OOP and UML Fundamentals
C++ OOP Aspects

Fundamental OOP Concepts. UML Class Diagrams

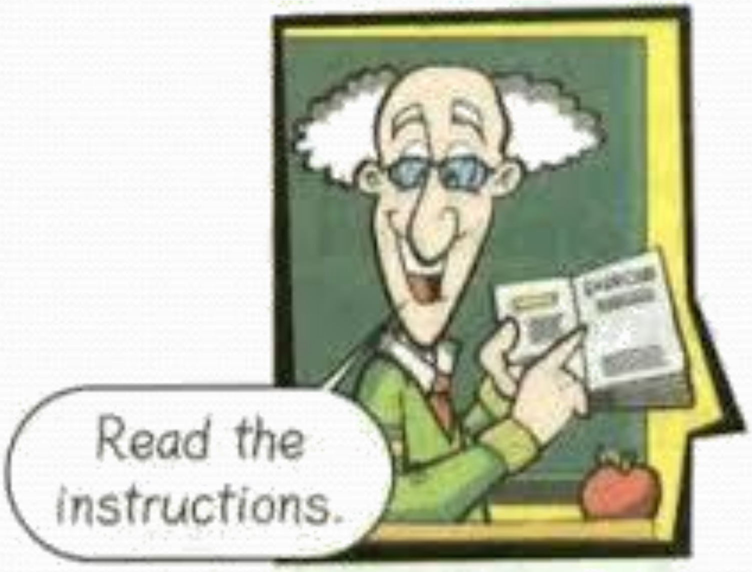
Programming Paradigms

How do you approach a problem?

Programming Paradigms

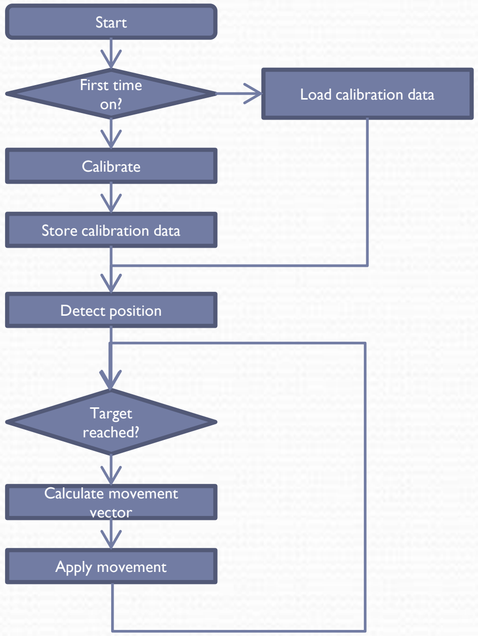
Imperative Programming – Introduction

- *"In computer science terminologies, **imperative programming** is a **programming** paradigm that describes computation in terms of statements that change a **program** state."*
- "Do this – do that" approach



Programming Paradigms
Imperative Programming – contd.

- A problem is approached using "step-by-step" modeling: break down the functionality into series of steps
- Best in products where sequence of steps is fixed or rarely changed
- Finer-level steps can be combined into larger blocks (***procedures***) to produce clear, concise and manageable sequence at each level



Programming Paradigms
Imperative Programming – Procedural

- Focus on ***procedures*** – a series of instructions that can be called from any point in the program
  - In *C/C++* - "***functions***"
- ***Procedures*** help achieve some degree of ***modularity***
- Modularity allows for ***testability***, ***reusability***, ***maintainability***, …

Programming Paradigms
Imperative Programming – Pros & Cons

- Pros
  - Matches the underlying technology
  - Easy to create or comprehend, especially for small systems
  - Easy to derive directly from user requirements
  - (? pro/con) Supported by modern object-oriented and (some) functional languages
- Cons
  - No direct way to communicate the relation (and consistency) between data and procedures/functions
  - Basic reusability unit is ***function*** which is not very stable and is prone to frequent changes for both requirement and technical reasons
  - Does not reflect correctly the ***real world***, which is made of ***interacting entities***, not ***separate*** activities and data

Programming Paradigms
Declarative Programming

- *Unofficially: "Style of programming that is* ***not*** *imperative"*
- *"In computer science, declarative programming is a programming paradigm, a style of building the structure and elements of computer programs, that expresses the* ***logic*** *of a computation* ***without*** *describing its* ***control flow****."* - *Wikipedia*
- *"A program that describes* ***what*** *computation should be performed and not* ***how*** *to compute it"*

Programming Paradigms
Declarative Programming – Examples

- *SQL Example*

  SELECT ***

  FROM *Students*

  WHERE *age > 35*

  ORDER BY *name*
- *Linq*
- *Functional programming*

Programming Paradigms
Object-Oriented Programming – Intro

- Where does it stand? Is it *Imperative*, or *Declarative*?
- Based on the concept of ***objects*** which combine ***data*** and ***behavior***
- When solving a problem, the primary focus is on ***objects*** and their ***relations/interactions***
- Example: a ***Car*** *has an* ***Engine***, ***Chassis***, ***Wheels***, … Then ***Engine*** itself has ***Cylinders***, ***Sensors***, etc.

Programming Paradigms

Object-Oriented Programming – Pros & Cons

- Pros
  - Closer to the real world – **entities** that we want to model are often directly **object-oriented** ready
  - The link between data and processing code is built into the syntax – the system is ***aware*** about it
  - Reusability units are much more obvious (often a ***class*** is directly reusable) and, with proper design, much more stable than functions
- Cons
  - Requirements are not very object-oriented; from them, suitable ***objects*** / ***classes*** need to be extracted and defined by the software Designers / Architects
  - The underlying hardware, storage, communications, etc. operate as series of operations, i.e. closer to ***Imperative*** paradigm

Object-Oriented Programming Fundamentals

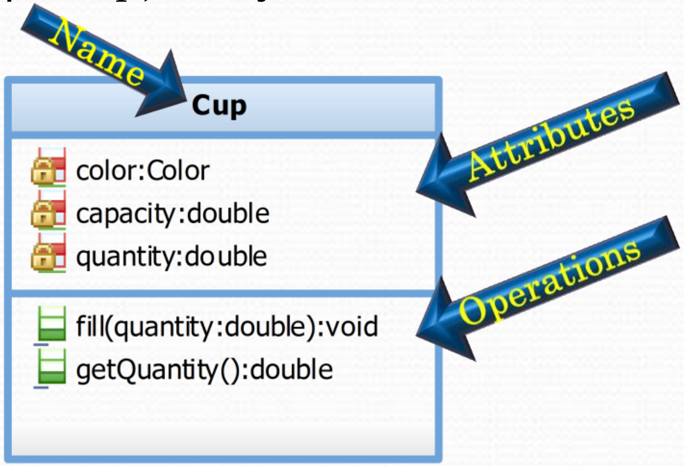
No objections

Object Oriented Programming
Introduction

- Completely different way to approach a problem
- Start by describing ***objects***, their ***relations*** and ***interactions***
- Key question to start in OOP way: *"**What** are we talking about?" (instead of "what the program will **do**"!)*
- At the end, objects' behavior is still ***imperatively*** described (i.e. method implementations)
- In short, ***objects*** combine ***data*** and ***behavior***
- A developer can think in object-oriented manner and still use "procedural" language such as C
- However, in ***object-oriented*** languages there is suitable syntax to ***express*** the link between data and behavior of an ***object***
  - This helps ***find errors***, ease the process of ***development*** and developer ***testing***, ***redesign*** and/or expand the system more safely, etc.

