Object Oriented Programming Lecture

Classes and Data Abstraction

School of Computing and Information Technology
Faculty of Engineering and Computing
University of Technology, Jamaica

Email: dwwhite@utech.edu.jm, taedwards@utech.edu.jm

Object Oriented Programming

Expected Outcome

At the end of this lecture the student should be able to:

- model individual class diagrams using UML
- model relationship class diagrams using the UML modeling language
- perform an object-oriented analysis on a set of requirements, extracting classes and their relationships

Object Oriented Programming

Expected Outcome

At the end of this lecture the student should be able to:

- build a solution model using UML,
- transform the UML model into programming code in an object oriented language.

Object Oriented Programming

Topics to be covered in this lecture:

- Define the terms “class” and “object”
- Members of a class – attributes and methods
- List and explain the four types of methods
- Identify the three compartments of a UML class diagram and state their contents
- Model classes derived from OOA using individual UML class diagrams

Object Oriented Programming

Topics to be covered in this lecture:

- List and explain the following type of relationships among classes:
  - ✓ Dependency
  - ✓ Association
  - ✓ Generalization
- Represent the relationships among classes using a UML relationship diagram

Object Oriented Programming

Topics to be covered in this lecture:

- Translate a class diagram from UML into Java or C++.
- Define object-based programming
- Show how the four types of class methods are implemented in a class

Classes and Objects

- A class in OOP is like a blue-print (architectural drawing) for building a house – which specifies the design for the house
- An object is like an actual house built from the blue-print. The same blue-print can be used to create more than one house from the same design - for example, house in a housing scheme

Classes and Objects

- In the Object-Oriented Paradigm, the central focus is on objects which are sets of data items
- A class is the description of a set of data and the operations that can be performed on the data contained in the class
- An object is an instantiation of a class. A class can be used to create many objects of the same type, but each such object created has its own identity and state.

Classes and Objects

- Example: Student might be a class. The Student class may have Student ID Number, Name and GPA. The class might contain the operation Show GPA.
- John might be a specific object of the Student class. The John object might have Student ID Number being “012345”, Name being “John Brown” and GPA being “3.9”. Sending the Show GPA message to the object called John would cause John's GPA to be printed on the screen

Members of a Class:
Attributes and Methods

- The data elements in a class are called attributes
- The operations than can be performed by a class are called methods
- In the previous example:
- - Student ID Number, Name and GPA are all attributes of the Student class
- - Show GPA is a method in the Student class

Members of a Class:
Attributes and Methods

- Attributes give an object its state
- Methods give an object its behaviour

State

At a particular moment in time, the John object might contain the Values:

Student ID Number: 012345
Name: John Brown
GPA: 3.9

Behaviour

When the Show GPA message is sent to the John Object

The Show GPA method displays the GPA for the student John Brown:

GPA: 3.9

Abstraction, Encapsulation and Data Hiding

- The Student class is an abstraction for a real-world entity called a Student (a real person who attends a school or is involved in learning). Unnecessary details such as hair colour and blood type are excluded from the abstraction
- The attributes and methods of the Student class are encapsulated in the class, effectively hiding the data and structure of the class. The outside world can interact with the class by interacting with its interface – that is by sending messages to the class through its methods and obtain behaviour

The Four Types of Methods

There are four (4) types of methods that a class can have, namely:

- Constructors
- Destructor
- Accessors
- Mutators

Sometime a fifth type is included:

- Utility

Constructor Methods

- A constructor is a special type of method that is used to initialize the attributes in an object
- It initializes the state of the object
- It is called automatically when a new object is instantiated (defined and created)
- Has the same name as the class of the object
- Can automatically call constructors for inherited classes

Destructor Method

- A destructor is a special type of method that is automatically called when an object is destroyed (removed from memory)
- It is used by the programmer to free any memory previously allocated by the object and any other operation that must be performed when the object is destroyed
- It has the same name as the class of the object

Accessor Methods

- An accessor is a type of method that is used to access the overall state of an object, or a particular attribute in the object
- Accessors are sometimes called “get” methods since they get, display or return the value of an attribute

Accessor Methods

- Accessors are necessary because in keeping with OOP principles, attributes are normally hidden inside the class away from the outside world, so when the outside world needs to know the value of the attribute, the accessor provides it, without given direct access to the attribute itself.
- The accessor can perform checks to see if access should be granted to the user to get the value of the attribute. They retrieve an attribute's value while preventing it from being changed

Mutator Methods

- A mutator is a type of method that is used to set one or more attributes in an object
- Mutators are sometimes called “set” methods since they set or change the value of an attribute
- Mutators can perform data validation before setting the attribute to the specified value

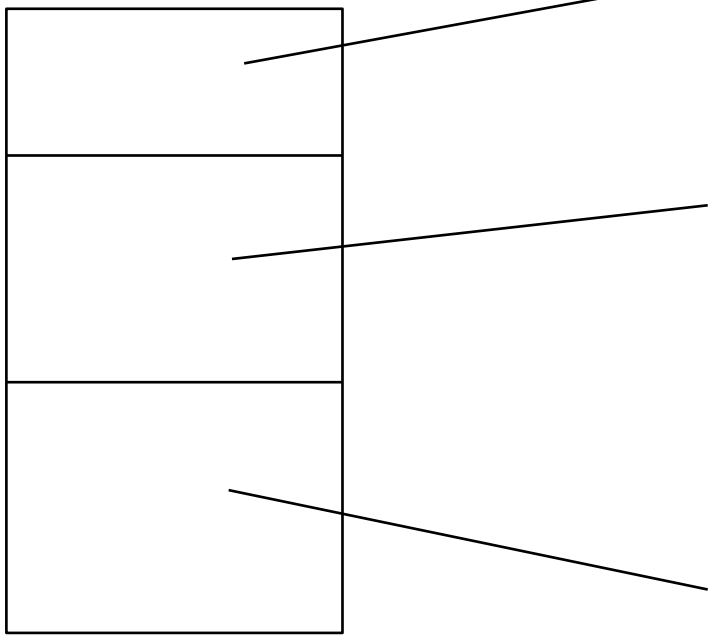
Utility Methods

- Sometimes a helper method is needed by a class. This helper method supports the work of the public (exposed) methods of the class
- Such a method is called a utility method
- This method is not available for direct use by outside users of the class
- However, friends of the class can call the method

