ÖZYEĞİN UNIVERSITY

CS105
Introduction to Object-Oriented Programming

Prof. Dr. Nizamettin AYDIN

naydin@itu.edu.tr

nizamettin.aydin@ozyegin.edu.tr

Classes

Outline

- Object-Oriented Languages
  - Simula
- What Exactly Is an Object?
- Object Data and Object Behavior
- Methods
- What is a class?
- UML Diagrams
- Basic Characteristics of OO Systems
  - Classes and Objects
  - Methods and Messages
  - Encapsulation and Information Hiding
  - Inheritance
  - Polymorphism and Dynamic Binding

Some Programming Languages

Domain-specific abstraction

Fortran

Cobol

General-purpose abstraction

Simula

Java

Direct mapping to hardware

C++

C++11

C#

Assembler

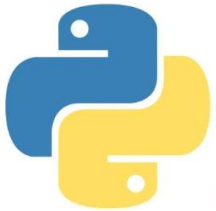
BCPL

C

Source: B.Stroustrup, The Essence of C++

Object-Oriented Languages

- OO Languages have been around since 60s.
  - First example is Simula (first stable release in 1967.)
  - Java
  - C#
  - Python
  - Ruby
  - PHP
  - TypeScript
  - Kotlin
  - R
  - Swift
  - C++
  - Julia
  - Go
  - Perl
  - Smalltalk
  - ...
    - https://en.wikipedia.org/wiki/List_of_object-oriented_programming_languages

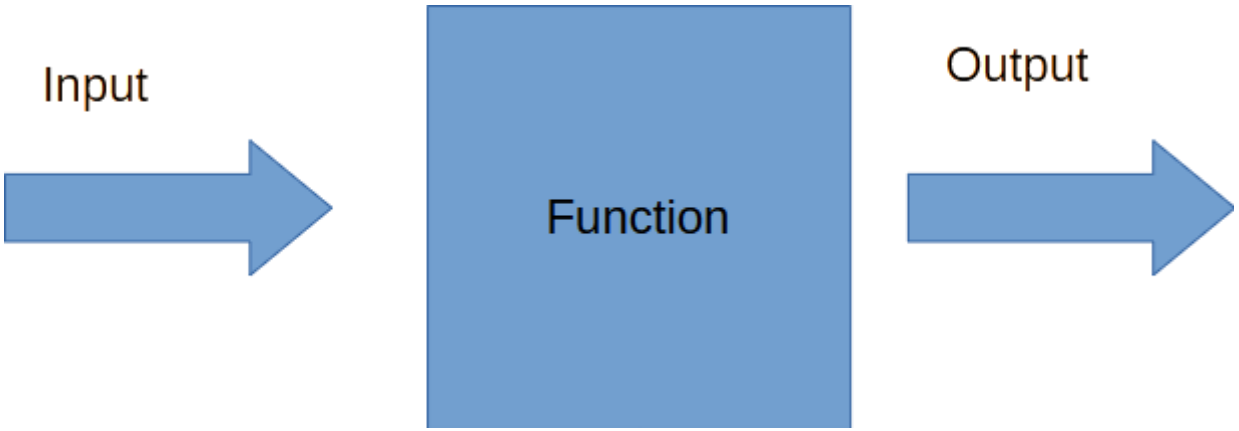


Simula

- the name of two simulation programming languages, Simula I and Simula 67,
  - developed in the 1960s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard.
  - Syntactically, it is an approximate superset of ALGOL 60
  - Simula 67 introduced objects, classes, inheritance and subclasses, virtual procedures, coroutines, discrete event simulation, and featured garbage collection.
  - considered the 1st object-oriented programming language.
  - the 1st Simula version by 1962 was designed for doing simulations;
  - Simula 67 though was designed to be a general-purpose programming language and provided the framework for many of the features of object-oriented languages today.

Procedural vs. OO Programming

- Are attributes and behaviors separate or combined?
- In procedural approach, data and methods are separate.



- In object-oriented approach, data and methods are combined and exist together within an object.

What Exactly Is an Object?

- Central concept in OO Paradigm.
- A program that uses OO technology is basically a collection of objects.

- A structure that both contains data and behaviors.

- To illustrate;
  - –consider that a corporate system contains objects that represent employees of that company. ”
    - Each of these objects is made up of the data and behavior.

Object Data and Object Behavior

- **Object Data**:
  - The data stored within an object represents the state of the object.
    - In OO programming terminology, this data is called attributes.
      - "For the example, employee attributes could be Social Security numbers, date of birth, gender, phone number, and so on."
      - The attributes contain the information that differentiates between the various objects, in this case the employees.
  - In daily language, we usually use adjectives to define these attributes hence state of real-life objects.
- **Object Behavior**:
  - The behavior of an object represents what the object can do.
    - In procedural languages the behavior is defined by procedures, functions, and subroutines.
  - Behavior is often called as methods.