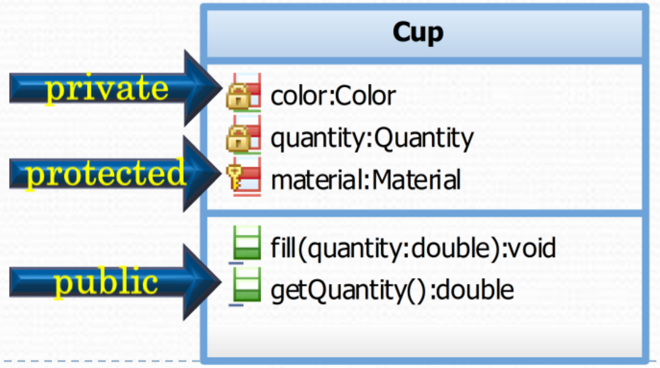
Object Oriented Programming
Objects & Classes. UML Class diagrams

- ***Objects*** can often be *categorized* into ***classes*** – groups of objects having the same general *attributes* and *operations* (behavior), but possibly differing in their ***values***
- *Example of a simple* ***UML class diagram*** *(with just 1 class in it):*
  - *Class* ***name*** *– Cup. Example objects may be myCup, coffeeCup, … (not shown below). Choose class and object* ***names*** *carefully!*
  - All ***Cup*** objects (***instances***) have the same set of ***attributes*** *(color, capacity and quantity)* but possibly different *values* (i.e. *color*=***RED*** for myCup, vs. *color* = ***BLACK*** *for coffeeCup*). They also have the same operations**.**



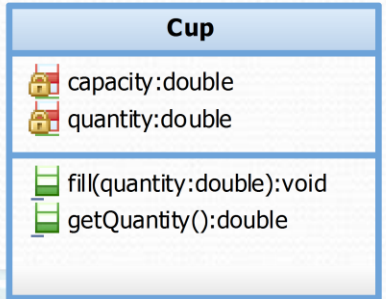
Object Oriented Programming
Objects & Classes – Access Modifiers

- ***Access modifiers*** (in a class) specify from where is a certain member (attribute or operation) accessible.
  - In example, for an ***attribute*** the term "access" can mean read its value, write its value, get address of, etc. For an ***operation*** "access" usually means to *execute the operaton (call* the method), but might also mean get its address.
- *Strongest* specifier is ***private***: only code belonging to the class itself can access a ***private*** member. *"Code"* here roughly means functions / methods
- *Weakest* specifier is ***public***: all functions and methods can access a ***public*** member
- There is a number of intermediate levels of access, in C++ only ***protected*** ()



Object Oriented Programming

Objects & Classes in C++



Simplified

```cpp
// main.cpp
class Cup {
public:
    void fill(double quantity) {   // inline
        this->quantity += quantity;
    }

    double getQuantity() {      // inline
        return quantity;
    }
private:
    double capacity;
    double quantity;
};
```

Real-world

```cpp
// cup.hpp
#ifndef CUP_HPP_
#define CUP_HPP_

class Cup {
public:
    void fill(double quantity);
    double getQuantity();

private:
    double capacity;
    double quantity;
};

#endif /* CUP_HPP_ */
```

```cpp
// cup.cpp
#include "Cup.hpp"

void Cup::fill(double quantity) {
    this->quantity += quantity;
}

double Cup::getQuantity() {
    return quantity;
}
```

Object Oriented Programming

C++ fundamentals. Default arguments. Overloading

- **Default arguments**

```
void printCoordinates(double x, double y = 0.0, double z = 0.0) {
      cout << x << ", " << y << ", " << z << endl;
}
```

  - printCoordinates() can be called with 1, 2 or 3 arguments
  - *printCoordinates() is* ***the same function / sequence of instructions*** *taking 3 parameters, just they have default values sometimes*

- **Function & operator overloading**

```
int add(int x, int y) {
    return x + y;
}
int add(int x, int y, int z) {
    return x + y + z;
}
```

  - add() **appears to be the same from user perspective**
  - Actually they are 2 (or more) **different functions**
  - **Operators** are basically functions with special names (i.e. ***operator+***) and a few added specifics

More on overloading

Object Oriented Programming

C++ fundamentals. Exceptions

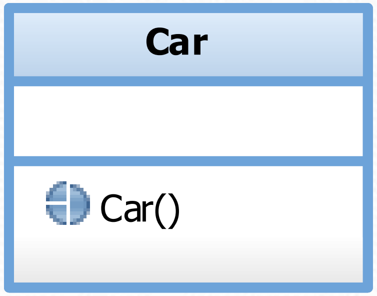
1. Error handling. Compare strategies:
    1. Return codes
    2. Exceptions

More on exceptions

Object Oriented Programming

Objects & Classes. Constructors

- ***Constructor*** is a special method that is called when object is created
  - In C++, the constructor has the name of the class (i.e. ***Cup***()).
    - No return type, not even ***void***
    - Can have parameters. Can be ***overloaded***
  - The compiler takes care of the call, whenever an object is being created
- A ***default*** constructor is such that can be called without arguments
    - ***Either*** it has ***no parameters***
    - ***Or*** *all* of its *parameters* have ***default values***
- An ***implicit*** *public* ***default*** constructor is created by the compiler, if the programmer does not define ***any*** constructor.