The Three Compartments of a UML Class Diagram

A UML class diagram is a rectangle with three sections as portrayed below

A UML Class Diagram



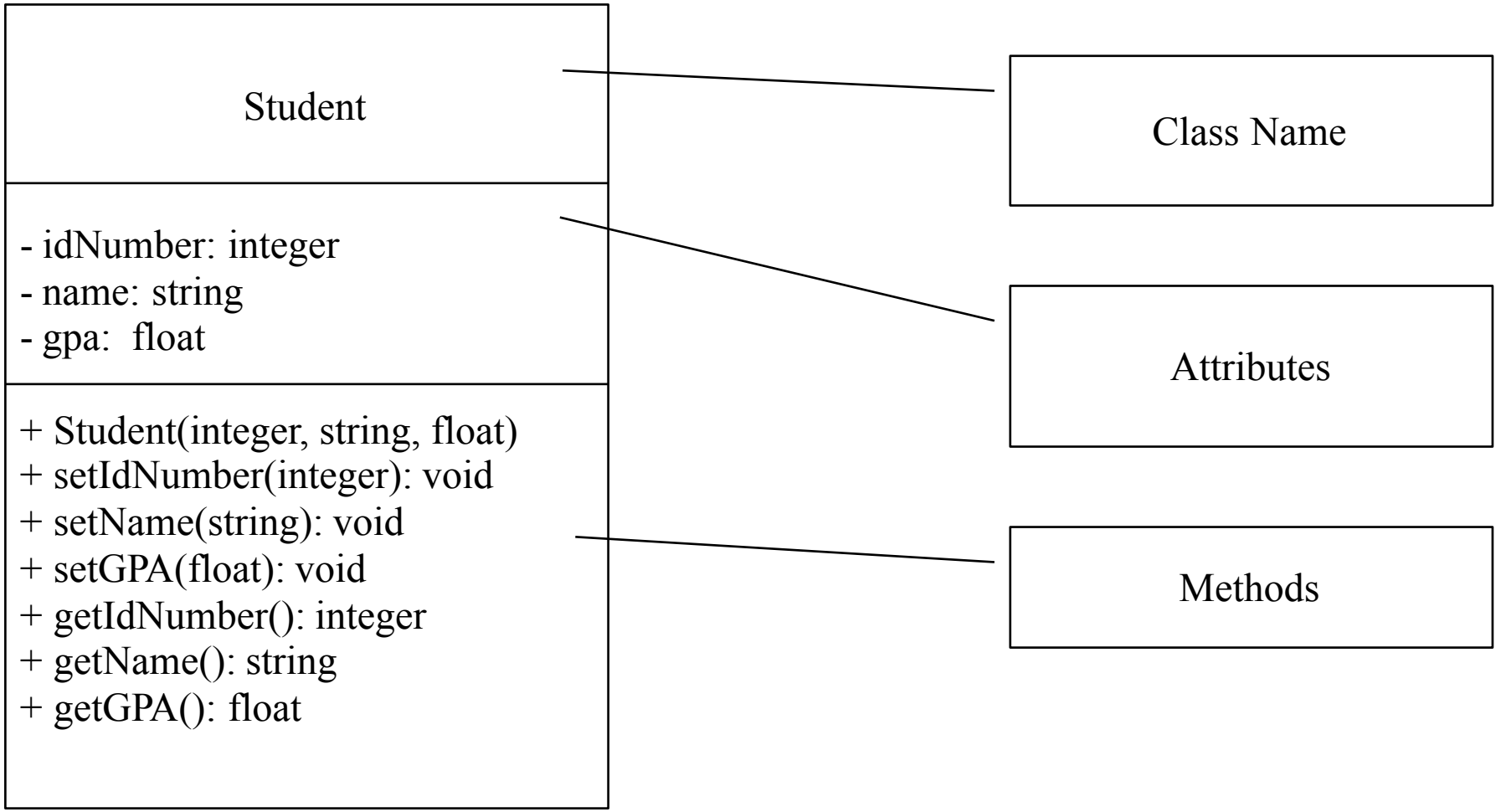
The name of the class goes in the top section. The name of the class is centered.

The attributes of the class are placed in the middle section. The data type and access specifier of each attribute may be specified along with each attribute

The methods are placed into the bottom section. Each method may be specified with its access specifier, parameters, and return type

The Three Compartments of a UML Class Diagram

A sample UML class diagram



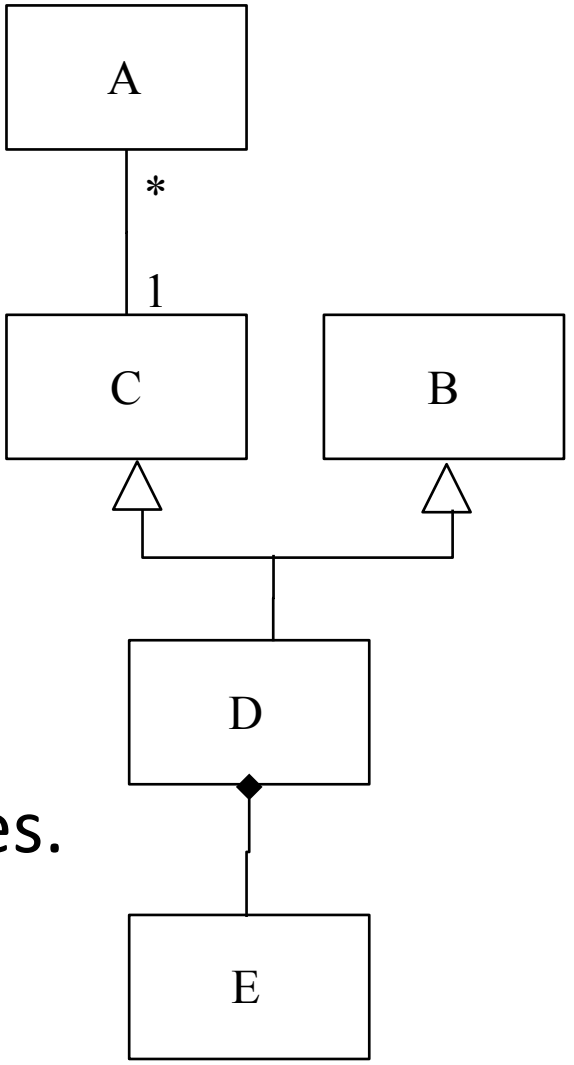
Types of Relationships among Classes

Relationships between classes can be depicted with UML a relationship diagram

Types of relationships among classes are:

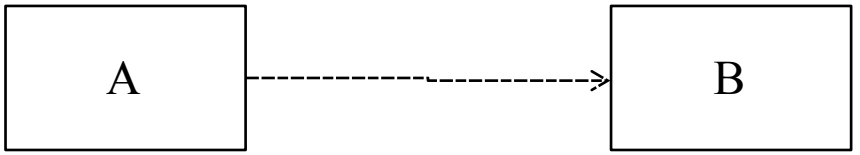
- Dependency
- Association
- Generalization
- Realization

A solid or dashed line linking two classes depicts a relationship between those classes. Sometimes the line has numbers, an arrow or a diamond at one end



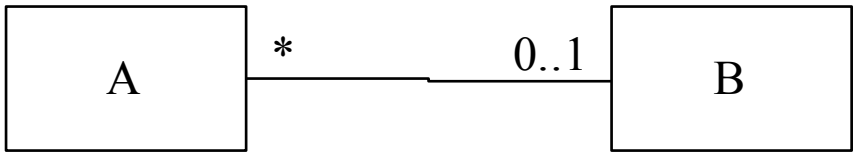
Dependency Relationships

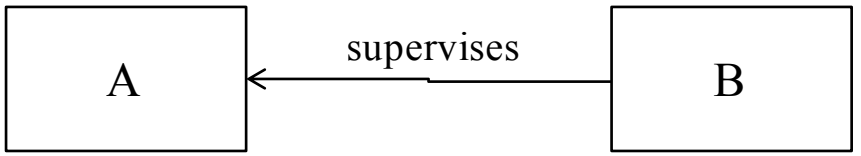
- In a dependency, one class A, depends on or uses another class B, but class A does not contain an instance of class B in itself.
- For example, occurs when a method in class A accepts an object of class B as a parameter
- If the interface to class B changes, this might affect class A, since A depends on B
- Represented by dashed line connecting the classes, with optional arrow point to class that is depended on



Association Relationships

- An association depicts a link between classes
- That link is represented by a solid line
- Multiplicity symbols may be placed on both ends of the line to how many objects of one class are connected with the other class

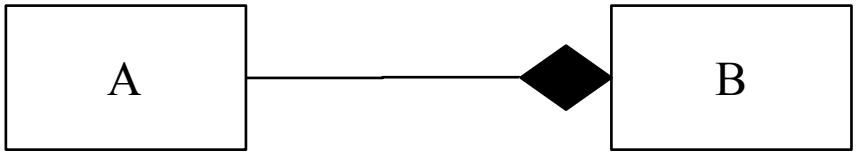




Multiplicity	**Description**
1	One
0..1	Zero to one
n	Exactly n
m..n	m to n
*	Zero to infinity
1..*	One to infinity