Object Data and Object Behavior

- **Object Behavior**:
  - In OO programming terminology, these behaviors are contained in methods,
    - you invoke a method by sending a message to it.
      - In our employee example, consider that one of the behaviors required of an employee object is to set and return the values of the various attributes.
      - Thus, each attribute would have corresponding methods, such as setGender() and getGender().
      - In this case, when another object needs this information, it can send a message to an employee object and ask it what its gender is.
  - In daily language, we usually use verbs to define the methods of real-life objects.

- One of the most interesting/powerful advantages of using objects is that the data is part of the package—it is not separated from the code.

Methods

- The interface of methods are defined using the following concepts:
  - Name
  - Input parameters
  - Return type

- A method definition takes the general form:

```
<access-modifier> <return-type> method-name(<formal-parameters>)
{
      // Body of method
      <return-statement-if-not-void>
}
```

What is a class?

- a template from which distinct objects are derived.
- an organizational unit of an OO design and program
- A good class would exhibit the following characteristics:
  - Highly cohesive:
    - the class represents a single useful entity or organizational unit and does that job well.
  - Minimally coupled:
    - the class limits its interactions with other classes to only those that are really necessary for it to do what it is designed to do.
  - Encapsulation:
    - the class keeps information necessary to its internal operation private and does not expose it to other classes, and only makes public the information necessary for it to interact with other classes in the intended manner.

Classes & Objects

- You need a class to instantiate an object.
- Instantiation is analogous to making cookies from a cookie cutter.
  - The cookie cutter is the class which specifies the shape of each cookie
  - Cookies are objects, the values of their fields (e.g. color of buttons) may be different.
  - Each object has its own identity, but they are created from the same specification.

What is a class?

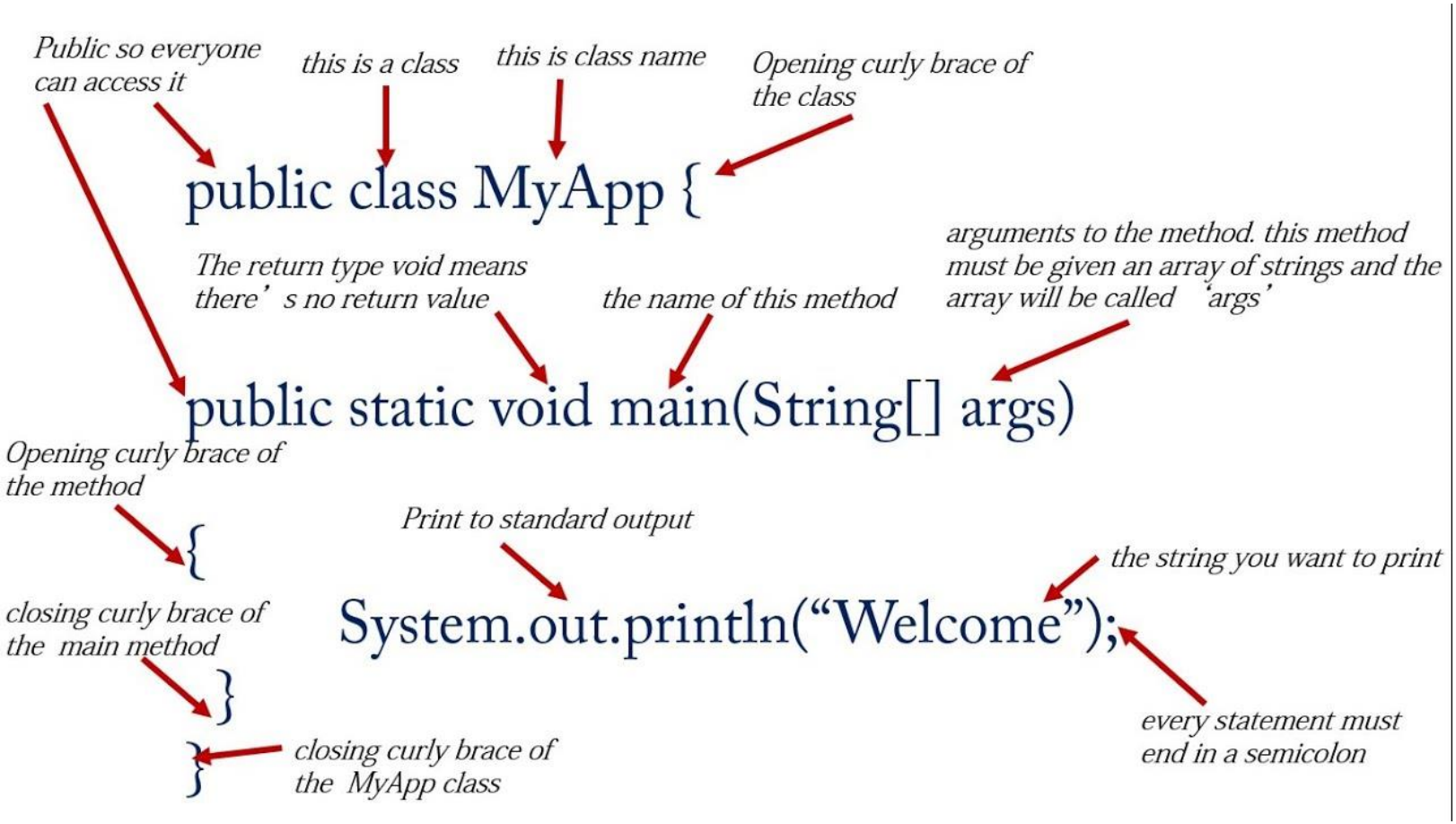
- In a general sense, any “thing”, “entity” or “class” (these words are often used interchangeably in many books and articles) can be defined by two key components:
  - The things that they “are” (the state)
  - The things that they “do” (the behaviour)