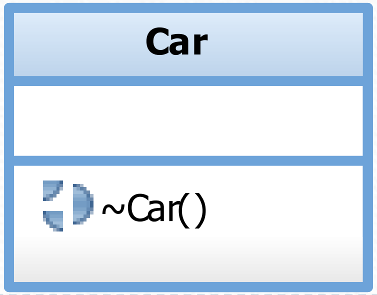


Object Oriented Programming
Objects & Classes. Destructors

- ***Destructor*** is a special method that is called when object is destroyed
  - In C++, the destructor has the name of the class prefixed by ~ (i.e. **~Cup**())
    - No return type, not even ***void***
    - Can **not** have parameters. ***No*** overload possible!
  - The compiler takes care of the call ***almost*** always, with 1 "exotic" exception in C++

Placement new

- If the programmer does not define a destructor, the compiler declares and defines an ***implicit*** *public* ***destructor***

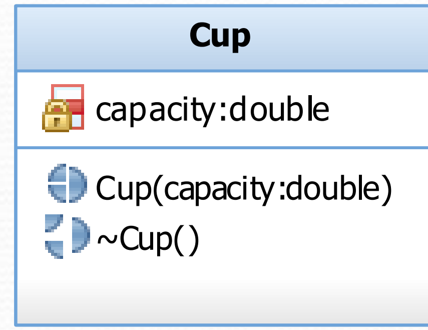


Object Oriented Programming

Constructors & Destructors – *Live Exercise*

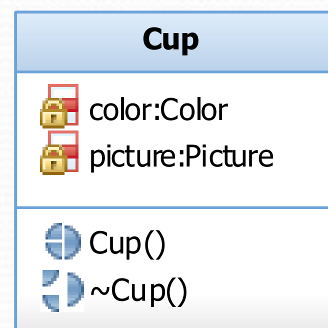
1. Demonstrate (in C++) in what order are called constructors and destructors of 2 automatic-duration objects ("stack objects")
   - Live example:

```
int main() {
    Cup cup1(100);
    Cup cup2(200);

    return 0;
}
```



2. Demonstrate the order in which constructors/destructors of ***attributes*** are called, w.r.t. each other and to their containing object's constructor/destructor.
   - Hint: define your own classes Color and Picture

```
int main() {
    Cup cup1();

    return 0;
}
```



Object Oriented Programming

Constructors & Destructors – *Order Explanation*

- The *order of construction/destruction* was demonstrated for ***automatic-duration objects*** relative to one another
- The *order of construction/destruction* was also demonstrated for ***members*** of an object with respect to each other, as well as relative to object's own constructor/destructor
- As it was demonstrated, the simplest scenario of automatic-duration objects ensures ***nested lifetimes*** of objects
- The purpose of having ***nested lifetimes*** is to ***avoid broken dependencies***
  - Broken dependency: object A trying to use object B after B's destruction
  - In the simplest example objects cup1, cup2 did not keep pointer/reference to each other
  - However in the real world it is often the case that an object keeps pointer/reference to another object
- That's why a ***stack*** (LIFO) data structure is used for *calling functions*, *passing arguments* and *automatic duration objects*!

Object Oriented Programming

Constructors & Destructors – *Copy Constructor*

- A ***copy constructor*** is a constructor allowing an instance of a class to be created from another instance of the same class
- ***First parameter*** of a *copy constructor* is ***always*** a ***reference*** of the same type (***Type&***), with or without ***cv (const*** and/or ***volatile)*** qualifiers. Example:

```
class Demo {
public:
    Demo(const Demo& other);    // Copy constructor declaration
};
```

- *Next parameters* either have default values or not present at all
  - Most often, no next parameters

Object Oriented Programming
Constructors & Destructors – *Temporary Objects (C++)*

- In C++ there are many situations where a ***temporary object*** is created. A ***temporary object*** can be created either explicitly by the developer, or implicitly by the compiler.
  - ***Explicitly*** using the form ***Type()*** (possibly with arguments to the constructor)
  - Returning by value from a function
  - Some casts
  - Intermediate values during expression evaluation
  - In some forms of initialization
  - Others
- When created, a temporary object has no name (or at least, initially…)
  - Hint: How ***any*** object can receive a name later on during its lifetime?

Object Oriented Programming

Temporary Objects – Exercises

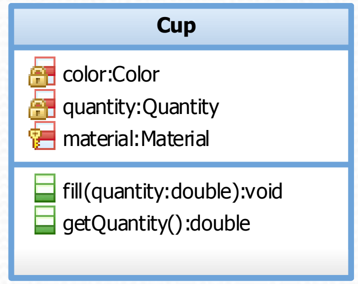
- Define a ***Demo*** class. Demonstrate ***explicit*** temporary object creation with automatic duration (*"in the stack"*). Trace object construction and destruction using console messages.
- Define a ***Demo*** class and a ***Helper*** class. Make it possible that a ***Helper*** object can be created from an ***int***. Make it possible that a Demo object can be created from a Helper object (but not directly from an ***int)***. Demonstrate implicit object creation by instantiating a Demo from an ***int***

Object-Oriented Programming Encapsulation

Hiding Secrets

Object Oriented Programming

Encapsulation – *Why & How*



- Expose as little as possible to the outside world
  - ***Why***?
- In a successful project, ***everything*** you expose (*attributes* and *operations*), ***will*** be used ***from outside***. And then the exposed attributes and operations become much harder to change (i.e. to improve, optimize, react to requirements change, fix bugs, etc.) because a lot of "foreign code" depends on them
- *How* to expose/hide? We already introduced the *access modifiers*. In C++
  - ***public*** – every piece of code can access members having this modifier
  - ***protected*** – only methods of the class itself and methods of its ***descendants*** can acces members having this modifier
  - ***private*** – only methods of the class itself can access members having this modifier

Object Oriented Programming

Encapsulation – *Interfaces. Best Practices*

- Several different, but logically close meanings of the term "Interface"
- ***Interface*** of a class – the ***exposed members*** of that class (operations & attributes)
  - *Usually "exposed" means **public**; but **protected** members are also a (special) interface for the descendants; and **package** members (not present in C++) – special interface to classes inside that package/assembly*
- ***Interface*** can also be a special construct in a given language, allowing to have just the "interface part" as a completely separate entity from implementation
  - In ***Java*** this is achieved using the ***interface*** keyword
  - In ***C++*** this is achieved by defining ***pure abstract classes*** (presented later in next chapters)
- *Synonym – **API** (Application Programming Interface)*
- What is better to expose (include in the ***interface***): ***attributes*** or ***operations***?
  - ***Why?***
  - The importance of having *control*
- Besides ***operations***, what else can be *safely* included in an ***interface***?
- ***Think carefully*** when designing interfaces!
  - *Think now, avoid problems later!*
- ***!!!** Correct usage of **OOP** **relies on** **interfaces** being **more stable** than **implementations** !!!*
  - *So in order to use OOP **correctly**, your emphasis **must be** on designing the **interfaces** instead of implementation details*
  - *Make OOP your ally*