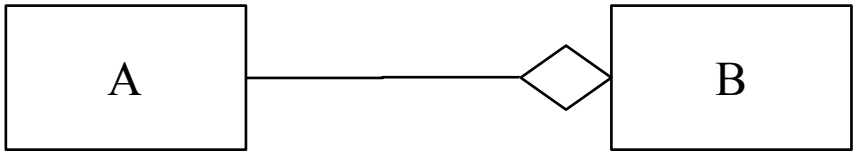
Composition Relationships

- A composition relationship is a type of Association relationship
- In this type of relationship, one class A is contained inside another class B (B has an A)
- For this reason, it is called a "has-a" relationship
- Every time an object of B is created, an object of class A is created as well e.g. (B) Student (A) Id#
- Composition is represented by joining both classes with a solid line and placing a shaded diamond on the end of the "container" class



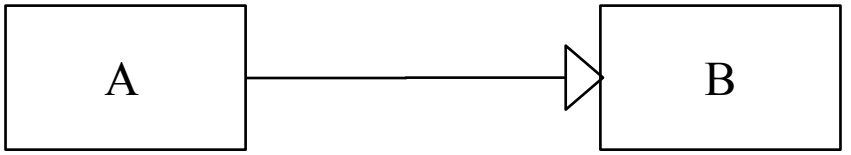
Aggregation Relationships

- An aggregation relationship is also a type of Association relationship
- In this type of relationship, one class A is may or may not be contained inside another class B
- Sometimes B can have an A
- But there are times when a B can exist with out having an A, e.g. (B) Student and (A) Cell Phone
- Aggregation is represented by joining both classes with a solid line and placing a unshaded diamond on the end of the "container" class



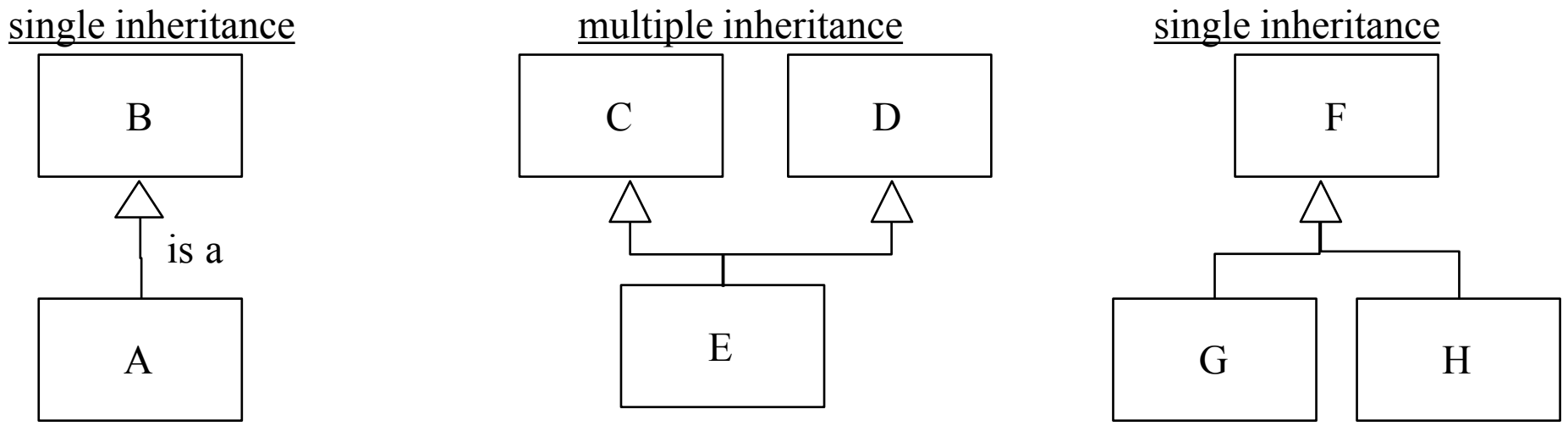
Generalization Relationships

- A generalization relationship specifies that a class is the parent of another class
- The parent class is also called the Generalized class or Base class
- The child class is also called the Specialized class or the Derived class
- Specified in UML by connecting the parent and child classes with a solid line and placing an unshaded triangle on the parent end, making B the parent of A in the example below



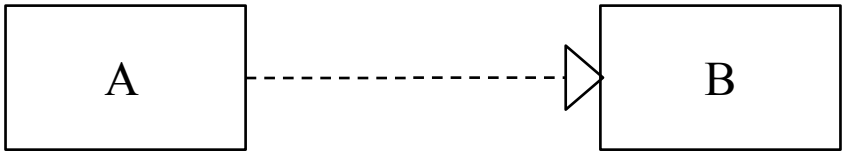
Generalization Relationships

- A generalization relationship is also called an inheritance relationship because the child classes inherit the members of the parent class
- A generalization is also called an "is-a" relationship e.g. A is a B below
- Can have single-inheritance (one parent) and multiple-inheritance (multiple parents)



Realization Relationships

- A realization relationship exists between two model elements when one of them must realize, or implement, the behavior that the other specifies.
- The model element that specifies the behavior is the supplier, and the model element that implements the behavior is the client.
- Specified in UML by connecting the client and supplier with a dashed line and placing an unshaded triangle on the supplier end, making B the supplier and A the client in the example below



Modeling classes derived from OOA

Sample Requirements

An office worker has an employee number, a first name, last name, date employed, and telephone number. The date employed contains the day, month and year. Each telephone number has an area code, a prefix and a sequential number. A receptionist is a type of office worker that has an extension number. A supervisor is a type of office worker that has a numeric supervisory code.

Modeling classes derived from OOA

Underlining the nouns in noun-verb analysis

An office worker has an employee number, a first name, last name, date employed, and telephone number. The date employed contains the day, month and year. Each telephone number has an area code, a prefix and a sequential number. A receptionist is a type of office worker that has an extension number. A supervisor is a type of office worker that has a numeric supervisory code.

