CS1412

Introduction into programming principles II

Max Berger

CS1412: Introduction into programming principles II

Max Berger

Table of Contents

List of Figures

List of Examples

Overview of CS1412 / Spring 2006

Syllabus

Course Objective

Short Synopsis

Object-oriented programming in C++ with emphasis on evaluation of alternative program design strategies. Class design, recursion, linked and dynamically allocated structures. This class will deepen the students understanding of designing and evaluation of larger programs.

Learning Outcomes

1. Students will analyze a problem and develop an object-oriented solution.
2. Students will use basic UML diagrams for their design.
3. Students will master C++ class syntax and semantics.
4. Students will confidently apply the Standard Template Library containers and algorithms.
5. Students will compare and evaluate alternative software designs for at least one project.

Methods of Assessment

1. Participation in class through excercises
2. Participation in the lab through lab excercises
3. Work on individual programming / design projects
4. C++ Programming homework excercises
5. Written and final exams

Announcements / Assignments

Announcements are posted on the class web page [http://max.berger.name/teaching/s06/]. Please check them frequently.

Students are required to be responsible for knowing about oral announcements or requirements not listed in this syllabus.

Class Hours

Section	Activity	Time	Location
92	Lecture	TR 9:30 - 10:50 am	CS 201
505	Lab	T 12:30 - 1:50 pm	PE 118

Attendance policy

Attendance in class will be checked every day. Attendance is not a certain percentage of the grade, but may be bring a grade down if to many classes are missed.It is the students own responsibility to acquire the material covered in classes she / he missed.

There will be excercises in class that can not be made up, unless you have a valid excuse. Proof (e.g. doctors note) may be required.

If, for any reason, you are absent on an exam / programming test date, you will have to notify the instructor **on the same day** at the latest. This can be done by a roommate, friend, parent, etc.

Civility in the classroom

Unacceptable behavior in the classroom includes: cellular phones or beepers, demanding special treatment, excessive tardiness, making offensive remarks, prolonged chattering, sleeping, "I paid for this" mentality, leaving the lecture early, dominating discussions, making unnecessary jokes, speaking out of turn, shuffling backpacks and notebooks, reading other material during class.

Lab policy

Lab attendance is required. Your grade will be affected by missed labs.

The Lab assignments have to be done during the lab period in the Lab. The students may leave early if the assignments are completed. If the assignments are not completed students may submit them after the lab, but only if they have worked on it during lab time!

Instructors

Lecturer: Max Berger [mailto:max@berger.name], CS 306 J , Office hours: Tuesdays 11 - 12

Teaching Assistant: TBA

The best way to reach me is to schedule an appointment via e-mail.

Text Books

Mandatory:

- Fundamentals of Object-Oriented Design in UML By Meilir Page-Jones. Addison Wesley, ISBN: 020169946X

Strongly suggested:

- UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3/E Martin Fowler Addison Wesley, ISBN: 0-321-19368-7

And either

- C++ Pocket Reference By Kyle Loudon 1st Edition May 2003 Series: Pocket References ISBN: 0-596-00496-6

or

- C++ in a Nutshell A Desktop Quick Reference By Ray Lischner 1st Edition May 2003 Series: In a Nutshell ISBN: 0-596-00298-X

Textbook policy

If you can't find the mandatory text book at the book store please get it online. There will be tests over material in the text book that might not be covered in class!

Software

Your C++ programs will need to able to run under one of the following environments:

- Eclipse CDT on Windows with MinGW GCC 3.2
- Eclipse CDT on Macintosh with Apple GCC 3.4 or 4.0 (installed in the lab)

Please read the installation instructions for eclipse [http://max.berger.name/teaching/cdt/].

You will need a program to draw UML diagrams. Two of them are:

- Poseidon for UML Community Edition [http://gentleware.com/]
- MagicDraw Community Edition [http://www.magicdraw.com/] (installed in the lab)

You will also need:

- A program that can create / unpack ZIP files, example: FilZip [http://www.filzip.com] (not needed in Mac OS X / WinXP)
- Adobe Acrobat Reader [http://www.adobe.com/] or another PDF reader.

All necessary software is installed in the lab. You may use the provided links / information to install these on your personal computer.

Grading

Item	Weight (MidTerm Grade)	Weight (Final)
Labs	20%	15%
Tests & Final	20%	30%
Programming Excercises	30%	15%
Projects	20%	30%
Participation	10%	10%
Total	**100%**	**100%**

Numeric to letter grades: 0-59: F, 60-69: D, 70-79: C, 80-89: B, 90-100: A. A higher grade may be given if students shows promise of success, A lower grade may be given if student shows lack of discipline (e.g. many missed classes).

Zero-Rule: Should any of the individual grade averages be zero (e.g. never turned in any projects) the student will receive an F regardless of the total average!

Expectation from students

Students should:

- Attend every class and be on time
- Bring the following material to class:
  - Letter sized paper
  - A pencil and an eraser (ideally) or a pen
  - A ruler

  These items may become unnecessary if the class can be held in the lab (still working on it)
- Students are not to use their own laptops in class

- First lab is Tue, January 17th
- Starting next class we will have self-assigned seats

Organization of this lecture

The lecture is organized in three parts:

- Part I, “Object Orientation” introduces object-orientation and defines the nine concepts of object orientation. It is completely language independet.
- Part II, “Object Orientation in C++” shows how to use these object oriented techniques in the C ++ language.
- Part III, “wxWidgets” introduces wxWidgets, an object-oriented cross-plattform GUI development framework written in C++.

Schedule

	Week	**Material**	**Misc**	**Projects**	**Prog. Excercises**
12-Jan	1	Introduction			
17-Jan	2	Encapsulation	First Lab		
19-Jan		Information Hiding, Attributes			P1
24-Jan	3	Associations			
26-Jan		State retention Object identity			
27-Jan		X	Drop with refund		P2
31-Jan	4	Messages			
2-Feb		Classes			P3
7-Feb	5	Inheritance			
9-Feb		Polymorphism			P4
14-Feb	6	Genericity			
16-Feb			Test over Part I		P5
21-Feb	7	Classes in C++			
23-Feb		Classes in C++		Project I	P6
28-Feb	8	Classes in C++			
2-Mar		Classes in C++			P7
7-Mar	9	Constructors			
8-Mar		X	Midsemester Grades		
9-Mar		Destructors			P8
14-Mar		X	Spring Break		
16-Mar		X	Spring Break		
21-Mar	10	STL			

	Week	**Material**	**Misc**	**Projects**	**Prog. Excercises**
22-Mar		X	Last day to drop / Pass Fail		
23-Mar		STL			
28-Mar	11	STL			
30-Mar		Templates	no Test here		
4-Apr	12	STL			
6-Apr		STL		Project II	
11-Apr	13		Test here		
13-Apr		wxWidgets			
18-Apr	14	wxWidgets			
20-Apr		wxWidgets			
25-Apr	15	wxWidgets			
27-Apr		wxWidgets			
2-May	16	Review for Final		Project III	
15-May		X	Final grades		

Part I. Object Orientation

Table of Contents

Chapter 1. Introduction to object orientation

What is object orientation?

Definitions from a dictionary:

Object	A thing presented to or capable of being presented to the senses.

In other words: Just about anything!

oriented	directed toward.

therefore we can deduct:

Object oriented	directed toward just about anything you can think of.

Properties of object orientation

Since this is very ambiguos, we define object orientation as 9 properties:

- Chapter 2, *Encapsulation*
- Chapter 3, *Information / implementation hiding*
- Chapter 4, *State retention*
- Chapter 5, *Object identity*
- Chapter 6, *Messages*
- Chapter 7, *Classes*
- Chapter 8, *Inheritance*
- Chapter 9, *Polymorphism*
- Chapter 10, *Genericity*

Terminology

To understand what whe are talking about we need to define some terms:

Important

attribute	represents information about an object (something you can describe, has / is relationship, e.g. this computer has and intel chip, the cpu speed is 4000 mhz)
operation	an action, with a possible return value (somehting you can do, e.g. i can turn on this computer)
class	the type of the object (is-a relationship, e.g. this is a computer, this is a table)
object	a specific instance of a class (with a distinguishing word, e.g. this computer, my laptop, my chair, Ryan's chair)

Conventions:

- attributes, operations, and objects start with lowercase letters (e.g. size, type, myComputer)

- classes start with uppercase letters (e.g. Computer, Chair, Keyboard)
- names containing multiple words are concatenated, each word (starting with the second) is capitalized (e.g. chipType, turnOn, ComputerScreen, myMpegDecoder, myTv)
- operations have a set of parenthesis after the name (e.g. turnOn(), plugIn())

Please note: In this class these conventions are enforced

What do the following things represent?

Example 1.1. Define these items as class, operation or attribute

size
chair
numberOfLegs
color
sitOn
milesDriven
car
gasUsed
turnLeft
direction
name: size
type: attribute
name: chair
type: class
name: numberOfLegs
type: attribute
name: color
type: attribute (could also be a class, as in green, red, ...)
name: sitOn
type: operation
name: milesDriven
type: attribute
name: car
type: class
name: gasUsed
type: attribute
name: turnLeft
type: operation
name: direction
type: attribute (could also be a class)

UML notation

UML stands for Unified Modelling Language.

UML consists of different types of diagrams.

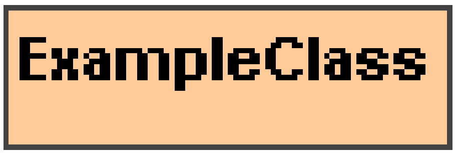
We will be looking into class diagrams.

Two types of information:

- Information about the class
- Relations between classes (later)

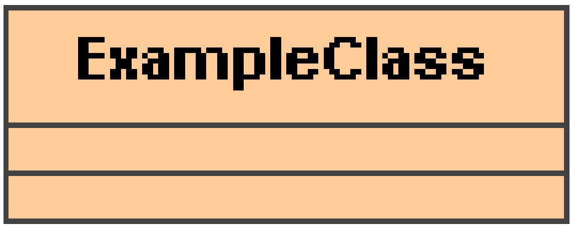
There are two different notation for classes. The shortened notation and the full notation.

Figure 1.1. UML short class notation



A class is denoted by a rectangle. The name of the class is written in bold font. The color is just for illustration.

Figure 1.2. UML long class notation

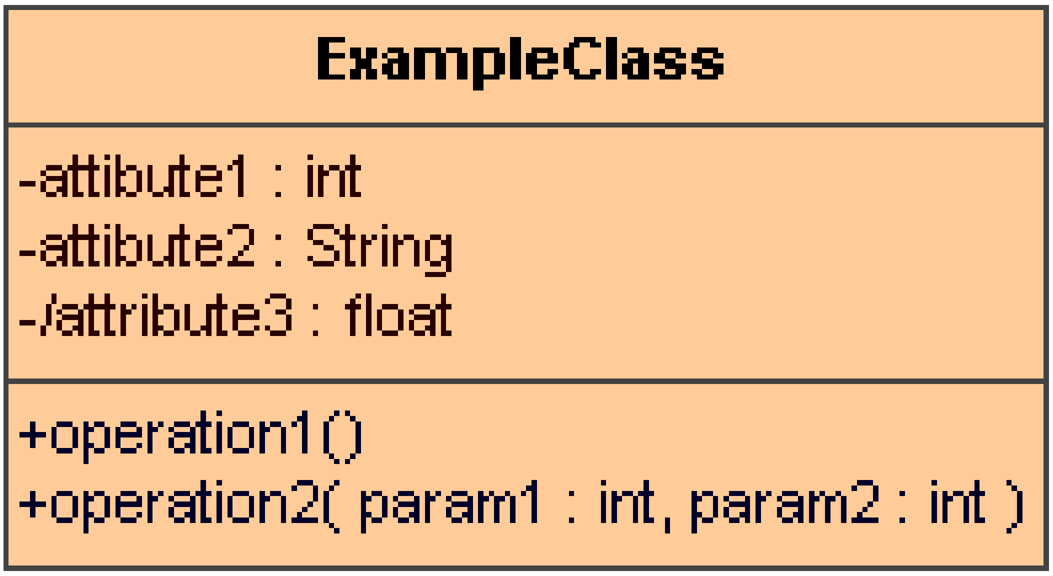


In the long notation the UML class gets divided vertically into 3 compartments:

- the top compartment contains the class name, still in bold
- the second compartment contains the attributes
- the third compartment contains the operations

Compartmets (except for the class name) can be shown as empty, even if there are attributes / operations present, however, all compartments need to be shown.

Figure 1.3. UML long class notation with some attributes and operations



Attributes are shown as follows:

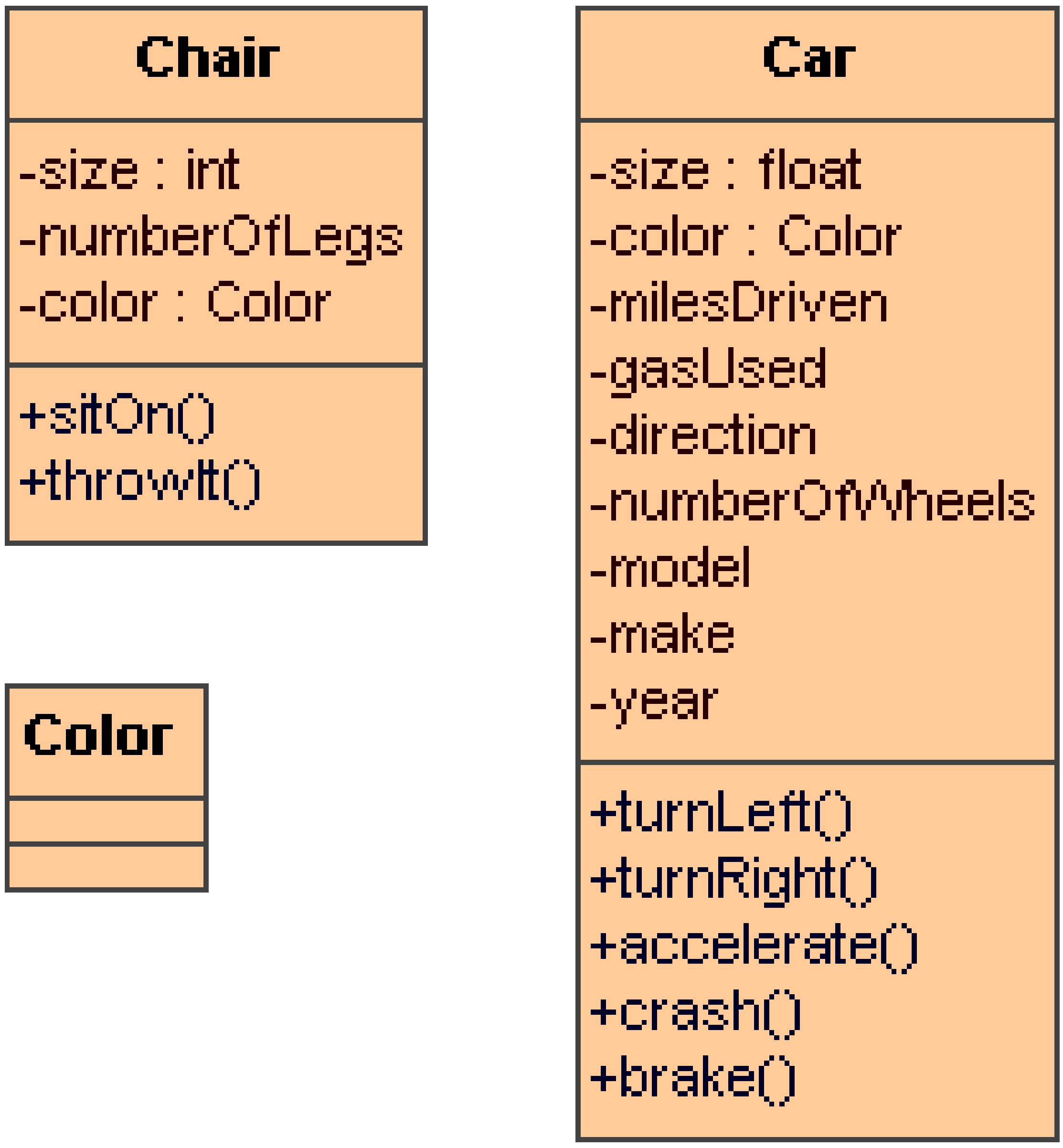
- First the visibility modifier (explanation later) (optional)
- then optional modifiers (explanation later)
- then the name of the attribute (remember: lower case!)
- a colon (:) and the type of the attribute (optional)

Operations are shown as follows:

- First the visibility modifier (explanation later)
- then the operation's name
- in parenthesis the list of parameters
- a colon (:) and the type of the return value (optional)

Example: Draw class diagrams for the classes, operations, and attributes from the list earlier.

Example 1.2. Example Class Diagrams



Book: Chapter 3.1 - 3.3

Chapter 2. Encapsulation

What is encapsulation?

Important

Encapsulation	is the grouping of related ideas into one unit, which can thereafter be referred to by a single name.

Came from the early 40's: The same pattern of instructions appear multiple times, why not give them a single name and re-use them?

Subroutine was born. Saves computer memory. Then people realized: Saves human memory too!

Encapsulation in OO has a similar purpose, but is more sophisticated.

Important

Object-Oriented encapsulation	is the packaging of operations and attributes representing a state into an object type so that it is accessible or modifiable only via the interface provided by the encapsulation

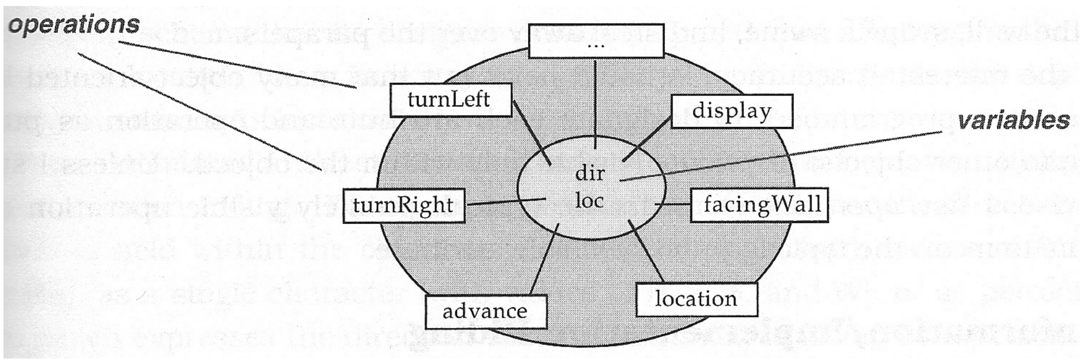
Example: Hominoid class, has operations such as:

turnLeft	with turns hominoid to the left by 90 degrees
advance	moves hominoid one step forward

Each operation is a procedure or function that is normally visible to other objects, which means it can be called upon by other objects.

Attributes represent information that an object remembers. Attributes are only accessed via object's operations. In other words, no other object can acces an attribute directly by grabbing the underlaying variables. Another object has to go though the object's operations.

Figure 2.1. Operations and attributes of Hominoid



Only the operations may acces attibutes, they form a protective ring around the core variables implemented in the object.

Example: location operation is used to access the loc variable.

Important

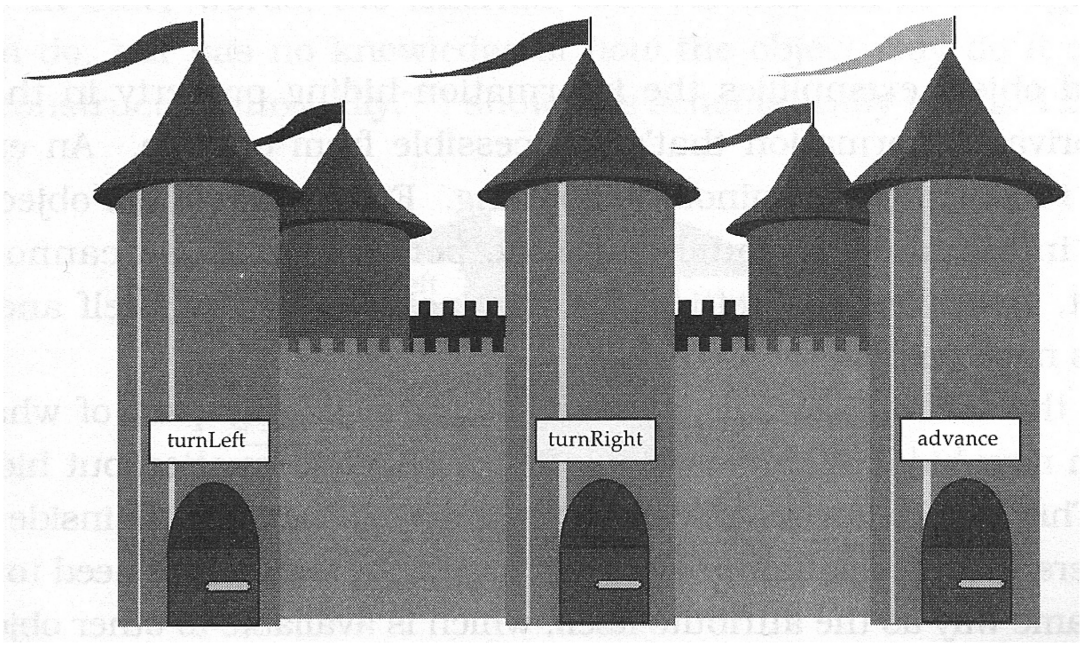
variable(s)	One or more variables implement one or more attributes.

Examples:

- a cpuSpeed attribute can be implemented with one variable cpuSpeed
- a location attribute can be implemented with two variables: locationX, locationY
- the attributes location and facingWall are both implemented with one location variable

an object structure resembles a medieval city, with a protective wall. It can be accessed only through well-defined and well-guarded gates.

Figure 2.2. The Hominoid's City



Staunch, hones yeopersons would enter the city via the gates. They would buy their pigs in the marketplace and then leave through a gate. Only the most villainous villeins and scrofulous scalawags would scale the wall, swipe a swine, and steal away over the parapets.

Book: Chapter 1.1

Chapter 3. Information / implementation hiding

Two views: public view (from the outside), private view (inside). Private view should be suppressed from others.

information hiding: information within cannot be seen from the outside. Implementation hiding: implementation details cannot be perceived from the outside.

Important

Information / Implementation hiding	is the use of encapsulation to restrict from external visibility certain information or implementation decisions that are internal to the encapsulation structure.

Back to the example from the book: hominoid: hides some information, example: direction. can be changed form the outside, but cannot be read. (however, can be displayed, and we may be able to check the position) -> information hiding.

However, encapsulation goes beyons that: can reveal information but hide implementation: Variable inside does not need to be implemented the same way as the attribute.

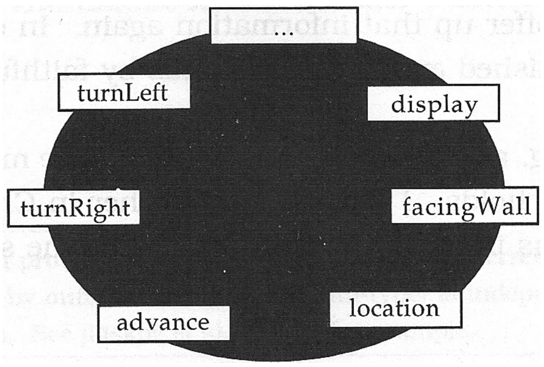
Example: hominoid reveals locatin via location attribute, but how is it stored internally? could be (xCoord, yCoord) or (yCoord, xCoord), polar coordinates or some scheme the designer dreamed up at 1 am. However: location attribute exports the location in a specific form. We don't care how the information is stored. This is implementation hiding.

The direction is an example implementation and information hiding. Direction could be degrees (0-359), characters (N, E, S, W), percentDirection (0.00.. 99.999), or anything else.

In a redesign, if we decide to release direction, we have to chose how to release it, however, this is independent from the actual implementation.

implementation / information hiding provides a black-box view. External object have view of what the object can do, but no knowledge how.

Figure 3.1. Black Box View



Benefits:

Localizes design decisions: Private design decision have little or no impact on the rest of the system. Local decision can be made and changed without impact on the rest of the system. This limits "ripple of change" effect.

Decouples content of information from its form of representation. No information internal is tied to external format. this prevents external users / programmers from meddling. if also prevents unstable connections.

Visibility Modifiers

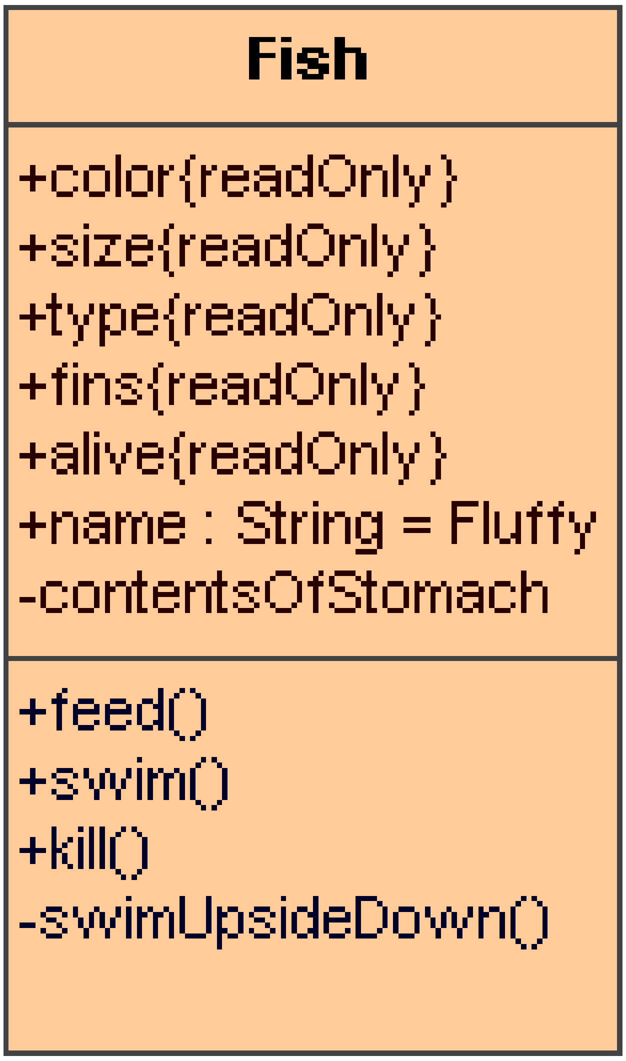
Visibility modifiers describe who can see and / or modify / execute an attribute or operation.

Notation in UML: shown with the visibility modifier:

- private (-)
- public (+)

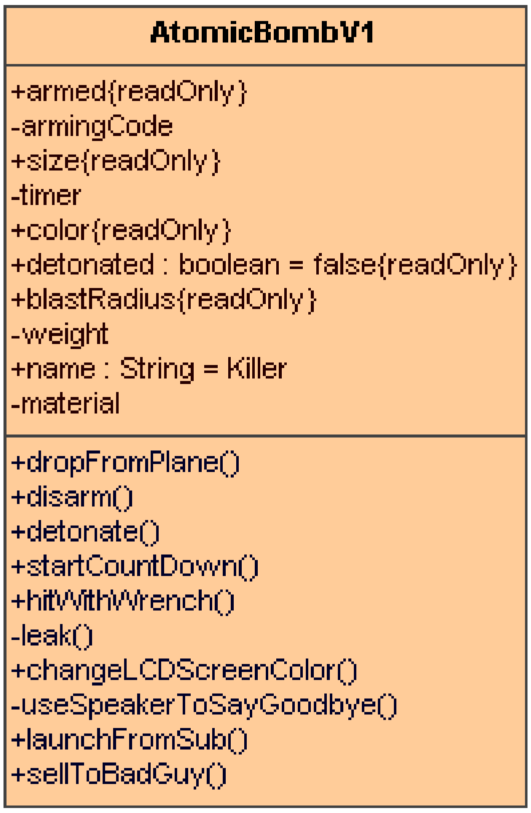
Special attribute property {readOnly} to indicate read-only attributes.

Example 3.1. Public, private, and read-only attributes



Practice: implement atomic bomb.

Example 3.2. Combination of students results



However, actual implementation is different!