Object-Oriented Programming
Object Relations & Interactions

I know you

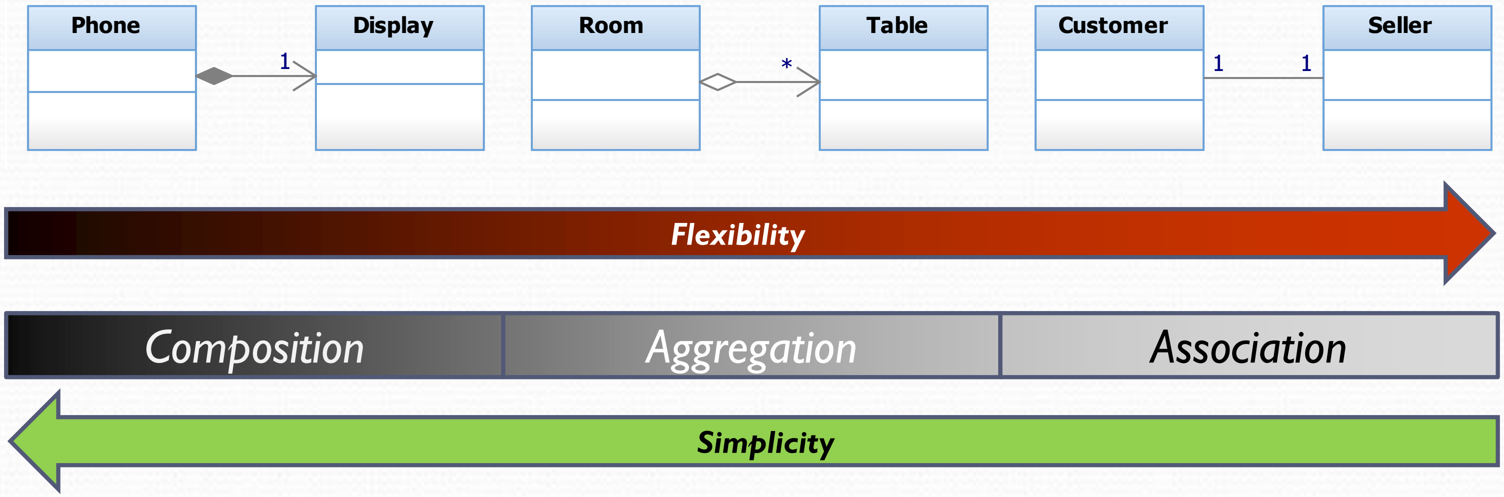
Object Oriented Programming
Object Relations

- ***Instance level relations***
  - ***Composition***
  - ***Aggregation***
  - ***Association***
- ***Class-level relations***
  - ***Inheritance***

Object Oriented Programming

Instance-level relations: *Overview*

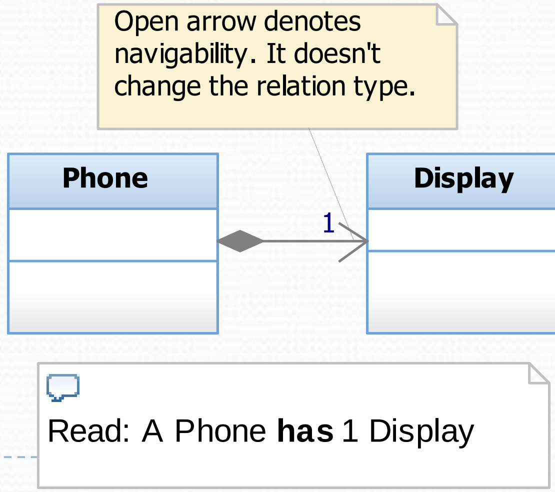
- ***Composition*** is the strongest relation – *ownership* ***(has-a) with*** *lifecycle dependency*. Simplest relation, but no flexibility.
- ***Aggregation*** is weaker *ownership* ***(has-a)*** ***without*** *lifecycle dependency*
- ***Association*** is the weakest relation of the three. ***No*** *ownership*, ***no*** *lifecycle dependency*, only knowledge / loose usage of the associated object (s).



Object Oriented Programming

Instance-level relations: *Composition*

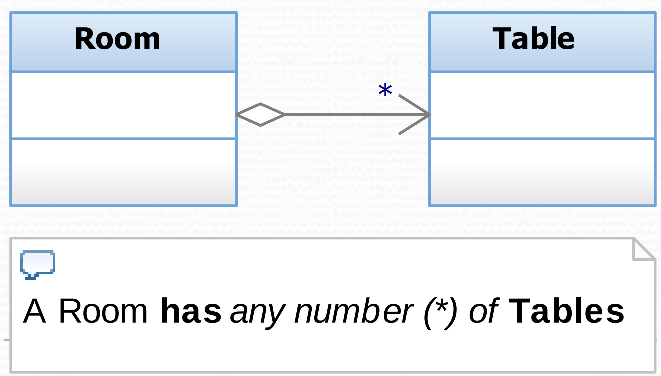
- ***Composition*** is the strongest ownership relation
- It is ***never*** *shared* – the owned object belongs exclusively and solely to the owner
- There is ***lifecycle dependency***: destroying the *owner* in a ***composition*** relation typically means destroying the *owned object(s)*
- Either the programmer does not need to take any special care of the life of the *owned object(s)*, or the cleanup work is easy to implement
- However the owned object cannot be changed, replaced, reused outside of its exclusive owner. Usually it cannot be retained after owner's death
- Therefore use ***composition*** in situations where you need ***simplicity*** but ***not flexibility***.



Object Oriented Programming

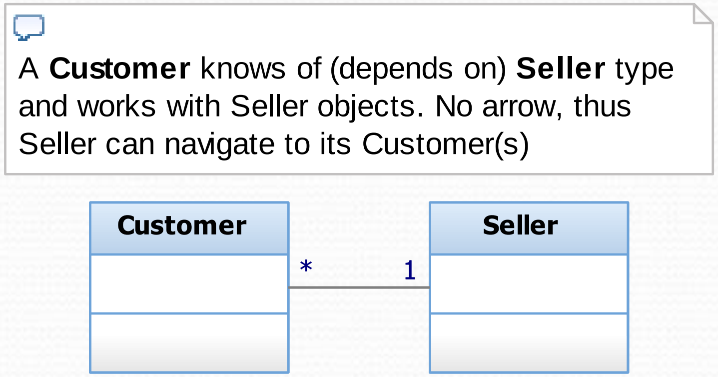
Instance-level relations: *Aggregation*

- ***Aggregation*** is weaker ownership relation
- It ***can be shared*** *(but not always is)*. If *shared*, the owned object potentially belongs to multiple owners.
- There is ***no lifecycle*** relation: the owned object can outlive its current owner. Often the owner can be created "empty" and receive owned objects later
- The programmer often needs to ***take care of lifecycles*** of both owner and owned objects separately. The programmer also must ***prevent the risk*** of calling into already dead object. ***Resource leaks*** must be prevented, too.
  - *In C++11 and above, there are very useful smart pointers (i.e.* ***std::shared_ptr****) to simplify all those tasks.*
- The *owned object typically* ***can be changed, replaced and/or reused outside*** of its current *owner*
- Use ***aggregation*** in situations where you need more ***flexible ownership***



Object Oriented Programming
Instance-level relations: *Association*

- ***Association*** is weakest relation of the three. It is ***no ownership*** at all.
- There is ***no lifecycle dependency***. An object just knows the type of another object and collaborates with it (i.e. request a service/send data)
- In its simplest form, an object ***objA*** of class ***A*** receives a message (has a method call) with object ***objB*** of class ***B*** passed as an argument. Then ***objA*** *uses* ***objB*** to perform a job and returns.
- If ***objA*** keeps a reference/pointer to ***objB*** then this is a stronger form of association, and sometimes is considered as aggregation.
- Use ***association*** to get 2 classes collaborate with each other, without ownership between them.

