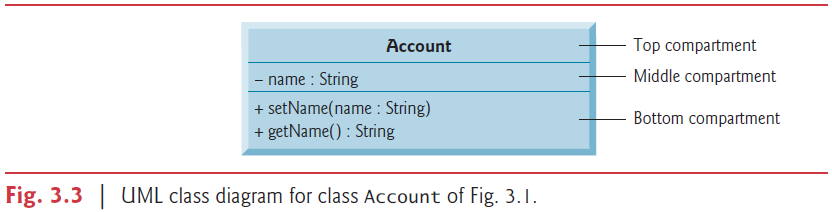
**Uml parts in books:** 55, 119, 126, 131, 146, 148, 149, 156, 200, 206, 212, 405,

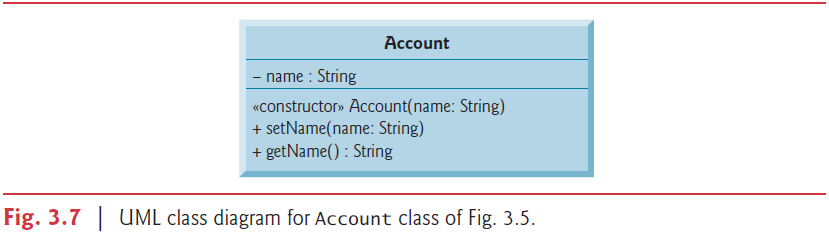
* **Chapter 3**

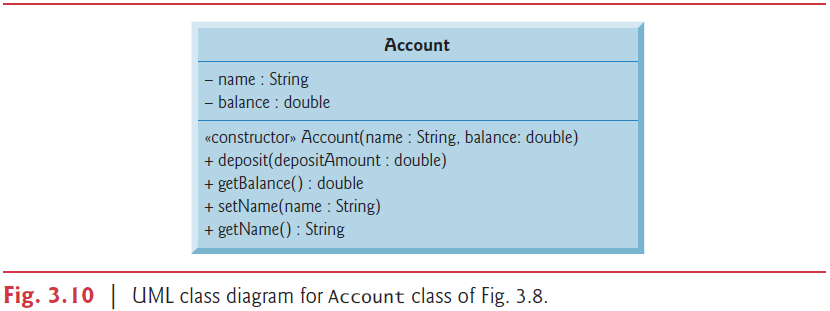
**-Adornments** / **Access modifiers:** See “Adornments / Access modifiers” section of “Java Notes” file.

UML marks of adornments: public (+), protected (#), no modifier, private (-)

**-UML**:We will use UML class diagrams to summarize a class’s attributes and operations. In industry, UML is being used to design a system before starting to program it.







**-Top compartment:** Contains class name centered horizontally in boldface type.

**-Middle compartment:** Contains names of class’s instance variables, preceeded by access modifiers, types after the colon.

**-­Bottom compartment:** Contains names of methods of class, each preceeded by access modifier, parameter names and types inside paranthesis, return type is after the colon. There are no return types for methods with no return value(void).

**­-Constructors:** Like methods, constructors are modelled in third compartment of class diagram too. It is listed before methods. The text “constructor” inside guilliments, class name and parameter list. There are no return types for constructors.

**-Return Types:** The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. If there is no return type than it is not specified. (Not even with void.)

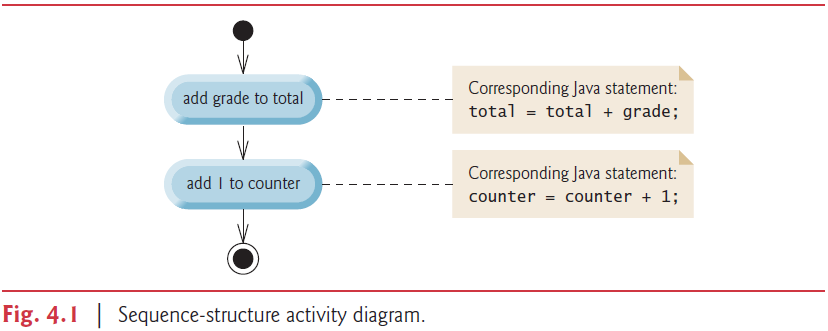
**-Parameters:** The UML models a parameter a bit differently from Java by listing the parameter name, followed by a colon and the parameter type in the parentheses after the operation name.

The UML has its own data types similar to those of Java, but for simplicity, we’ll use the

Java data types.

* **Chapter 4-5**

**-UML Activity Diagram**



-A UML activity diagram models the workflow (also called the activity) of a portion

of a software system. Such workflows may include a portion of an algorithm, like the

sequence structure in Fig. 4.1. Activity diagrams are composed of symbols, such as action state

symbols (rectangles with their left and right sides replaced with outward arcs), diamonds

and small circles. These symbols are connected by transition arrows, which represent

the *flow of the activity*—that is, the *order* in which the actions should occur.