Modeling classes derived from OOA

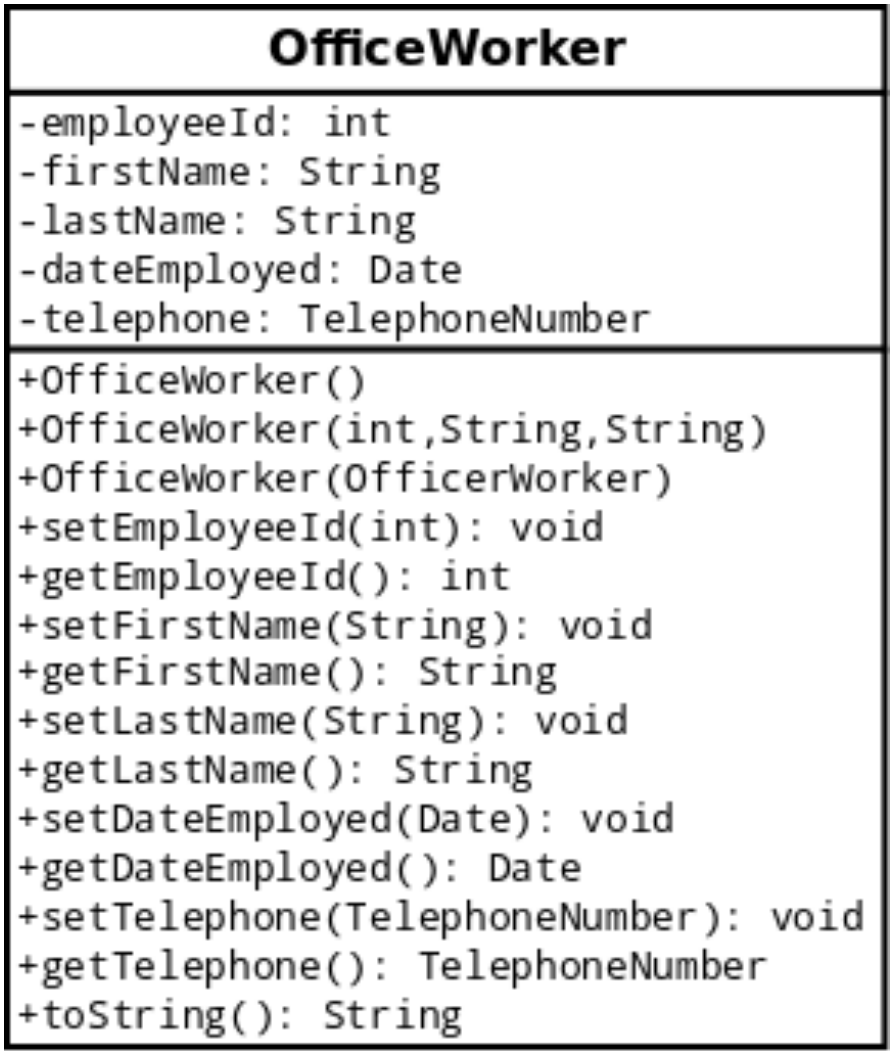
Resulting classes and attributes

- class **OfficeWorker** contains attributes:
- employee number, first name, last name
- class **Date** contains attributes:
- day, month and year
- class **TelephoneNumber** contains attributes:
- area code, prefix and sequential number
- Class **Receptionist** contains attribute:
- extension number
- Class **Supervisor** contains attribute:
- supervisory code.

Modeling classes derived from OOA

Resulting classes and attributes

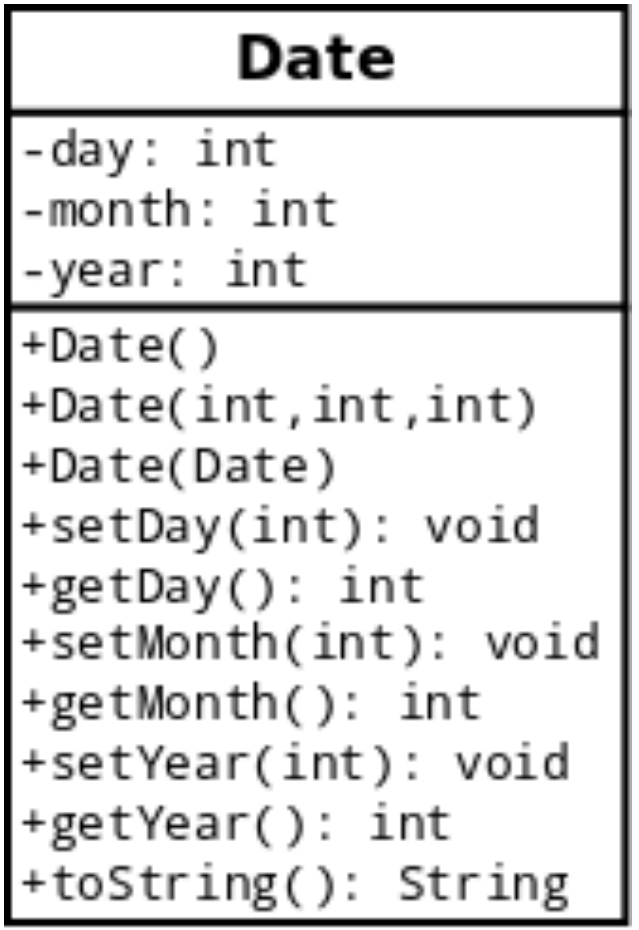
- class **OfficeWorker** contains attributes:
- employee number, first name, last name
- class **Date** contains attributes:
- day, month and year
- class **TelephoneNumber** contains attributes:
- area code, prefix and sequential number
- Class **Receptionist** contains attribute:
- extension number
- Class **Supervisor** contains attribute:
- supervisory code.



Modeling classes derived from OOA

Resulting classes and attributes

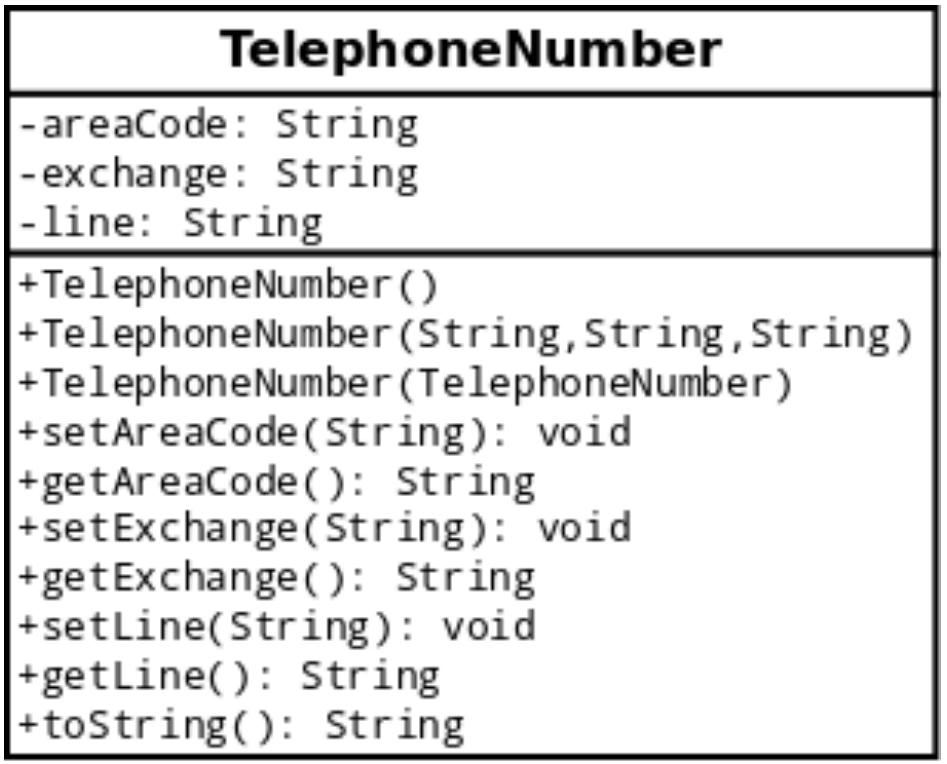
- class **OfficeWorker** contains attributes:
- employee number, first name, last name
- class **Date** contains attributes:
- day, month and year
- class **TelephoneNumber** contains attributes:
- area code, prefix and sequential number
- Class **Receptionist** contains attribute:
- extension number
- Class **Supervisor** contains attribute:
- supervisory code.



