Introduction

Object-Oriented Programming

236703

Winter 2015-6
Course Staff

Lecturer in charge:
Prof. Yossi Gil

Lecturer:
Eran Gilad

TAs: Nurit Moscovici (in charge), Helal Assi
Course Regulations

Read online
Abstract vs. Concrete
The lecture starts soon, where are the slides? 😞
Homework

Topics

- Squeak: 40%
- Java: 40%
- C++: 20%

Late submission penalty

- On time: 0
- 1 day: 5
- 2 days: 5
- 3 days: 5
- 4+ days: 0

Effect of cheating excuses

Final grade given exam + HW

Final grade vs. Exam grade vs. HW grade

Final grade: 90.00
   80.00
   70.00
   60.00
   50.00
   40.00
   30.00
   20.00
   10.00
   0.00

Exam grade: 100
   90
   80
   70
   60
   50
   40
   30
   20
   10
   0

HW grade: 100
   90
   80
   70
   60
   50
   40
   30
   20
   10
   0

Effect of cheating excuses: 0
Final Exam

Lectures and tutorials

Every covered material

Recycled question

Based on requirements and topics

Homework

From previous exams

Can’t answer

Lose points

Rely on partner

Final Grade

See your future here...
This Course

• ... is about object oriented *languages* and *features*
  • focus on abstraction *mechanisms*
  • not on object oriented *design*
• A comparative approach
  • comparing different languages
• Learn language features that make writing high quality code easier
  • This course will make you better programmers!
Motivation

• Software development is a practical endeavor
• Software development is difficult
  – How can we make it a little bit simpler?
• 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?
Object Oriented Programming

• OOP is a programming paradigm that tries to confront on-going coding difficulties

• The core solution: **Abstraction** through **objects**
  • Represent entities from both problem and program space
  • Match the way people think

• An **abstraction** of **objects** is called a **class**

• An **abstraction** of **classes** is called a **class hierarchy**
Abstraction in the Abstract

• Abstraction – the art of forgetting
• Allows the programmer to ignore details that are beyond the current perspective
  • Have the code express ideas, not operations
  • This is how humans think!

```java
while (HaveMoreElements())
    ProcessNextElement();
NotifyDone();
```
Abstraction in Practice

• Represent distinct entities as objects
  – So the code resembles real world

• Avoid code duplication and hide details by:
  – Identifying similarities between code fragments
  – Eliminating these similarities by making a single instance of the repeated code
    • Possibly parameterized
Abstraction Using Inheritance

- **Subclassing**: avoid code duplication by having related classes inherit a shared parent class
- **Subtyping**: operations that use the parent class can be run on child classes as well
  - *Is a* relation between child and parent

```cpp
class Employee {...}
class Manager : public Employee {...}

void GiveRaise(Employee* e) {
    e->incSalary(1000);
}
```
Abstractions Is Everywhere

• Cumbersome operations can be simplified by hiding the gory details
  • Less details → less to worry about!

```
// col can be any std
// collection –
// but not an array
for (auto iter = col.begin();
    iter != col.end();
    ++iter)
{
    cout << (*iter);
}

// col can be any
// std collection,
// an array,
// and other types
for (auto val : col)
{
    cout << val;
}
```
Encapsulation

Encapsulation is a central mechanism in OOP

- Separates interface from implementation
  - Interface more stable than implementation
- Reduces syntactic dependencies by hiding implementation details

```cpp
class Time {
    int h, m, s;   // int totalSecs;
public:
    int getSeconds();
    int getMinutes();
    int getHours();
    ...  
};
```
Encapsulation

- Also ensures single responsibility by controlling access
- Single responsibility $\rightarrow$ consistent behavior
- Language feature $\rightarrow$ code of higher quality

```cpp
void Time::setSeconds(int s) {
    if (s > 59)
        throw "too many secs";
}
```

Can encapsulation eliminate semantic dependencies? (e.g., assuming a throw upon bad argument)
The OOP Manifest

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.
OO Abstraction Mechanisms

• Types
  – Joined 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
OO 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
  – Relies on polymorphism

• Genericity
  – Similarity between operations on different types (or other parameters)
Static vs. Dynamic: Types

- **Static type**: the compile-time type of a variable, parameter or return type
- **Dynamic type**: the run-time type of the object being held or passed by the above

```
Base* base;
base = new Derived;
base->foo();
```

Understand those definitions – they are fundamental!
Static vs. Dynamic: Languages

• **Statically-typed language**: variables have static types
  o Compiler can provide safety and efficiency
• **Dynamically-typed language**: no static types
  o Less code, inherently generic

Base\* b = new Derived;
// fast and safe:
b->foo();
// compiler error:
b = new Unrelated;

v = “2”;
// no problem:
v = 2;
// run-time error:
v.Append(“unlimited”);
Static vs. Dynamic: Binding

- **Static binding**: method is selected at compile-time
  - Fast execution at run-time
    - Allows optimizations such as inlining
- **Dynamic binding**: method is selected at run-time
  - Flexible behavior

<table>
<thead>
<tr>
<th></th>
<th>Static binding</th>
<th>Dynamic binding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statically-typed</strong></td>
<td><code>Base b;</code> <code>b.Foo();</code></td>
<td><code>Base* b = GetB();</code> <code>b-&gt;virtualFoo();</code></td>
</tr>
<tr>
<td><strong>Dynamically-typed</strong></td>
<td><strong>N/A!</strong></td>
<td><code>foo: theArg</code> <code>theArg bar.</code></td>
</tr>
<tr>
<td><strong>language</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Great code is written using other paradigms

- Ever heard about an OS written in C?

But, object oriented languages have features that encourage good habits, e.g.:

- By default, all members in a C++ class are private
- By default, all methods in a Java class are virtual
- Smalltalk control structures are statements, which rely on polymorphic behavior