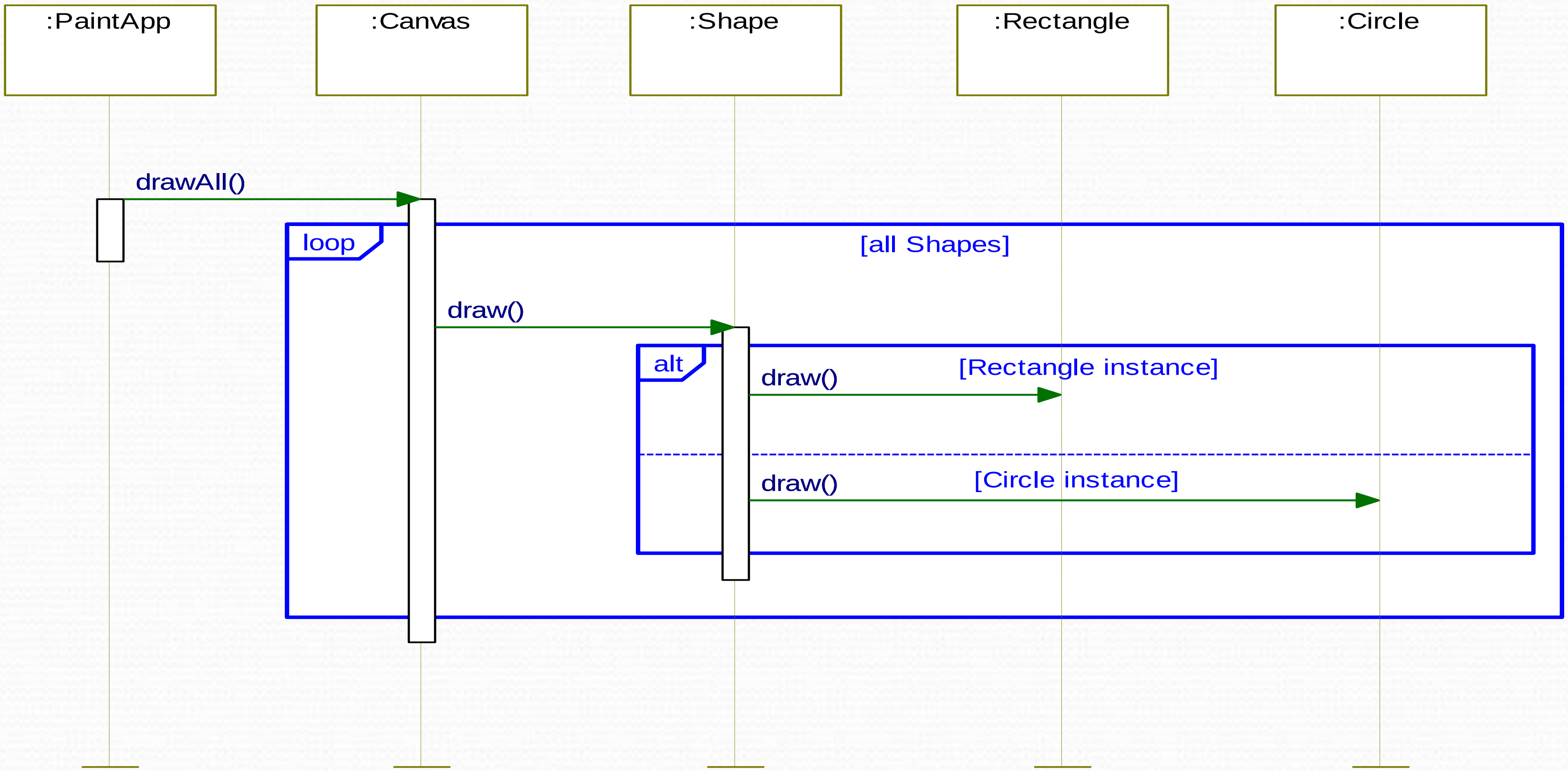


Object Oriented Programming
Instance-level relations. Discussion

- Create a UML diagram of a (part of) car racing game. You have the following requirements:
  - There is a Garage in the game; it provides maintenance to all Cars
  - The player is a Driver who owns Car(s) and goes to the Garage from time to time to change the oil, etc.
  - The Garage keeps information about what cars have been served there, but doesn't need that information until an Inspector comes and requests to check it
- After the class diagram is ready, discuss the possibilities for change and how much would various changes affect the existing design and implementation
- Implement the diagram

Object Oriented Programming
Object interactions. UML Sequence Diagrams (briefly)

- Objects (***objName:Class***) with lifelines vertically
- Messages (usually method calls)
- Closer to the implementation – ***sequence*** of method calls along the lifeline

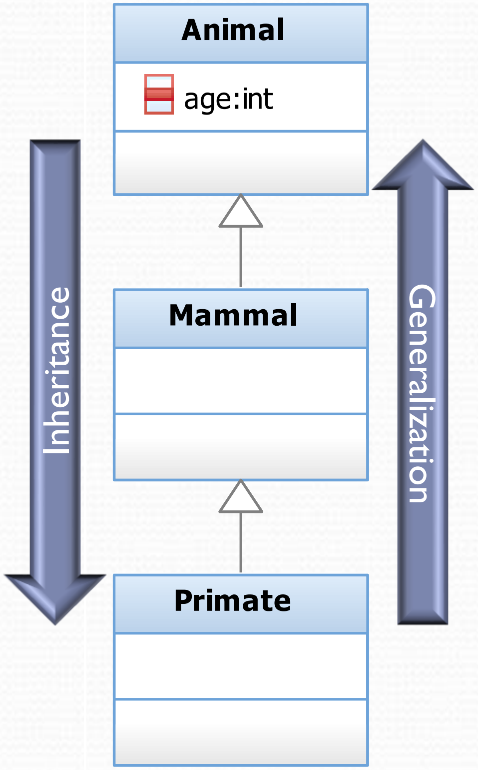


Object-Oriented Programming
Inheritance & Generalization

Two sides of the same coin

Object Oriented Programming
Inheritance, Subtyping, Generalization – *Fundamentals*