Modeling classes derived from OOA

Resulting classes and attributes

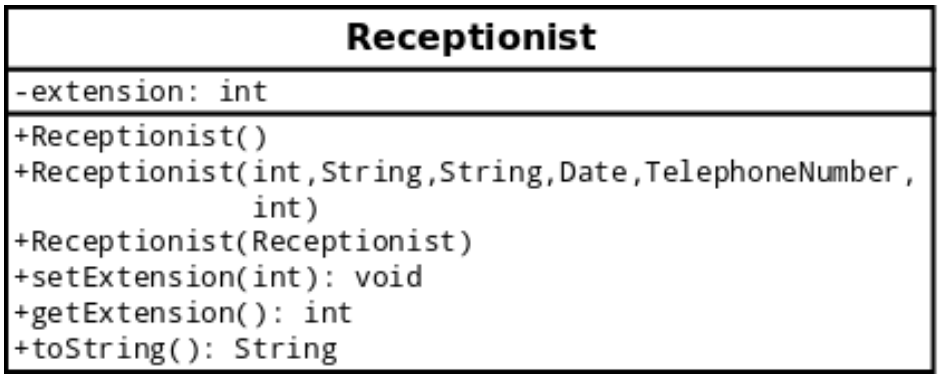
- class **OfficeWorker** contains attributes:
- employee number, first name, last name
- class **Date** contains attributes:
- day, month and year
- class **TelephoneNumber** contains attributes:
- area code, prefix and sequential number
- Class **Receptionist** contains attribute:
- extension number
- Class **Supervisor** contains attribute:
- supervisory code.



Modeling classes derived from OOA

Resulting classes and attributes

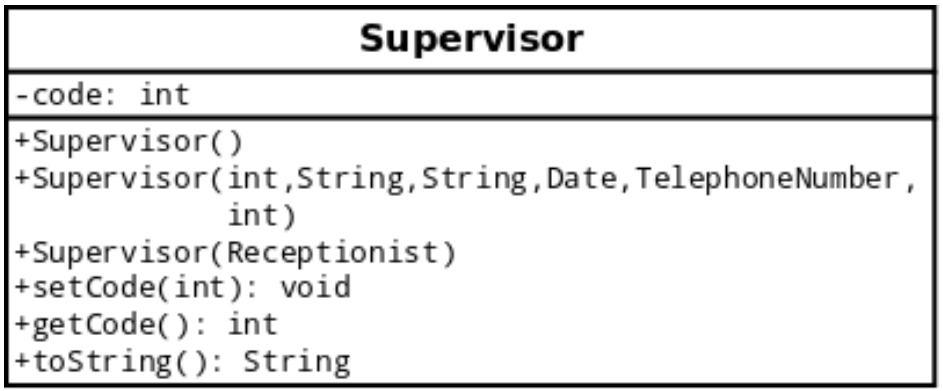
- class **OfficeWorker** contains attributes:
- employee number, first name, last name
- class **Date** contains attributes:
- day, month and year
- class **TelephoneNumber** contains attributes:
- area code, prefix and sequential number
- Class **Receptionist** contains attribute:
- extension number
- Class **Supervisor** contains attribute:
- supervisory code.



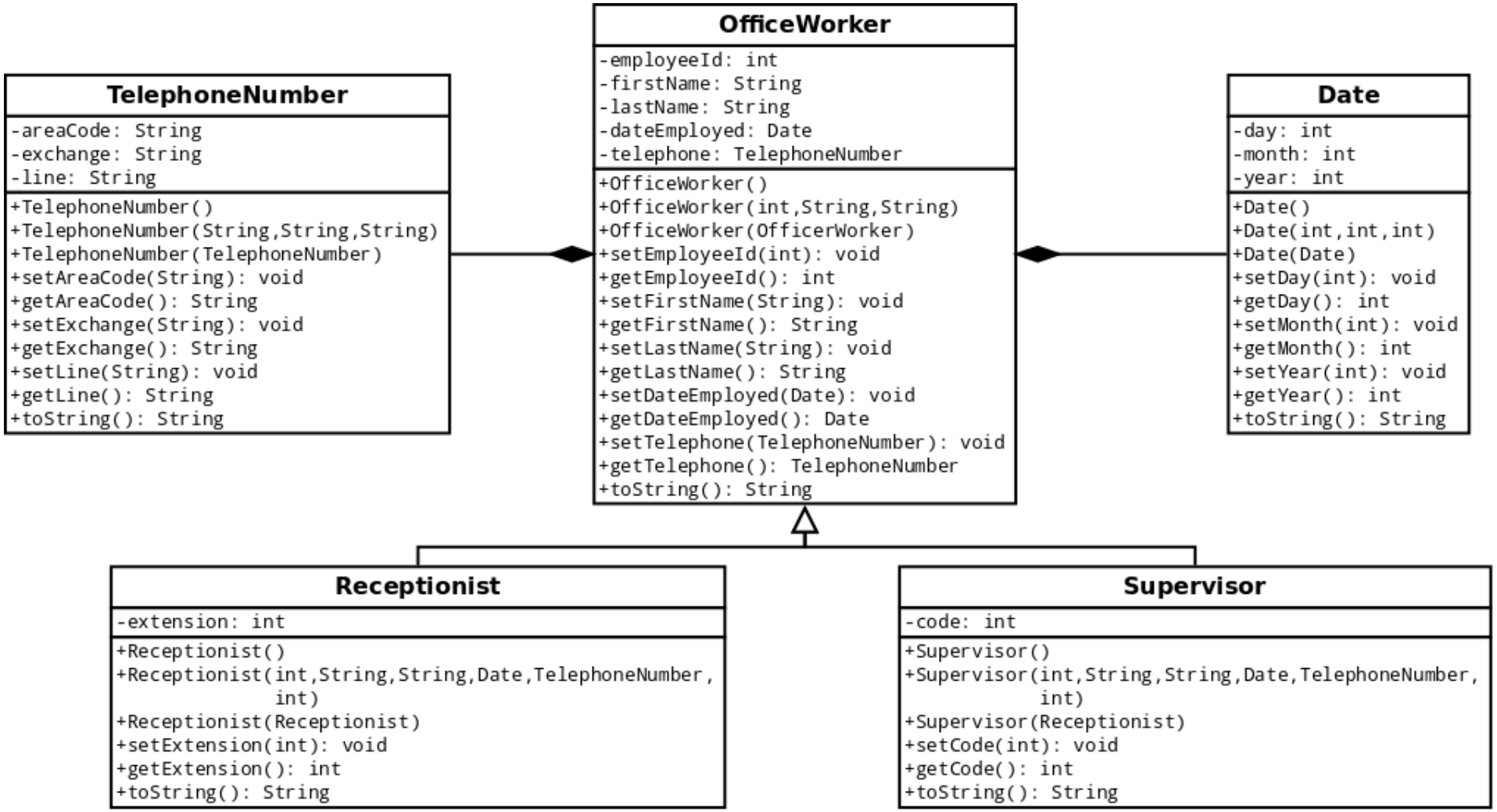
Modeling classes derived from OOA

Resulting classes and attributes

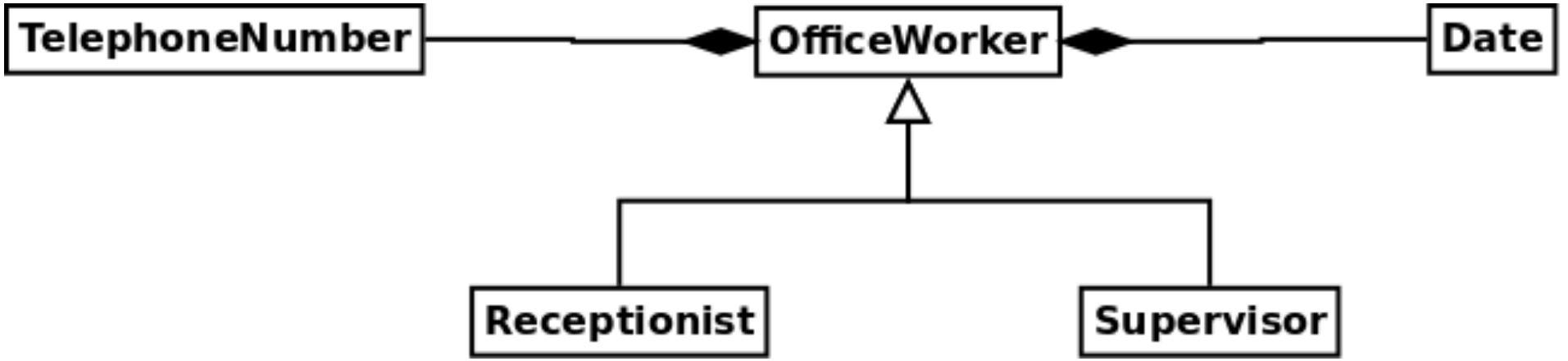
- class **OfficeWorker** contains attributes:
- employee number, first name, last name
- class **Date** contains attributes:
- day, month and year
- class **TelephoneNumber** contains attributes:
- area code, prefix and sequential number
- Class **Receptionist** contains attribute:
- extension number
- Class **Supervisor** contains attribute:
- supervisory code.