Like pseudocode, activity diagrams help you develop and represent algorithms. Activity diagrams clearly show how control structures operate.

Consider the sequence-structure activity diagram in Fig. 4.1. It contains two action states, each containing an action expression—for example, “add grade to total” or “add 1 to counter”—that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows in the activity diagram represent transitions, which indicate the *order* in which the actions represented by the action states occur. The program that implements the activities illustrated by the diagram in Fig. 4.1 first adds grade to total, then adds 1 to counter.

The solid circle at the top of the activity diagram represents the initial state—the *beginning* of the workflow *before* the program performs the modeled actions. The solid circle surrounded by a hollow circle at the bottom of the diagram represents the final state—the *end* of the workflow *after* the program performs its actions.

Figure 4.1 also includes rectangles with the upper-right corners folded over. These are

UML notes (like comments in Java)—explanatory remarks that describe the purpose of symbols in the diagram. Figure 4.1 uses UML notes to show the Java code associated with each action state. A dotted line connects each note with the element it describes. Activity diagrams normally do *not* show the Java code that implements the activity. We do this here to illustrate how the diagram relates to Java code.

*-Diamond*, or decision symbol, which indicates that a *decision* is to be made. The workflow continues along a path determined by the symbol’s associated guard conditions, which can be *true* or *false*. Each transition arrow emerging from a decision symbol has a guard condition (specified in *square brackets* next to the arrow). If a guard condition is *true*, the workflow enters the action state to which the transition arrow points.  
 UML activity diagram for the if statement.

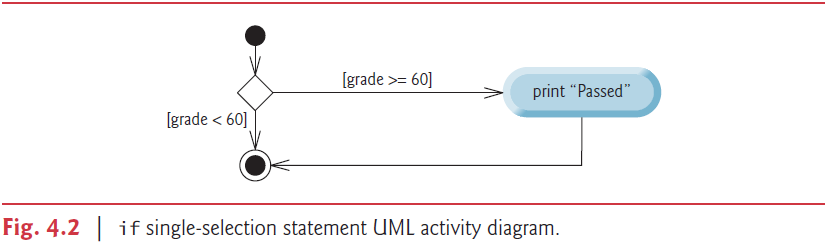
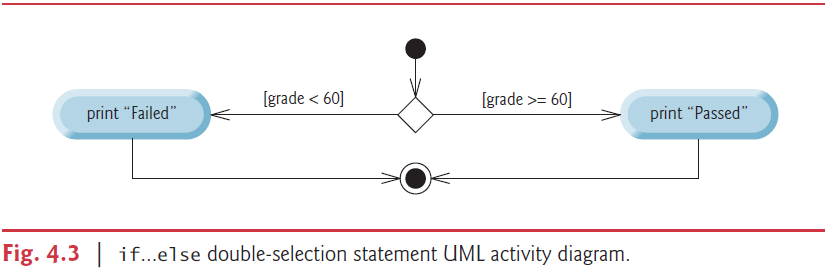


Figure below illustrates the flow of control in the if…else statement. Once again, the symbols

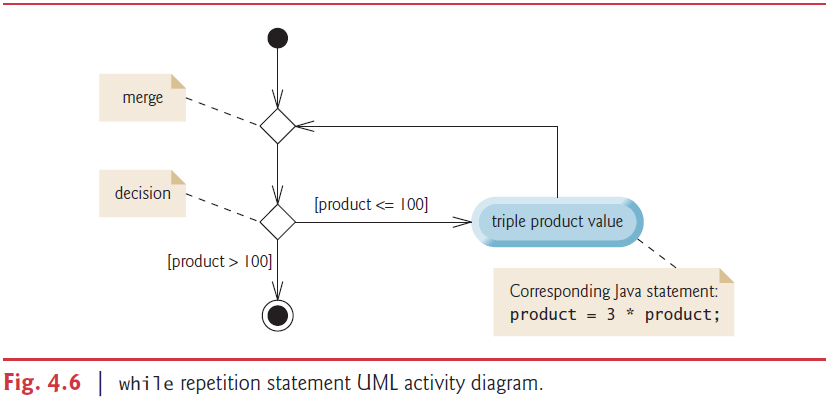
in the UML activity diagram (besides the initial state, transition arrows and final

state) represent action states and decisions.