- ***class-level*** type of relation (relation between ***entire classes***, not between instances/objects).
  - Note: There is "*object-based inheritance*" (***prototyping***), that is another story…
- Defines an ***"is-a"*** relationship between classes.
  - In theory, ***is-a*** relationship refers to ***subtyping*** which is different concept than ***inheritance***; however in *C# / Java / C++* world these terms are usually used as synonyms, as in this presentation.
  - Constrast that to the ***"has-a"*** relationship in ***composition*** & ***aggregation***
- Used to ***classify*** classes into larger groups, which are also classes
  - Example: a *Mammal* ***"is-an"*** *Animal* (both Mammal and Animal are classes)
  - *Deeper* hierarchies are possible: a *Primate* ***is-a*** *Mammal*, which ***is-an*** *Animal*. This is ***multi-level*** inheritance (not to be confused with ***multiple inheritance***!)
- Each ***descendant*** (i.e. the *Primate* class) receives members of its base class(es) (i.e. *Mammal*)
  - … but it might not be able to access all of them
  - Explain ***protected*** access modifier



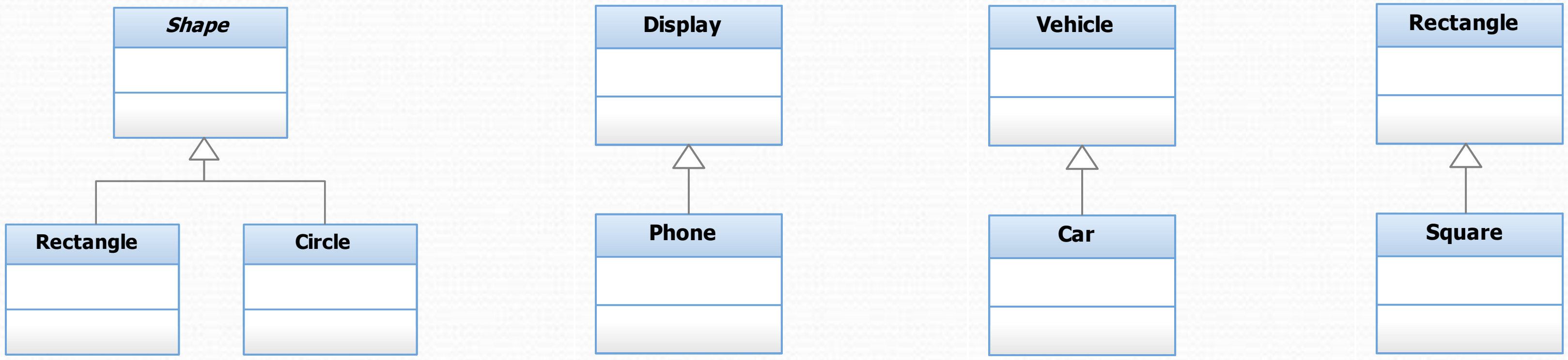
Object Oriented Programming
Inheritance, Subtyping, Generalization – *contd.*

Animal kingdom	Common features
Phylum Chordata	• a notochord (a rodlike structure that supports the body).
Subphylum Vertebrata	• a notochord which develops into a backbone.
Class Mammalia	• a notochord which develops into a backbone; and • nursing (feeding milk to) their young.
Order Primates	• a notochord which develops into a backbone; • nursing their young; and • flexible hands and feet.
Family Hominidae	• a notochord which develops into a backbone; • nursing their young; • flexible hands and feet; and • two legs.
Genus *Homo*	• a notochord which develops into a backbone; • nursing their young; • flexible hands and feet; • two legs; and • habitually walking upright.
Species *Homo sapiens*	• a notochord which develops into a backbone; • nursing their young; • flexible hands and feet; • two legs; • habitually walking upright; and • a large brain.

Object Oriented Programming

Inheritance/Generalization – *Exercises*

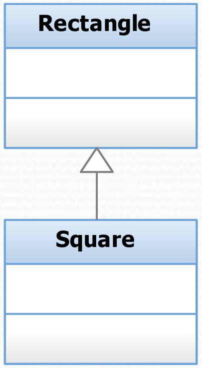
- Which of the below examples correspond to ***is-a*** relationship, and which – don't?
  - Code exercises
  - Up-casting & down-casting. C++ casts



Object Oriented Programming

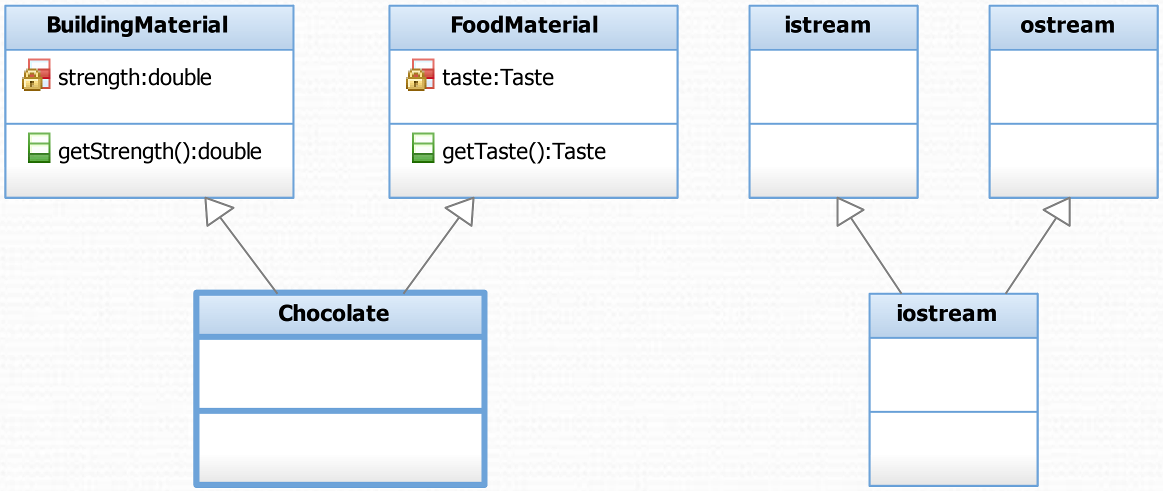
Composition or *Inheritance? **Liskov Substitution Principle***