Getters / Setters

In actual implementation, all attributes should be private. Attributes are read and set with getter and setter methods.

Different notations: ObjC and Java Notation.

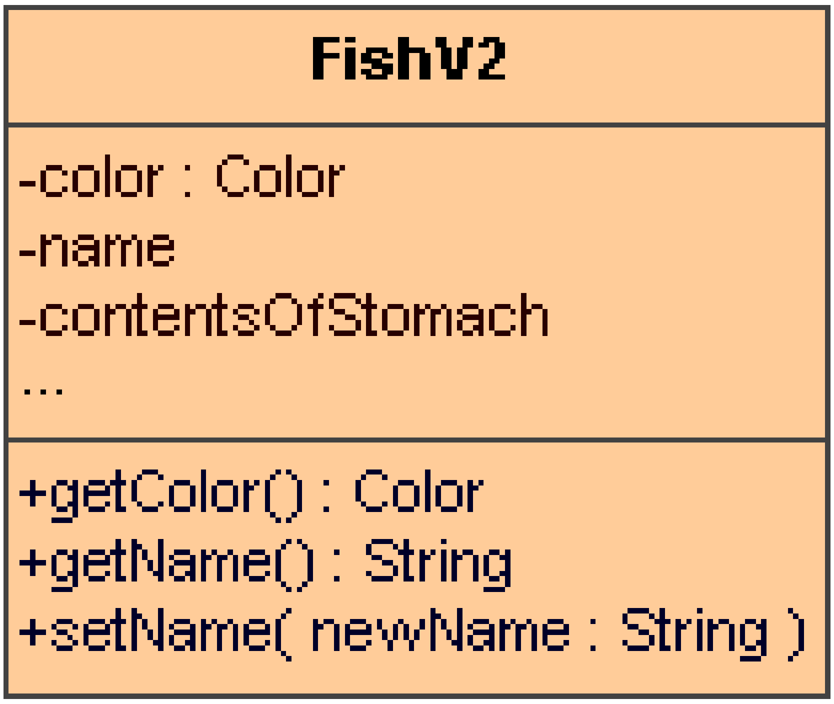
Setter: Both: add "set", capitalize attribute, e.g. color -> setColor()

Getter: Java (to use in this class), add "get", capitalize attribute, e.g. color -> getColor().

ObjC (just fyi): use the same name as the attribute, OR use "is" prefix in case of booleans, e.g. : color -> color(), bw -> isBw()

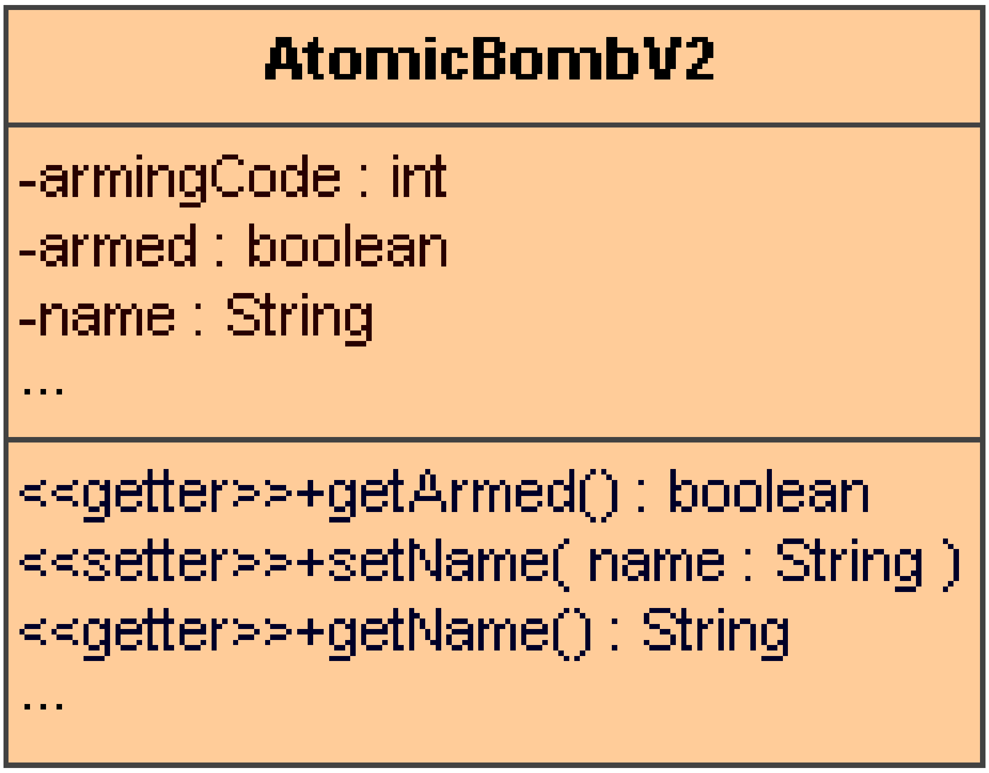
Read-Only attributes: have no "setter"

Figure 3.2. Example of the same attributes, with getters and setters



Practice: do the same thing (bomb), now with getters and setters.

Figure 3.3. Excerpt from the atomic bomb, now with getters and setters


AtomicBombV2
-armingCode : int
-armed : boolean
-name : String
...
<<getter>>+getArmed() : boolean
<<setter>>+setName(name : String)
<<getter>>+getName() : String
...

No matter which notation you use, in the actual implementation you should always user getters / setters to do the implementation hiding!

Stereotypes

New UML notation: Stereotype (the thing with the in the << >>)

Important

Stereotype	A UML term for "a new kind of model element defined within the model based on an existing kind of model element". Stereotypes may extend the semantics but not the structure of pre-existing meta-model classes.

Steteotypes

- Are always written in << and >>
- are completely optional
- can be applied to anything (classes, attributes, operations, etc.)
- add additional classification information

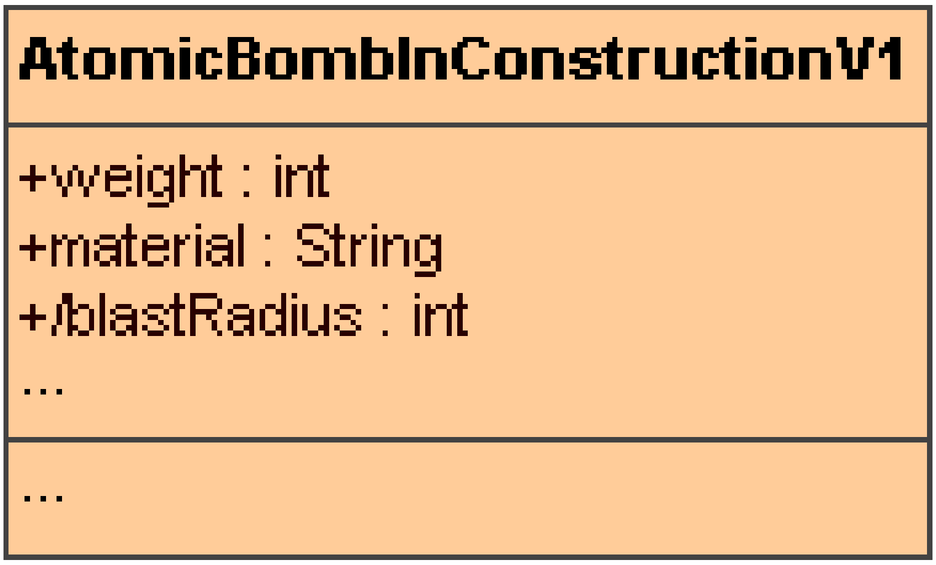
In the example given above the use of stereotypes makes absolutely no sense, since everyone should be able to figure out that these methods are getter and setter methods. I just wanted to show you stereotypes.

Derived Properties

A derived property is a property that is not stored directly, but is derived from other properties. Most of the times it is calculated.

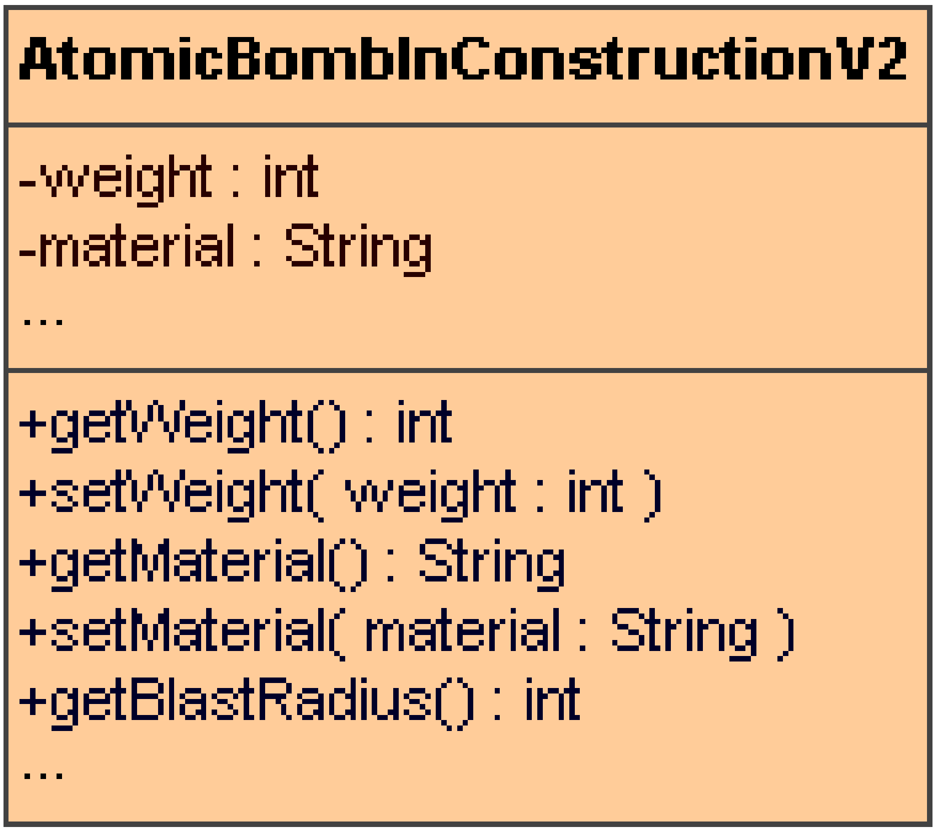
UML Notation: noted with a "slash" and public, or just as a getter method. These are usually readonly, so {readOnly} can be omitted.

Example 3.3. Atomic bomb with a derived attribute



For the actual implementation derived attributes do not show up at all but provide a getter method:

Example 3.4. Actual implementation of derived attribute



Book: Chapter 1.2, UML Destilled: Chapter 3 (attributes) (36-37, 68-69)

Associations

Some properties may be modeled as their own class. This makes sense:

- When the class can be reused (e.g. both cars and bikes have wheels)
- When the class has a lot of things that should be logically grouped together

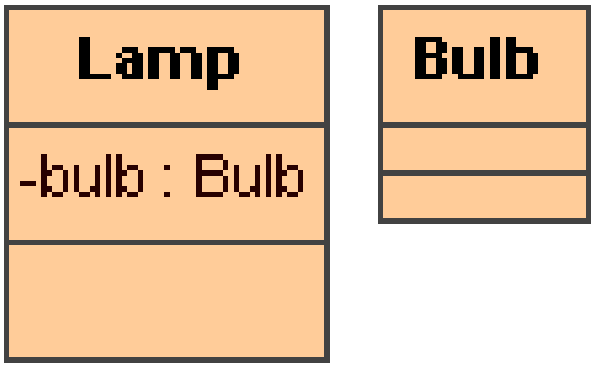
Beware:

- In most cases, the containing class will have a reference to the contained class (Car has an attribute of type Wheel, Lamp has an attribute of type Bulb, etc)
- If the classes are equal, the attribute may be in either class (Person has an attribute leftNeighbor)
- The attribute may also be in both classes. In this case special care has to be taken whn setting the attribute (when actually coding it). (Person has leftNeighbor and rightNeighbor, Car has attribute wheel : Wheel, Wheel has attribute onCar: Car)

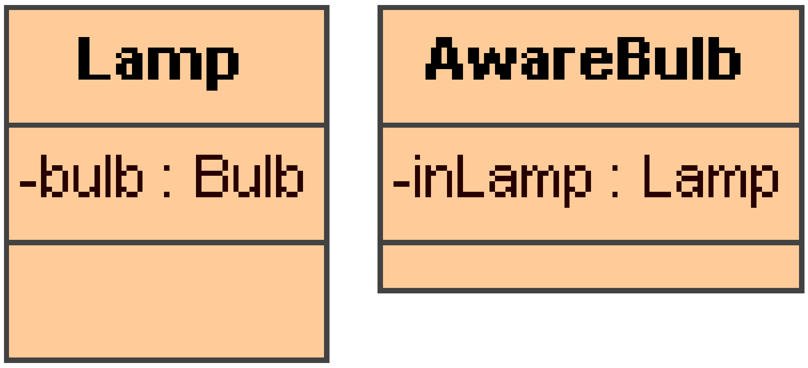
Important

Navigable	An association from one class to another is said to be navigable. One way navigation goes from one class to another, while two-way navigation goes in both directions.

Example 3.5. Lamp with a bulb one-way navigable



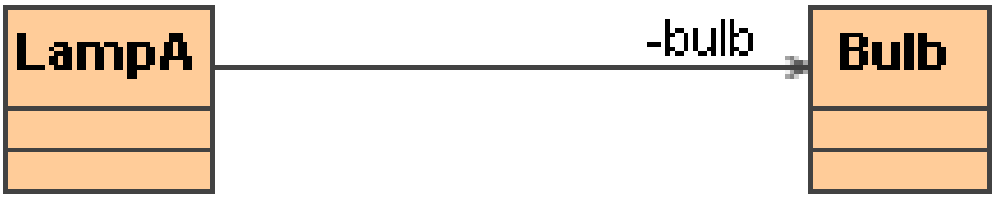
Example 3.6. Lamp with bulb, two-way navigable



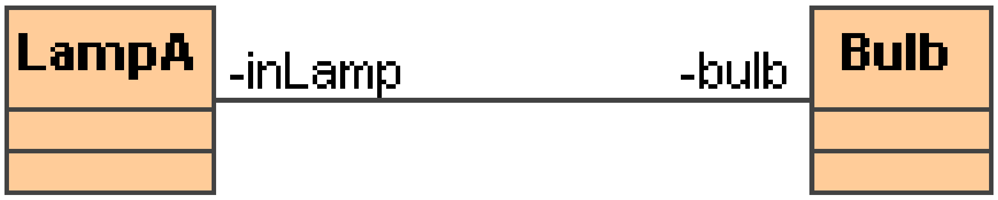
In UML, associations are usually shown with a line between the classes.

- If the association is one-way navigable, the will be an open arrow at the class that the association is navigable to.
- If the association is two-way navigable there are no arrows
- Instead of showing an attribute in the attribute compartment, it may also be shown at the end of the association. Type will be omitted, since it is always the class its pointing to. In a two-way navigable association there will be names on both ends.

Example 3.7. Lamp with bulb, one-way navigable, shown as association

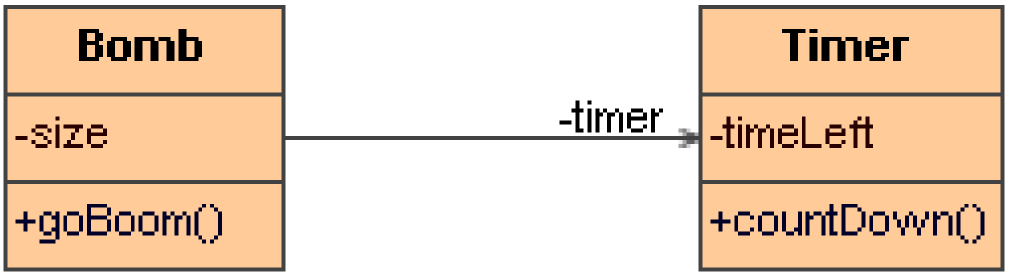


Example 3.8. Lamp with bulb, two-way navigable, shown as association



Example 3.9. Atom bomb and timer as two separate classes

Practice: Draw a diagram of an atom bomb and its timer, connect them with an association. Include at least one attribute and operation in each class. How many attributes does your bomb have?



Bomb has two attributes: size and timer.

Multiplicity

So far we hae only talked about one-to-one relationships (Each lamp has one bulb, each bulb is in one lamp), but there may of course be different multiplicities.

Important

Multiplicity	for an attribute or association defines "how many" of this attribute or association there are. Multiplicity is given with as mininum number, dots (..), and maximum number.

Multiplicity only takes whole numbers, no negatives!

Examples: A car usually has 4..5 wheels, a class has 0..75 students.

Shortcuts:

- if both numbers are the same, the .. and the second number can be omitted, e.g. a twin-lamp has 2 bulbs
- if the upper limit is unlimited a * is used: A city has 1..* houses, a novel has 100..* words
- 0..* can be written as *

Common multiplicities:

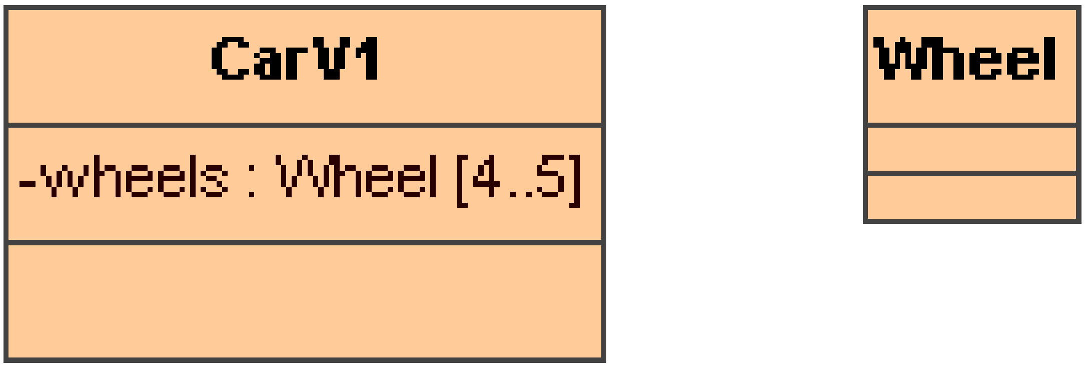
- 0: A class has never any of this attribute (not really used)
- 0..1: A class has none or one of the specified (e.g. a lamp has a bulb or has none, a person has a job or has none)

- *: Any arbitray number of attributes: A file has any number of bytes
- 1: Exactly one: The hominoid has exactly one position
- 1..*: At least one, but any number: A poem has at least one word, but can be infinitely long.

If multiplicity is ommited, it usually means 1 or 0..1.

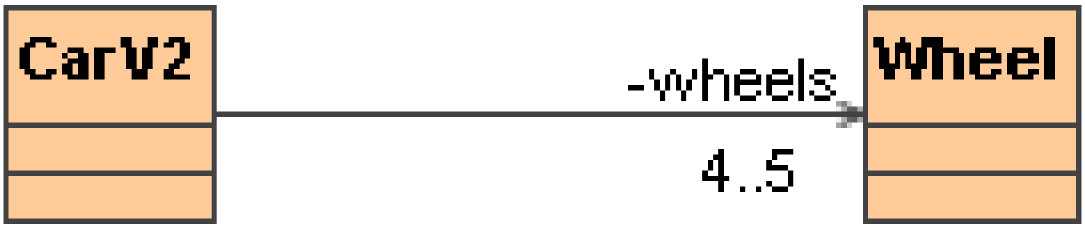
On attributes, multiplicity is shown in square brackets after the attribute, e.g. [2], [*], [1..4]

Example 3.10. Multiplicity on attributes



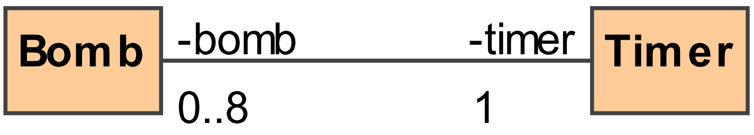
On associations, the multiplicy is shown without square brackets on the class that is multiplied:

Example 3.11. Multiplicity on associations



Example 3.12. A timer for multiple bombs

Practice: assume a modern timer that can detonate up to 8 bombs at a time. Show the classes for Timer and Bomb and their association. You may use the shortened notation for the classes.



Chapter 4. State retention

A class is a general description: Every Hominoid has a position.

An object is an "instance" of a class and has specific values for the attributes: This Hominoid is at position 2x5

Objects keep their state (in attributes).

Traditional methods need to have parameters. They don't know about their previous existence.

But objects are aware of the past. they retain information inside -> objects don't die, they are ready to execute again

Example from Monopoly: A property knows its value. So when buying or selling a property, there is no need for the parameter "price".

Example from Hominoid. A Hominoid is at position 2x5, facing up. After the operation "forward" it is not at 2x4, facing up. Another invocation of forward will move it to 2x3 (so the results are different).

Techspeak: an object retains state. state is the set of values that an object holds. example: hominoid retains knwoledgde about the square and the location.

State can be compared to global variables, but here they are are encapsulated with their functions.

Encapsulation, information hiding and state are the core of object orientation. However, these concepts are not new, they used to be called ADT (abstract data type).

Practice: Assume a Hominoid with the following state:

- Direction : (N,E,S,W) = N
- PosX : int = 5
- PosY : int = 5

Where the location (1,1) is the top left (most north, most west) and location (20,20) is the bottom right (most south, most east), X goes east - west and Y goes north - south.

The operation advance() moves the hominoid one step forward, the operations turnLeft() and turnRight() turn the hominoid by 90 degress.

Assume the following function calls:

1. advance()
2. turnRight()
3. advance()
4. advance()
5. turnRight()
6. advance()
7. turnLeft()
8. turnLeft()
9. advance()

What is the Hominoids state after this code?

Chapter 5. Object identity

Each object has its own identity

Important

Object Identity	is the property by which each object (regardless of its class or current state) can be identified and treated as a distinct software entity.

There is "something unique" about any object that distinguishes it from other objects. This unique thing is the object-handle (sometimes calles object reference).

Example of an object creation (books notation):

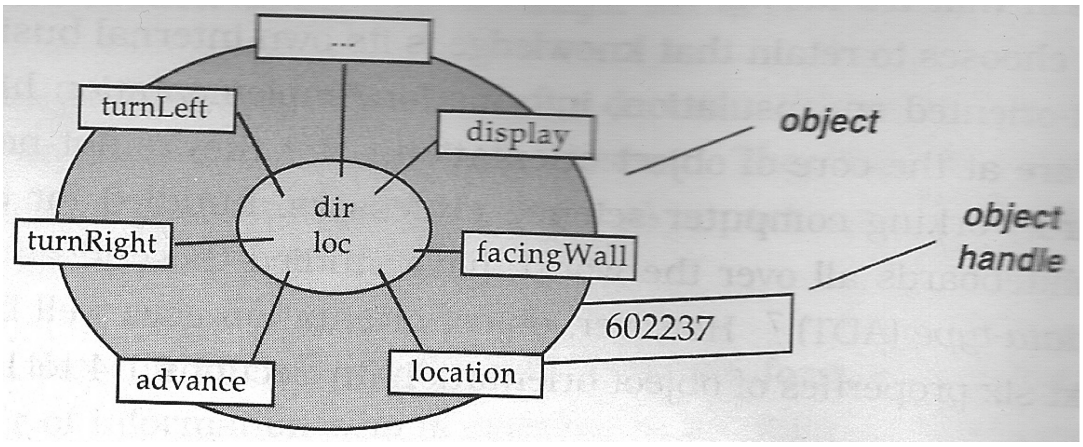
```
var hom1: Hominoid := Hominoid.New;
```

Example of an object creation (C++ notation):

```
Hominoid *hom1 = new Hominoid();
```

The right hand side creates a new object of class Hominoid, and returns an object handle. In the case of this example the handle is 602237.

Figure 5.1. An object with its handle



Two rules apply for handles:

1. The same handle stays with the object for all time!
2. No two objects can have the same handle. Every new object gets its own handle that is different. Objects may look identical (same state, same class) but if they have different handles they are different objects! If they have the same handle they are the same.

   In most OO languages (Java for example) handles are different from all existing or having existed objects. In C++ handles can be re-used if the original object is dead.

   C++ uses the memory address as object handle.

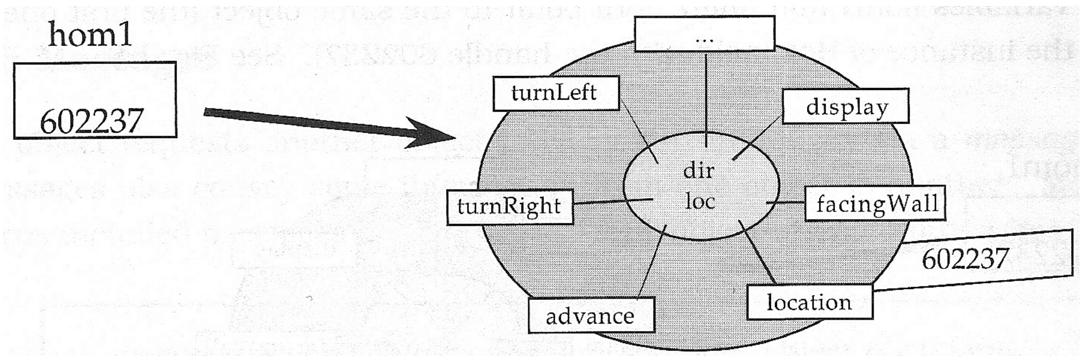
The left part of the line (var hom1:Hominoid / Homoid *hom1) is a declaration that gives a programmer-meaningful name (hom1) to a space that can hold an object handle.

The assignment (:= / =) causes hom1 to hold the handle to the object. Read: "now points to" or "now refers to"

In the case above: hom1 now referes to a new object of class Hominoid.

Usually you don't see the handle, but just use the variables.

Figure 5.2. hom1 pointing at the object with handle 602237



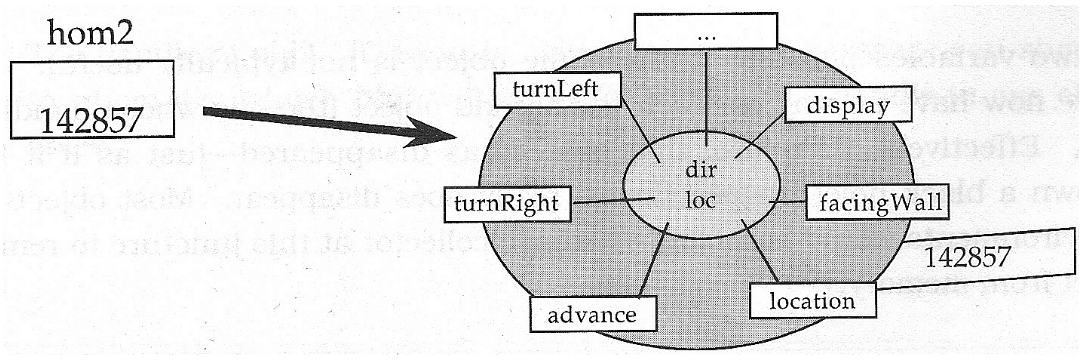
Some OO languages (C++ for example) use the location in memory as a handle (remember pointers?)

Example2: Lets create another object:

```
var hom2: Hominoid := Hominoid.New; // books notation
Hominoid *hom2 = new Hominoid(); // C++ notation
```

This will create a second object (of class Hominoid) with a new handle (e.g. 142857) and stores that handle in hom2.

