance hierarchy*. 
Additional Abstraction Mechanisms

• Types
  – Join fields and operations
  – Well defined instantiation process

• Methods
  – Allows the programmer to concentrate on what is being done, not how it is being performed
  – Syntactically (and semantically) bound to a certain type

• Encapsulation
  – Minimize dependencies
  – Allow consistent behavior
Additional Abstraction Mechanisms

• Inheritance and polymorphism
  – Avoid code duplication
  – Represent relations and allow substitution

• Dynamic Binding
  – A method call is not tied up with a specific method body
  – Provides polymorphism

• Genericity
  – Similarity between operations on different types (or other parameters)
What is a Type?

• A set of values
• A set of operations on those values

• Properties
  – Set of possible values
  – Set of operations (arithmetic? assignment? display?)
  – Relations with other types
  – Representation (location, stack, heap..)
  – Coercions to other types

• Classes in an object-oriented language are an elaboration of this notion to include relations among types and operations (class inheritance, method inheritance)
Types

• Often broken down into
  – Elementary (primitive, atomic types)
    • C types: int, char, double, float...
  – Compound: generated using *type constructors* – operators that accept types and generate a new type.
  • C constructs: struct, enum, union, arrays, pointers...
Types and Operations

• Certain operations are legal for values of each type

  – It *doesn’t* make sense to add a function pointer and an integer in C
  – It *does* make sense to add two integers
  – It *might* make sense to add an integer and a pointer (this is what an array reference might be in some languages)

  – But all have the same assembly language implementation!
Type Systems

• A language’s type system specifies which operations are valid for which types

• Type checking ensures that operations are used with the correct types

• Three kinds of languages:
  – *Untyped*: No type checking (machine code)
  – *Statically typed*: The compiler computes the type of each expression. All or almost all checking of types is done as part of compilation (C, Java)
  – *Dynamically typed*: Almost all checking of types is done as part of program execution (Python, Smalltalk).
The Type Wars

• Static Typing:
  – Catches many programming errors at compile time (safety)
  – Enforces the design decisions (clarity)
  – Avoids overhead of runtime type checks (performance).

• Dynamic Typing:
  – Flexible
  – Less code – no need to declare types
  – Rapid prototyping easier in a dynamic type system.
Supported? Yes/No?

DT = Dynamic Typing; ST = Static Typing

interface List { void add(Object o); }
class ArrayList implements List { void add(Object o) { ... } }
class LinkedList implements List {
    void addFirst(Object o) { ... }
    void add(Object o) { ... }
}

LinkedList w = new LinkedList();  // DT ✔️ ST ✔️
w.addFirst("1");

List x = new LinkedList();  // DT ✔️ ST ✗
x.addFirst("1");

List y = new ArrayList();  // DT ✗ ST ✗
y.addFirst("1");

LinkedList z = new ArrayList();  // ST ✗
z.addFirst("1");  // DT ✗
Typing Strength

• Strong typing:
  – An unbreakable association between a value and its type
  – An operation fails on arguments which have the wrong type

• Weak typing:
  – One can change the value-type association
  – Language implicitly converts types when used
Strong Typing -- What does it look like?

Strong typing prevents mixing abstractions.
Nominal vs. Structural Type Systems

- **Nominal equivalence**: two types are equivalent if they have the same name in the same context
- **Structural equivalence**: two types are equivalent if they have the same structure

```c
struct {int x; int y;} a1;
struct {int x; int y;} b1;
a1 := b1; // S
```

```c
// N
a: array [1..100] of Integer;
b: array [1..100] of Integer;
a := b;  // N S
```
Nominal vs. Structural Type Systems

• Pascal’s type system is almost purely nominal

• C
  – struct, union – nominal equivalence
  – All other type constructors (e.g., arrays, function pointers) – structural

• C++
  – Just as C (class types: nominal equivalence)
  – Templates – structural

• Java
  – class – nominal equivalence
  – Arrays, templates – structural
C as a Hybrid Language

• `struct` (and unions) are nominal...

define struct {int i;} S1;
define struct {int i;} S2;

S1 s1;
S2 *p = &s1; //error!

• All other type constructors create structural types

define int A1[100];
define int A2[100];

A1 a1;
A2 *p = &a1; //OK
Intra-program Communication in C

• It is possible to declare the same type in different modules
  – Either by #including a common header file
  – Or by manually defining the type twice

• The compiler views these definitions as the same type
  – Even if they are different.
  – The reason: C’s type system is weakly typed.