- For example, we can break the car entity into state and behavior as follows:

State	Behaviour
Colour (text)	Press the start button
Engine power (number of BHP)	Press the accelerator
Convertible? (yes/no)	
Parking brake (on/off)	

What is a class?

- There are three main fields in a class definition:
  - –Name, Attributes, Methods



UML Diagrams

- Unified Modeling Language (UML) is a visual modeling language that
  - provides developers with diagrams and methods to
    - visualize, create, and document software systems, and model the organizations that use these systems.
- participants participate in the development of software systems, each with a role
  - Analysts
  - Designers
  - Programmers
  - Testers
- Each of them is interested in different aspects of the system and needs a different level of detail.
- UML attempts to provide a highly expressive language so that participants can take advantage of system diagrams.

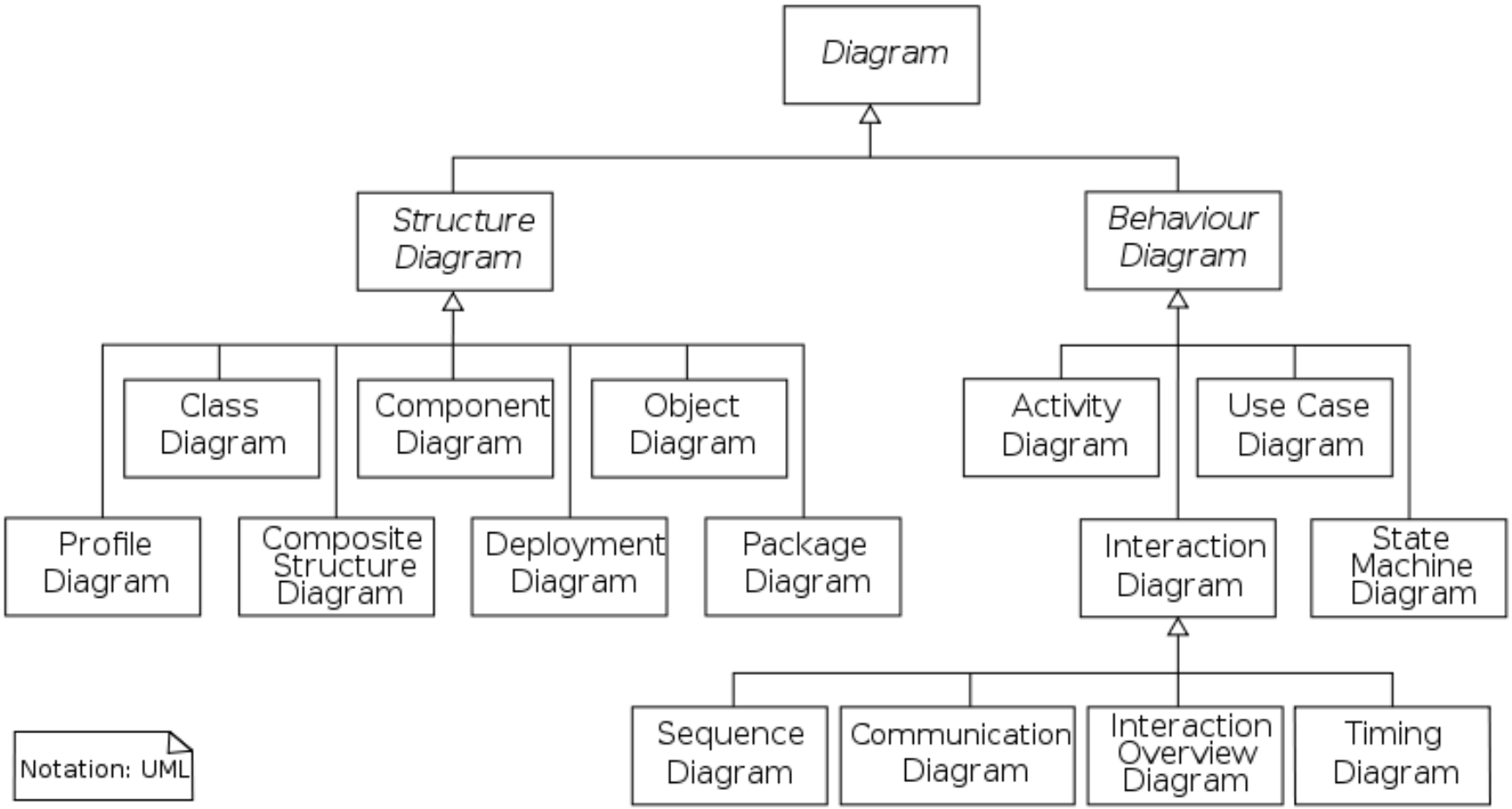
UML Diagrams

- UML consists of a set of diagrams representing the parts of a system.
- These diagrams are divided into two main categories:
- **Structure Diagrams**:
  - represent the architectural components of the system such as objects, relationships and the dependence of system elements on each other.
  - Class Diagram: A diagram shows the classes within the system and the relationships between them.