Figure 5.3. hom2 pointing to the object with handle 142857

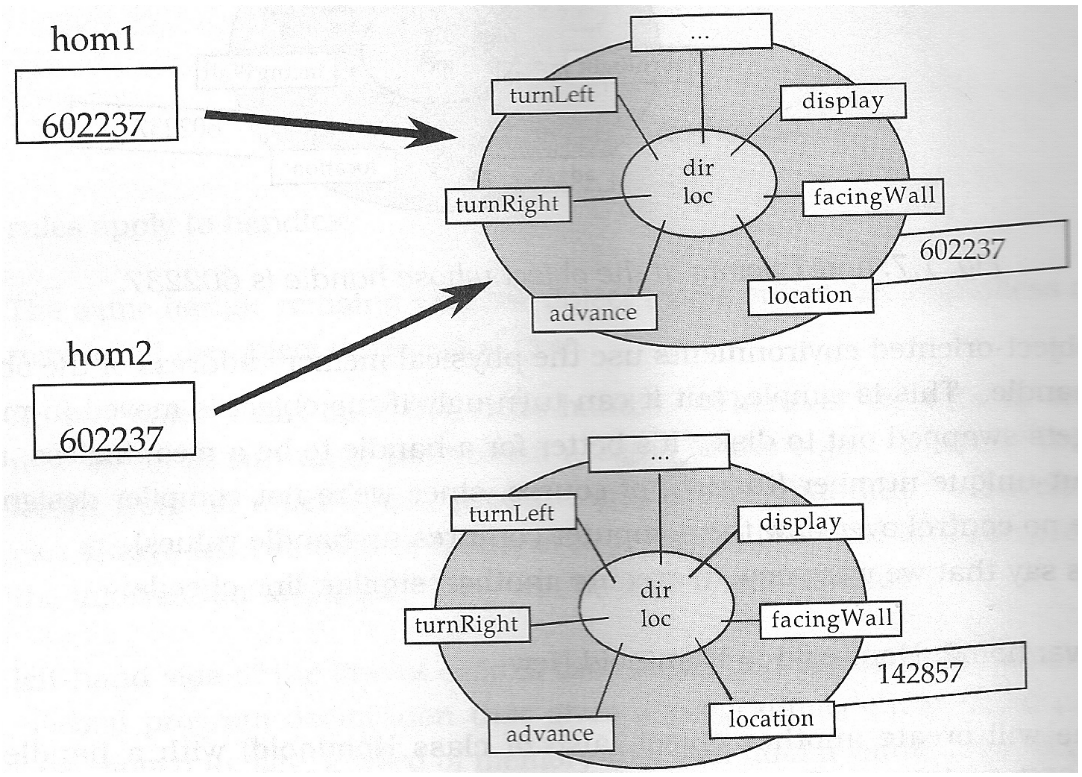


Now what happens after the assignment?

```
hom2 := hom1; // book
hom2 = hom1; // C++
```

Now both variables hom2 and hom1 hold the same object handle, they both point to the same object!!!

Figure 5.4. hom1 and hom2 pointing to the same object, object 142857 is unreachable



Having two variables pointing at the same object makes very seldom sense.

Even worse: The object at 142857 is now unreachable! We have no variable that holds its handle! That object has now disappeared. In C++: Memory Leak. In Java: Garbage collector.

Practice: which object handles are stored in these variables after this code has executed? Which object(s) is/are "lost" ?

```
Hominoid *hom1 = new Hominoid();  // assume handle 111
Hominoid *hom2 = new Hominoid();  // assume handle 222
Hominoid *hom3 = new Hominoid();  // assume handle 333
hom3 = hom2;
hom2 = hom1;
```

Solution:

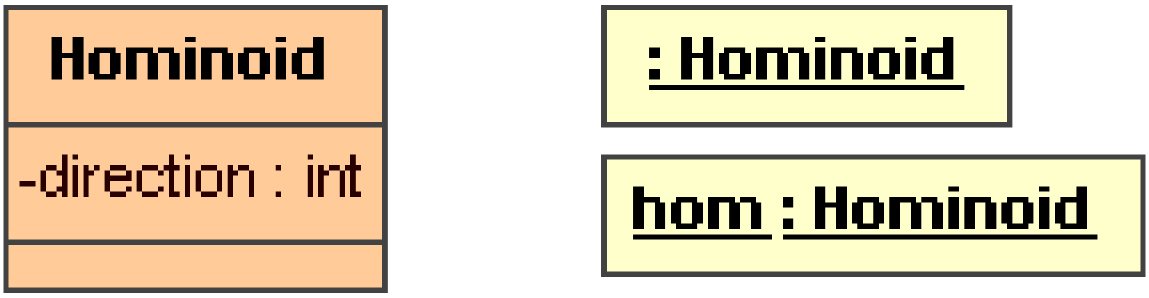
- hom3 point to handle 222
- hom2 points to handle 111
- hom1 points to handle 111
- the object at handle 333 is lost

Notation in UML: Similar to class, but different!

- Name is underlined to signify instance.
- The format for the name is object handle (or variable name), then a colon, then the class, e.g. hom1 : Hominoid
- object handle / variable name may be ommited, e.g. : Hominoid (notice the colon is still there). In this case we talk about an "anonymous instance"

- Short notation: Just the name and its class in a box
- Long notation: shows one additional compartment that lists the attribtues with name, an equal sign and the value, e.g. direction = 180
- Instances are shown in an object diagram

Example 5.1. Homioid class shown with 3 instances



This Hominoid has 3 instances:

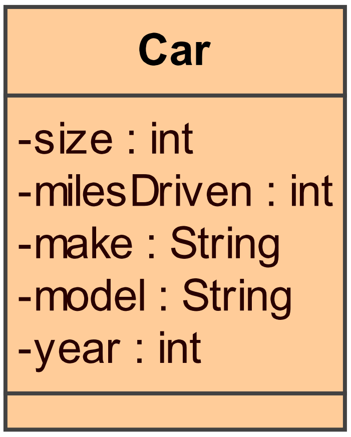
- Two of then are anonymous, one has a name (hom)
- All of them have a value for the attribute direction, but it is only shown on one of them

Notes for MagicDraw:

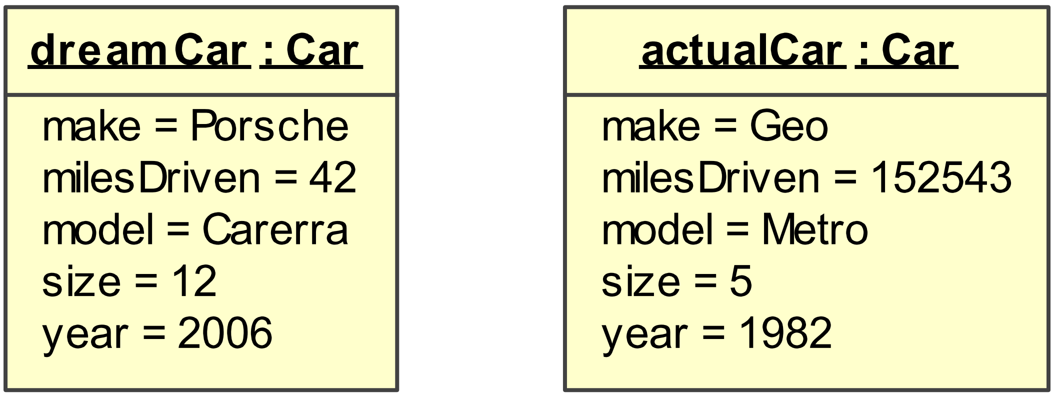
- MagicDraw calls the instance attributes "slots"
- Slots are not shown by default (must be turned on)
- No attribute has a value by default (must go into menu and "create value")
- MagicDraw shows classes in orange, instances in yellow (this is completely optional, the underline is what makes the real difference!)

Example 5.2. Instances for Cars

Practice: Draw two instances for the class Car, as given in this diagram:

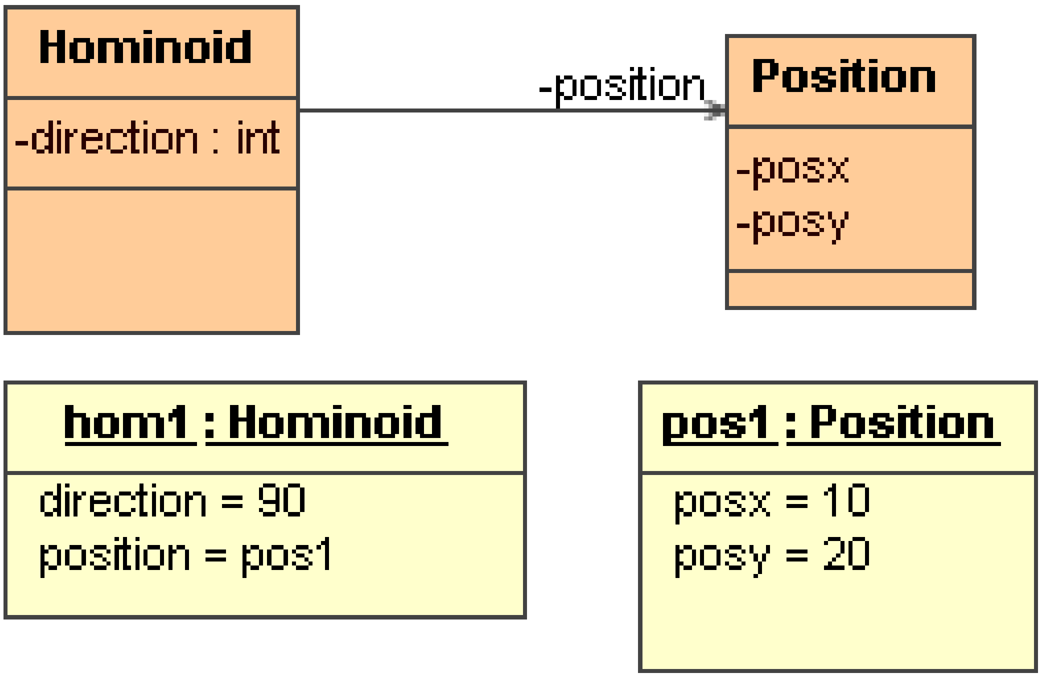


A possible solution:



Since every object (must) have an object handle, relations can be shown like other attributes:

Example 5.3. Class diagram and object diagram for Hominoid with Position



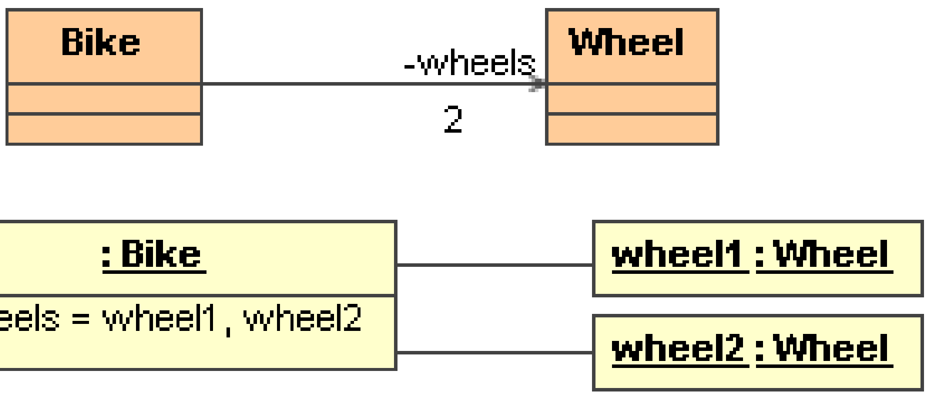
In this case we show the relation Hominoid - Position as attribute (slot), where the value is the object handle of position.

Warning: We use human-readable handles in UML diagrams, but in reality these are the internal object handles (strange numbers that make no sense). The handles shown on UML diagrams do not necessarily need to match the names used for the variables in actual programming!

Assiciations can still be shown, but they have no labels. Rather the information should be shown in the slots

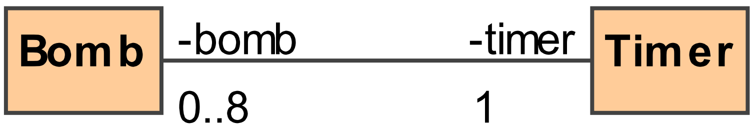
When an attribute has multiple values (e.g. a car has 4 wheels), then the elements are listed separated by a comma.

Example 5.4. A bike instance with two wheel instances

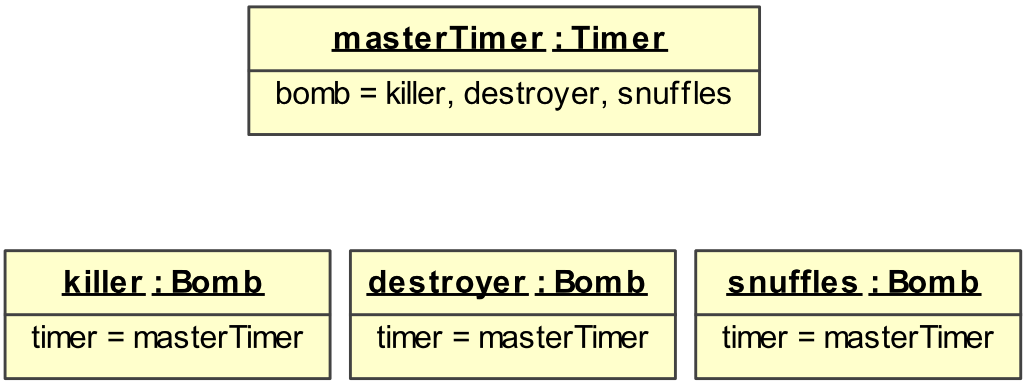


Example 5.5. Instances for the bomb and the modern timer

Now draw an object diagram for the modern bomb timer. Use 3 bombs and 1 timer. Add all slots needed (on Timer and Bomb)



A solution:



Chapter 6. Messages

Objects communicate via messages. Messages can ask to carry out an activity. They may also carry information from one object to another object.

Important

message	a message is the vehicle by which a sender obejct obj1 conveys to a target object obj2 a demand for object obj2 to apply one of its methods

obj1 and obj2 may be the same objects, so objects may send messages to themselves.

We will see:

- the anotomy of a message
- the characteristics of message arguments
- the role of an object sending the message
- the role of an object receiving the message
- the 3 types of messages

Message structure

In order for obj1 to send a message to obj2, obj1 must know 3 things:

1. The handle of obj2. (You need to know who to talk to). obj1 will usually store obj2's handle in one of its variables (remember the section called “Associations”)
2. The name of the operation of obj2 that obj1 wishes to execute
3. Any additional information (arguments) that obj2 requires to execute the operation

Important

sender	the object sending the message (obj1 in this example) is called the sender.
target	the object receiving the message (obj2 in this example) is called the target

Example from the Hominoid:

```
hom1.turnRight;       // Books notation
hom1->turnRight();    // C++ notation
```

hom1 points to (contains the handle of) the target object of the message. turnRight is the name of the operation. In this case, turnRight has no arguments.

The sender in this case the the object from which this line is called.

Compare to procedural programming:

```
call turnRight(hom1);   // Books notation
turnRight(hom1);        // C++ notation
```

Traditional: the operation is important, the data is just an argument. What do I want to do? Oh, and by the way, this is what you need to do it with.

OO: the target is important, the message is second. Hey, you object! Do this!

More like real life: If you shout out "someone should turn on the lights", nothing will happen. They you call on a certain person, and then they may wake up and do it... However, if you point at someone and say "you! please turn on the light" it magically happens.

Reasoning: Different objects may provide the same operation.

Example operation: sitOnIt() may be provided by the classes Table, Chair, and Floor. Both are different objects, and the actual method is very likely differnt.

Example: operation: takeMeToSchool() may be provided by the classes Car, Truck, SchoolBus, Bike, Feet, Helicopter.

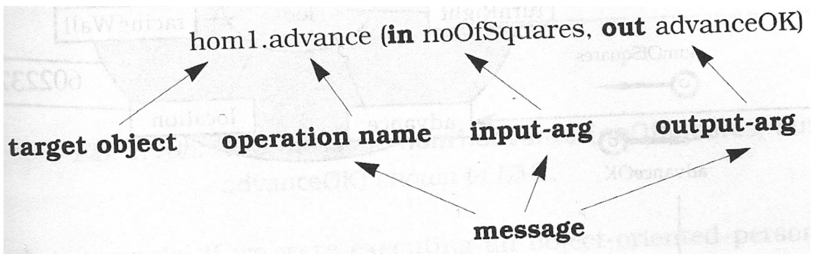
Message arguments

Like functions in traditional programming, messages pass arguments back and forth.

Example of an advance function that has two arguments:

- noOfSquares that tells the function how many numbers of squares to advances (input)
- advanceOk tells the sender if it worked (or if there was a wall) (output)

Figure 6.1. Dissection of a message and its components



The same in C++ notation

```
hom1->advance(noOfSquares,advanceOk);  // C++ notation with pass-by-reference
advanceOk = hom1->advance(noOfSquare); // C++ notation with return value
```

Rember: One output value may be done as a return value, multiple output values have to be done with pass-by-reference.

There are 3 types of arguments:

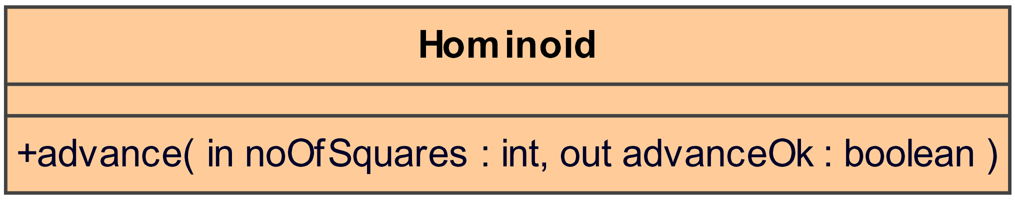
in	input arguments (pass-by-value)
out	output arguments (return value in C++ if only one, otherwise pass-by-reference)
inout	combination of in and out. (pass-by-reference)

If the direction is not shown, and is not clear from the context, it usually means "in"

In UML, in and out can be shown before the parameter name

Note for MagicDraw: In Magicdraw set "show operation parameters direction kind" to true.

Figure 6.2. Hominoid with one operation with in and out parameters



In pure OO languages (smalltalk), all arguments are object handles.

In mixed programming languages (C++, Java) there are two types of arguments

- object handles (in C++: pointers, e.g. Car*)
- data (int, boolean, float, ...)

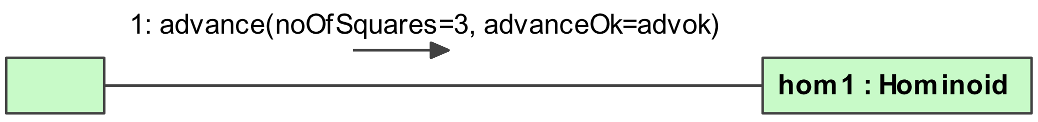
In UML, messages are shown in "communication diagrams"

We now know 3 types of UML diagrams:

- Class diagrams, contains classes (Chapter 3 in UML distilled). Defines a static view.
- Object diagrams, contains instances (Chapter 6 in UML distilled). (In MagicDraw this is also in class diagram, however in "real application" these should not be mixed). Defines a snapshot view of the system.
- Communication diagrams, contain instances and messages (Chapter 12 in UML distilled), Defines a "dynamic view" of the system.

Generally, these 3 should not be mixed!

Figure 6.3. Call to a Hominoid Object

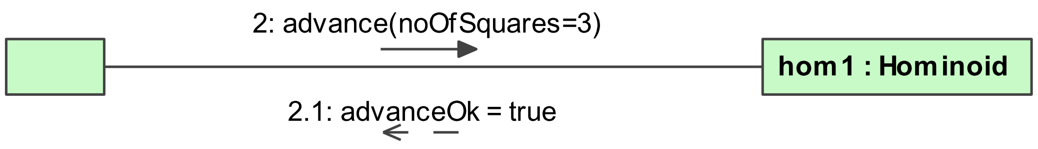


In a communication diagram:

- Instances are shown (Magicdraw does these in green)
- Associations are shown where messages are sent
- A message is an arrow next to an association. The arrow has to be a solid triangle! (this denotes a synchonous call, which is what we'll do in this class)
- The label for the arrow shows:
  - The sequence number (in what order do the messages appear?)
  - The name of the operation to call
  - The input and output arguments, as needed.

Alternate Notation for return values:

Figure 6.4. Call to a Hominoid object with return arrow

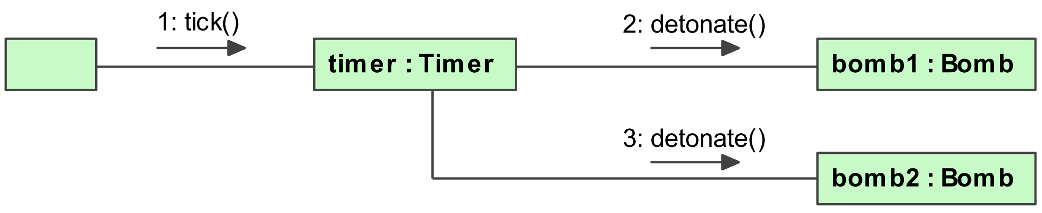


In this case, the return values are shown on a return arrow. A return arrow:

- is an open arrow
- is dashed

Sequence numbers describe the order in which the events are sent. Sequence numbers are consecutive (1, 2, 3, 4, 5, ...). They may contain sub-sequences (2, 2.1, 2.2, 3, 4, 4.1, 4.2, 4.3, ...)

Figure 6.5. Timer detonating two bombs

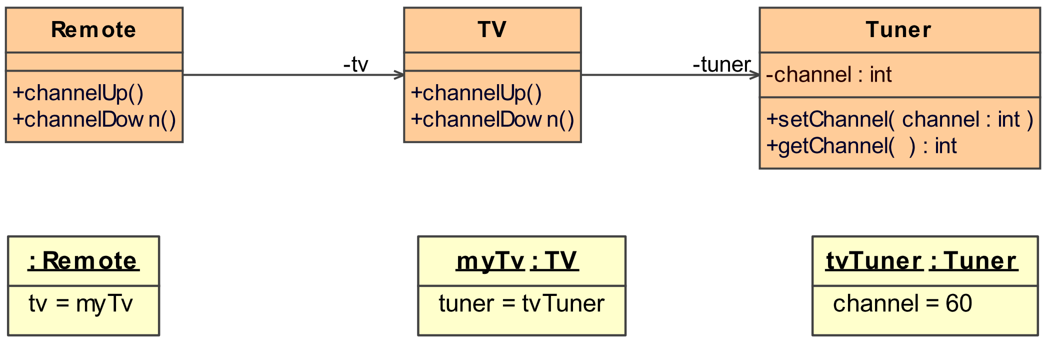


Messages from the outside are usually shown with no originator.

Note: MagicDraw does not allow this. We have to use an empty class.

Practice: Assume the following class diagram and the sate given in the following object diagram

Figure 6.6. A TV with a Remote and Tuner

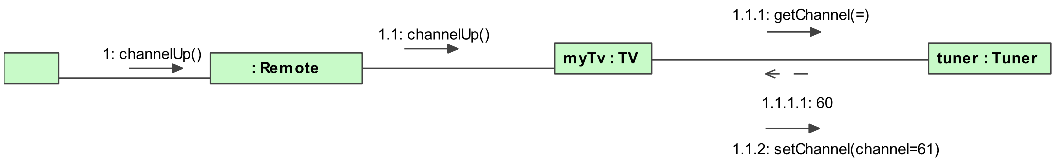


Assume the user presses the channelUp button on the remote. Show a communication diagram of what happens.

- You will need 3 links (one going to nowhere)
- You will need 5 messages (4 calls + 1 return message)

Possible Solution:

Figure 6.7. Communication diagram for TV



The roles of objects in messages

Objects may be

- the sender of a message (in C++ / Java: not available to the target)
- the target of a message
- pointed to by a variable (attribute) within another object
- pointed to by an argument passed back and forth in a message

Objects can play all these roles in their lifetime. However, some objects tend to always receive, while others almost always send.

Sender and Target are dependent on the message!

Pure OO (e.g. smalltalk) has only objects, no data. Even things like Integer are an object in smalltalk! However, in C++ we have to distinguigh between object and data. Data may

- be contained in a variable
- be passed back and forth as arguments
- not be the sender of a message
- not be the target of a message

Types of messages

There are 3 types of messages that an object may receive: informative messages, interrogative messages, and imperative messages.

Important

informative message	An informative message is a message to an object that provides the object with information to update itself. (It is also known as an update, forward, or push message.) It is a "past-oriented" message in that it usually informs the object of what has already taken place elsewhere.

Example: emplyee.gotMarried(marriageDate:Date, toWhom:Person).

Most "setter" methods fall in this category.

An informative message tells an object something that's happend in the part of the real world represented by that object.

Important

interrogative message	An interrogative mesaage is a message to an object requesting it to reveal some information about itself. (It is

also known as a read, backward, or pull message). It is a "present-oriented" message, in that it asks the object for some current information.

Example: hom1.location message asks hom1 to tell us the location

most getter messages fall in this category.

Important

imperative message	An imperative message is a message to an object that requests the object to take some action on itself, another object, or even the environment around the system. (It is also known as a force or action message.) It is a "future-oriented" message, in that it asks the object to carry out some action in the immediate future.

Example from the hominoid: hom1.advance().

This kind of action usually has some (more or less) complicated algorithm behind it.

Example: hom1.goToLocation(square:Square, out feasable: Boolean) would have to find a way to that location first...

Real-World example: robotLeftHand.goToLocation(x,y,z:Length, theta1, theta2, theta3:Angle) would ask a robot arm to actually move.

Example from the bomb:

- timer.tick() may tell the timer that another second has passed. This is an informative message.
- timer.setTime(time:int) sets the timer to a certain value. This is an informative message. may also be imperative (if something else happens, like the timer starts)
- bomb.detonate() is called when the timer is expired. This is an imperative message.

Practice: Classify each one of the 4 messages from the earlier TV example (in this case the return doesn't count)

1. channelUp() (user -> Remote) imperative (demand action) / informative (real-world -> computer)
2. channelUp() (Remote -> TV) imperative (demand action)
3. getChannel() (TV -> Tuner) interrogative (need to know information)
4. setChannel() (TV -> Tuner) imperative (demand action)

- A procedural progam consits of a set of instructions
- An object oriented program consists of a set of messages between objects

Book: Chapter 1.5

Chapter 7. Classes

Whenever we excute the "new" command on a class we instanciate an object that is structually identical to every other object created by the "new" statement on the same class.

Example: new Hominoid() creates an instance that is stucturally identical to every other new Hominoid().

Structurally identical means: same operations and variables

Important

class	A class is the stencil from which objects are created (instantiated). Each object has the same structure and behavior as the class from which is is instantiated. If object obj belongs to class C we say "obj is an instance of C."

There are two differences between two objects of the same class:

- Each object has a different handle
- Each object will probably have a different sate (= values for its variables)

The difference between objects and classes:

- A class is what you design and program
- Objects are what you create (from a class) at run-time

Architecture: Class is the blueprint, object is the house.

A class may spawn 3, 300, 3000, a lot of objects. A class resembles a stencil: Once the shape of the stencil is cut, the same shape can be traced from it thousands of times. All of the tracings will be identical (but still different objects).

But why bother?

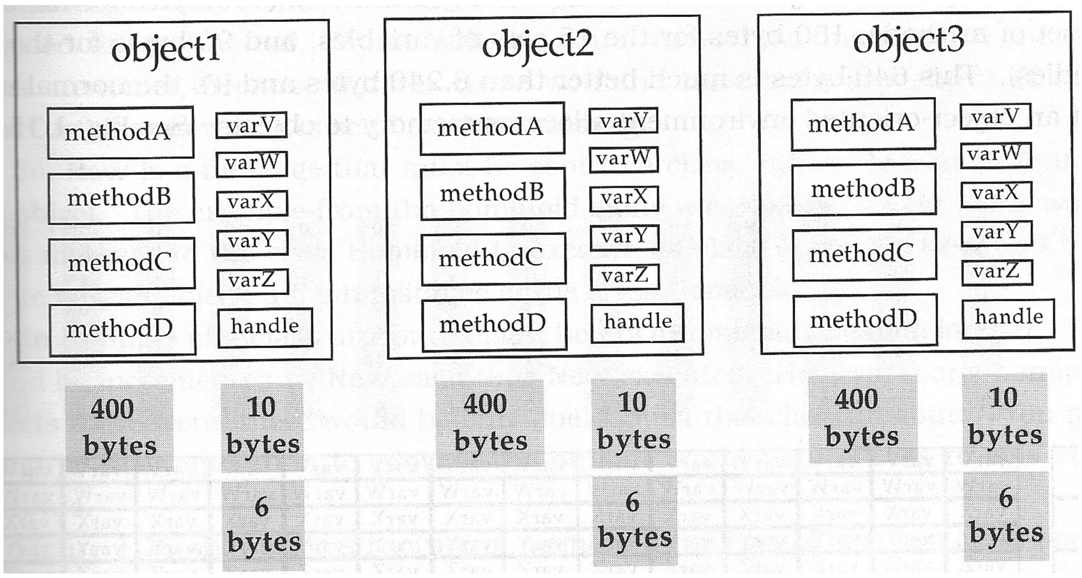
Look at the memory requirements: Lets assume that an object of a particular class has

- 5 variables that require 2 bytes each = 10 bytes
- 4 methods that require a total of 400 bytes
- a handle requiring 6 bytes [1]

This means a total of 416 bytes required for that object.