- "Favor ***object composition*** over ***class inheritance***"
  - This means to ***avoid code reuse by inheritance*** *(Phone-Display example)*
  - In that case, we inherit only ***interfaces***, thus following true ***subtyping,*** and reuse code only via ***composition/aggregation***
- ***Liskov Substitution Principle (LSP)***
  - *Subtypes* must be able to ***substitute*** their *supertypes* without altering program's **correctness**
  - Example with *Rectangle and Square*, breaking ***LSP***



Object Oriented Programming
Multiple Inheritance

- ***Multiple Inheritance (MI)*** – a scenario where a ***descendant*** has *more than 1* ***base*** *classes*

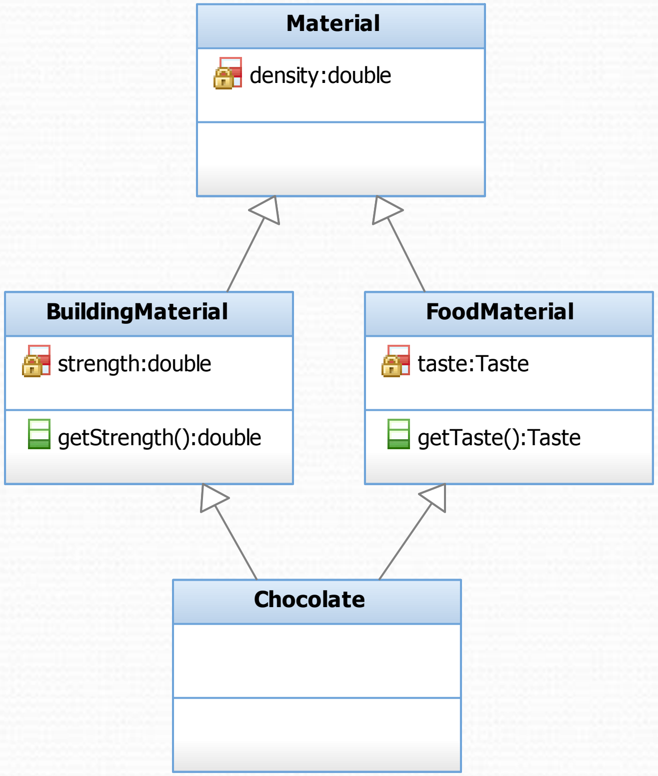


- The descendant (i.e. ***Chocolate***) inherits base classes' "features"
  - "is-a" relation to all base classes (i.e. both ***BuildingMaterial*** and ***FoodMaterial)***
  - ***Ambiguities*** may occur (i.e. having members with the same name). If that happens, use scope operator to resolve (i.e. ***Base1::x*** *vs.* ***Base2::x***)

Object Oriented Programming

Multiple Inheritance – *Dangers*

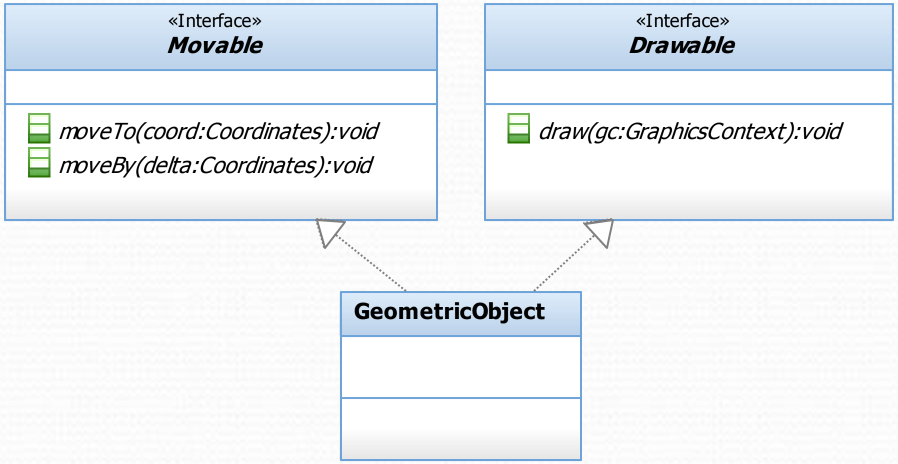
- ***Dreaded Diamond***



- How many copies of ***density*** does a ***Chocolate*** instance have?
- Virtual Inheritance

Object Oriented Programming
Interfaces – Revisited

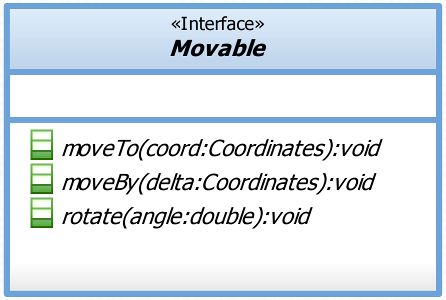
- Interface as a separate entity. UML notation
  - ***implements*** (*realizes*) relationship. *Implementation* vs. *Inheritance*
- When to extract a separate *interface* and when to use the class directly
- *"Service classes"* vs. *"Data classes"*
  - Some of the classes carry ***data*** (i.e. some ***Coordinates*** class, or ***std::string***). They might have several simple operations (accessors – getters/setters), but their purpose is to contain and carry relevant information. In simplest case, if no changes expected, it's OK to even expose attributes (i.e. ***x*** and ***y*** coordinates)
  - Other classes ***do some job*** for us: ***DBConnector*** (i.e. to create a connection to a database), ***ShapeFactory*** (create an object – instantiate a ***Shape*** descendant), etc.



Object Oriented Programming

Interface Segregation Principle. Program to Interfaces

- ***Interfaces*** in this sense (as a language construct, or some substitute such as pure virtual classes) become a unit of reusability
- A common mistake is to combine logically separate methods into one interface
  - Interfaces, similarly to classes, shall obey *__S__ingle __R__esponsibility __P__rinciple*
- *During design phase, split **interfaces** into small reusable units*
  - *This is called **segregation***
- Better err on the side of smaller, than larger interfaces
  - Exercise: How would you rework the following interface:



Object-Oriented Programming
Polymorphism

"I'll send an SOS to the world"

Sting – "Message in a bottle"

Object Oriented Programming

Polymorphism – *Definition*

"Sit down"

- Ability to ***send a message*** to an ***object***, *without* knowing its ***class***
  - *But we **have to** know something… What is it?*
- *Why* we want to be unaware of object's ***exact*** class?
- ***Polymorphism*** as basis of almost all modern OOP techniques