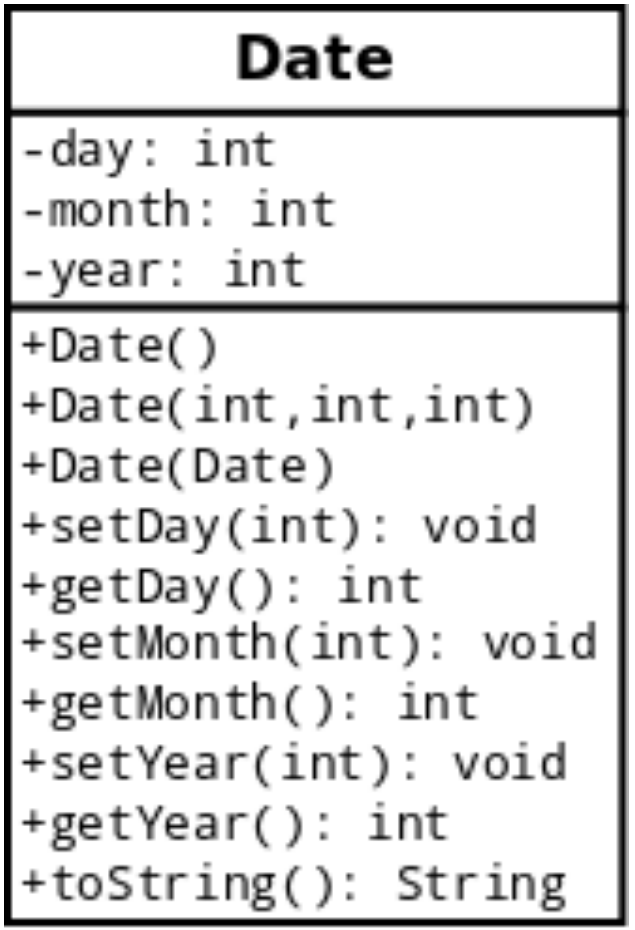
Represent relationships among classes using a UML relationship diagram



Represent relationships among classes using a UML relationship diagram

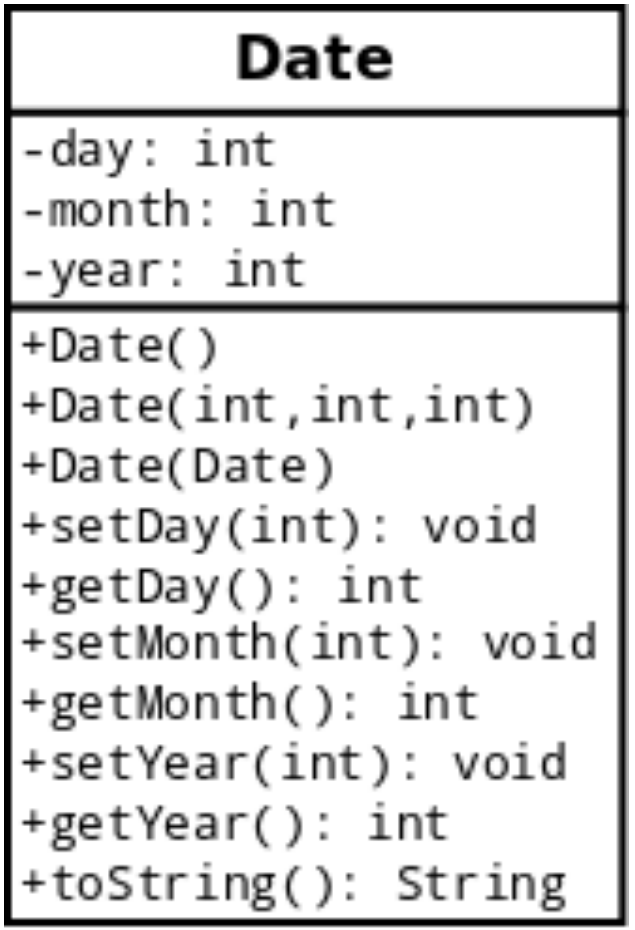


Convert Class Diagram into Code: C++



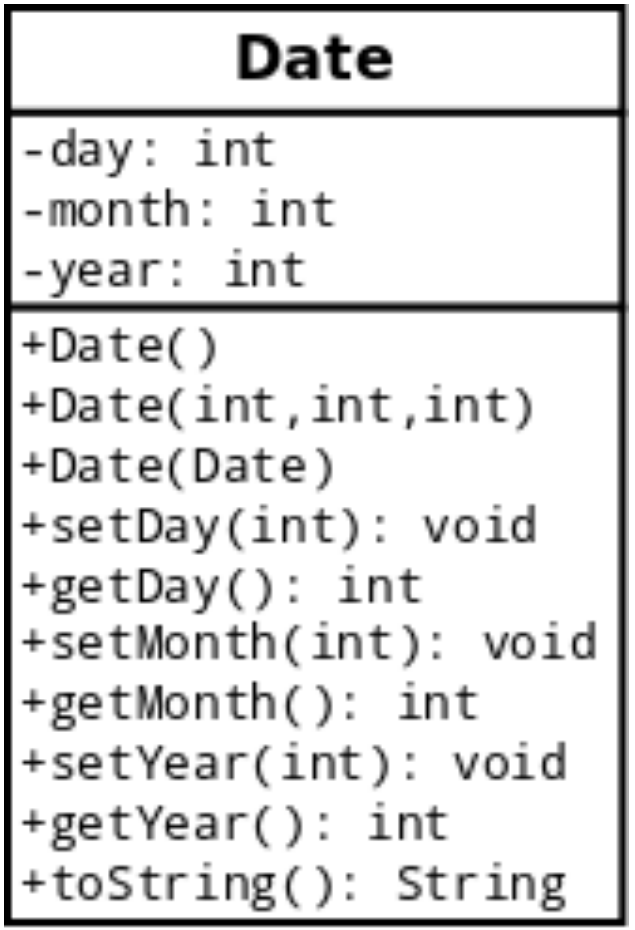

```
class Date{

   private:

      int day;

      int month;

      int year;


   public:

      // Default constructor

      Date(){

         day = 1;

         month = 1;

         year = 1900;

      }
```