  - Package Diagram: It breaks the system down into smaller and easier-to understand parts, and this diagram allows us to model these parts.
  - Component Diagrams: to encode how the system is separated or partitioned and how each model depends on the other.
    - It focuses on the actual components of the program.

UML Diagrams

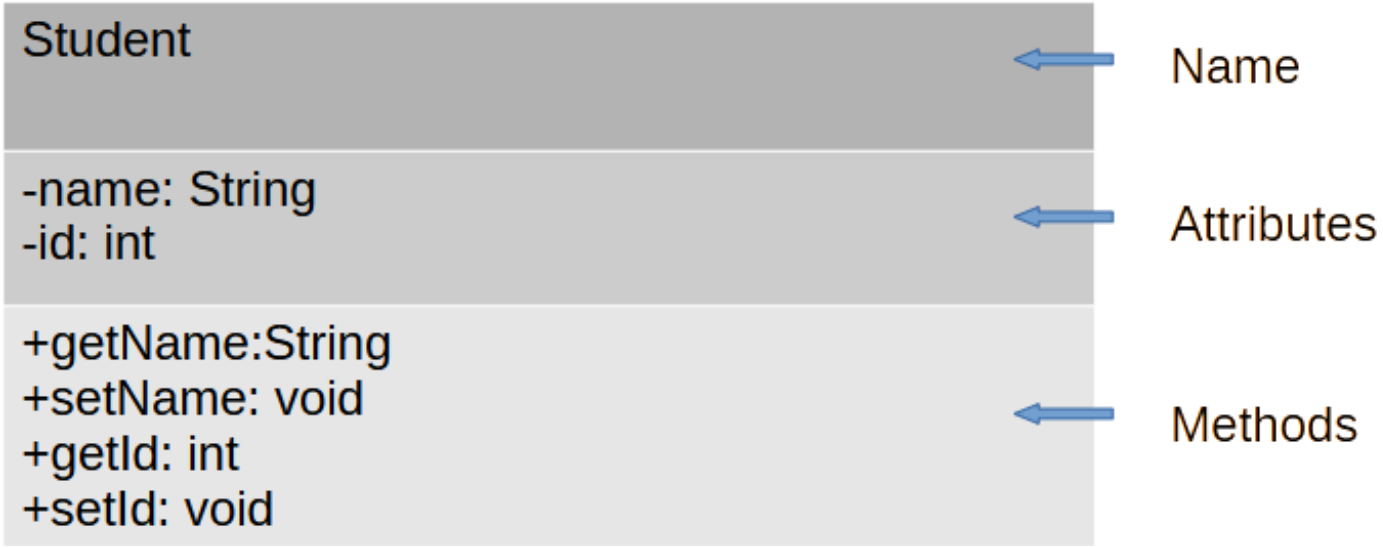
- **Behavior Diagrams**:
  - represent the dynamics of the system and the interaction between the system's objects and the changes that occur in the system.
  - Use Case Diagram: A description of the behavior of the system from the user's point of view.
  - Collaborative Diagram: to describe how the objects we build collaborate and use numbers to show the sequence of messages.
    - Also referred to as Communication diagrams in UML.
  - Sequence Diagrams: describes how objects in the system interact over time.
    - It displays the time sequence of the objects participating in the interaction
  - State Diagrams: to show the succession of transitions between different states of an organism during its life cycle and the events which force it to change its state.
  - Activity Diagram: shows a special case where most of the states are states and actions, and most transitions are triggered by the completion of the verbs of the source states.
    - This diagram focuses on internal processing-driven flows.

UML Diagrams



UML Diagrams

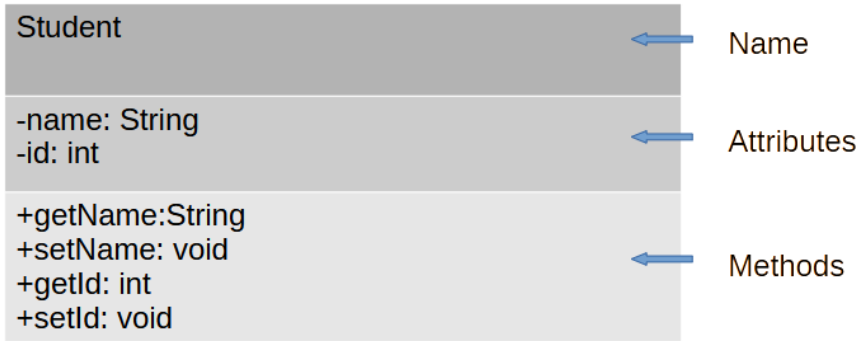
- UML diagrams are frequently used to model class designs.