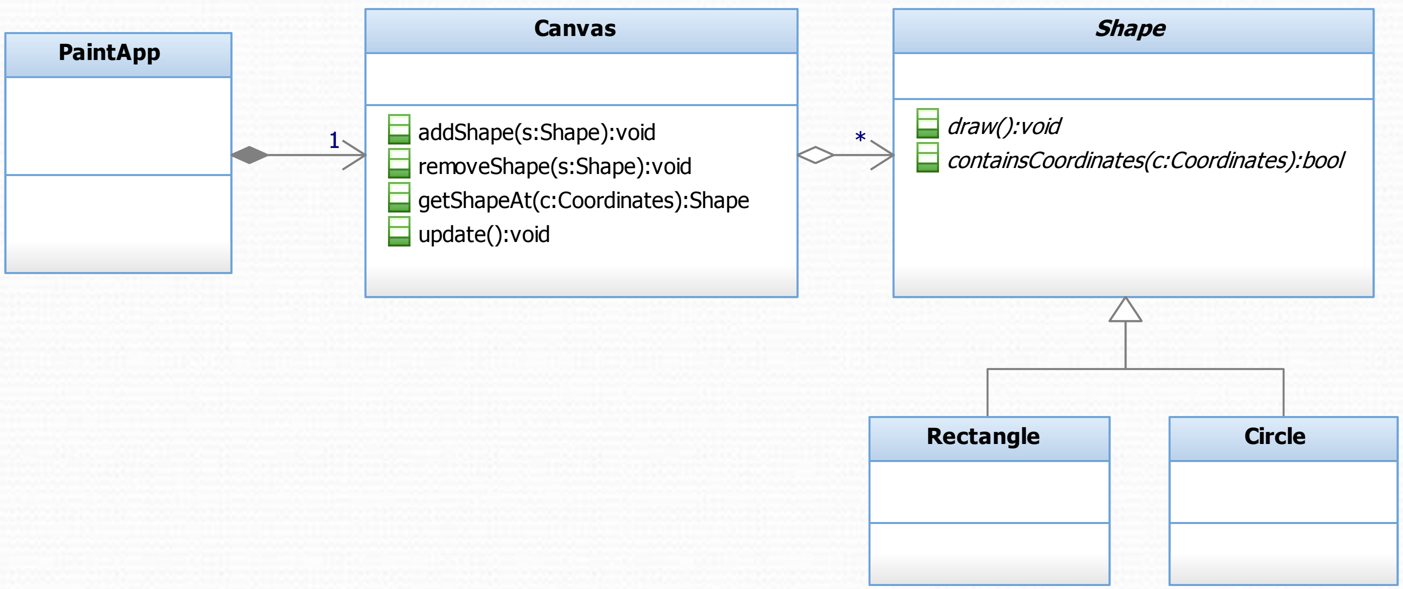
Object Oriented Programming
Polymorphism – *How-to (C++)*

- In C++, "send a message" within the same program means (usually) calling a method
- To call a method ***polymorphically*** in C++ the following is necessary:
  - An instance of some class (i.e. a ***Rectangle*** object)
  - Either a ***pointer*** or a ***reference*** of some base type (direct or indirect) – i.e. a ***Shape* pSomeShape*** or ***Shape& shape*** that refer to the concrete object (the ***Rectangle*** object)
  - The method is called via that base-type *pointer* or *reference*
  - The method ***must be present*** in the base class and ***must be virtual***

Object Oriented Programming

Polymorphism – *Example (UML Class Diagram)*

- ***Canvas*** containing graphic figures which are descendants of ***Shape***
- Implement the UML diagram (specifically, ***update()*** method that draws all graphical figures and everything necessary to call it) by making sure Canvas class doesn't "know about" (depend on) Rectangle, Circle and other Shape descendants



Object-Oriented Programming Abstractions

Abstraction in concrete terms

Object Oriented Programming
What is "Abstraction"?

- Focus on some aspects of a system (concepts; "important stuff") while safely ignoring others ("details")
  - This is related to characteristics of human mind, the way we think
- In OOP world, both ***interfaces*** and ***abstract classes*** are called ***"abstractions"***
- *Abstractions* are supposed to be designed carefully and be ***more stable*** than details. They are more *dependable.*
- ***Rule: Depend on abstractions, not on concrete classes***

Object-Oriented Programming
Genericity (Templates)

Brief intro

Object Oriented Programming

Templates – *Intro*

- What if there is ***the same*** algorithm for processing several ***different types*** of data? Code ***manually*** several different functions?

```
void myswap(int& x, int& y) {
        int temp = x;
        x = y;
        y = temp;
}
void myswap(double& x, double& y) {
        double temp = x;
        x = y;
        y = temp;
}
```

- We need to **avoid** code **duplication** (one of *"code smells"* – a sign for poor quality)
- Therefore we'd like to write the function code ***once*** replacing some of the specific type(s) with ***parameters*** (i.e. instead of ***int*** or ***double*** write ***T***)
  - Provide ***T*** "argument" where the template is used, somewhat similar to function arguments
- This is yet another way to achieve better ***reusability*** and ***maintainability***

Object Oriented Programming
Function Templates

```
#include <iostream>

template <typename T>            // template <class T> is the same, but class would be a little misleading here
void myswap(T& x, T& y) {
    T temp = x;
    x = y;
    y = temp;
}

int main() {
    int x = 5, y = 6;
    double z = 8.7, t = 1.2;
    int p = 18, q = 3;

    myswap<int>(x, y);           // Pass int as template argument. The compiler generates a function myswap(int&, int&) for you!
    myswap<double>(z, t);        // Pass double as template argument. The compiler generates myswap(double&, double&)!
    myswap(p, q);                // The compiler deduces the template argument (here int). Thus myswap(int&, int&) is used.

    std::cout << x << ", " << y << std:: endl;
    std:: cout << z << ", " << t << std::endl;
    std:: cout << p << ", " << q << std::endl;

    return 0;
}
```

Object Oriented Programming
Class Templates

```
// In a header file (i.e. MyPair.hpp)
template <typename T>  // Again, template <class T> works the same
class MyPair {
public:
    MyPair(const T& first, const T& second)
        : m_first(first), m_second(second) {}
    T& max();                                   // Can be defined outside the class...
private:
    T m_first;
    T m_second;
};

// Definition of a method outside the template class, but still in the header (MyPair.hpp)!
template<typename T>
T& MyPair<T>::max() {
    return m_first > m_second ? m_first : m_second;
}
```

Object Oriented Programming

Class Templates – *contd.*

```
// How to use the template class MyPair - i.e. in main.cpp
#include <iostream>
#include "MyPair.hpp"

int main() {
    MyPair<int> intPair(5, 3);                // The compiler generates a class MyPair<int> for you!
    MyPair<double> doublePair(5.4, 11.9);     // The compiler generates a class MyPair<double> for you!
    MyPair<char> charPair('q', 'a');          // The compiler generates a class MyPair<char> for you!

    std::cout << intPair.max() << std::endl;
    std::cout << doublePair.max() << std::endl;
    std::cout << charPair.max() << std::endl;

    return 0;
}
```