Convert Class Diagram into Code: C++



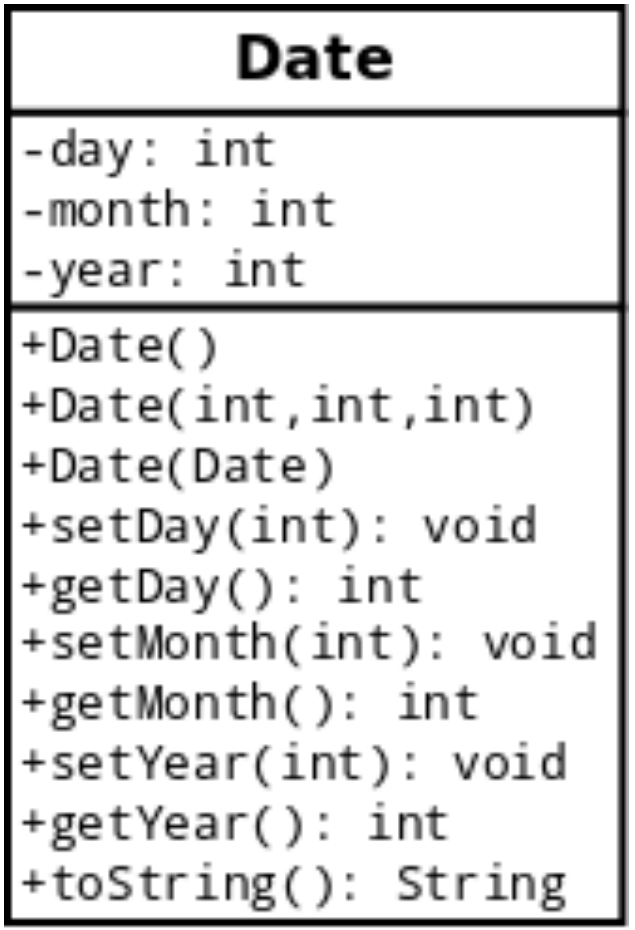

```
// Primary Constructor
Date(int day, int month, int year){
   this->day = day;
   this->month = month;
   this->year = year;
}

// Copy Constructor
Date(const Date & date){
   day = date.day;
   month = date.month;
   year = date.year;
}
```

Convert Class Diagram into Code: C++



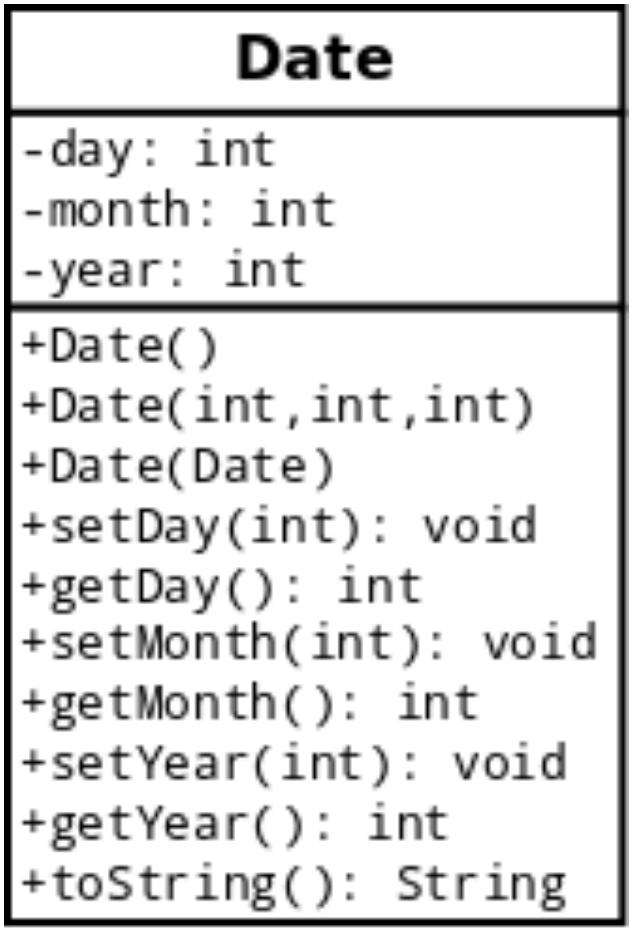

```
// Mutators
void setDay(int day){
   this->day = day;
}

void setMonth(int month){
   this->month = month;
}

void setYear(int year){
   this->year = year;
}
```

Convert Class Diagram into Code: C++



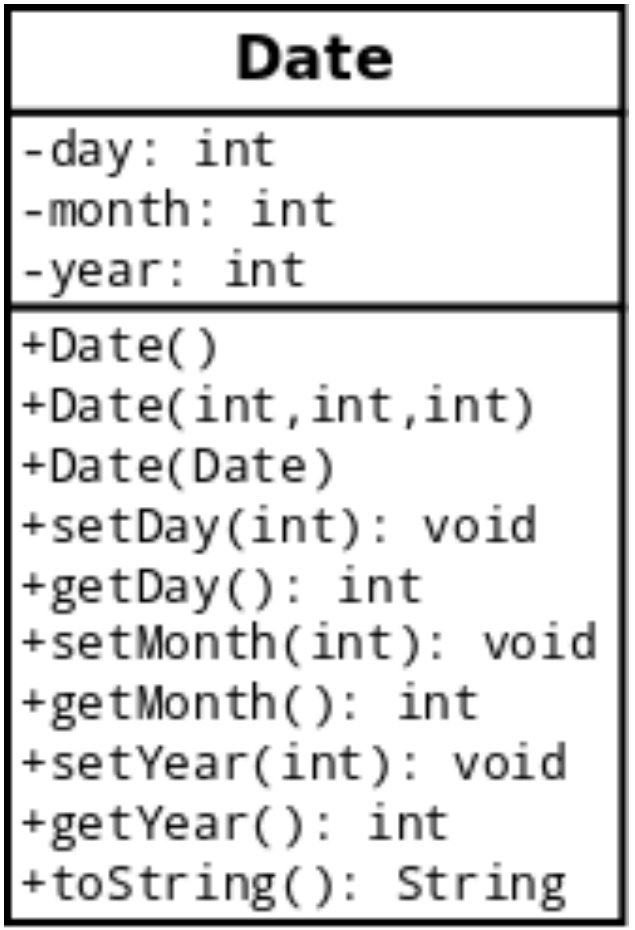

```
// Accessors
int getDay() const {
   return day;
}

int getMonth() const {
   return month;
}

int getYear() const {
   return year;
}
```

Convert Class Diagram into Code: C++



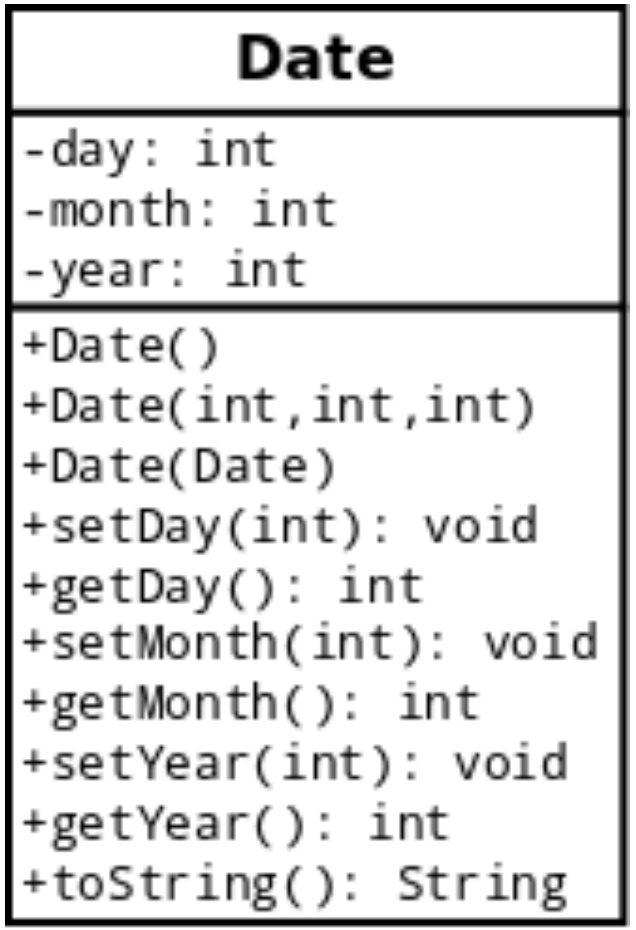

```
        /* A method to show the state
         * of the object, formatted
         * as user friendly output
         */
        void show() const {
           std::cout << day << "/"
                     << month << "/"
                     << year;
        }
};// Never forget this closing
  // semicolon that terminates the
  //class definition
```