- Class diagram
  - It is referred to as a static diagram because it describes the classes with their properties, methods, and their static relationships with each other, but it does not display the interactions between them.

Scope of attributes or methods

- The following symbols are used to define the scope of attributes or methods.
  - –These symbols show us the accessibility of the attribute or method within the class.
    - known as modifiers



  - –Public (**+**): Any element of the class or outside of the class can access this member (attributes or operation).
  - –Private (**-**): The member can only be accessed within the class itself.
  - –Protected access specifier (#): The property can only be accessed within the class or from another class that inherits this class.

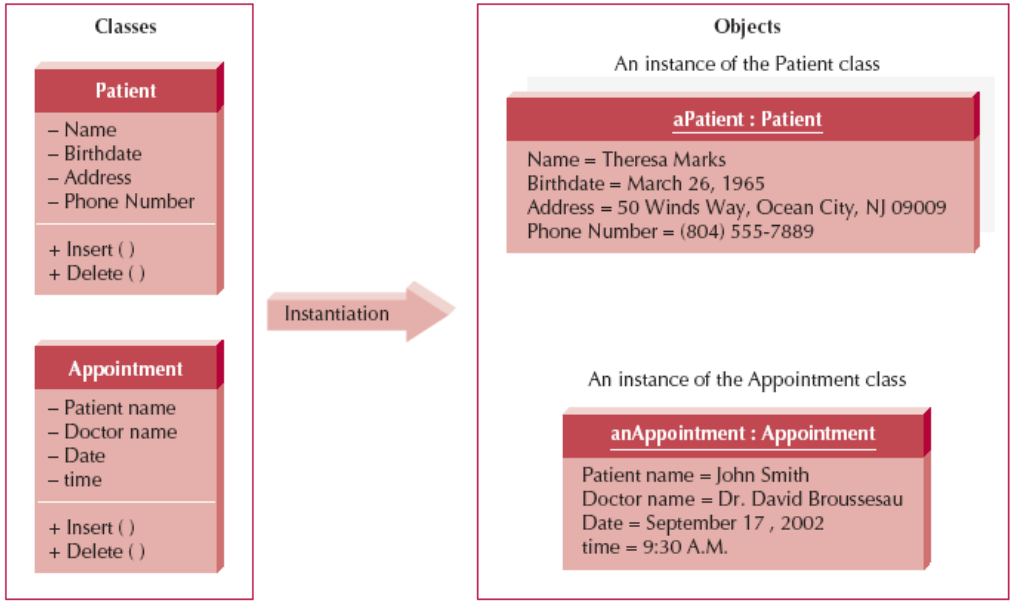
Basic Characteristics of OO Systems

- OO systems focus on capturing the structure and behavior of information systems in little modules that encompass both data and process.
- These little modules are known as objects.
- The basic characteristics of OO systems include
  - classes,
  - objects,
  - methods,
  - messages,
  - encapsulation,
  - information hiding,
  - inheritance,
  - polymorphism,
  - dynamic binding.

Classes and Objects

- **Class:**
  - –the general template we use to define and create specific instances, or objects.
- Every object is associated with a class.
  - –"For example, all the objects that capture information about patients could fall into a class called Patient, because there are
    - attributes (e.g., names, addresses, and birth dates)
    - methods (e.g., insert new instances, maintain information, and delete entries)

    that all patients share"



Classes and Objects

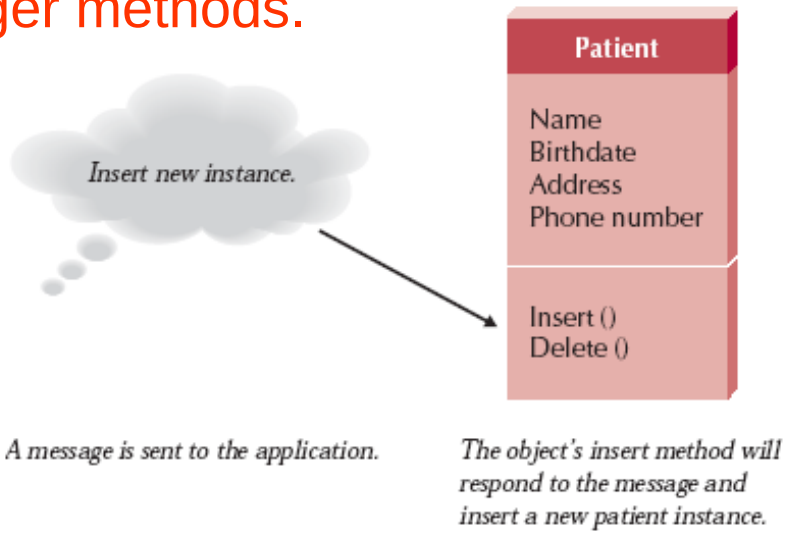
- **Object:**
  - an instantiation of a class.
  - a person, place, event, or thing about which we want to capture information.