Now lets assume we have three objects of that class:

Figure 7.1. Three objects of the same class and their memory requirements



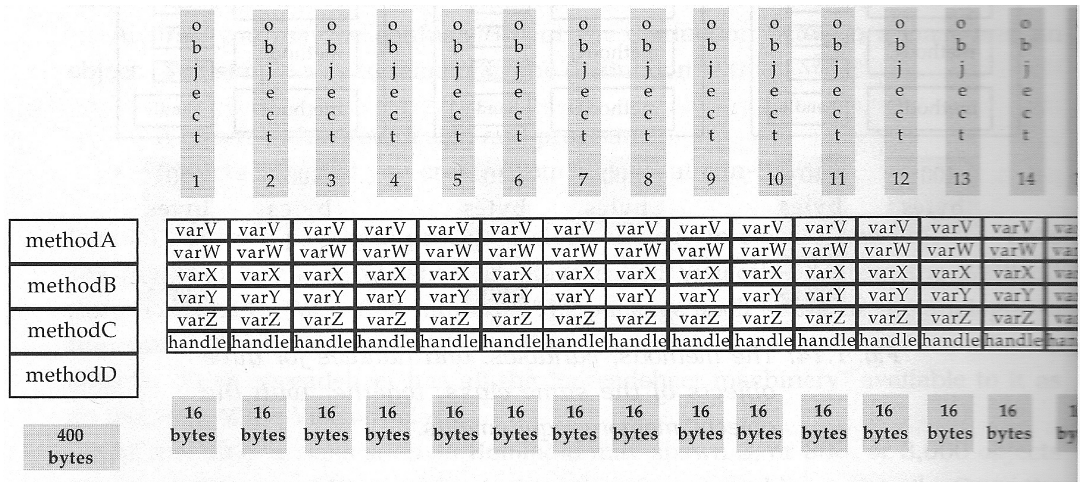
Now each one of them requires 416 bytes, making a total of 416 * 3 = 1248 bytes, right?

No! Because all of these objects are of the same class they are structurally identically, they may share the common code, which is the methods. (Each object has its own handle, and its own values for the variables, so they can not be shared)

So by sharing the actual code, which is the largest portion in most cases, we are able to save big!

Lets assume we now have 15 instances:

Figure 7.2. 15 objects sharing the same class



We need:

- 400 bytes for the class methods (* 1)
- 16 bytes for handle + values (* 15)

Making a total of 400 + 16 * 15 = 640 bytes!

Practice: Assume a class with the following memory requirements:

- methods: 500 bytes
- variables: 96 bytes
- handle: 4 bytes

How much memory is used for 1 instance? for 5 instances?

600 bytes / 1 and 1000 / 5

So far we have talked about so called "object instance operations" and "object instance variables".

Class operations and class attributes

Object instance operations and object instance variables work with instances of classes.

What happens if we want something where we know we have only one of it? Do we need to create an object for every "helper" class and pass it around to every other object?

No, we can use class operations and class variables instead!

They work much like in procedural programming (there can be only one), but are still encapsulated in a "class".

Object instance and class operations / attributes can even be mixed in one class.

Class operations are used:

- When there is only one of a certain thing.
- When there is no state.

Example of only one: Bank in Monopoly (guess what we're doing in next lab :)). Screen on the computer.

Example of no state: Utility functions, such a math functions, helper functions, etc. but also creation functions.

Class operation can only work with class attributes (they have no instance)!

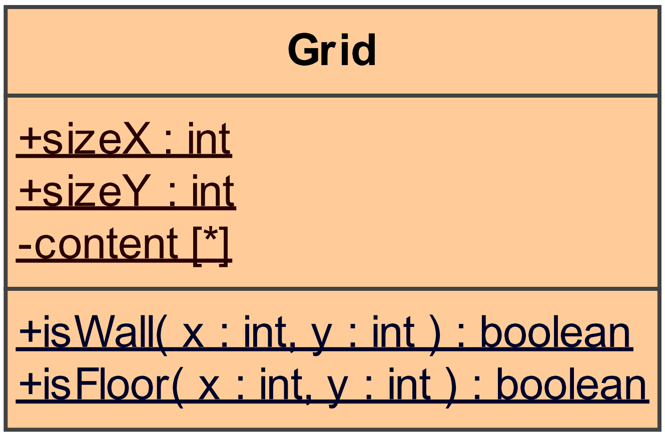
Object instance operations can work with class attributes or object instance attributes.

You can call class operations without creating an instance first! So anyone can send a message to a class operation without having a handle.

Sometime the notion "static" is used for class attributes and operations. (MagicDraw uses the flag "is static" for class attributes and operatons, C++ and Java use the keyword "static")

Notation in UML: In UML class attributes and operations are underlined.

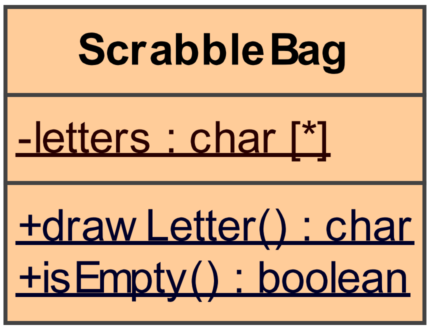
Figure 7.3. Grid for the Hominoid, implement completely static since there is only one


Grid
+sizeX : int
+sizeY : int
-content [*]
+isWall(x : int, y : int) : boolean
+isFloor(x : int, y : int) : boolean

Practice:

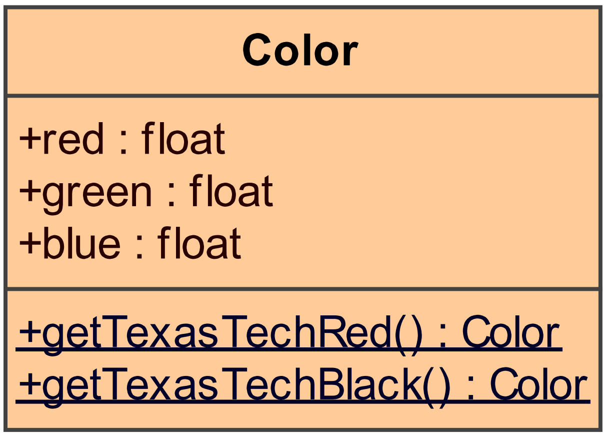
Assume a game of scrabble. The game has exactly one bag. The bag contains a bunch of letters (char) and provides an operation for retrieving a letter, and one to check if the bag is empty. Draw a UML diagram for the class "ScrabbleBag".

Figure 7.4. Solution for Scrabble



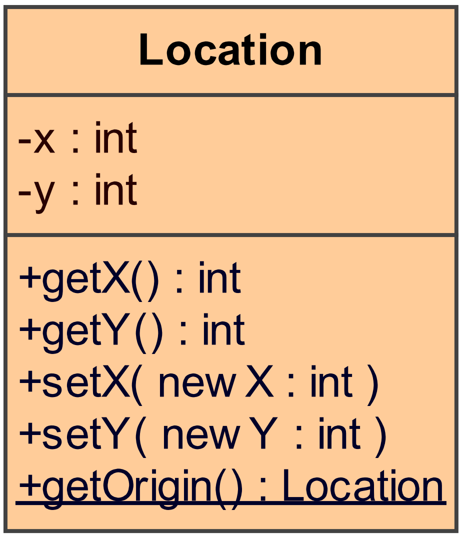
Example: Using static methods for creation of instances:

Figure 7.5. A color class with static methods for creation



Practice: Draw a UML diagram of a "Location" class. It should provide attributes for x and y and a class operation for "get origin"

Figure 7.6. Example solution for Location class



Book: Chapter 1.6, 3.6

Chapter 8. Inheritance

The problem: You write a class C and later discover a class D that is almost identical to C but has some extra attributes and operations.

You could just duplicate C's operations and attributes into D, but: This is extra work and make maintenance difficult

Better solution: Ask class D to re-use operations / attributes of C. This is called inheritance.

Important

inheritance	Inheritance (by D from C) is the facility by which a class D has impliclity defined upon it each of the attributes and operations of class C, as if those attributes and operations had been defined upon D itself.
superclass	C is a superclass of D.
subclass	D is a subclass of C.

Through inheritance, objects of class D can make use of attributes and operations that where defined in class C (and D).

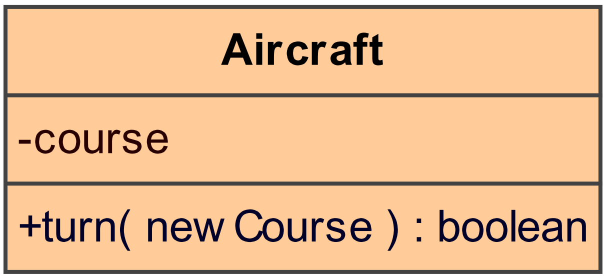
Inheritance allows to build software incrementally, allowing you to:

- First build classes for the generic case
- Then build classes for special cases that inherit from the generic classes.

Example:

Class Aircraft which defines an operation turn() and an instance attribute course. This works well for the all air crafts.

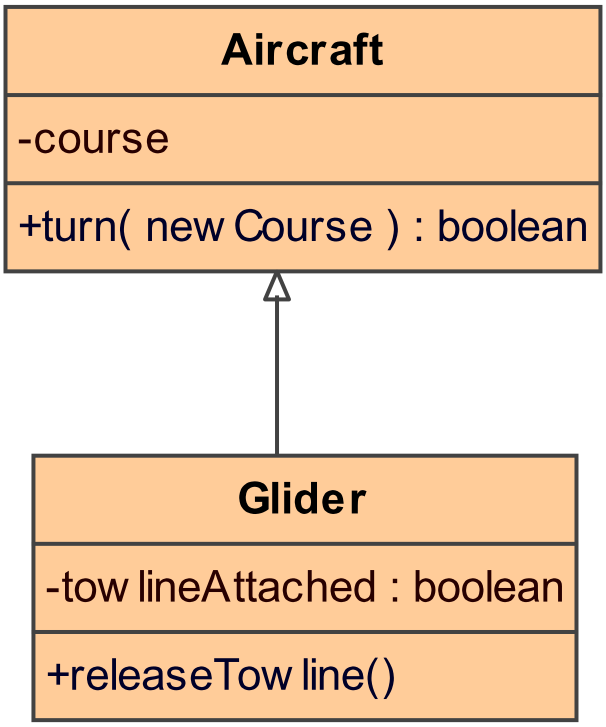
Figure 8.1. UML diagramm for class Aircraft



However, there may be special aircraft, such as glider. A glider need to know its course and be able to change it, but it also needs special operations (e.g. release towline) and special attributes (e.g. isTowlineAttached).

We can have Glider inherit from Aircraft. Then Glider will have all its instance attributes and operations and the ones from Aircraft.

Figure 8.2. Glider inheriting from Aircraft



The glider now has the attributes:

- course
- towLineAttached

and the operations:

- turn()
- releaseTowline()

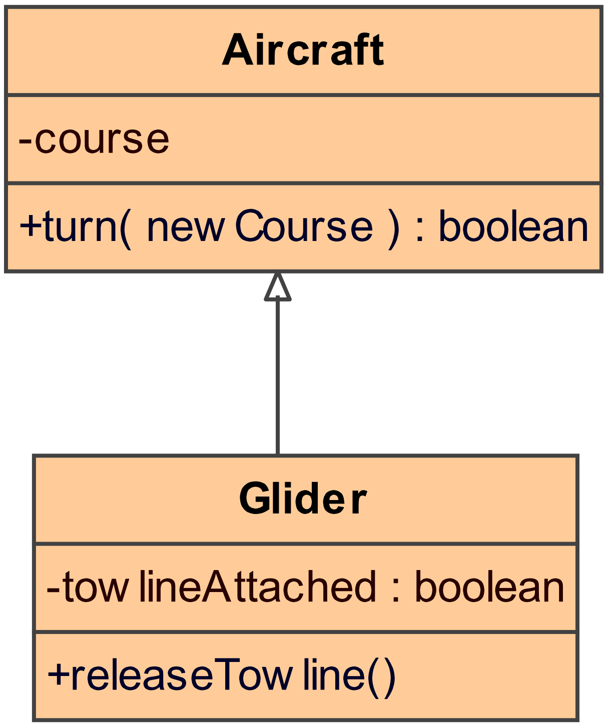
Notes for the UML notation:

- inheritance is shown with a directed arrow.
- the arrowhead must be an unfilled triangle!

MagicDraw calls this "Generalization".

Practice: Model a cable box and a dvr cable box. The cable box has a current channel, which can be set with an "channelUp" and a "channelDown" button. A dvr cable box has all the features of a regular cable box, but also has a "record" button and has a certain number of minutes left for recording. Draw a UML diagram for both classes, using inheritance.

But lets look again at the aircraft / glider example and some sample code:



```
var ac: Aircraft := Aircraft.New;
var gl: Glider := Glider.New;
...
ac.turn(newCourse, out turnOk);
gl.releaseTowline;
gl.turn(newCourse, out turnOk);
ac.releaseTowline;
...
```

Or the same in C++ notation:

```
Aircraft *ac = new Aircraft();
Glider *gl = new Glider();
...
turnOk = ac->turn(newCourse);
gl->releaseTowline();
turnOk = gl->turn(newCourse);
ac->releaseTowline();
...
```

The object pointed to by ac receives the message "turn", which causes it to apply the operation turn(). Since ac is an instance of Aircraft, it will use the operation turn() defined on the class Aircraft.
The object pointed to by gl receives the message "releaseTowline" which causes it to apply the operation releaseTowline(). Since gl is an instance of Glider, it will use the operation releaseTowline() defined on the class Glider.
The object pointed to by gl received the message "turn", which causes it to apply the operation turn(). Glider does not define turn(), but since Aircraft is a superclass of Glider, it will use the operation turn() defined on the class Aircraft.
This will not work! ac refers to an instance of Aircraft, but Aircraft does not have an operation "releaseTowline". Inheritance does not help in this case!

Distinction between object and instance:

Any object is an instance of its class and of all of its classes superclasses!

- ac is an object of class Aircraft. ac is an instance of class Aircraft.
- gl is an object of class Glider. gl is an instance of the classes Aircraft and Glider.

Compare to real world: If you own a glider, you own an aircraft at the same time (even though it is the same object).

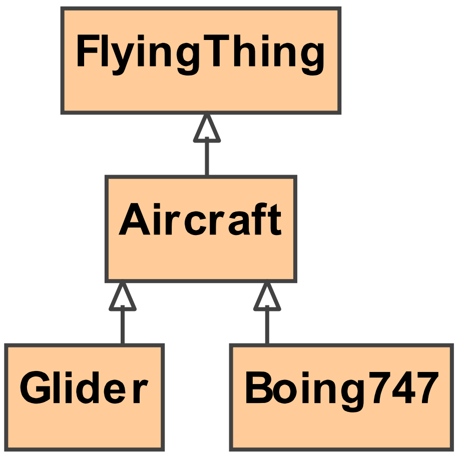
is-a relationship: Inhertiance is usually useful whenever an is-a relationship can be used. Example: Every glider is-an aircraft.

Common mistake: Using inheritance instead of attributes. For example, every Hominoid has a location. So why not just inherit from Location? Then Hominoid would have all the attributes and operations needed for it's location, right? Technically - Yes. Logically, is every Hominoid also a Location? No! A Hominoid has a location. Therefore, also technically feasable this is bad design!

Inheritance can span multiple levels. Example: FlyingThing, AirCraft, Glider. Every glider is an aircraft and a flying thing. Every aircraft is a flying thing.

Every class may have multiple subclasses! Example: a Boing 747 is an aircraft, a Glider is an Aircraft.

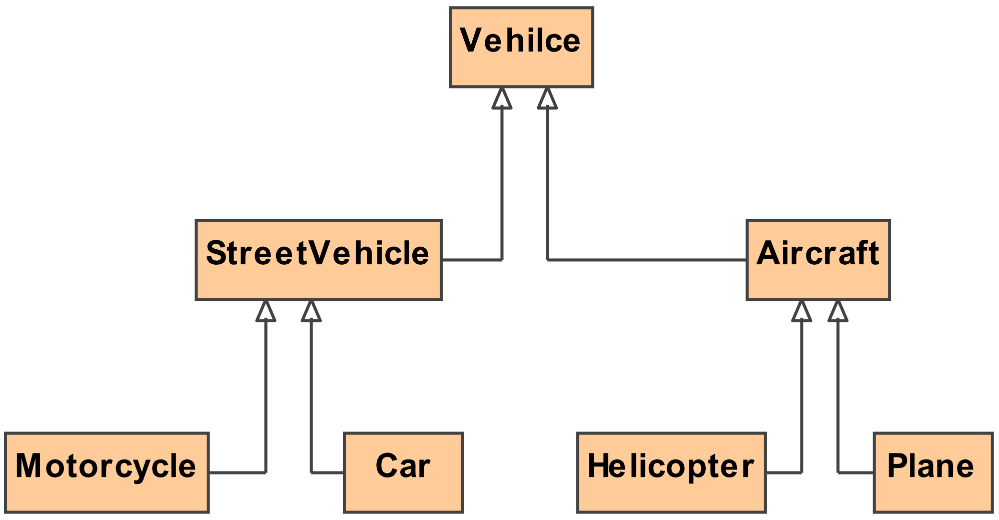
Figure 8.3. Larger example of inheritance



Practice: Draw a larger inheritance diagram (use short class notation), for the classes:

- Motorcycle
- Helicopter
- StreetVehicle
- Car
- Aircraft
- Plane
- Vehicle

Figure 8.4. Example solution for vehicle inheritance

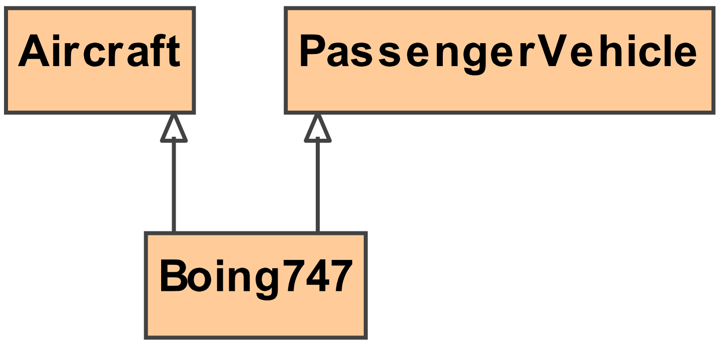


Multiple Inheritance

So far we have seen single-inheritance: Each class had only one direct superclass. However, each class could have multiple subclasses. This can be shown in an inheritance tree (The example we have seen so far).

But can a class also have multiple (direct) superclasses? Yes, this is called "multiple inheritance".

Figure 8.5. Example of multiple inheritance



Warning! Multiple inheritance can cause mayor design problems (= headaches). Why? what happens if operations or attributes "clash"?

Example: Assume Aircraft defines a "size" attribute which defines the length of the aircraft in feet (float). PassengerVehicle defines a "size" atrribute which defines how many passenger fit in this vehicle (int). When Boing747 inherits the "size" attribute, what does it mean?

As a matter of fact, multiple inheritance can cause so many problems that some languages forbid it! C++ and Eiffel allow it, Java, Delphi, ObjC and Smalltalk do not. (Java / Delphi / ObjC have a workaround with interfaces / protocols, but more about that later).

Multiple inheritance, if used, should be used very cautiously! Only use it if there is no other way!

Chapter 9. Polymorphism

Polymorphism comes from greek:

- poly = many
- morph = form

Example: Odo from "Deep Space Nine"

In OO there are two definitions:

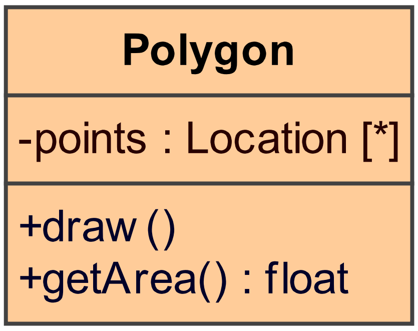
Important

Polymorphism	1. Polymorphism is the facility by which a single operation or attribute name may be defined upon more than one class and may take on different implementations in each of those cases.
	2. Polymorphism is a property whereby an attribute or variable may point to (=hold the handle of) objects of different classes at different times.

So what does that mean?

Example: Imagine a class for polygons:

Figure 9.1. A class for polygons

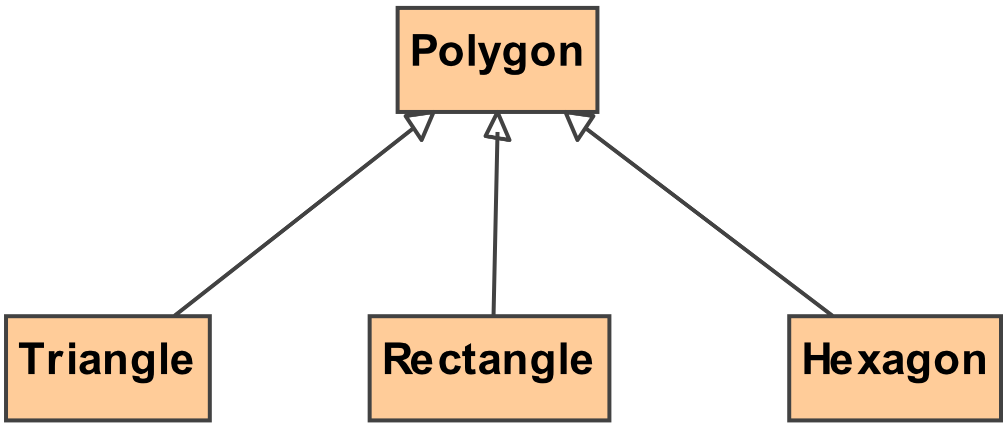


Sidenote: Remember derived attributs? area is a derived attribute.

If we where to look into the implementation of getArea() is would have to be very very very very very complicated to support any polygons.

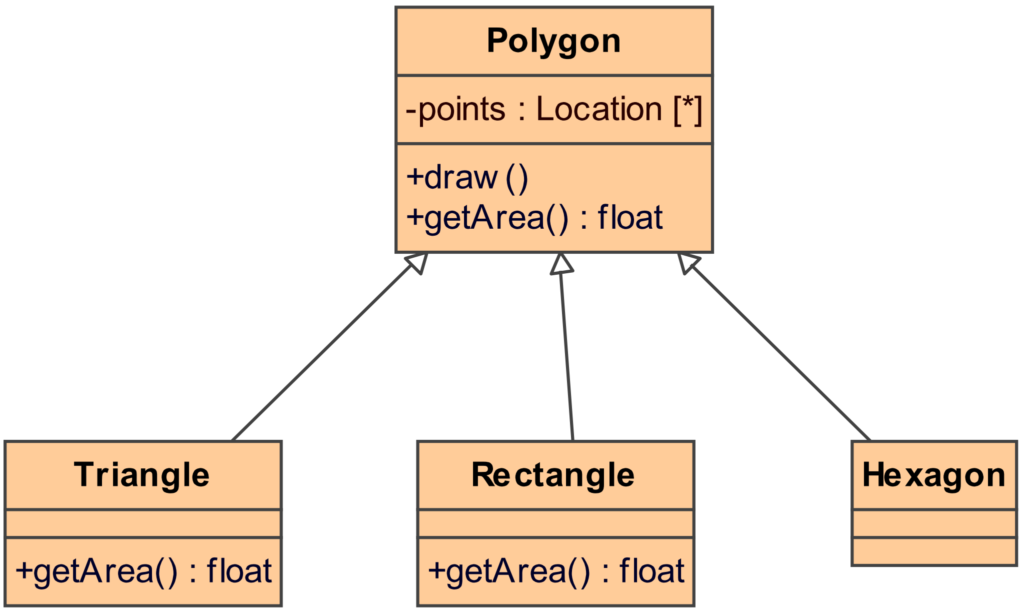
Now how about some special cases of polygons?

Figure 9.2. Polygon with some subclasses



getArea() for Triangle and for Rectangle are very easy (for Hexagon it is still complicated). Can't we just redefine them for the subclasses with some more efficient algorithms? Yes, we can!

Figure 9.3. Polygon with subclasses that overwrite methods



Now we can implement an easier version of getArea() for Triangle and for Rectangle. We say "Rectangle overrides the method getArea()"

Important

Overriding	Overriding is the redefinition of a method defined on a class C in one of C's subclasses.

Rules for overriding:

- Methods can be overridden

- Attributes can never be overridden
- In C++ (and most other languages, e.g. Java) overridden methods must have the exact same signature (=name, capitalization, parameter list)
- In C++ (but not in most other languages) overridden methods may have a different return type (although you should use that with caution)
- In C++ overridable methods have to be marked with the "virtual" keyword (but more of that later)

Practice: Show an inheritance diagram of the classes "Aircraft", "Glider" and "Boing747". Assume that every aircraft has an operation "startEngines()". Boing747 will just use that operation. Glider will have to override it, since it has no engine.

Let's assume the following code:

```
twoDShape.getArea; // Book's notation
twoDShape->getArea(); // C++ notation
```

We may not know which algorithm gets executed, this depends on the Class that the object two2DShape belongs to. We have to look at several cases:

1. twoDShape is an instance of Triangle, the operation getArea() from Triangle will be executed.
2. twoDShape is an instance of Recangle, the operation getArea() from Rectangle will be executed.
3. twoDShape is an instance of Hexagon. Hexagon has no getArea() function, so the one from Polygon is used.
4. twoDShape is an instance of Polygon, the operation getArea() from Polygon is used.
5. twoDShape is an instance of another class C that is not related to Polygon. Since that class does not have an operation getArea() we will get an error (compile-time in C++)

But how can I not know which class an object belongs to???

The answer is: Since every object is an instance of all of its superclasses, we can use it every place where we can use the superclass (Assignments, as parameters)! Here are some examples:

```
var p: Polygon;
var t: Triangle := Triangle.New;
var h: Hexagon := Hexagon.New;
...
if user sayas OK
then p := t
else p := h
endif;
...
p.getArea; // P may be a Triangle or a Hexagon object
...
```

Or in C++ notation:

```
Polygon *p;
Triangle *t = new Triangle();
Hexagon *h = new Hexagon();
...
if (userSaidOk)
  p = t;
```

```
else
  p = h;
...
p->getArea(); // P may point to a Triangle or a Hexagon
...
```

Even though we assign an object of class Triangle (or Hexagon) to an object handle for Polygons, the object will still keep its class, so it will still execute the getArea() function from Triangle (or Polygon in the case of Hexagon)

In classical programming we would need to test of what type p really is and have some complicated switch-statement.

Advantage: We can now add new classes that are subclasses of polygon and overwrite getArea() without changing any of the existing code!

The general declaration "var p:Polygon" (Polygon *p) is a safety restriction. It makes sure that:

- p can only hold handles to objects of class Polygon or its subclasses. (e.g. we cannot assign it an object of class "Consumer")
- we can only execute the operations defined on Polygon, or we will get a compiler error.