Convert Class Diagram into Code: Java



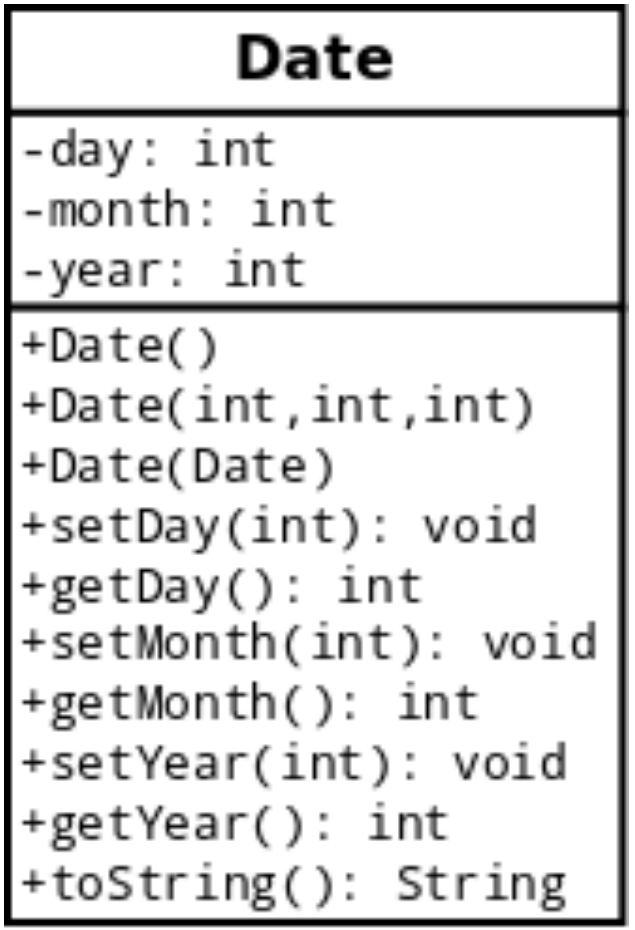

```
public class Date{
    private int day;
    private int month;
    private int year;

    // Default constructor
    public Date(){
        day = 1;
        month = 1;
        year = 1900;
    }
```

Convert Class Diagram into Code: Java



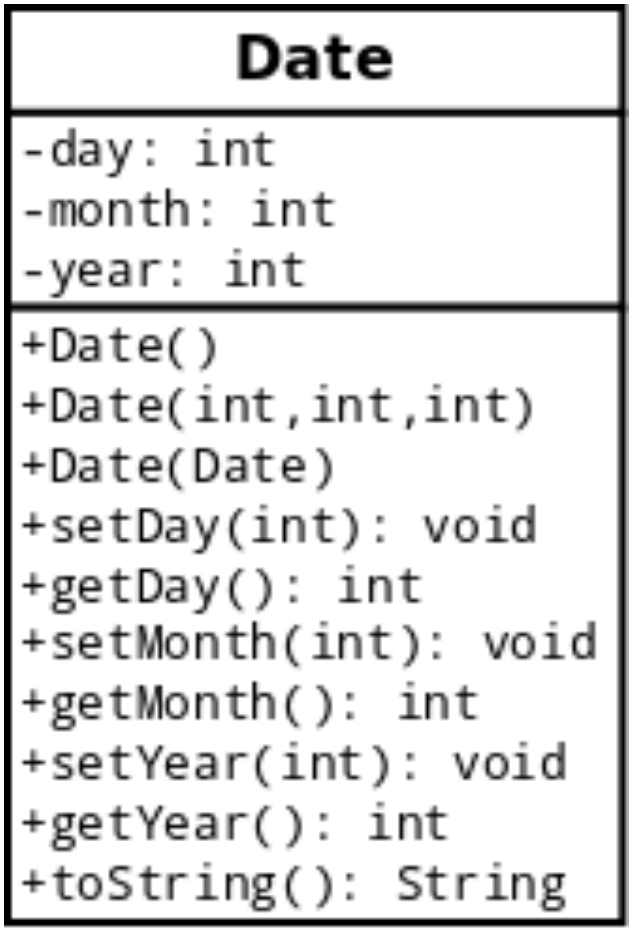

```
// Primary Constructor
public Date(int day, int month, int year)
{
    this.day = day;
    this.month = month;
    this.year = year;
}

// Copy Constructor
public Date(Date date){
    day = date.day;
    month = date.month;
    year = date.year;
}
```

Convert Class Diagram into Code: Java



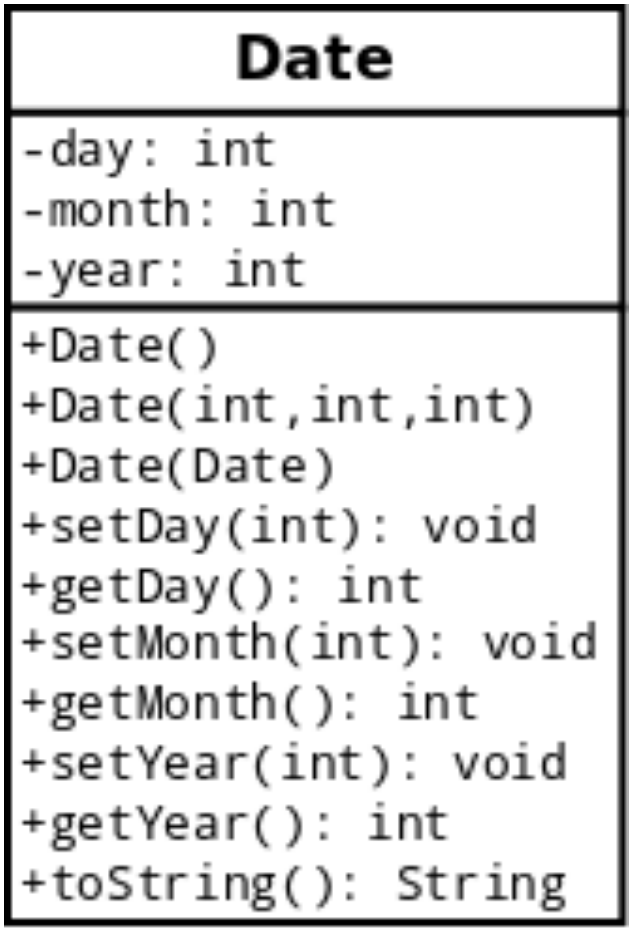

```
// Mutators
public void setDay(int day){
   this.day = day;
}

public void setMonth(int month){
   this.month = month;
}

public void setYear(int year){
   this.year = year;
}
```

Convert Class Diagram into Code: Java




```
// Accessors
public int getDay() {
    return day;
}

public int getMonth() {
    return month;
}

public int getYear() {
    return year;
}
```

Convert Class Diagram into Code: Java




```
    public String toString() {
        String out;
        out = day + "/";
        out += month + "/";
        out += year;
        return out;
    }
}// end class Date
```

Define object-based programming

- Sometimes a language supports encapsulation of attributes and methods in classes and objects but does not support inheritance and polymorphism
- These programming languages are said to be object-based but not object-oriented
- In object-based languages sometimes classes are usually already provided so the programmer can instantiate object and call their methods e.g. VB6 but not VB.net