- If we were building an appointment system for a doctor's office, classes might include doctor, patient, and appointment.

- The specific patients like Jim Maloney, Mary Wilson, and Theresa Marks are considered instances, or objects, of the patient class.

Methods and Messages

- **Methods:**
  - –implement an object's behavior.
    - nothing more than an action that an object can perform.
    - analogous to a function or procedure in a traditional programming language such as C, Cobol, or Pascal.
- **Messages:**
  - –information sent to objects to trigger methods.
  - –a function or procedure call from one object to another object.
    - "For example, if a patient is new to the doctor's office, the system will send an insert message to the application.
    - The patient object will receive a message (instruction) and do what it needs to do to go about inserting the new patient into the system."



Encapsulation and Information Hiding

- **Encapsulation:**
  - –the combination of process and data into a single entity.

- Traditional approaches to information systems development tend to be either
  - –process-centric
    - e.g., structured systems

  or
  - –data-centric
    - e.g., information engineering.

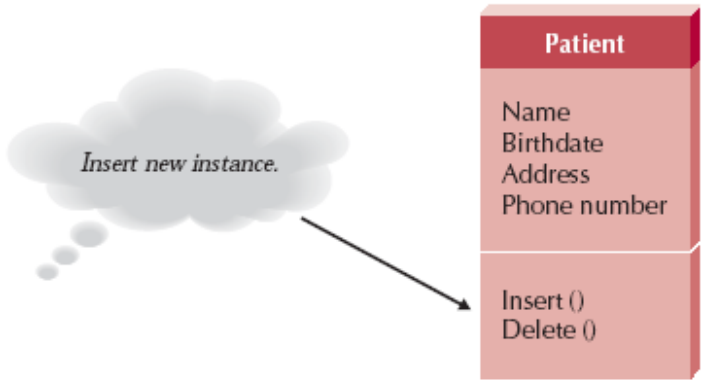
- Object-oriented approaches combine process and data into holistic entities (objects).

Encapsulation and Information Hiding

- **Information hiding:**
  - –first promoted in structured systems development.
  - –only the information required to use a software module is published to the user of the module.
    - This implies the information required to be passed to the module, and the information returned from the module is published.
    - How the module implements the required functionality is not relevant.
- In OO systems, combining encapsulation with the information hiding principle suggests that the information hiding principle be applied to objects instead of merely applying it to functions or processes.
  - –Objects are treated like black boxes.

Encapsulation and Information Hiding

- **Reusability Key:**
  - –Use an object by calling methods
    - because it shields the internal workings of the object from changes in the outside system, and it keeps the system from being affected when changes are made to an object.
  - –In the following figure, notice how a message (insert new patient) is sent to an object, yet the internal algorithms needed to respond to the message are hidden from other parts of the system.
  - –The only information that an object needs to know is the set of operations, or methods, that other objects can perform and what messages need to be sent to trigger them.



A message is sent to the application.

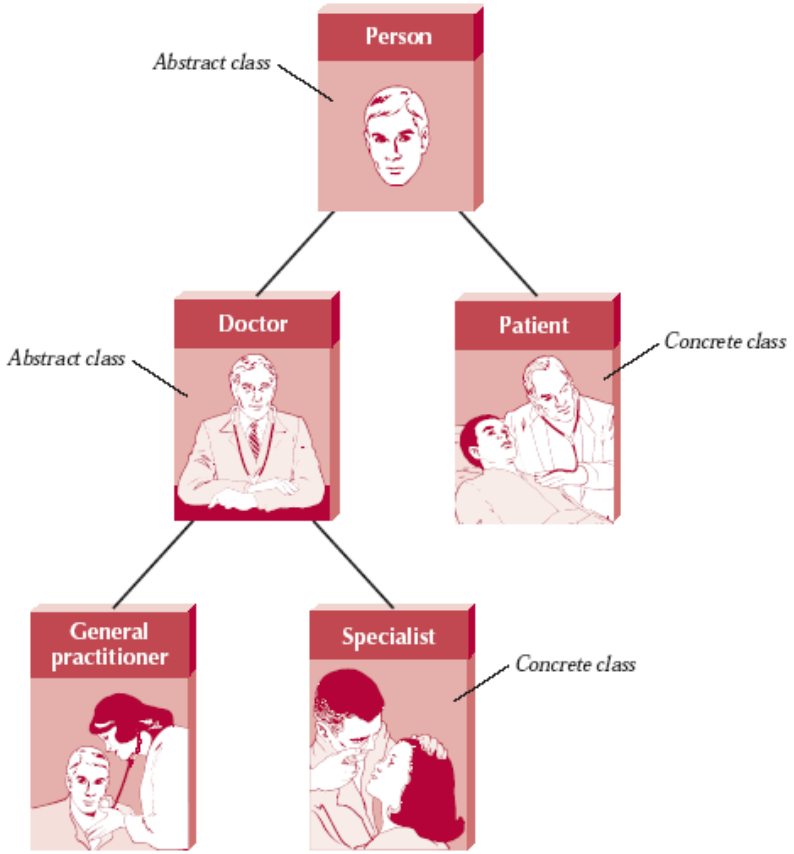
The object's insert method will respond to the message and insert a new patient instance.