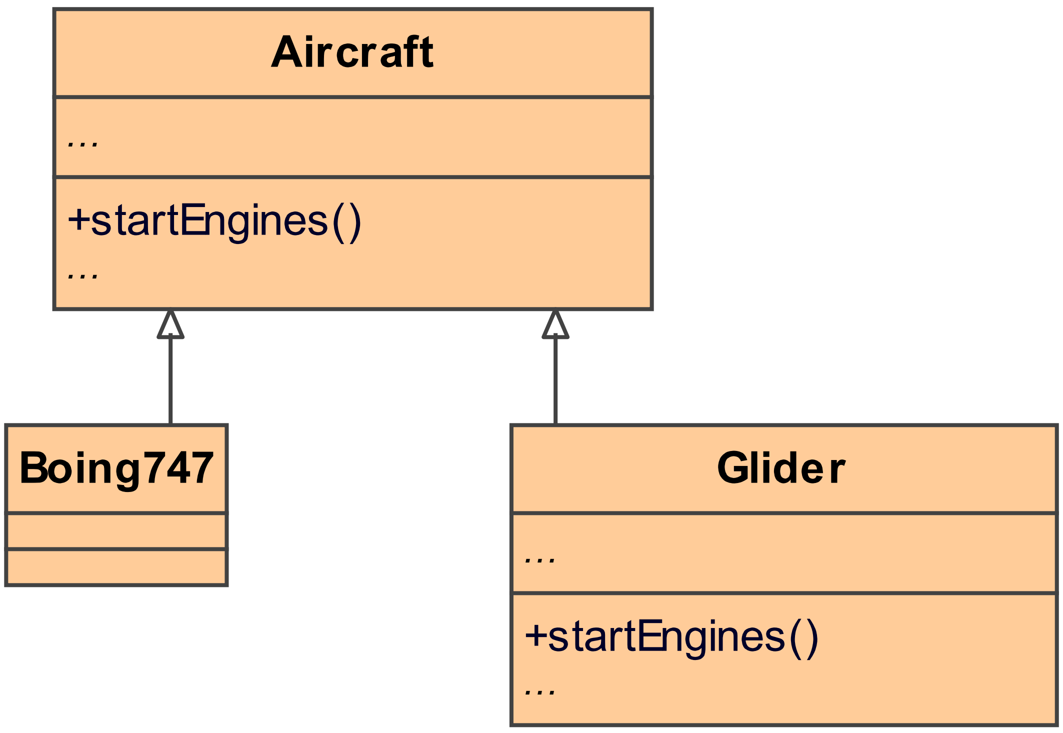
The operation getArea() is an example of the first definition of polymorhpism.

The variable p is an example of the second definition of polymorphism.

Practice:

Assume the following classes:

Figure 9.4. Inhertiance diagram for Aircrafts



And the following code:

```
Aircraft *a = new Aircraft();
Boing747 *b = new Boing747();
Glider *g = new Glider();
...
a->startEngines();
b->startEngines();
g->startEngines();
...
a = g;
a->startEngines();
...
a = b;
a->startEngines();
```

which startEngines() functions are called in this code? Given an answer for the numbered lines.

Sidenote: Here we are losing an object, which one?

How does polymorphism work? Through dynamic binding:

Important

dynamic binding	Dynamic binding (or run-time binding or late binding) is the technique by which the exact piece of code to be executed is determined only at run-time (as opposed to compile-time)

The environment inspects the class at the last possible moment: at run-time, when the message is sent.

Do not confuse overriding with overrloading. Remember overloading from the intro class? We had function overloading and operator overloading.

overloading	overloading of a name or symbol occurs when several operations (or operators) defined on the same class have that name or symbol but different parameters. We say the name or symbol is overloaded.

Overriding: Superclass declares operation, subclass overrides it. Example: Aircraft declares startEngines(), Glider overrides startEngines().

Overloading: In the same class, but with different arguments! Example: Aircraft defines startEngines() and startEngines(toWhichPower:float).

Overloading is done by signature and can actually be checked at compile-time.

Some people (including me) sometimes call overloading polymorphism but that is incorrect! Only overriding is polymorphism.

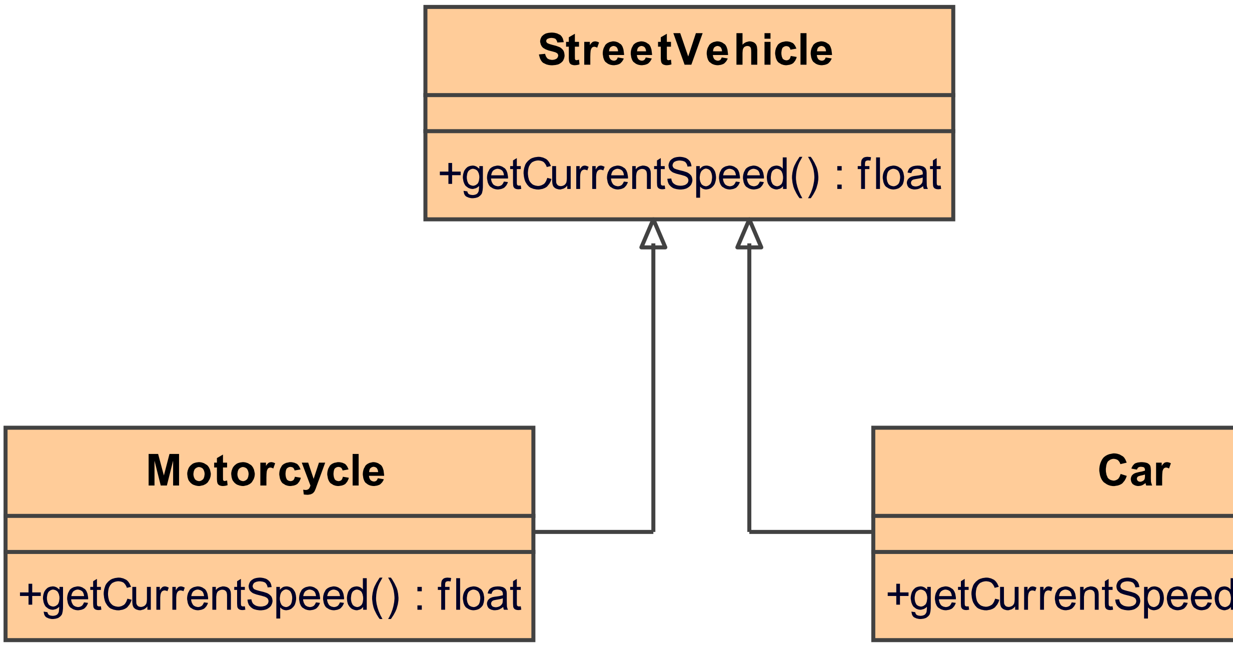
Book: Chapter 1.8

Abstract operations

No we can override methods and declare superclasses with operations.

But what if an operation does not make sense on a superclass? What if the superclass is actually "useless" and just used for polymorhpism?

Figure 9.5. Example operation on vehicles



In this example, we would never instantiate an object of class "StreetVehicle". We would use the class StreetVehicle only to combine Motorcyle and Car. We would instantiate objects of class Car or Motorcycle and use them as StreetVehicle. The importance of every StreetVehicle is that is has an operation getCurrentSpeed() !

Then why even bother implementing getCurrentSpeed() on StreetVehicle? Instead, we can make it an abstract operation.

Important

abstract operation	An operation is lacking a a workable implementation. Normally, a descendent class will override this inherited abstract operation with its own concrete operation

Unfortunately, when a class has at least one abstact operation we will also need to make the class abstract

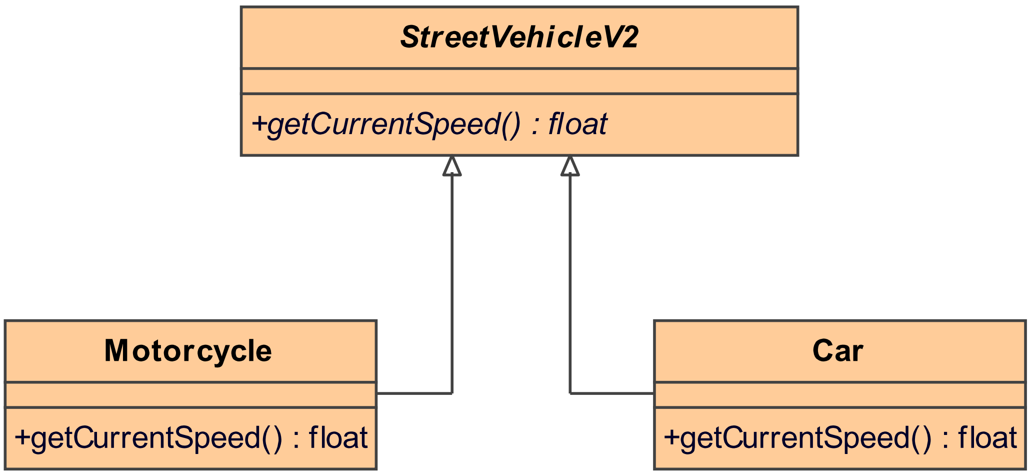
Important

abstract class	A class from which objects cannot be instantiated (normally because the class has one or more abstract operations). An abstract class is usually used as a source for descendant classes to inherit its concrete (nonabstract) operations.

The reason: Would we instantiate StreetVehicle and call its operation getCurrentSpeed(), but Streetvehicle does not implement it, we don't know what to do.

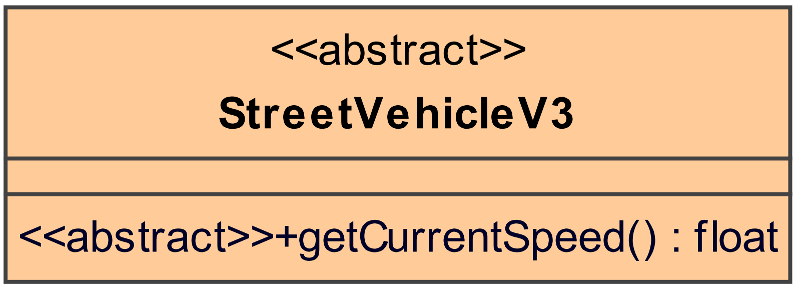
Abstract classes and operations are shown in UML in italics.

Figure 9.6. Vehicles with abstract superclass


StreetVehicleV2
+getCurrentSpeed() : float
Motorcycle
+getCurrentSpeed() : float
Car
+getCurrentSpeed() : float

Unfortunately you can not write italics on paper (you may be able to but I am not able to distinguish your italics from non-italics. Therefore if you want to denote abstract classes or operations in this class on paper, you have to use the stereotype <<abstract>>. I do not want you to write things like "this is italics" or something, since that would not be correct UML!

Figure 9.7. Abstract object as done on paper



The book uses an alternate notation. The use UML constraints (similar to stereotypes, but written in { }) to denote abstract classes on paper. They would write {abstract}.

While stereotypes go in front of the method / class names, constraints go after the the declaration. Example: StreetVehicle {abstract}. See the book for an example.

You may use the books notation if you like or the stereotypes notation! But deceide on either one and do not mix them in the same diagram!

Book: Chapter 3.7

Practice: From the Vehicle example, draw an class diagram showing the classes Aircraft, Helicopter, and Plane. Add an operation getCurrentHeight() that is only defined for Helicopter and Plane, but specified on all Aircraft.

Pure abstract classes (interfaces)	A class that has has no attributes and only abstract operations is called a pure abstract class or interface.

Notes

- A pure abstract class may be a subclass of one or more pure abstract classes. (but not of abstract classes or regular classes).
- Since pure abstract classes define no attributes and only abstract operations they cause no problems with multiple inheritance

To denote a pure abstract class, you may use the stereotype <<interface>> on the class name. If you do your design on paper, thats enough. If you use a modelling tool you still have to italicize your class name and operations!

Practice: Draw a class diagram for the classes Chair, Table, Floor, and the pure abstract class Seat. Seat defines an operation sitOn() that the other three classes override.

Chapter 10. Genericity

Important

Genericity	Genericity is the construction of a class C so that one or more of the classes that it uses internally is supplied only at run-time (at the time that an object of class C is instiated)

Problem:

Sometimes you want to write an class that supports "generic" types of classes.

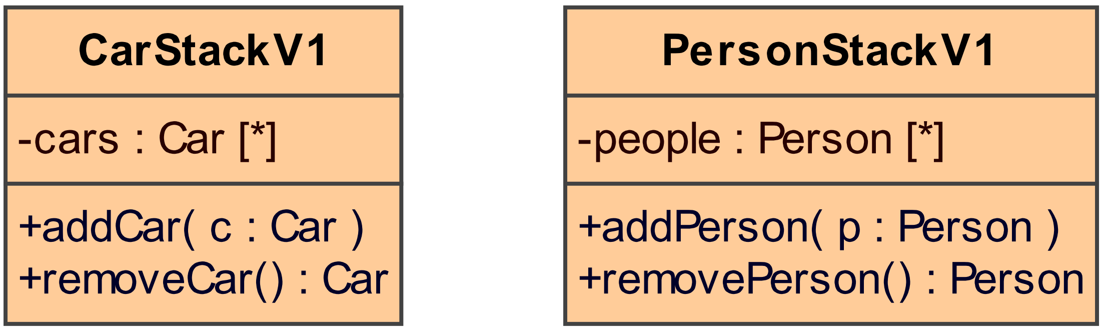
Example: List of something. Provides the same operations, but takes different parameters.

Assume we have the following classes:

Figure 10.1. Two classes that have nothing in common

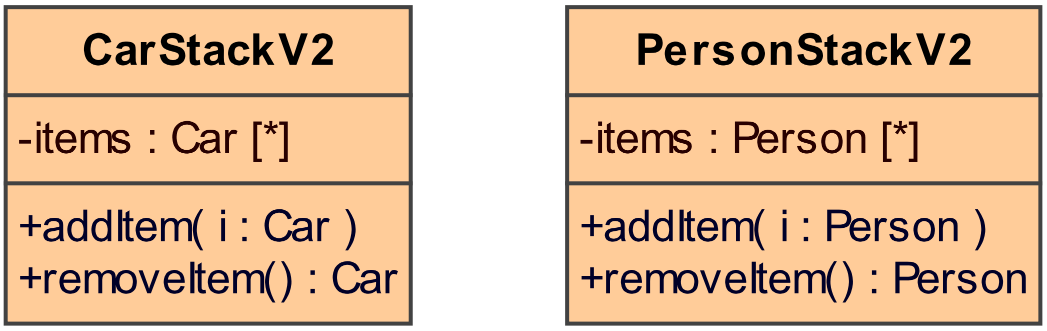
Unfortunately these two things have nothing in common. So if we want to provide a "Stack" of cars and a "Stack" of People, we'd have to implement two classes:

Figure 10.2. Two different stacks for two different types



However, if we look at the functionality they provide, they are actually very similar. As a matter of fact, the implementation of both would be very much the same. If we would replace "Car" or "Person" with a more generic term like "item", we get:

Figure 10.3. Two similar stacks for two different types

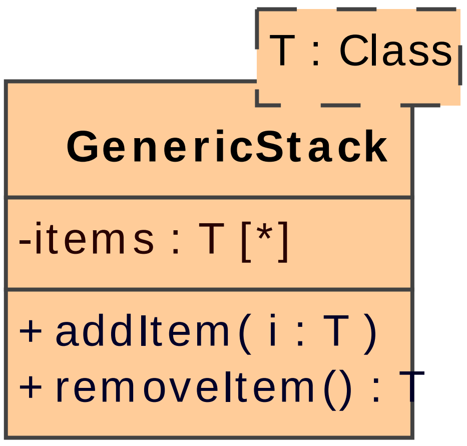


Please note, that the only difference between these two classes is now the class on which they operate. One still has "Car", the other one "Person". The actual implementation of the functions will now be exactly the same!

So can we not make a class that somehow has a parameter of which class to operate on?

Yes, we can! (Thats the point of this lecture). These classes are called "Generic classes".

Figure 10.4. A generic Stack Class



UML Notation:

- A generic is "added on" as an extra box with a dashed outline
- The format is name : type
- You can use any name you like
- The name can be used inside that class like a type
- In this class, type will always be "Class"
- Every class may have multiple generics, they each go on their own line.

Note for MagicDraw: In MagicDraw, Generics are called "Template Parameters".

We are now not only specifying one class, but actually a set of classes. Each individual class can only use the data type provided! To use a particular class, we have to provide the template parameter.

Example in C++:

```
GenericStack<Car> *carStack = new GenericStack<Car>();
GenericStack<Person> *people = new GenericStack<Person>();
```

Then, using it is easy:

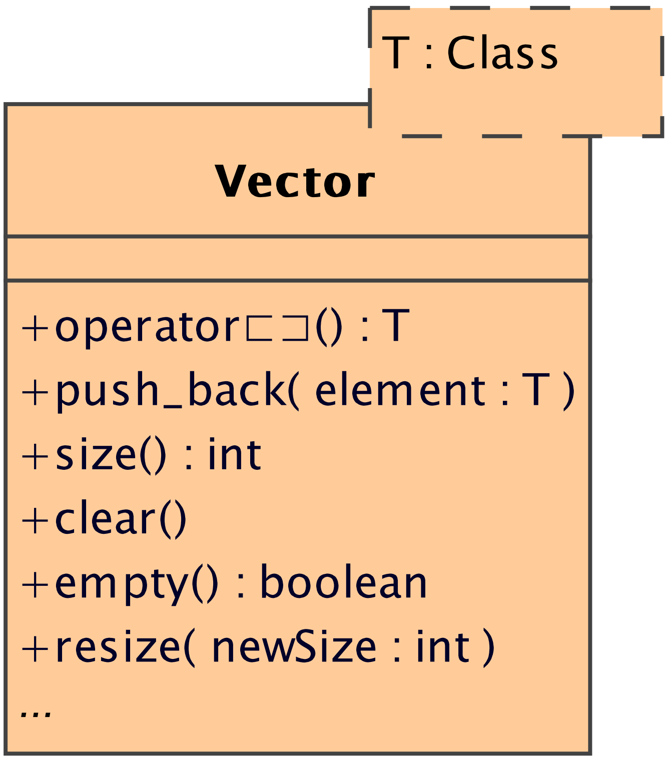
```
Car *c = new Car();
carStack.addItem(c);
carStack.addItem(new Car());
Person max = new Person();
Person someoneelse = new Person();
people.addItem(max);
people.addItem(someoneelse);
```

Until your data structures class, you will very likely not write generic classes, but you will use them. C++'s Standard Template Library (STL) provides many very useful generic classes.

For everyone who misliked arrays in C++, there is the generic class "Vector". This class provides:

- a [] operator for easy access to its elements
- an operation to add new elements
- an operations to check the current size
- an operation to wipe out the contents
- an operation to check it is empty (convenience for size()==0)
- a resize operation
- and many more

Figure 10.5. The STL class "Vector"



Note: Everytime where we have multiplicity with a given maximum size, you will most likely use static arrays in C++ (example: Car has 4 wheels)

Everywhere where we have multiplicity with a maximum size of * you will most likely use the Vector<T> class.

So whenever you model:

```
properties : Property [*]
```

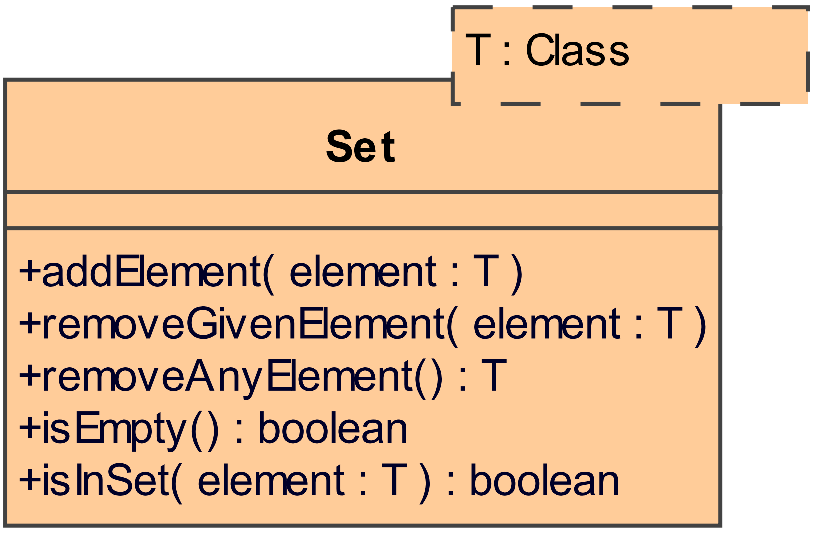
you would actually implement:

```
properties : Vector<Property>
```

Note for this class: Always use multiplicity in models, Vector<> in implementation. I do not want to see Vector<> in your model!

Practice:

Draw an UML diagram for a generic class "Set". A set contains elements of a certain class. It should provide operations to add an element, remove an element, check if the set is empty, and check if an element already exists in the set.


T : Class
Set
+addElement(element : T)
+removeGivenElement(element : T)
+removeAnyElement() : T
+isEmpty() : boolean
+isInSet(element : T) : boolean

Appendix A. Summary

That's it! If you understood these parts you've done well!

We covered chapters 1,3,4 and parts of 5 of the book.

Part II. Object Orientation in C++

Table of Contents

Object Orientation in C++ Overview

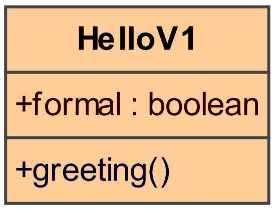
In this part, we will look into C++ and its implementation of Object Orientation. For this chapter we will mostly follow chapter 6 of C++ in a nutshell.

Chapter 11. Classes in C++

Class definitions

Before we can use classes, we have to define them. Assume the following simple class:

Figure 11.1. Hello World Class



In C++ there are two parts to every class:

- Class definition
- Class implementation

Important

Class definition	The class definition defines the structure of the class. It defines the class name, member variables and member functions.
Class implementation	The class implementation specifies the behavior of the class. It gives an implentation for its member functions.

Warning to all tha have used Java before: In Java they are both together, in C++ they have to be separated!

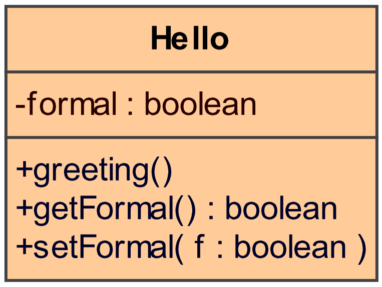
Sidenote: As you should be able to guess, pure virtual classes (interfaces) have no implementation.

But back to the class. The first step is redefining it for the actual implementation (we learned some of that earlier).

- Use getters / setters instead of public attributes
- Use Vector<> for multiplicity

In an actual project we would probably do this step in our head. But it doesn't hurt do to on paper:

Figure 11.2. Hello redefined for implementation



Although it is technically possible and perfectly legal to declare public attributes in C++, it is not legal in this class! For all projects, designs, implementations, etc. you do in this class you have to use private attributes!

The example here is intentionally simple. In reality we would probably declare a default value for formal, but we need to know about constructors first (in one of the next classes).

Given the definition in UML we can now translate it into C++.

Example for a class definition:

```
class Hello
{
private:
  bool formal;
public:
  void greeting();
  bool getFormal();
  void setFormal(bool f);
};
```

Lets look at the different parts:

> The class definition starts with the keyword "class" followed by the class name, followed by an open brace (similar to structs).
> Visibility modifiers in C++ are a given as the keyword (private or public) and then a colon (:).
> Note that there is no "boolean" datatype in C++, it is called "bool". Other than that an attribute defintion is similar to a variable definition.
> Visibility modifiers count for more than just the next definition. They count for every definition from that point onward until the end of the class (unlike Java).
> A function with no return has the "void" return type. Member function definitions look like C ++ prototype declarations.
> Remember, in C++ it is type, space, name
> The class definition ends with a closed brace and a semicolon! Do not forget the semicolon! (This is the same as with structs).

Other notes

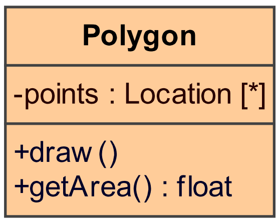
Order	As in UML, in C++ it is convention to declare all variables first, then all member functions.
Indentation	Usually the class definition starts with no indentatation. Visibility modifiers and the closing brace are on the same level as the class definition. Member attributes and operations are indented.

As you can see a C++ class definition shows the exact same thing as a UML class definition. This is not a coincidence. There are even programs that can produce one from the other. However, these are very expensive.

Practice:

Write a C++ class definition for this class:

Figure 11.3. Polygon class

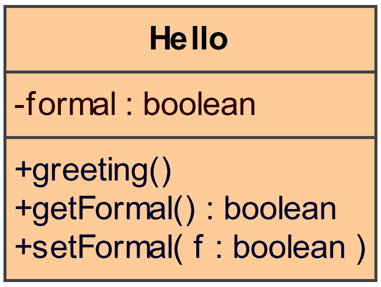


Hints: You will not need any getters / setters here. The corrent type for "points" would be "Vector<Location>".

We have now defined a class and its interface. But now we have to give actual implementations for the defined methods.

Lets look at the Hello class again:

Figure 11.4. Specification for hello



We have defined the class

And its attributes.

What is missing is the methods.

Fortunately, implementing the methods is much like we've seen in implementations before:

```
void Hello::greeting()
{
  if (formal)
    cout << "Hello, nice to meet you!" << endl;
  else
    cout << "What's up?" << endl;
}

...
```

The "Hello::" is borrowed from namespaces. In this case, a class is somewhat like a namespace (although a different thing)

The header line of a method implementation is:

Return data type (void if no return value), class name, colon-colon (::), method name, parameters

The body of a method is exactly the same as we learned in earlier.

In the method, we can make use of all attributes of the same class as if they were global variables. In the example given we can use "formal" because it is defined inside the class "Hello" and our greeting method is for the class "Hello".

Practice: Assume the following class definition:

```
class Point
{
private:
  float x;
  float y;
public:
  float distanceFromOrigin();
};
```

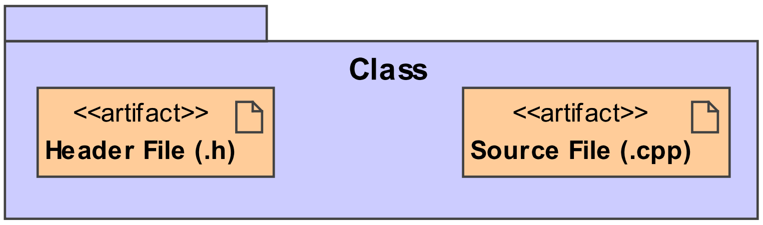
Give an implementation for the distanceFromOrigin function. Note: the formula is root(x^2+y^2) (our course this is NOT C++ notation).

Where do things go?

Now that we know how to define a class, and how to implement its methods, lets look at where things go.

Note: These things are conventions in real life, but for me (and that means for you in this class) they are unbreakable rules!

Figure 11.5. Overview of a Class in C++



Rules:

- The class defintion belongs in a header file (.h), that has the exact same name and capitalization of the class. Example: Hello.h
- The class implementation belongs in a source file (.cpp) that has the exact same name and capitalization of the class. Example: Hello.cpp
- The header file will be guarded against multiple inclusion with the #ifndef ... #define .. #endif construct
- The source file will always include its own class definition (e.g. #include "Hello.h")

This way, there are always 2 files for each class (exception: pure virtual classes).

Note: If you use eclipse you can have ecplise create these files for you (say New / Class)

To start the program, we still need a main() function. This function should go into its own file, preferably something like "main.cpp". In good OO programs this function is very short! Example:

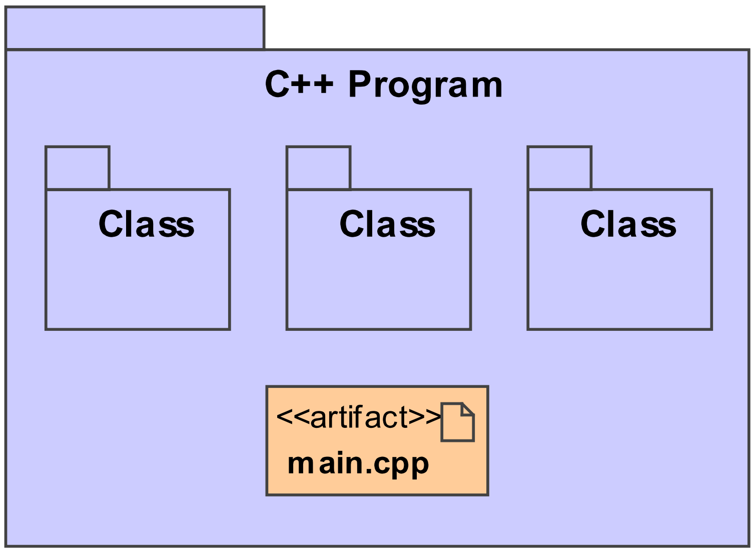
```
#include "SomeClass.h"
```

```
int main()
{
  SomeClass *myInstance = new SomeClass();
  myInstance->start();
  delete myInstance;
  return 0;
}
```

Remember to always include things where there are used!