Inheritance

- **Inheritance:**
  - –was proposed in data modeling in the late 1970s and the early 1980s.
- The data modeling literature suggests using inheritance to identify higher-level, or more general, classes of objects.
- Common sets of attributes and methods can be organized into superclasses.
- Typically, classes are arranged in a hierarchy whereby
  - –the superclasses, or general classes, are at the top,
  - –the subclasses, or specific classes, are at the bottom.
- It is useful for code reusability:
  - –reuse attributes and methods of an existing class when you create a new class.

Inheritance

- In the following figure,
  - person is a superclass to the classes Doctor and Patient.
  - Doctor, in turn, is a superclass to general practitioner and specialist.
- Notice how a class (e.g., doctor) can serve as a superclass and subclass concurrently.
- The relationship between the class and its superclass is known as the A-Kind-Of (AKO) relationship.
  - For example, in the figure,
    - a general practitioner is A-Kind-Of doctor, which is A-Kind-Of person.



Inheritance

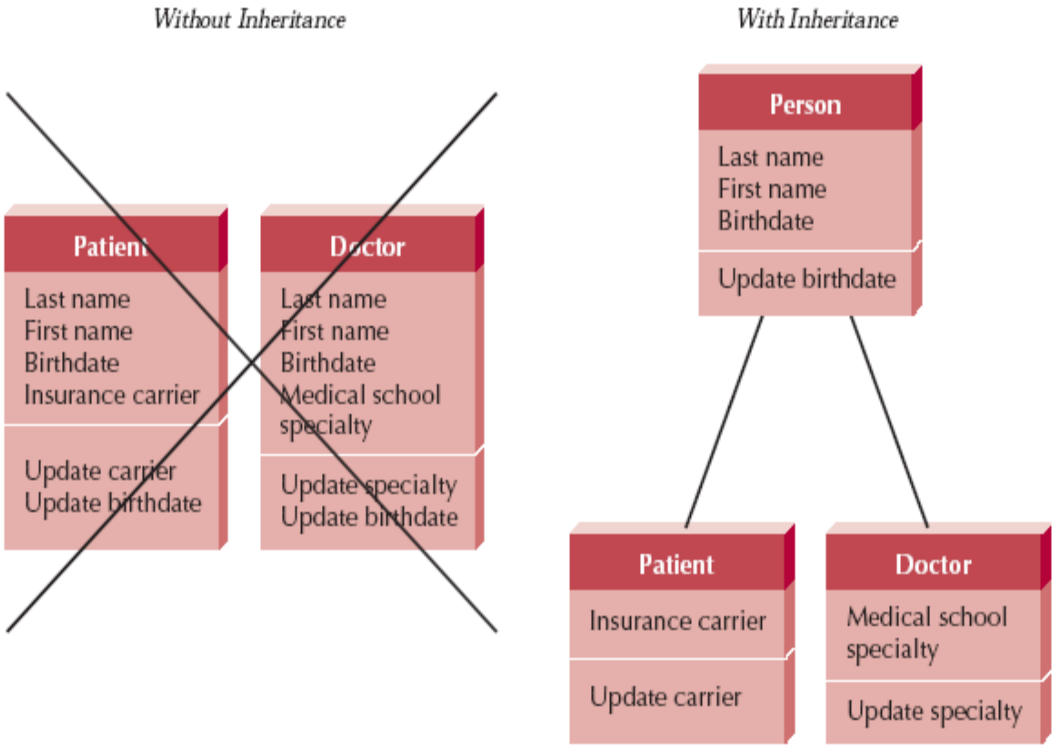
- Subclasses inherit the appropriate attributes and methods from the superclasses above them.

  - That is, each subclass contains attributes and methods from its parent superclass.
    - For example, previous figure shows that both doctor and patient are subclasses of person and therefore will inherit the attributes and methods of the person class.

  - Inheritance makes it simpler to define classes.
    - In previous figure, instead of repeating the attributes and methods in the doctor and patient classes separately, the attributes and methods that are common to both are placed in the person class and inherited by those classes below it.

Inheritance

- Notice how much more efficient hierarchies of object classes are than the same objects without a hierarchy in the following figure.



Polymorphism and Dynamic Binding

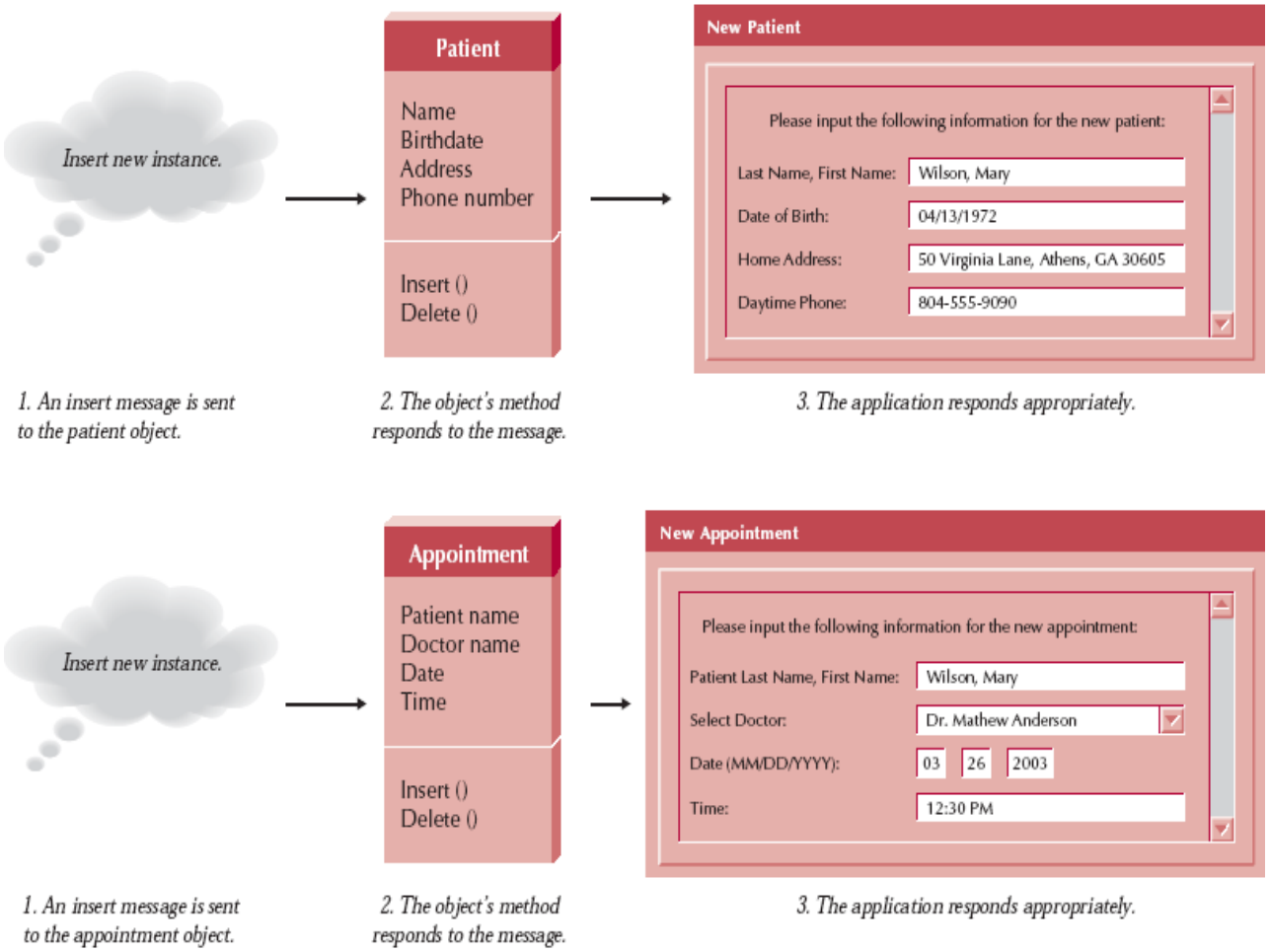
- **Polymorphism:**
  - –the same message can be interpreted differently by different classes of objects.
    - For example, inserting a patient means something different than inserting an appointment.
    - As such, different pieces of information need to be collected and stored.

- not have to be concerned with how something is done when using objects.

- simply send a message to an object, and that object will be responsible for interpreting the message appropriately.

Polymorphism and Dynamic Binding

- For example, if we sent the message “Draw yourself ” to a square object, a circle object, and a triangle object, the results would be very different, even though the message is the same.
- Notice in Figure how each object responds appropriately (and differently) even though the messages are identical.



Polymorphism and Dynamic Binding

- Polymorphism is made possible through dynamic binding (late binding)
  - a technique that delays typing the object until run-time.
    - As such, the specific method that is actually called is not chosen by the object-oriented system until the system is running.

- This is in contrast to static binding, in which
  - the type of object would be determined at compile time.
  - The developer would have to choose which method should be called instead of allowing the system to do it.
    - This is why in most traditional programming languages you find complicated decision logic based on the different types of objects in a system.

Polymorphism and Dynamic Binding

- For example, in a traditional programming language, instead of sending the message “Draw yourself ” to the different types of graphical objects in the previous figure, you would have to write decision logic using a case statement or a set of if statements to determine what kind of graphical object you wanted to draw, and you would have to name each draw function differently (e.g., drawsquare, draw-circle, or draw-triangle).
- This obviously would make the system much more complicated and more difficult to understand.

Any Questions?