Because I love graphics, here's another graphic showing the same thing:

Figure 11.6. A C++ program



But enough theory, here is a complete example:

Example 11.1. Hello.h

```
#ifndef HELLO_H_
#define HELLO_H_

class Hello
{
private:
  bool formal;
public:
  void greeting();
  void setFormal(bool f);
  bool getFormal();
};

#endif /*HELLO_H_*/
```

Example 11.2. Hello.cpp

```
#include "Hello.h"
#include <iostream>
using namespace std;
```

```
void Hello::greeting()
{
  if (formal)
    cout << "Hello, nice to meet you!" << endl;
  else
    cout << "What's up?" << endl;
}

void Hello::setFormal(bool f)
{
  formal = f;
}

bool Hello::getFormal()
{
  return formal;
}
```

Example 11.3. main.cpp

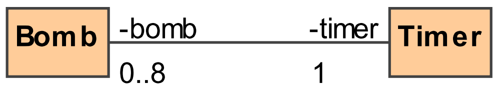
```
#include "Hello.h"

int main()
{
  Hello *h = new Hello();
  h->setFormal(true);
  h->greeting();
  delete h;
  return 0;
}
```

Incomplete class declarations.

Problem: Two classes reference each other.

Figure 11.7. Example of two classes referencing each other



Here's a fist attempt at an implementation in C++

```
// this is Bomb.h
#include "Timer.h"

class Bomb
{
private:
  Timer *timer;
}

// this is Timer.h
#include "Bomb.h"
```

```
class Timer
{
private:
  Bomb *bomb[8];   // This works due to what is
                   // stated about the * later
}
```

Alternate notion:

```
// This is an alternate Timer.h
#include "Bomb.h"
#include <vector>

class Timer
{
private:
  vector<Bomb*> bomb; // Much safer :)
}
```

Discussion: What seems to be the problem? How could you solve this?

Answer in C++: Incomplete class definitions.

Works when we only need to know that there is a class with a certain name, but do not need to know any details (like in the class definition / header file).

In this case we can tell C++ that there is a class with that name, and that it will be defined elsewhere.

Notion: class classname semicolon. Example: class Timer;

Example:

```
// this is Bomb.h

class Timer;

class Bomb
{
private:
  Timer *timer;
}

// this is Timer.h

class Bomb;

class Timer
{
private:
  Bomb *bomb[8];
}
```

Classes vs. POD

POD = plain old data

Classes that have only public attributes are POD.

By default (with no visibility given) members are public.

struct and class are the same in C++.

Convention: we use struct for POD and class for everything else.

Example:

```
struct LocationAsPod {
  int x;
  int y;
};

class LocationAsClass {
private:
  int x;
  int y;
public:
  int getX();
  void setX(newX:int);
  int getY();
  void setY(newY:int);
};
```

Object handles

For the following examples we will assume a simple Location class:

```
class Location {
private:
  int x;
  int y;
public:
  int getX();
  void setX(newX:int);
  int getY();
  void setY(newY:int);
};
```

Dynamic object handles

In most cases we want to use dynamic object handles.

Creating an object:

```
  new Location(); // or
  new Location;
```

Please note: In this case the parenthensis () are optional.

This would create the object and then immediately loose it.

Assigning that new object to a variable for use:

```
Location *here = new Location();
```

Now the variable "here" holds a handle to the object.

To access member functions / variables, we use the "->" operator:

```
here->setX(10);
here->setY(5);
```

```
cout << here->getX() << " " << here->getY();
```

To pass object handles to other methods, they have to take object handles as parameters.

```
class Hominoid
{
...
public:
  goToLocation(Location *target);
};

...

Hominoid *bob = new Hominoid();
Location *here = new Location();
...
bob->goToLocation(here);
...
```

To return object handles, use the pointer reference data type:

```
class Hominoid
{
...
public:
  Location *whereAreYou();
};
...
Hominoid *bob = new Hominoid();
...
Location *bobsLocation = bob->whereAreYou();
...
```

Practice:

Define a class "Computer" that contains an attribute "cpu" of type "Cpu". Show the class definition with the attribute cpu and its getter and setter method. Show the implementation for the getCpu() and setCpu() functions.

```
class Computer
{
private:
  Cpu *cpu;
public:
  void setCpu(Cpu *);
  Cpu *getCpu();
};
...
void Computer::setCpu(Cpu *newCpu) {cpu = newCpu;}
Cpu *Computer::getCpu() { return cpu; }
```

About the position of the *:

In C++, the * can either be with the classname OR with the attribute / function name:

```
Location *pos; // is the same as
Location* pos;  // is the same as
Location * pos; // is the same as
Location*pos;
```

```
...
Location *whereAreYou(); // is the same as
Location* whereAreYou();
```

So logically it would make more sense to put the * to the class name, since we're defining a reference to an object and not to the variable.

The only case where this is not equivalent is when we declare multiple variables / attributes in one line.

```
Location* y, z;
```

Here, we would assume this to be equivalent to:

```
Location* y;  // THIS IS NOT THE CASE
Location* z;  // THIS IS NOT THE CASE
```

Unfortunately it is equivalent to:

```
Location* y;
Location z;
```

Therefore you have two choices:

- Either always put the * with the attribute / variable name
- Or never declare multiple attributes / variables in one line.

Since I'm lazy, I chose the first solution. You may pick whichever one you like, but be consistent within the same project!

Disposing of objects:

Since we created new dynamic object, we have to dispose of them properly:

```
Location *loc = new Location();
...
delete loc;
```

We use "delete" as learned in an earlier class with pointers.

Practice:

Go back to the Cpu / Computer example. Create one instance of each. Call your setter method. Dispose of both objects properly.

Warning: In OO it is sometimes where complicated to find out WHERE or WHEN to delete objects.

Two problems:

- Forget to delete
- Delete multiple times

Forget to delete: Sometimes we can not delete an object immediately, but someone else has to do it. In this case, we may forget to delete.

Example:

```
class Hominoid
{
...
public:
  Location* dreamPlace();
...
}
...
Location *Hominoid::dreamPlace()
{
  Location *l = new Location();
  l->setX(1);
  l->setY(1);
  return l;
}

...
Hominoid *bob;
...

Location loc = bob->dreamPlace();
// Do something with loc

// now loc should be deleted.
delete loc; loc = null;
```

But why can't we always delete?

Because then we may run into the next problem, deleting to many times / in the wrong place. Example:

```
class Hominoid
{
private:
  Location *l;
...
public:
  Location *getLocation();
...
}
...
Location *Hominoid::getLocation() {
  return l;
}
...
Hominoid *bob;
...
Location *loc = bob->getLocation();
// do something with loc;

// loc should not be deleted, bob still needs it!
```

Unlike most other languages (ObjC, Java, ...) C++ does not have a good solution for this problem. The only solution is very very very very careful programming and good documentation.

Static object handles

So the whole deleting gives us headaches. Can't we just use static data members instead? Aren't they just much easier?

They are easier, alright, because they get automatically allocated and deleted!

Example:

```
{
  Location l; // l gets automatically allocated here
  l.setX(5);
  l.setY(7);
  ...
}  // l gets automatically deleted here when it runs out of scope
```

Unfortunately, whenever we use l it gets copied. Here's a larger example:

```
...
void advance(Location l) // This would usually be somewhere in some class
{
  l.setX(l.getX()+1);
}
...
Location z;
...
z.setX(1);
z.setY(1);
advance(z);
cout << z.getX(); // Prints 1
```

We can get around the problem here by using pass-by-reference. However, this only works in some cases. Imagine a setter method:

```
void Hominoid::setLocation(Location &l)
{
  Mylocation = l; // would still copy
}
```

Also please note: Static object references can not be used with incomplete class definitions! You have to have either a full class definition or a dynamic object reference (pointer).

Conclusion:

- Static references are easier
- Unfortunately they may copy objects
- Use with caution!
- You need to practice with new and delete anyways, so you may just use dynamic handles everywhere.

A good rule:

Use static object references when you

- Use that object in only one function
- Never pass that object as a parameter.

Example: Main function

```
int main() {
  Game g;
  g.play();
  return 0;
}
```

We have already used some classes with static references:

- string is a class.

- ifstream is a class.
- ofstream is a class.

Static Member Variables

Rember class attributes?

OO: class attributes <-> C++: static member variables

OO: object instance attributes <-> C++: non-static member variables.

- Every object instance has its own set of object instance attributes
- All object instances share the same class attributes.

In C++ we use the "static" keyword.

Example:

```
class MakesNoSense {
private:
  static int counter;
  int someVar;
public:
  void doSomething();
  int getCounter();
}
...
void MakesNoSense::doSomething(int a)
{
  counter = a;
  someVar = a;
}
...

MakesNoSense *a = new MakesNoSense(), *b = new MakesNoSense();
a->doSomething(1);
b->doSomething(2);

cout << a->getCounter();  // Discussion: What will this print?

delete a; delete b;
```

Static Member Functions

As seen in the discussion about object orientation there can also be static member functions.

OO: Class operations <-> C++: static member functions

They also uses the keyword "static".

Usage:

- When there is "only one"
- When the function does not depend on any non-static class atttributes
- For constructing pre-defined object.

Example:

```
class Color {
private:
  int red, green, blue;
public:
  static Color* createTTURed();
};

Color* Color::createTTURed()
{
  Color *c = new Color;
  c->red = 204;
  c->green = 0;
  c->blue = 0;
  return c;
}
```

Notes for static member variables:

- Since they are part of the class they are allowed to use private member data.
- However, they do not have an object with them. So they can only access static member variables directly, all others only if they have an object.

To use a static member variable use them as if they where in a namespace. From within the class this is unnecessary.

Example:

```
Color* c = Color::createTTURed();
```

Practice:

define a class "Location" that has the two member variables posX and posY. provide a static member function called "getOrigin()" that creates a new location with posX=0 and posY=0.

Show the class definition and the implementation for getOrigin(). Show an example of how this could be called.

```
class Location {
private:
  int posX,posY;
public:
  static Location* getOrigin();
};

Location* Location::getOrigin() {
  Location *l = new Location();
  l->posX = 0;
  l->posY = 0;
  return l;
}
...
Location *o = Location::getOrigin();
```

Intermission: So how do I call a method again?

Assume the follwing class definiton:

```
class Bla {
```

```
public:
  static int doSomething(bool really);
  string getName();
}
```

To call the static function, we have 3 options:

```
// Preferred way:
int i = Bla::doSomething(true);
// This works also, but I do not like it.
int i = b->doSomething(true);  // Assuming b is of type Bla*
int i = c.doSomething(true);   // Assuming c is of type Bla
```

To call the non-static function we have two options:

```
string s = b->getName();       // Assuming b is of type Bla*
string s = c.getName();        // Assuming b is of type Bla
```

Inline Member Functions

Getters and setters are small, but implementing them takes an extra step. This can be very annoying.

C++ allows implementing functions where the actual definition is. This is called an "inline" implementation.

There is also another kind of "inline" implementation with the keyword "inline". Do not confuse these two. (they actually do the same thing).

Example:

```
class Location
{
private:
  int x,y;
public:
  int getX() { return x; }
  int getY()
  {
    return y;
  }
  void setX(int newX)
  {
    x = newX;
  }
  void setY(int newY) { y = newY; }
};
```

Even though inline functions are very often written in one line the formatting has absolutely nothing to do with the name. It is just a more clear way to do it, especially with one-line implementations.

When calling an inline function, instead of jumping into the function the compiler actually inserts the code wherever the function is called. This makes the code faster, but larger. For short functions this is advisable, for long functions this is a bad idea.

Why not always use inline?

- Inline makes the executable larger.
- One some compilers, inline code can not have conditional statements (if, case) or loops (for, while, do..while). If you use such statements in your inline functions your code will be less portable.

Guidelines for this class:

- It is no problem if you never use inline functions
- You should use them only for short functions that contain at most 3 statements, and no conditional / loop statements.
- I recommoned to use them on simple getters and setters.

Practice:

Complete this class definition by providing inline getters and setters:

```
class Color
{
private:
  int red, green, blue;
public:
  int getRed() { return red; }
  void setRed(int r) { red = r; }
  int getGreen() { return green; }
  void setGreen(int r) { green = r; }
  int getBlue() { return blue; }
  void setBlue(int b) { blue = b; }

  void setToBlack() { red = 0; green = 0; blue = 0; }
};
```

this

When talking about messages, we used the terms "sender" and "target". When we're actually sending messages, we use the notion target->messageName(parameters). This does not include the sender of a message. But what if we want the system to know who the sender is?

Solution: We could add the sender as an additional parameter. Example:

```
class ListOfCars {
...
  void addCar(Car *c);
}
```

Now if a car want's to add itself to the list, it can do so. All it needs is to have an object reference to itself.

This object reference is called "this". ("self" in most other OO languages)

- This is only valid in non-static methods (because it needs an object)
- It will always hold the reference of (point to) the current object.

Example:

```
void Car::registerMe()
{
  list->addCar(this);
}
```

The use of "this" in setters.

Assume the following class:

```
class Bla
{
private:
  int attr;
public:
  void setAttr(int);
  // ...
}

//  ...


void Bla::setAttr(int attr)
{
  this->attr = attr;
}
```

Chapter 12. Constructors and Destructors

Constructors and destructors are special forms of member functions.

Important

Constructor	A constructor is used to initialize an object
Destructor	A destructor is used to finalize an object (= clean it up)

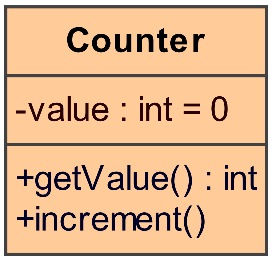
Constructors

When an object instance is created, all member variables have unitizialized values. To make proper use of an object these member variables should be given some values.

Since you do not want to set all attributes manually, we use "constructors".

Imagine the following class:

Figure 12.1. A Counter class



This class is very useful to count. It is (purposely) written so that the counter can not be set from the outside. Unfortunately C++ does not allow us to initialize variables with values. So how do we do it?

Answer: Constructors

Default Constructor

We define and implement a so called default constructor. A default constructor:

- Is called by default when an object instance is created.
- Needs no parameters

To define a default constructor we define a function with

- the exact same name as the class
- no parameters
- no return type (not even void)

In most cases the default constructor will be public. However there are cases where you want it private (more about that later)

Usually consturctors are defined as the first methods. Usually default constructors are the first of the constructors

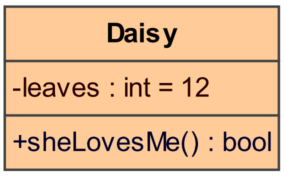
Example:

```
class Counter
{
private:
  int value;
public:
  Counter();
  // ...
};
```

Practice:

Define the following class, add a default constructor.

Figure 12.2. A daisy



Implementing a default constructor is again the same as implementing that method that has the same name as your class.

What should you do in a constructor?

- All member variables should be initialized
- If a default value is given in the model, use that
- If not, make up good starting values (0 for numbers, null for pointers).
- If there is something else your object needs to do before it starts, this is the place to do it.

Example:

```
Counter::Counter()
{
  value = 0;
}
```

Practice:

Implement the default constructor for the Daisy class defined earlier.

Calling a default constructor:

A default construcotr is called everytime the object is initialized and we don't do anything special. Examples:

```
Counter *c;
c = new Counter;     // calls the constructor
Counter *c2 = new Counter();  // calls the constructor
Counter c3; // calls the constructor
Counter c4(); // calls the constructor
```

The parentensis () are optionional in this case. I prefer to use them to show visually that a constructor is called. You may do as you like.

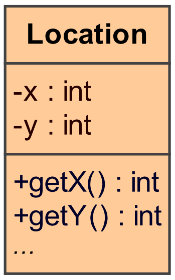
Practice: Create a (dynamic reference to a) Daisy object and call its default constructor

Constructors with parameters

By this time at least one of you has heopfully already asked about it: What happens if we want to initialize variables, but we only know the values at construction? What if we need parameters to initialize the values to something that makes sense?

Example:

Figure 12.3. Location Class



What should we use as starting values for x and y? 0 and 0 may be good values for a default constructor, but what we really want is a parameterized constructor where we can pass in the values.

Fortunately defining and implementing parameterized constructors is the same as default constructors, only that now we have parameters:

```
class Location
{
private:
  int x,y;
public:
  Location(int, int);
  // ...
};

...

Location::Location(int nx, int ny)
{
  x = nx;
  y = ny;
}

//alternate:
Location::Location(int x, int y)
{
  this->x = x;
  this->y = y;
}
```

Practice: Change "Daisy" to define an implement a constructor with one parameter (for number of leaves).

Just as with overloading (not overriding) we may have multiple constructors with different parameter lists. In most cases there will be a default constructor and one that takes parameters.

```
class Location
{
private:
  int x,y;
public:
  Location() { x = 0; y = 0; }
  Location(int nx, int ny) { x = nx; y = ny; }
  Location(int d) { x = d; y = d; }
  // ...
};

alternate, but not the SAME!!!!!!
  Location(int nx=0;int ny=0) { x = nx; y = ny; }
```

Calling a parameterized constructor:

We call parameterized constructors during object creation. Example:

```
Location *la = new Location(1,2);  // for dynamic objects
Location lb(4,5);                  // for static references
```

Practice: Create another new (dynamic) Daisy object and call its parameterized constructor.

Please note:

If we define no constructors, C++ will automatically provide an empty a default constructor.

If we define at least one constructor (default or parameterized) C++ will complain if we want to use the default constructor. This sometimes happen accidentaly.

Example:

```
Location l; // calls the default constructor
            // even though its not obvious
```

The copy constructor

This part is merely for completenes, but I will not ask you for that in any test. Assume Location object as given earlier, and all static object references.

```
Location la(1,2);
Location lb;
lb = la;
```

What happens here?

In the first line an object is created with the parameterized constructor.

In the secont line an object is created with the default constructor.

In the third line the object previousely referenced by lb is thrown away. A new object is created using the "copy constructor" to copy the variable values from la. This new object is then assigned to la.

C++ provides this copy constructor for us. If we want to override it we need to decare a constructor that takes one parameter whos type is a reference to the class type.

Example:

```
class Location
{
...
  Location(const Location &l); // const is optional but recommended
...
};
```

Destructors

So far we have talked about

- using objects
- creating objects

But we also need to know how to:

- dispose of objects properly

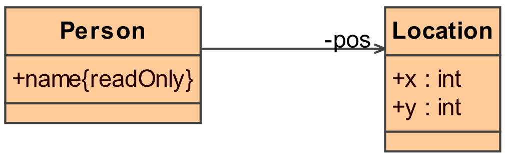
Destructors are used to finalize an object.

The main purpose of a destructor is to clean up after the object, in most cases this means deleting memory.

To find out where to delete, we need to find out who is "responsible" for a certain object. In most cases this will be the creator, but it doesn't have to be.

Assume the following classes:

Figure 12.4. A Person with a Location



In this case we can clearly assume that objects of class "Person" are responsible for their pos object (of class Location).

So whenever a Person is deleted we want their Location to be deleted as well.

Defining a destructor:

A destructor is defines like a method that has no return type (just like the constructor), the name tilde-classname (e.g. ~Person) and no parameters. There are no parameterized destructors.

Example:

```
class Person
{
...
public:
...
  ~Person();
...
}
```

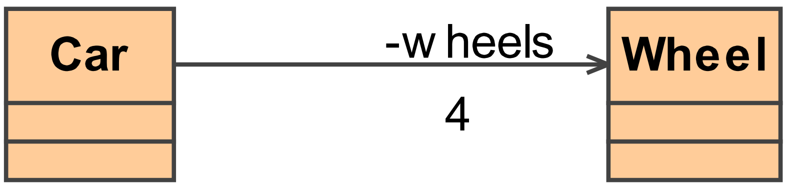
Note on the location of the definition:

- Some people define the destructor after all constructors before other methods.
- Some people define the destructor after the very last method.

You may do either or, put please again: Do not mix within one project!

Practice!

Figure 12.5. Car and Wheels



Example solution

```
class Car
{
private:
  Wheel *wheels[4];
public:
  Car();
  ~Car();
};

// ...

Car::Car()
{
  for (int i=0;i<4;i++)
    wheels[i] = new Wheel();
}
```

Implementation for destructors:

Just like any other method. Example:

```
Person::~Person()
{
  delete pos;
}
```

A destructor should

- Delete all objects this object was responsible for
- Clean up every thing else needed. If the object resembles hardware is should close the connection. (Example on fstreams: deleting an fstream closes the file)

Practice: Implement destructor for car.

```
Car::~Car()
{
  for (int i=0;i<4;i++)
    delete wheels[i];
}
```

How to call destructors:

Destructors are automatically called whenever an object is deleted.

- For dynamic object references that means delete ...;
- For static object references that means the variable runs out of scope.

Example:

```
Person *p = new Person("Waldo");
// ...
delete p;   // <-- calls the destructor

{
  Person q("Waldo");
  // ...
}  // <-- call of the destructor
```

Note for eclipse:

When you create classes in eclipse via the New/Class function it creates a default constructor and a destructor for you. It also follows all naming conventions and provides guards for multiple inclusions.

The destructors in eclipse have the "virtual" keyword. For now you may ignore it. It is important when we talk about inheritance.

Implicit destructors:

If you do not provide a destructor C++ will provide one for you. Unfortunately this one won't do much (unless you're using inheritance).

Here is a complete example:

```
// Location.h
#ifndef LOCATION_H_
#define LOCATION_H_

class Location
{
private:
  int x,y;
public:
  Location();
  Location(int x,int y) { this->x = x; this->y = y; }
  ~Location();
  int getX() { return x; }
  int getY() { return y; }
};

#endif /*LOCATION_H_*/

// Person.h
#ifndef PERSON_H_
#define PERSON_H_

#include <string>
```

```
using namespace std;

class Location;

class Person
{
private:
  string name;
  Location *loc;
public:
  Person(string name);
  Person(string name, Location *l);
  ~Person();
  void cheat();
};

#endif /*PERSON_H_*/

// Location.cpp
#include <cstdlib>
#include "Location.h"

Location::Location()
{
  // Randomizer is initialized in main.cpp
  x = rand()%100;
  y = rand()%100;
}

Location::~Location()
{
  // Nothing to do
}

// Person.cpp
#include <iostream>
#include <string>
#include "Person.h"
#include "Location.h"

Person::Person(string name)
{
  this->name = name;
  // No location given? Ok, create a new one
  // Our destructor will make sure it gets deleted.
  this->loc = new Location();
}

Person::Person(string name, Location *l)
{
  this->name = name;
  // Location given? Use it!
  // Please note: In this case we decided that Person
  // is responsible for the location. So it must NOT
  // be deleted on the caller, but here!
  this->loc = l;
}
```

```
Person::~Person()
{
  // We're responsible for loc, so let's delete it!
  delete loc;
}

void Person::cheat()
{
  cout << name << " is at "
       << loc->getX() << ", " << loc->getY()
       << endl;
}

// main.cpp
#include <cstdlib>
#include <ctime>
#include <iostream>
#include "person.h"
#include "location.h"

int main()
{
  // Initialize randomizer
  srand(time(0));

  // We don't know anythign about location
  Person *w = new Person("Waldo");
  w->cheat();
  delete w;

  // Creating a location and immediately passing the
  // responsibility to the Person object
  Person *m = new Person("Max",new Location(12,34));
  m->cheat();
  delete m;

  // Creating a Location
  Location *fiftyfifty = new Location(50,50);
  // Passing it (and the responsibility for it) to
  // Person
  Person *md = new Person("Middleman",fiftyfifty);
  md->cheat();
  // This also deletes the location
  delete md;

  // Typical mistakes! Do not do any of those!

  // the object reference by fiftyfifty just got deleted!
  cout << fiftyfifty->getX() << endl;

  Location *lc = new Location(11,22);
  Person *p1 = new Person("Person1",lc);
  Person *p2 = new Person("Person2",lc);
  // Now both p1 and p2 are responsible for lc. So both
  // will try to delete it! Bad idea....

  // deletes p1 and lc
```

```
    delete p1;

    // will crash
    p2->cheat();

    return 0;
  }
```

Chapter 13. Inheritance

In C++ a class can derive from zero or more base classes.

Unlike Other languages (ObjC, Java), there is no common base class. A class that is not derived is really not derived.

derived class (subclass)	a class with at least one base class
direct base classes (direct superclasses)	are the classes are the classes that are immediate base classes
indirect base classes (indirect superclasses)	are the classes that are superclasses of the direct base classes (and their superclasses, etc.)

Remember: A base class inherits all the functions and variables of all of its superclasses!

Inheritance in C++:

```
class Max { ... };
class SomeClass { ... };
class LittleMax : public Max { ... };
class MultiDerived : public Max, public SomeClass { ... };
class LinearDerived : public LittleMax { ... };
```

To declare inheritance:

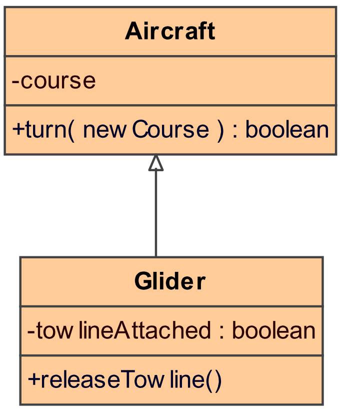
between the class name and the opening brace, insert a colon, the keyword public, and the name of the base class.

There are also other types of inheritance (other than "public") which we will talk about later. Other OO languages (ObjC, Java) only have public inheritance

If you have multiple direct base classes, join them with a comma.

Practice:

Define these two classes with inheritance. You may omit all contents of the actual class (the attribtues and methods), I am only interested in the definition line (as in the example).



Virtual

Remember when we talked about polymorhpism and function overriding? Unfortunately, C++ is a grown languages, and therefore does not allow function overriding by default. We need to do two things:

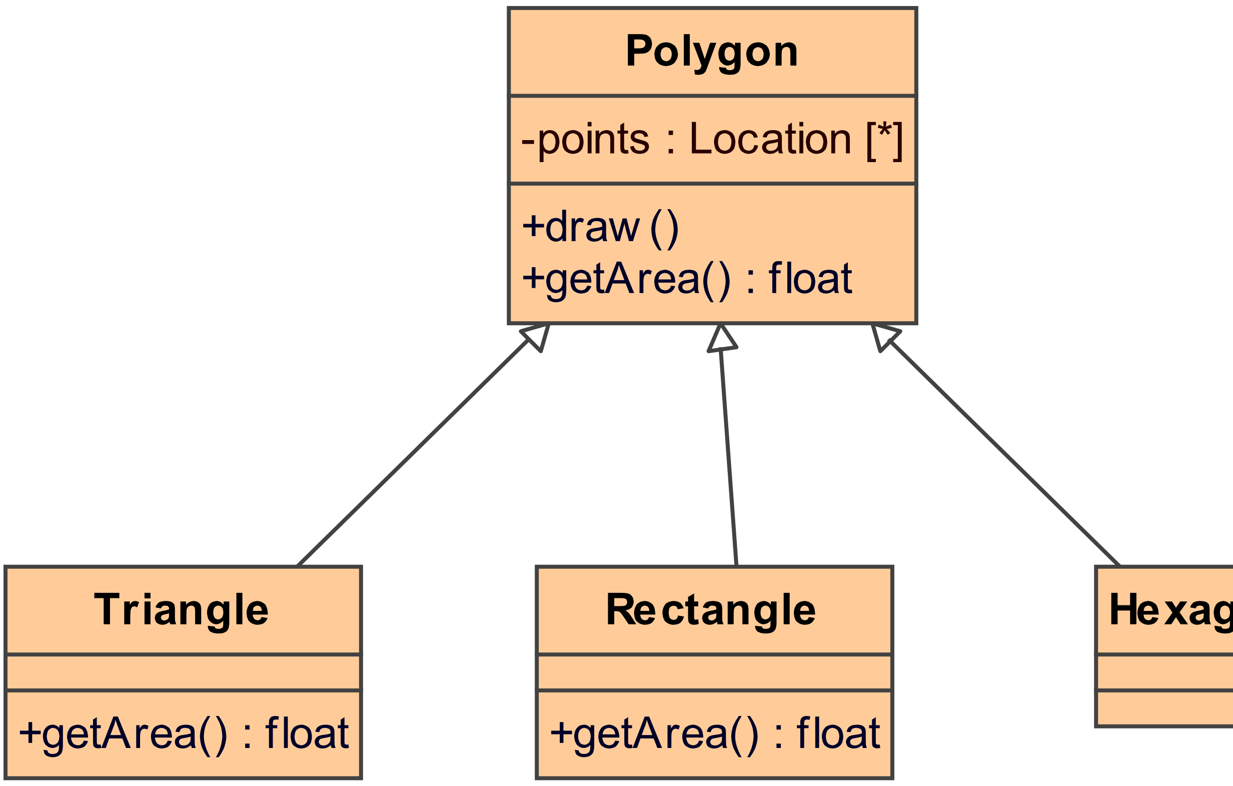
- Use dynamic object references. If we use static references, classes get "sliced". This is a very strange concept. If your interested, read page 156 of C++ in a Nutshell. Just remember: To use function overriding, and to make proper use of inheritance, always use dynamic references.
- Functions to be overridden or that are overridden are declared with the "virtual" specifier (only in the declaration, not the implementation).

In other languages (ObjC, Java), virtual is the default.

Therefore, for this class:

- Specify ALL functions (including destructor, excluding constructor) as virtual in all classes that are subclasses or superclasses!

Example 13.1. Example: Implementation of polygon



Here are the class definitions:

```
class Polygon {
private:
  vector<Location *> points;
public:
  Polygon(vector<Location *> points);
  virtual void draw();
  virtual float getArea();
};

class Triangle : public Polygon {
public:
  Triangle(Location *p1, Location *p2, Location *p3);
  virtual float getArea();
};

class Rectangle : public Polygon {
public:
  Rectangle(Location *topLeft, Location *bottomRight);
  virtual float getArea();
};

class Hexagon : public Polygon {
public:
  Hexagon(Location *center, float radius);
};
```

And some code that uses these classes:

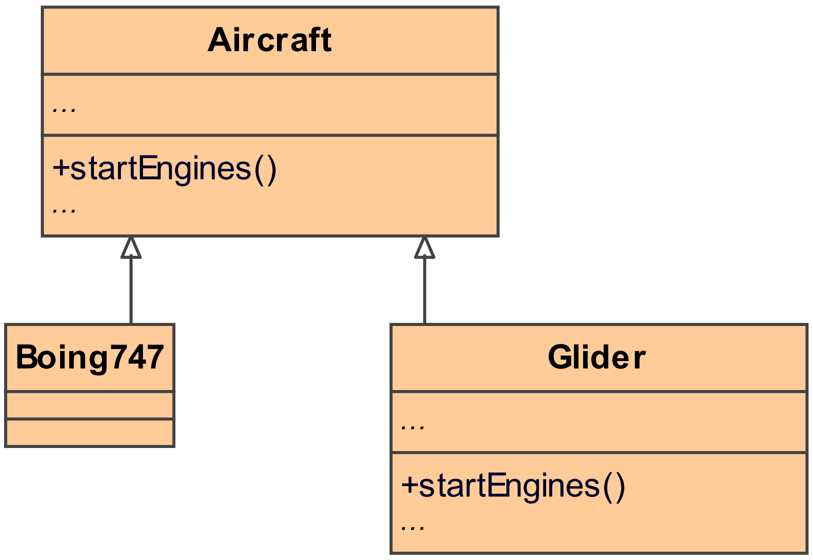
```
// assume v1 is a vector<Location *> with some useful values.
// assume p1,p2, .. are Location* with useful values;

Polygon *pg1 = new Polygon(v1);
Triange *t1 = new Triangle(p1,p2,p3);
Rectangle *r1 = new Rectangle(p4,p5);

// Using oo:
Polygon *pg2 = new Triangle(p6,p7,p8);
Polygon *pg3 = new Rectangle(p9,p10);
Polygon *pg4 = new Hexagon(p11,1.0);

cout << pg1->getArea() << endl; // calls getArea() from Polygon
cout << pg2->getArea() << endl; // calls getArea() from Triangle
cout << pg3->getArea() << endl; // calls getArea() from Rectangle
cout << pg4->getArea() << endl; // calls getArea() from Polygon
```

Practice:



Define these three classes. Show the class definitions and the definitions for the methods given here (all other methods / attributes / constructors / etc. may be ommited)

```
class Aircraft {
public:
  virtual void startEngines();
};

class Boing747 : public Aircraft {
};

class Glider : public Aircraft {
public:
  virtual void startEngines();
};
```

Base constructors and protected

Lets go back to the geometry example:

Remember Polygon and Rectangle:

```
class Polygon {
private:
  vector<Location *> points;
public:
  Polygon(vector<Location *> points);
  virtual void draw();
  virtual float getArea();
};

class Rectangle : public Polygon {
public:
  Rectangle(Location *topLeft, Location *bottomRight);
  virtual float getArea();
};
```

With the knowledge we have so far we would implement the polygon constructor as follows:

```
Polygon::Polygon(vector<Location *> points)
{
  this->points = points;
}
```

And something like this for the rectangle:

```
Rectangle::Rectangle(Location *topLeft, Location *bottomRight)
{
  points.push_back(topLeft);
  points.push_back(new Location(topLeft->getX(),bottomRight->getY()));
  points.push_back(bottomRight);
  points.push_back(new Location(bottomRight->getX(),topLeft->getY()));
}
```

Unfortunately C++ gives us strange compiler errors:

- `std::vector<Location*, std::allocator<Location*> > Polygon::points' is private`
- `error: no matching function for call to `Polygon::Polygon()`

What is happening here?

Answer (to the first one): points is private. Everything that is private can only be used in this particular class, and none of its superclasses. There are two options:

- Add getters / setters and use those instead
- change the visibility from "private" to "protected"

protected	protected methods and variables are available to all members of this class and all of its subclasses.

Answer (to the second one): Every constructor implicity calls the constructor of its superclass(es). If we don't tell it which constructor to use it tries to call the default constructor.

Unfortunately it calls the superconstructor BEFORE we can do anything.

We have therefore two choices:

- Call a specific superconstructor
- add a default superconstructor

In this case I decided on the default superconstructor. Here is the complete new definition:

```
class Polygon {
private:
protected:
  vector<Location *> points;
  Polygon() {};
public:
  Polygon(vector<Location *> points);
  virtual void draw();
  virtual float getArea();
};
```

- We made "points" protected so that subclasses can use it directly
- We added a default constructor that doesn't do anything
- The default constructor is protected. The subclasses should be able to call it, but no one from the outside!

So the notes for inheritance in C++ are:

- make all functions virtual
- maybe add a default constructor
- change some items from "private" to "protected" to give subclasses access.

Calling base destructors

This is easy. We don't need to. We just need to make sure all destructors are virtual. Then they will automatically be called in reverse order.

So what is the order? Assume the following classes (with all virtual destructors:

```
class Base { .. } ;
class SubClass : public Base { ... };
class SubSubClass : public SubClass { ... };
```

Now when we call

```
Base *b = new SubSubClass();
```

It will execute:

1. The constructor of Base
2. The constructor of SubClass
3. The constructor of SubSubClass

And when we clean up:

```
delete b;
```

1. The destructor of SubSubClass
2. The destructor of SubClass
3. The destructor of Base

The point here is: Always make destructors "virtual" and you have no problem.

Calling Specific Base constructors

Sometimes when you write a constructor, you would like to pass some of the parameters to a base constructor.

Example:

```
class Person {
private:
  Date *birthday;
public:
  Person(Date *bd);
};

class Student : public Student {
private:
  School *attends;
public:
  Student(Date *birth, School *sch);
};
```

In this case, the "bd" parameter for the school constructor could just be passed on to the parent constructor. And we can do that by calling the parent constructor:

```
Student::Student(Date *birth, School *sch) : Person(birth) {
  this->school = sch;
}
```

Pros and Cons:

- Works only if we can put the parameters for the superconstructor within one expression
- You don't have to provide a default constructor in your superclass
- You can keep your attribute private

Note: If you're in multiple inheritance, you can call multiple superconstructors by separating them with a comma. E.g. : Person(birth), OtherConstructor(params).

Practice: Assume these two given class definitions. Implement both constructors, with one calling its superconstructor.

```
class Computer
{
private:
  int cpuSpeed;
public:
  Computer(int cpuSpeed);
};

class Laptop : public Computer
{
private:
  int batteryLife;
public:
  Laptop(int cpuSpeed, int batteryLife);
};

Computer::Computer(int cpuSpeed)
{
  this->cpuSpeed = cpuSpeed;
}

Laptop::Laptop(int cpuSpeed, int bl) : Computer(cpuSpeed)
{
  batteryLife = bl;
}
```

Pure Virtual / Abstract

We can now define classes with inheritance, call super constructors, and super destructors. By now we know almost everything needed to implement all our class designs. The only thing left is "abstract" methods.

pure virtual function	C++ speak for "abstract method". Function, since this is what C++ calls methods in objects. Virtual, because it is to be overridden. And "pure" since it has to be overriden.

To denote pure virtual functions make then virtual and add a = 0 (equals zero) between the argument list and the semicolon. Example:

```
  virtual void someFunction(int param, float moreParam) = 0;
```

This is one of the rare cases where you actually have to use = 0 and NOT an expression that evaluates to 0 (e.g. = 1-1). However, whitespace doesn't matter, so =0 would also work.

Abstract class. Remember from the OO Section:

Abstract class	Abstract classes contain (or inherit) at least one abstract method. Abstract classes can not be instantiated.

In C++ there is no explicit notion for abstract classes. (But don't forget it when you model!)

So here's a complete example of an abstract class:

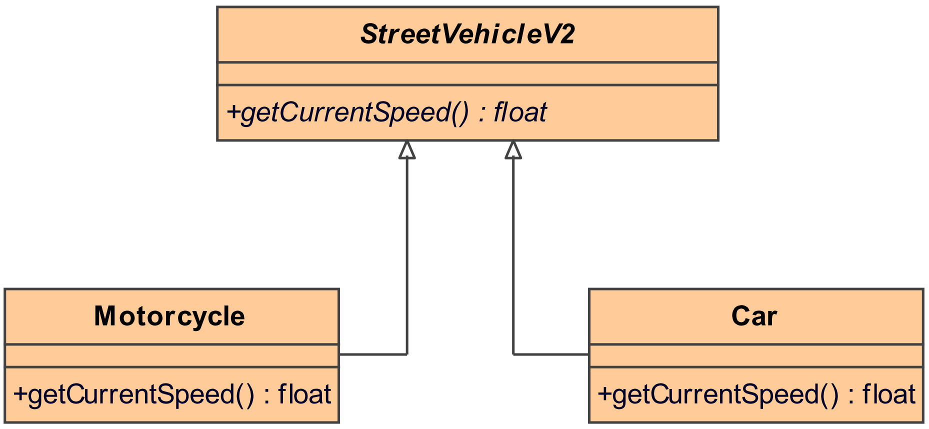
```
class FinancialManager {
public:
  virtual bool shouldIInvestIn(string what) =0;
};
```

And an example of overriding:

```
class BigSpender : public FinancialManager {
public:
  virtual bool shouldIInvestIn(string what) { return true; }
};
```

Practice:

Give the definition (not the implementation) of these 3 classes:


StreetVehicleV2
+getCurrentSpeed() : float
Motorcycle
+getCurrentSpeed() : float
Car
+getCurrentSpeed() : float

```
class StreetVehicleV2 {
public:
  virtual float gCS() = 0;
};

class Motorcycle: public StreetVehicleV2 {
public:
  virtual float gCS();
};

// somewhere else:
float Motorcycle::gCS() {
// ...
}
```

Something completely different: vector

You have seen it, and you may want to use it, so here is a complete short description of the "vector" datatype.

What is vector? vector is the answer to all the problems with arrays.

- It is in the header <vector>
- In the namespace std

To declare a vector, use the typename vector, and then the contained datatype in pointy brackets <>. Example:

```
vector<int> bla;    // as opposed to int bla[];
vector<Location *> loc;
vector<string> names;
vector<boolean> isItTrue;
vector<vector<int>> twoDVector;
```

To add elements to a vector, use its method push_back. push_back takes exactly one argument, which must be of the same type the vector enumerates. Examples:

```
bla.push_back(1);
loc.push_back(new Location(1,2));
names.push_back("Test");
isItTrue.push_back(true);
twoDVector.push_back(bla); // Probably. Need to test this!
```

Another important function is "size". Size takes no parameters and returns the number of elements. Example:

```
vector<int> x;
x.push_back(23);
x.push_back(42);
cout << x.size() << endl; // prints 2
```

To access elements you can use the [] operator just like with arrays. Just like arrays the first element is at index 0 and the last element is at .size() - 1. Do not access elements out of range! Example:

```
cout << x[0] << endl; // prints 23
cout << x[1] << endl; // prints 42
```

You may also write to elements, but only to existing ones!

```
x[1] = 7; // valid
x[2] = 123; // Will crash. Use .push_back to add elements!
```

vector has many more fun features. Read C++ in a Nutshell, pg. 722 - 726 if you are interested.

More complete example: The Coin Toss Game again.

```
// Coin.h

#ifndef COIN_H_
#define COIN_H_

#include <iostream>

enum Side { HEADS, TAILS };
std::ostream& operator<< ( std::ostream &o, Side s);

#endif /*COIN_H_*/
```

```
// ComputerPlayer.h

#ifndef COMPUTERPLAYER_H_
#define COMPUTERPLAYER_H_

#include "Player.h"

class ComputerPlayer : public Player
{
private:
 Side bet;
public:
 virtual void makeBet();
 virtual Side getBet() { return bet; }
 ComputerPlayer(std::string name);
};
```

```
#endif /*COMPUTERPLAYER_H_*/
```

```
// HumanPlayer.h

#ifndef HUMANPLAYER_H_
#define HUMANPLAYER_H_

#include "Player.h"

class HumanPlayer : public Player
{
private:
  Side bet;
public:
 HumanPlayer();
 HumanPlayer(std::string name);
 virtual void makeBet();
 virtual Side getBet() { return bet; }
};

#endif /*HUMANPLAYER_H_*/
```

```
// Player.h

#ifndef PLAYER_H_
#define PLAYER_H_

#include "Coin.h"
#include <string>

class Player
{
protected:
 std::string name;
 Player() {}
public:
 Player(std::string name) { this->name = name; }
 virtual ~Player() {}
 virtual void makeBet()=0;
 virtual Side getBet()=0;
 std::string getName() { return name; }
};

#endif /*PLAYER_H_*/
```

```
// TossGame.h

#ifndef TOSSGAME_H_
#define TOSSGAME_H_

#include <vector>

class Player;

class TossGame
{
private:
 std::vector<Player *> players;
public:
 TossGame();
 void play();
 virtual ~TossGame();
};

#endif /*TOSSGAME_H_*/
```

```
// Coin.cpp

#include "Coin.h"

std::ostream& operator<< ( std::ostream &o, Side s)
{
 if (s == HEADS)
  o << "heads";
 else
  o << "tails";
 return o;
}
```

```
// ComputerPlayer.cpp

#include "ComputerPlayer.h"
#include <cstdlib>

void ComputerPlayer::makeBet()
{
 if (rand()%2==0)
  bet = HEADS;
 else
  bet = TAILS;
}

ComputerPlayer::ComputerPlayer(std::string s) : Player(s)
{
 bet = HEADS;
}
```

```
// HumanPlayer.cpp

#include "HumanPlayer.h"
#include <iostream>
using namespace std;

void HumanPlayer::makeBet()
{
 char c;
 cout << "Would you like to bet on heads (h) or tails (t) ? ";
 cin >> c;
 if (c == 'h') bet = HEADS;
 else if (c == 't') bet = TAILS;
 else {
  cout << "I don't recognize that. I'll just assume tails." << endl;
  bet =TAILS;
 }
}

HumanPlayer::HumanPlayer(string s) : Player(s)
```

```
{
 bet = TAILS;
}

HumanPlayer::HumanPlayer()
{
 cout << "What is your name? ";
 cin >> this->name;
}

// main.cpp

#include "TossGame.h"

int main()
{
 TossGame g;
 g.play();
 return 0;
}

// TossGame.cpp

#include "TossGame.h"
#include "HumanPlayer.h"
#include "ComputerPlayer.h"
#include "NetworkPlayer.h"
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

TossGame::TossGame()
{
 srand(time(0));
 //char x;
 //unsigned int number;

 /*
 cout << "How many players? ";
 cin >> number;

 for (unsigned i=0;i<number;i++) {
  cout << "What type of player?";
  cin >>
 }
 */

 players.push_back(new HumanPlayer());
 players.push_back(new ComputerPlayer("Comp"));
 players.push_back(new ComputerPlayer("AnotherComp"));
 //players.push_back(new NetworkPlayer());
}

TossGame::~TossGame()
{
 for (unsigned int i=0;i<players.size();i++)
```

```
  delete players[i];
}

void TossGame::play()
{
 for (unsigned int i=0;i<players.size();i++) {
  cout <<  players[i]->getName() << " makes a bet..." << endl;
  players[i]->makeBet();
  cout << players[i]->getName() << " bets on " << players[i]->getBet() << endl;
 }

 cout << "Now I flip the coin..." << endl;
 Side landed = TAILS;
 if (rand()%2==0) landed = HEADS;
 cout << "It landed on " << landed << endl;

 for (unsigned int i=0;i<players.size();i++) {
  cout << players[i]->getName() << " has ";
  if (players[i]->getBet()==landed)
   cout << "won!";
  else
   cout << "lost!";
  cout << endl;
 }
}
```

And the Same without OO:

```
#include <iostream>
#include <ctime>
#include <cstdio>
#include <vector>
using namespace std;

enum PlayerType { HUMAN, COMPUTER, NETWORK };
enum Side { HEADS, TAILS };

std::ostream& operator<< ( std::ostream &o, Side s)
{
 if (s == HEADS)
  o << "heads";
 else
  o << "tails";
 return o;
}

struct Player {
 PlayerType playerType;
 string name;
 Side bet;
};

vector<Player *> players;

void initializeGame()
{
 srand(time(0));
 Player *m = new Player;
 m->playerType = HUMAN;
 m->name = "Max";
 players.push_back(m);

 Player *c = new Player;
 c->playerType = COMPUTER;
 c->name = "Comp";
 players.push_back(c);

 Player *n = new Player;
 n->playerType = NETWORK;
 n->name = "John Doe";
 players.push_back(n);

}

Side betHuman()
{
 Side bet;
 char c;
 cout << "Would you like to bet on heads (h) or tails (t) ? ";
 cin >> c;
 if (c == 'h') bet = HEADS;
 else if (c == 't') bet = TAILS;
 else {
  cout << "I don't recognize that. I'll just assume tails." << endl;
  bet =TAILS;
 }
 return bet;
}

Side betComputer()
{
 Side bet;
 if (rand()%2==0)
  bet = HEADS;
 else
  bet = TAILS;
 return bet;
}

void playGame()
{
 for (unsigned int i=0;i<players.size();i++) {
  cout <<  players[i]->name << " makes a bet..." << endl;
  if  (players[i]->playerType == HUMAN )
    players[i]->bet = betHuman();
  else {
    players[i]->bet = betComputer();
  }
  cout << players[i]->name << " bets on " << players[i]->bet << endl;
 }

 cout << "Now I flip the coin..." << endl;
 Side landed = TAILS;
 if (rand()%2==0) landed = HEADS;
 cout << "It landed on " << landed << endl;
 for (unsigned int i=0;i<players.size();i++) {
  cout << players[i]->name << " has ";
  if (players[i]->bet==landed)
   cout << "won!";
  else
   cout << "lost!";
  cout << endl;
 }
}

void cleanUp()
{
 for (unsigned int i=0;i<players.size();i++)
  delete players[i];
}

int main()
{
 initializeGame();
 playGame();
 cleanUp();
 return 0;
}
```

Multiple Inheritance and virtual inheritance

Unlike other languages, C++ allows multiple inheritance.

Unfortunately multiple inheritance leads to lots of problems: If a variable or method is declared in more than one superclass.

Therefore we wil not do discuss many details. If you're interested, read pages 163 - 166 of C++ in a nutshell.

Here are some guidelines to make multiple inheritance feasable. This is what Java and Delphi do.

A class may inherit from one "regular" superclass.

All other superclass must be pure abstract (interfaces) which means: Only pure virtual functions (no implementations) and no attributes (variables).

Why would you need multiple inheritance? Everytime something is multiple things at one time. E.g. a "FlyingCar" could be a subclass of "Car" and of "Airplane".

Example:

```
class Car {
public:
  virtual void driveTo(Location *l)=0;
};

class Airplane {
public:
  virtual void flyTo(Location *l)=0;
};

class FlyingCar : public Car, public Airplane {
public:
  virtual void flyTo(Location *l);
  virtual void driveTo(Location *l);
};
```

Now suddently getter methods make much more sense! A superclass can define virtual getter and setter methods without actually defining the attribute!

Example:

```
class Airplane {
public:
  virtual int getHeightAboveGround()=0;
  virtual void setHeightAboveGround(int h)=0;
};
```

//Practice: Define the following classes:

Notes on virtual inheritance

Chapter 14. Access specifiers

We have already talked about these, but let's define them again.

public

attributes and methods that are public are accessible to every user of the objects.

It is usually a bad idea to make attributes public.

protected

attributes and methods that are protected are accessible to methods of the class and every subclass (direct or inderect) of that class.

private

attributes and methods that are private are only accessible from methods within that class.

friends

Sometime you want to give other classes access to private and protected methods and attributes.

You can do that by giving another class "friend" privilege.

It does not matter where you declare the friend, as long as its in the class definition.

Friends are explicit. They are NOT transitive and do NOT inherit.

Friends are often used for private constructors.

In this case we do not want to make the default constructor public, since it does not initialize all attributes.

But if we know that another class is "good" enough to set all the attributes we can make that class a friend.

```
class A {
  friend class B;
private:
  int attr;
  A();
public:
  A(int a) { attr = a; };
  void setAttr(int b) { attr = b; };
};

class B {
private
  A* attr;
public:
  B() {
    attr = new A();
    attr->setAttr(14);
  }
}
```

"Friends" are especially useful in the Factory design pattern. You will learn about that in a software engineering class.

Chapter 15. Templates

As we have seen in the OO chapter, the last concept of object orientation is genericity.

Unfortunately, genericity is a "new" concept.

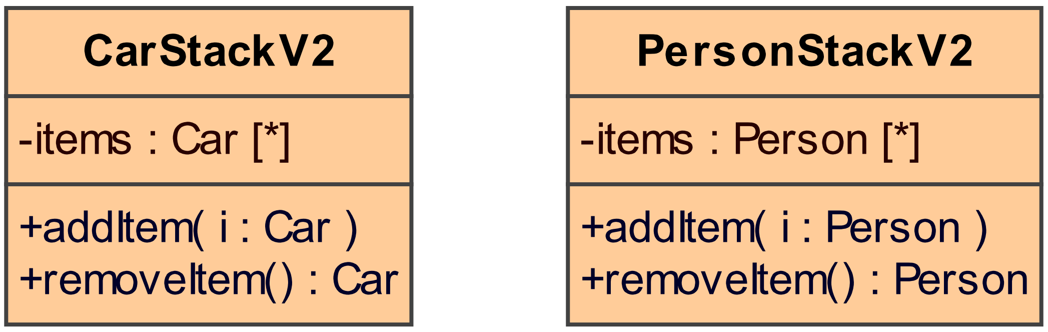
Genericity in C++ is supported through templates. Templates were added in the ANSI C++ standard (1999). They work on most modern compilers.

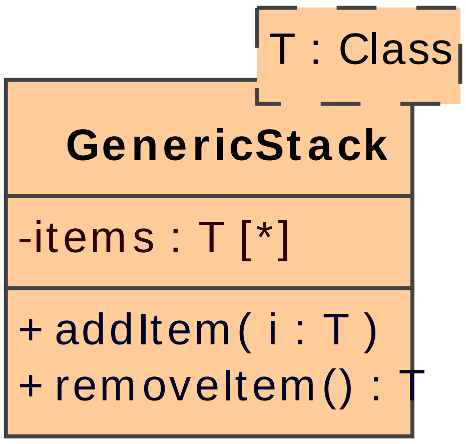
Genericity in Java was added in Java 1.5 (2004) through Generics. This is still pretty new.

So what is genericity again?

Genericity is the concept of writing classes that work with any datatype. The dataype is given whenever an object of that class gets instantiated.

Example: Remeber the CarStack, PersonStack and Generic stack:





In C++, Generics are implemented through class templates.

The most prominent use of class templates is in the standard template library (STL)

Here is an example of a class template:

```
template<class T>
class GenericStack {
private:
  vector<T> items;
public:
  void addItem(T i);
  T removeItem();
};
```

So, to define a template class

- Use the keyword "template"
- In pointy brackets <> define the template parameters. Every template parameter starts with the keyword "class" (or "datatype"), followed by a name of your choice. The most commonly used names are single upper case characters starting with T: T, U, V, ...
- Inside your class, you can use your template parameters just like other data types (int, float, String, ...)
- The actual data type is assigned when you "instantiate" this class.

Practice:

Define a generic "Location" class. This class should store two attribtues (x, y) which are of the same datatype, given during its instantiation. Show: The class definition, the attributes, the getter and setter functions for both.

```
template<class T> class Location {
private:
  T x,y;
public:
  void setX(T newX);  // same for Y
  T getX();           // same for Y
  addSomeThingToX(T addToX) { x = x + addToX; }
};
```

Default parameters: In the template definition, you may use default parameters. An example is the "basic_string" class from the STL. It defaults to string of the "char" type. Example:

```
template<class T = char> basic_string ...
```

Just like default parameters, you may or may not specify these when instantiating. Example:

```
basic_string s;
basic_string<> s3;  // does not work!
basic_string<char> s2; // same as above
basic_string<int> is;  // does not make much sense
basic_string<wchar> ws;   // wchar may not exist on your system
```

Which brings us directly to the next part: Instantiating template classes.

You instantiate a template class by using the class name, adding pointy brackets <> and adding the data types. Example:

```
GenericStack<Car> carStack;
GenericStack<Car *> dynamicCarStack;
GenericStack<Person> personStack;
```

Practice: Define a variable that instantates your "Location" class with the "int" datatype. Define a variable that instantiates your "Location" class with the "float" datatype.

```
Location<int> il;
Location<float> fl;
```

Implementing functions from template classes:

To implement a function from a template class, you have to repeat the template declaration (without default parameters), and add the the same template to the class name. Example:

```
template <class T>
void GenericStack<T>::addItem(T i)
{
  ///
}

template <class T>
T GenericStack<T>::removeItem()
{
  ///
}
```

Practice:

Provide the implementation for the getters and setters from the "Location" class.

```
template <class T>
void Location<T>::setX(T x)
{
  this->x = x;
}
```

Caveat:

Templates are not actually instantiated until they are used. They are instantiated once for every datatype used

```
GenericStack<Car> x;   // instantiates the GenericStack for cars
GenericStack<Car> y;   // resuses that
GenericStack<Person> p; // instantiates the GenericStack for people
```

Therefore, if you implement them in an extra file they are not instantiated.

Unfortunately every compiler handles this problem differently.

In gcc this can be fixed by putting the implementation into the actual header file!

Note: You can use use templates without classes, for more information see C++ In a Nutshell, pages 174 - 210.

Chapter 16. The STL

Once C++ had a good template mechanism, people started implementing data structures using these templates. The most widespread collection of templates came from SGI and HP and was called the STL.

Standard Template Library (STL)	A set of data structures and algorithms using templates. Now part of the C++ standard.

For a complete reference to the STL:

- Read C++ in a Nutshell
- Go to http://www.sgi.com/tech/stl/table_of_contents.html

Containers

A Container is an object that stores other objects (its elements), and that has methods for accessing its elements.

All Containers provide methods to create iterators (see iterators below).

All containers provide the following methods:

unsigned int size()	returns the current size of the container
bool empty()	returns true if the container is empty

Sequences

In most containers that we have seen so far, the order of elements is important. The STL provides several sequence containers.

Most of these containers support the same operations. Then why bother having two implementations for it? Because every container performs differently on different operations!

Example:

We'll talk about the two template types "vector" and "list" (in the <vector> and <list> includes)

A vector is based on an array.

- Getting the nth element of an array is fast -> getting the nth element of a vector is fast
- Adding an element to the end of the list is fast
- Adding an element in the middle requires moving all elements past this one back -> slow
- Adding an element to the beginning requires moving all elements back -> very slow

list is based on a liked list

- Getting the nth element of a linked list requires traversing all elements to that point -> slow
- Adding an element anywhere in the list does not require any movement -> fast (once the element is found)

You will learn more about the implementation issues in the data structures class.

Both vector and list support:

push_back(T x)	add an element to the end of the list
pop_back()	removes the last element
T back()	returns the last element
T front()	returns the first element

Only list supports:

push_front(T x)	add an element to the beginning of the list
pop_front()	removes the first element

Only vector supports:

[] or at()	access element at the given index.

Here are some examples:

```
#include <iostream>
#include <list>   // For list
#include <vector> // For vector

using namespace std; // All STL containers are
                     // in the std namespace.

int main()
{

  vector<int> a;  // Declare a vector that takes int
  a.push_back(3); // add to end of vector
  a.push_back(24);
  a.push_back(42);

  // as long as we still have elements
  while (!a.empty()) {
    // print the last element
    cout << a.back() << " ";
    // and remove it!
    a.pop_back();
  }
  cout << endl;
  // this (above) printed 42 24 3

  list<int> b; // declares a linked list that takes int
  // adds some elements to the end
  b.push_back(3);
  b.push_back(24);
  b.push_back(42);
  // as long as there are elements left
  while (!b.empty()) {
    // print the last element
    cout << b.back() << " ";
    // and remove it
    b.pop_back();
  }
  cout << endl;
  // this (above) printed 42 24 3

  list<int> c; // declares a linked list that takes int
  // adds some elements to the end
  c.push_back(3);
  c.push_back(24);
  c.push_back(42);
  // as long as there are elements left
  while (!c.empty()) {
    // print the first element
    cout << c.front() << " ";
    // and remove it
    c.pop_front();
  }
  cout << endl;
  // this (above) printed 3 24 42

  return 0;
}
```

So now that we see the use of different containers, lets see what we have available:

deque	A deque (double-ended queue) is a sequence container that supports fast insertions and deletions at the beginning and end of the container. Inserting or deleting at any other position is slow, but indexing to any item is fast. The header is <deque>. Pg 470 deque supports: [], at(), front(), back(), push_front(), push_back(), pop_front(), pop_back(), empty(), size(), ...
list	A list is a seuqence container that supports rapid insertion or deletion at any position, but does not support random access. The header is <list>. Pg 559
vector	A vector is a sequence container that is like an arrays, except that it can grow in size as needed. Items can be rapidly added or removed only at the end. At other positions, inserting and deleting items is slower. The header is <vector>. Pg 722

Practice: Implement a short program (the whole program) that

- declares a deque of type float
- add the elements 1.234, 12.34, and 123.4
- prints the first element
- removes the first element
- prints how many elements are in the deque

```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
  deque<float> f;
  f.push_back(1.234);
  f.push_back(12.34);
  f.push_back(123.4);
  cout << f.front() << " ";
  cout << f.at(0) << " ";
  cout << f[0] << endl;
  f.pop_front();
  cout << "There are " << f.size() << " elements left!";
  cout << endl;
  return 0;
}
```

Iterators

Since all the containers have a different implementation, we need a standard way of going through all the items.

These things are called "iterators".

We are already used to iterators, we just didn't know it: For vector and arrays we used integers. We ran theses from 0 to size()-1. Therefore this "int" was an iterator.

Iterators use a lot of operator overloading to behave similar to pointers.

Declaring: To declare an iterator use the subclass "iterator" for your specific data type. Example:

```
vector<int> a; // Your vector a
// ... a lot of a.push_back()

// The actual declaration:
vector<int>::iterator it;
```

There are two standard functions for iterators:

begin()	returns an iterator pointing to the first element
end()	returns an iterator pointing after the last element

Iterators can be advanced with the ++ operator (some can go backwards with --) and compared with the == or != operator. To run over all element, you can therefore use:

```
it = a.begin();
while (it!=a.end()) {
  // ...
  it++;
}
```

Or quicker:

```
for (it = a.begin();it!=a.end();it++) { ... }
```

To get the element an iterator points to, you act as if iterator would be a pointer and use the dereference operator *. Example:

```
cout << *it;
```

Practice:

Assume you have given the following declaration:

```
list<char> l;
l.push_front('l');
l.push_back('a');
l.push_front('b');
```

Write a for loop that uses an iterator to iterate over l and print all the contents of the list.

```
list<char>::iterator li;
for (li=l.begin();li!=l.end();li++)
  cout << *li;
```

Print just the second element (emulate at(1))

```
list<char>::iterator li;
li = l.begin();
li++;  // can not use li = li+1 in this case
cout << *li;
```

All of these containers support insert() and erase(), but we had to introduce iterators first.

iterator erase(iterator p)	erases the item that p points to. erase returns an iterator that points to the item that comes immediately after the deleted item or end().
iterator insert(iterator p, T x)	inserts x immediately before p and returns an iterator that points to the newly inserted element x.

Warning: Iterators may become "invalid" after an insert or a delete operation! You should therefore use the return value if possible!

Example:

```
vector<int> v;
// ...
// same as p.pop_front(), would it exist.
v.erase(v.begin());
```

A more complex example: Delete all occurences of "42" in a list:

```
list<int> l;
// ...
list<int>::iterator i = l.begin();
while (i!=l.end()) {
  if ((*i) == 42)
    i = l.erase(i);
  else
    i++;
}
```

Another example:

```
list<int>::iterator i = l.begin();
// insert the number 0 at the beginning
i = l.insert(i,0);
// make sure i points after the first element
i++;
```

Practice:

Assume this given deque:

```
deque<int> d;
// d gets filles with some values
```

Write a loop that inserts a 42 before every occurence of 0 in d. Two hints:

- iterators to a deque become invalid after insertion. Make sure you use the return value!
- don't write an infinite loop!

```
deque<int>::iterator i = d.begin();
while (i!=d.end()) {
  if (*i == 0) {
     i = d.insert(i,42);
     i++;
  }
  i++;
}
```

Possible test question:

For a server application you need to write a FIFO (first in, first out) queue, so that all incoming jobs are processed in the order they arrive. Which STL container would you use for such a queue and why (1-2 sentences)?

Associative Containers

An associative container contains keys that can be quickly associated with values. There is:

map	Stores a pair of keys and associated values. The keys determine the order of the elements in the map. map requires unique keys. Header: <map>. Pg 202
multimap	same as map, but allows duplicate keys. Header: <map>. Pg 608
set	Stores just keys in ascending order. Set requires unique keys. Header: <set>
multiset	same as set, but allows duplicate keys. Header: <set>

To declare a map we need two datatypes, the key and the value datatype:

```
map<string,int> m;
```

We can then use keys of the given type as index to store and retrieve contents:

```
m["Jan"] = 1;
m["Feb"] = 2;
cout << m["Jan"] << endl;
```

Most operations on map can take a key as parameter where we usually have to use iterators, e.g. erase:

```
m.erase("Jan");
```

Trying to use an element that was not set works fine. If the valuetype is a class, then it will even create a new object for you (calls the default constructor).

```
cout << m["Mar"] << endl;
```

Practice:

To map from student id's to name it is usually wise to use a map, since we do not want to create an array with 10000000000 elments.

- Define a variable of a map type with "long" as keytype and "string" as valuetype
- Fill in two random students of your discretion (do NOT use your real SSN!!!)

```
map<long,string> students;
students[123456789] = "Some";
students[987654321] = "One";
students[987654321] = "Else";
```

An iterator over a map<K,V> will give you a pair<K,V> for every element you access (this is the real pair which is differnt from the one we used in lab).

You can access the key in the member variable first, and the value in the member variable second. Thanks to operator overloading you may either use the dereference (*) or the dereference and access member (->) operator (or both, as you wish). Example:

```
for (map<string,int>::iterator i=m.begin();i!=m.end();i++) {
  cout << (*i).first << " " << i->second;
}
```

Practice:

Using an iterator, iterate over your students map defined earlier. For every student print something like this on the screen (remember: first is the key, second is the value):

```
Student: Max Berger
Student ID: 123456789
```

```
map<long,string>::iterator si;
for (si=students.begin();si!=students.end();si++) {
  cout << "Student: " << si->second << endl;
  cout << "ID: " << si->first << endl;
}
```

Iterator categories

There are five categories of iterators:

Input	Permits you to read a sequence in one pass. The increment operator (++) advances to the next element, but there is no decrement operator. The dereference operator returns an rvalue, not an lvalue, so you can read elements but not modify them.
Output	Permits you to write a sequence in one pass. The increment operator (++) advances to the next element, but there is no decrement operator. You can dereference an element only to assign a value to it. You cannot compare output iterators.
Forward	Permits unidirectional access to a sequence. You can refer to and assign to an item as many times as you want. You can use a forward iterator whenever an input or an output iterator is required.
Bidirectional	Similar to a forward iterator but also supports the decrement (--) operator to move the iterator back one position. Example: list<>::iterator.
Random access	Similar to bidirectional iterator but also supports the subscript [] operator to access any index in the sequence. Also, you can add or subtract an integer to move a random access iterator by more than one position at a time. Subtracting two random access iterators yields the distance between them. You can compare two random iterators with < or >.Thus, a random access iterator is most like a conventional pointer, and a pointer can be used as a random access iterator. Examples: deque<>::iterator, vector<>::iterator.

We will hardly every see anything but bidirectional and random access iterators. But it is important to know the other types exist.

With the exception of output each of these iterators includes all of the above. (a bidi is also forward and input, etc.).

In this example we use the fact that we can do math with iterators to find the index of elements that match a certain value.

```
vector<int> v;
```

```
v.push_back(1);
v.push_back(2);
v.push_back(2);
v.push_back(3);

for (vector<int>::iterator i=v.begin();i!=v.end();i++) {
  if ((*i) == 2) {
    cout << "I found the number 2 at index " << i - v.begin() << endl;
  }
}
```

Practice:

Assume the vector definition from above. Write a for loop that uses an iterator to output every other element. Actually advance the iterator by 2. Here you'll have to use the < operator instead of !=.

```
for (vector<int>::iterator i=v.begin();i<v.end();i+=2) {
  cout << *i << endl;
}
```

Algorithms

The STL also provides some standard algorithms. There are way to many to list here, but they all work pretty much the same way. I have selected some random ones:

unsigned count(InIter first, InIter last, T value)	counts all elements in the range [first, last[that match the given value.
FwdIter max_element(FwdIter first, FwdIter last)	returns an iterator that points to the larges element in the range [first,last[. (can you guess what min_element does?)
InIter find(InIter first, InIter last, T value)	returns an iterator to the first occurence of value in the range [first,last[.
void sort(RandIter first, RandIter last)	sorts the elements in [first,last[to be ascending. The value type of the container must support the operator<
void random_shuffle(RandIter first, RandIter last)	randomly shuffles the elements in the range [first,last[.

count and find require an Input Iterator or better, maxElement requires a forward iterator or better, sort and random_shuffle require a random access iterator.

There are about 66 algorithm in the STL. For a complete reference, read Pg. 270-274 and Pg 328-369.

To use algorithms, you have to include the <algorithm>

I want you to know the five algorithms given here, If you need to know other ones then I'll provide the reference.

Example:

```
list<int> l;
l.push_back(1);
l.push_back(2);
l.push_back(1);
l.push_back(3);

// There are two 1's in there, so this will print 2
cout << count(l.begin(),l.end(),1) << endl;
```

Practice: Print out the value of the largest element in l.

```
cout << *(max_element(l.begin(),l.end()));
```

Here is another example:

```
vector<int> v;
for (int i=1;i<100;i++)
  v.push_back(i);

random_shuffle(v.begin(),v.end());

cout << "The largest element is now at ";
cout << max_element(v.begin(),v.end())-v.begin() << endl;
```

Part III. wxWidgets

Table of Contents

Chapter 17. Introduction

What are GUI toolkits?

GUI	Graphical User Interface

The problem:

- There are many OSes out there
- Every OS looks different
- There must be some way to get buttons, windows, etc.
- There is no common standard!

Unfortuntely, every OS has a different "windowing system"

Unix / Linux	uses X11.
MacOS X	uses Aqua.
Windows	uses Win32 API.

Programming for these systems direct is not much fun. It involved plain C (no C++).

Therefore, people have written toolkits.

Toolkit	In computer programming, widget toolkits (or GUI toolkits) are sets of basic building elements for graphical user interfaces. They are often implemented as a library, or application framework.

Some very common Toolkits are:

Motif	build on top of X11, written in C.
GTK (Gimp Tool Kit)	build on top of X11, written in C. A windows version is available, but not as stable.
QT	Written in C++. The X11 version is free, the windows version is commercial.
Carbon	Build on top of Aqua. Written in C.
Cocoa	Build on top of Aqua. Written in Obj-C.
MFC (Microsoft Foundation Classes)	Build on top of Win32. Written in C++. Only available with Visual Studio
VCL (Visual Component Library)	Build on top of Win32. Written in C++. Only available with Borland compilers.
wxWidgets	Cross-Plattform. Works on top of X11, Carbon, or Win32. Written in C++. Freely available for most compilers.

There are many, many other toolkits. This is just a small selection.

Unfortunately most of these toolkits do not come standard on the respective OS. That means:

- When you develop, you need to make sure the development portion of the toolkit you are using is installed on the machine

- When you deploy, you need to ensure that the deployment portion of the toolkit is on the users machine. For windows, this means .dll files.

What is wxWidgets?

wxWidgets...

- was formerly called wxWindows
- has been around since 1992
- provides a common interface for GUIs on MacOS, Windows, Unix
- provides other functionality as well (threads, HTML viewing, etc.)

For more information and documentation, please see:

- www.wxwidgets.org [http://www.wxwidgets.org]
- max.berger.name/howto/wxWidgets [http://max.berger.name/howto/wxWidgets/]

Chapter 18. A small wxWidgets program

The code

```
/*
 * hworld.cpp
 * Hello world sample by Robert Roebling
 * Adapted for unicode by Max Berger
 */

#include "wx/wx.h"


class MyApp: public wxApp
{
    virtual bool OnInit();
};


class MyFrame: public wxFrame
{
public:

    MyFrame(const wxString& title,
           const wxPoint& pos, const wxSize& size);

    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

    DECLARE_EVENT_TABLE()
};

enum
{
    ID_Quit = 1,
    ID_About,

};

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(ID_Quit, MyFrame::OnQuit)
    EVT_MENU(ID_About, MyFrame::OnAbout)
END_EVENT_TABLE()

IMPLEMENT_APP(MyApp)

bool MyApp::OnInit()
{
    MyFrame *frame = new MyFrame( wxT("Hello World"),
         wxPoint(50,50), wxSize(450,340) );
    frame->Show(TRUE);
    SetTopWindow(frame);
    return TRUE;
}

MyFrame::MyFrame(const wxString& title,
       const wxPoint& pos, const wxSize& size)
: wxFrame((wxFrame *)NULL, -1, title, pos, size)
{
    wxMenu *menuFile = new wxMenu;
    menuFile->Append( ID_About, wxT("&About...") );
    menuFile->AppendSeparator();
    menuFile->Append( ID_Quit, wxT("E&xit") );

    wxMenuBar *menuBar = new wxMenuBar;
    menuBar->Append( menuFile, wxT("&File") );

    SetMenuBar( menuBar );

    CreateStatusBar();
    SetStatusText( wxT("Welcome to wxWindows!") );
}


void MyFrame::OnQuit(wxCommandEvent& WXUNUSED(event))
{
    Close(TRUE);
}

void MyFrame::OnAbout(wxCommandEvent& WXUNUSED(event))
{

    wxMessageBox(wxT("This is a wxWindows Hello world sample"),
        wxT("About Hello World"), wxOK | wxICON_INFORMATION, this);
}
```

GUI Elements as objects

GUI programming fits very nicely in the object-oriented methodology.

Every type of thing that can be displayed on the screen is a class. Example: for the wxFrame class for windows.

If we need a specialiced version, we can subclass the default class for our own behavior. For example: MyFrame in the example. Could also be DocumentWindow, SettingsWindows, GameFrame, etc.

The actual item displayed on screen is an instance of that class. For some types of windows, there will be only one instance (example: preferences), for some, there wil be multiple instances (example: multiple browser windows).

Event driven programming

In classical programming, we are in control of the control flow.

In GUI programming the user is.

So what do we do? We would need to do something like this (this is not actual code!):

```
while (true) {
  event = waitForUserToClickSomething();
  processEvent(event);
}
```

and this is acutally the way it is done in classical GUI programming. But we are thinking in terms of classes and objects, and messages between these objects.

So when the user clicks somethings, we want a message to be sent to some object.

We need to tell the computer what event should be sent to which object.

This is the purpose of the event table. It describes which methods should be called on which event.

The EVENT macros implement this behavior for us.

It can also be done programmatically (see wxWidgets documentation).

Inside your declaration, use this:

```
   DECLARE_EVENT_TABLE()
```

Inside your implementation file, use something like this:

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(ID_Quit, MyFrame::OnQuit)
    EVT_MENU(ID_About, MyFrame::OnAbout)
END_EVENT_TABLE()
```

This will implement an event handler for the class "MyFrame" which is subclassed from "wxFrame". For the menu event with the id "ID_Quit" it will call the method "OnQuit" from "MyFrame". For the menu event with the id "ID_About" it will call the method "OnAbout".

Prefixing event methods with "On" is just a convention and not required. But it is a good idea.

Unicode

Regular ASCII character codes use values 0..255 (char datatype). This is good for english letters (there are 26 lower case, 26 upper case, 10 numbers, and some symbols), but it is inappropriate for languages like chinese.

Therefore a larger character set was defined by the UNICODE consortium. (see www.unicode.org [http://www.unicode.org/]). In unicode, each character uses 4 (sometimes 2) bytes. Modern GUI toolkits should provide support for unicode.

When compiling wxWidgets, you have a choice to do so. If you have the Mac OS X version of you have compiled it with my instructions, your wxWidgets is the unicode version.

Unfortunately, when we define a string using "" it is by default the old version of the string (an array of char). So we need a way to convert from classical strings to unicode strings. We use the wxT() macro to do so. Example: wxT("Hello") would result in a unicode-string representing Hello.

To represent unicode characters, use the wxChar datatype.

To represent unicode strings, use the wxString datatype. It behaves almost the same as the STL string datatype.

Short rules:

- Always use wxChar instead of char.
- Always enclose literal string constants in wxT() macro unless they're already converted to the right representation or you intend to pass the constant directly to an external function which doesn't accept wide-character strings.

- Use wxString instead of C style strings.

Please note: If you have compiled wxWidgets yourself, you version may not use unicode. In this case, the above macros will still work, and default to the non-unicode version. (e.g. wxT() will not do anything). However, I will test your program on an unicode version! So unless you have the wxT() macros, you I will get compiler errors and you loose points. Please either recompile wxWidgets or test your program in the lab!

Multiple files

The above program should be broken up into multiple files. The definitions should be in the header files, the implementation in implementation files. The IMPLEMENT_APP and EVENT_TABLE macros should be in the implementation files for that class.

wxApp

Is the base class for your application. You usually want to override it, and override implementations for:

- virtual bool OnInit()
- virtual int OnExit()

There are also some other functions that can be overriden. Please see the wxWidgets documentation if you're interested.

The most important step in the initialization is to create a frame, make it visible, and make sure it is the top window.

wxFrame

Base class for all windows on the screen (wxWidgets calls them frames). You will override this at least once, for your main window type. If you have multiple different windows, you will have multiple wxFrame subclasses.

wxMenuBar

used to add a menu to your window.

The top most level is a wxMenuBar. A menu bar may have multiple Menus. You can set the menubar for a windows with the SetMenuBar() method.

A Menu has

- regular entries
- separators
- submenus

Mac users "think different". In their view, it is not the windows that has a menu, but the active application. They also want "Exit" to be called "Quit" and to be in the application menu instead of the File menu. Fortunately we do not need to worry about this, wxWidgets will do that autmatically for us. We do need to know about it, since our menus may move without us noticing. Please note that wxID_ABOUT and wxID_EXIT are predefined by wxWidgets for that exact purpose.

There are many standard event identifiers for all the default menu items. Please use them whenever possible! See www.wxwidgets.org/manuals/2.6.3/wx_stdevtid.html [http://www.wxwidgets.org/manuals/2.6.3/wx_stdevtid.html] for a complete list.

Status Bar

A special part of any wxFrame. Can be used to display messages. wxFrame offers two methods for use with status bars:

void CreateStatusBar()	creates a status bar on that window. Call it only once and in the constructor for your wxFrame.
void SetStatusText(wxString text)	sets the text in the status bar. May be called from everywhere in your wxFrame, e.g. when a menuitem was selected.

Use the status bar to display additional information.

Chapter 19. Filling a window

Adding a button

lets add a button to our sample program. In the constructor for the frame, we add:

```
new wxButton(this, wxID_YES);
```

This will create a new button. The parent window for the button will be the frame (this). The id will be wxID_OK which is the default ID for any OK button. the wxButton constructor has many more optional arguments:

```
wxButton(wxWindow* parent, wxWindowID id,
const wxString& label = wxEmptyString,
const wxPoint& pos = wxDefaultPosition,
const wxSize& size = wxDefaultSize,
long style = 0,
const wxValidator& validator = wxDefaultValidator,
const wxString& name = "button")
```

The individual parameters are:

parent	the parent window for the button. In this case, it is the current frame. In many cases, this can be a panel, a notebook, etc.
id	the ID for the item. Connect this via your event table to call your functions. If you use default ids you will get default behavior.
pos	the position of the item, given from the top-left corner of the parent. Use wxPoint(int x, int y) to create a point object.
size	the size of the item. Use wxSize(int width, int height) to create a size object.
style	extra style attributes. can be any combination of: wxBU_LEFT, wxBU_TOP, wxBU_RIGHT, wxBU_BOTTOM, wxBU_EXACTFIT, wxNO_BORDER joined with a logical or (|). Example: `wxBU_LEFT | wxNO_BORDER`
validator	a validator can be used to check input data and have input data automatically retrieved and stored in a variable. This is useless for buttons, but useful for text fields.
name	a name for the control. Absolutely useless on modern systems. Leave the default value in there.

Please note: Even though we did not set the label, and it defaults to an empty string, the button will still display "Yes". This is because we use the default id for a "yes" button. On an international machine this will automatically translate into the language that is set, e.g. "Ja" on a German machine, "Si" on a Spanish one, etc.

Practice: Add a new button labeled "Bla" with the event id "ID_BLA". It should be at position 50,30 in the window, and have no border.

To connect a button, use the EVT_BUTTON in your event table:

```
EVT_BUTTON(wxID_YES, MyFrame::buttonPressed)
```

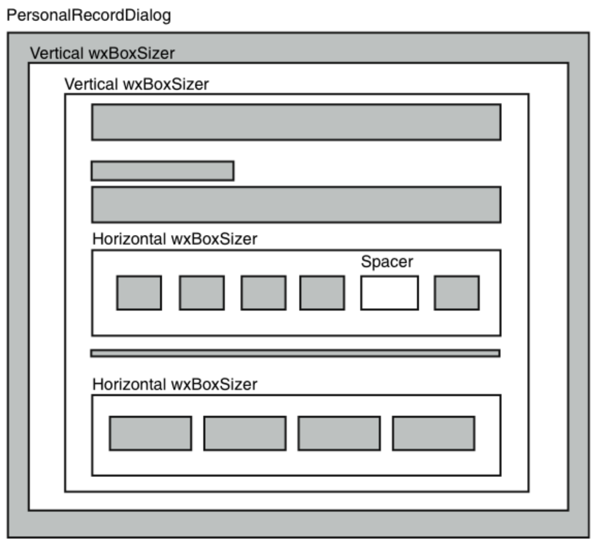
Layouting

Using the pos and size parameters we can now add buttons to a window and specify where they should be at. We can manually control the layout of a window.

But what if the window is resized? What about the difference in button size on OS X / Linux / Windows? For these things, we need automatic layout. wxWidgets offers this through "Sizers". A more common term is "Layout Managers".

We will look at the wxBoxSizer class. The boxSizer class supports horizontal or vertical stacking of controls. Each boxSizer may contain other boxSizers.

Figure 19.1. Sample stacked box Sizers



To support automatic layout, each control has a minimum size, or sometimes called "best size". It will report that back to the sizer for layouting.

To use sizers, first create an object of the wxSizer class, using either wxVERTICAL or wxHORIZONTAL as parameter.

```
wxBoxSizer *sizer = new wxBoxSizer( wxVERTICAL );
```

You may then add elements, either controls, or other spacers:

```
void Add(wxWindow* window, int stretch=0, int flags=0, int border=0);
void Add(wxSizer* window,  int stretch=0, int flags=0, int border=0);
```

Where the parameters mean:

window	the element to be added to that sizer	
stretch	a stretch factor. If the sizer is larger than the sum of its contents, what should happen? If the stretch factor is 0, the elements will stay the same size. If it is > 0, then each element will be stretched in relation to the sum of all stretch factors.	
flags	defines additional info for resizing:	
	0	the client window will preserve its size
	wxGROW	the client window will grow to fill the space
	wxLEFT, wxRIGHT, wxTOP, wxBOTTOM	specifies where the borders go
	wxALL	border on all edged (= wxLEFT | wxRIGHT | wxTOP | wxBOTTOM)
border	size of the border	

To set a sizer, use the SetSizer(wxSizer sizer) method

Example:

Adding two buttons, on top of each other, with a 10 pixel border around them:

```
wxBoxSizer *sizer = new wxBoxSizer(wxVERTICAL);
sizer->Add(new wxButton(this,wxID_YES),0,wxALL,10);
sizer->Add(new wxButton(this,wxID_NO),0,wxALL,10);
SetSizer(sizer);
```

Practice:

Create two buttons "Ok" and "Cancel" (wxID_OK and wxID_CANCEL) and add them to a horizontal sizer, with Ok on the left and Cancel on the right. Put a small border (2 px) around the buttons.

You can create "static text" (text that cannot be changed by the user) using the wxStaticText class.

```
wxStaticText(wxWindow* parent,
  wxWindowID id,
  const wxString& label,
  const wxPoint& pos = wxDefaultPosition,
  const wxSize& size = wxDefaultSize,
  long style = 0,
  const wxString& name = "staticText")
```

Most parameters are similar to the ones for the button. Label describes the contents of the text. You may change the label later using the void SetLabel(const wxString& label) function. Please note: To change the label later, you will need to keep a reference to it. A good idea would be to declare an attribute for the changing label, and to use that. Example:

```
class MyFrame: public wxFrame
{
private:
    wxStaticText *someText;
//...
wxBoxSizer *sizer = new wxBoxSizer(wxVERTICAL);
someText = new wxStaticText(this,wxID_ANY,wxT("This is a Text"));
sizer->Add(someText);
sizer->Add(new wxButton(this,wxID_OK),0,wxALL,2);
SetSizer(sizer);
//...
someText->SetLabel(wxT("Ok, Ok, i give up!"));
```

Warning! If you're widgets get much bigger / smaller, you will have to tell youre sizer(s) to re-layout. For this, you will have to keep them in an attribute as well. Example:

```
private:
  wxBoxSizer *sizer;
//...
sizer->Layout();
```

And, last but not least you may resize your window to fit the contents. Use the Fit(wxWindow&) method for that. Example:

```
sizer->Fit(this);
```

data input

There are two methods of data inputs:

- Reading directly from the control widgets
- Providing validators