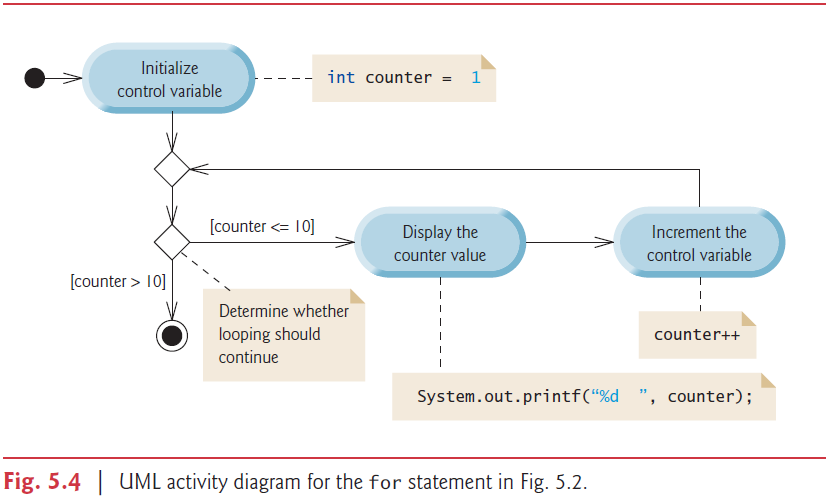


The UML represents both the merge symbol and the decision symbol as diamonds. The merge symbol joins two flows of activity into one.

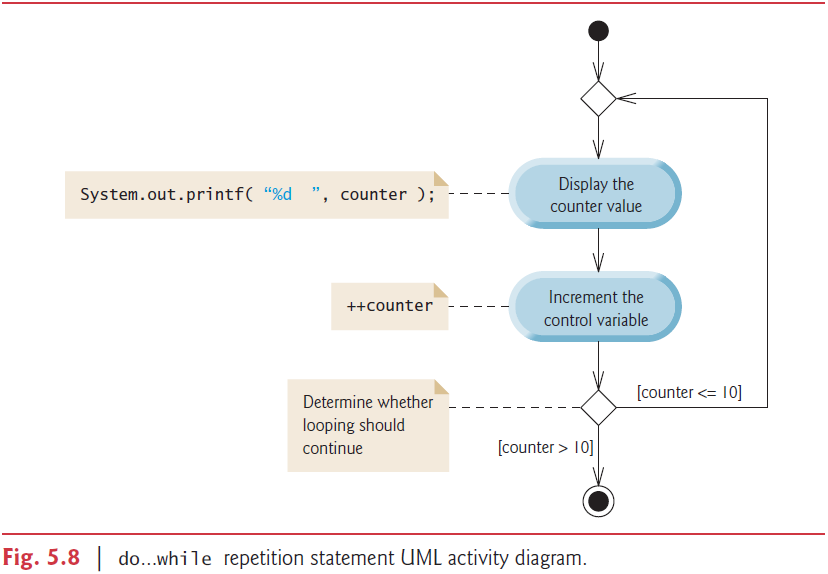
The decision and merge symbols can be distinguished by the number of “incoming” and “outgoing” transition arrows. A decision symbol has one transition arrow pointing to the diamond and two or more pointing out from it to indicate possible transitions from that point. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it. A merge symbol has two or more transition arrows pointing to the diamond and only one pointing from the diamond, to indicate multiple activity flows merging to continue the activity. *None* of the transition arrows associated with a merge symbol has a guard condition.  
 UML activity diagram for the while statement.



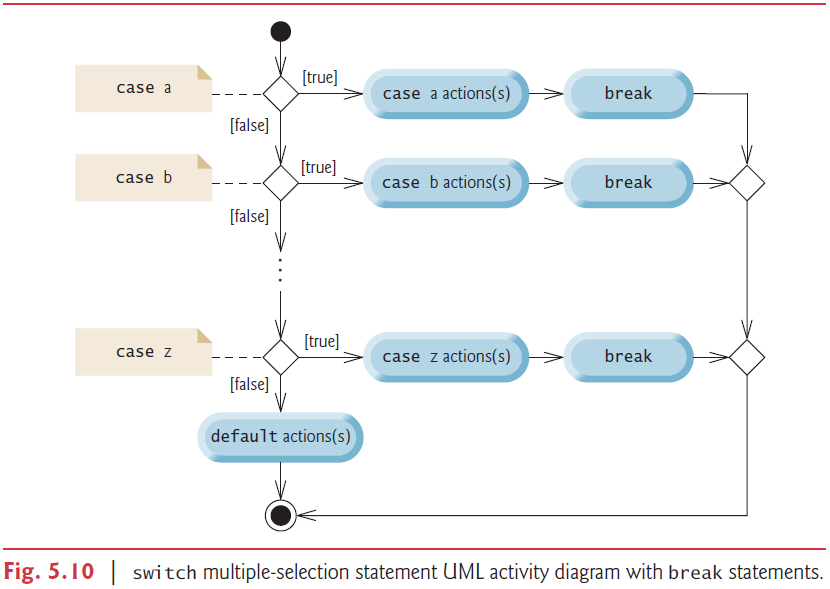
The for statement’s UML activity diagram is similar to that of the while statement. The diagram makes it clear that initialization occurs once before the loop-continuation test is evaluated the first time, and that incrementing occurs each time through the loop after the body statement executes.



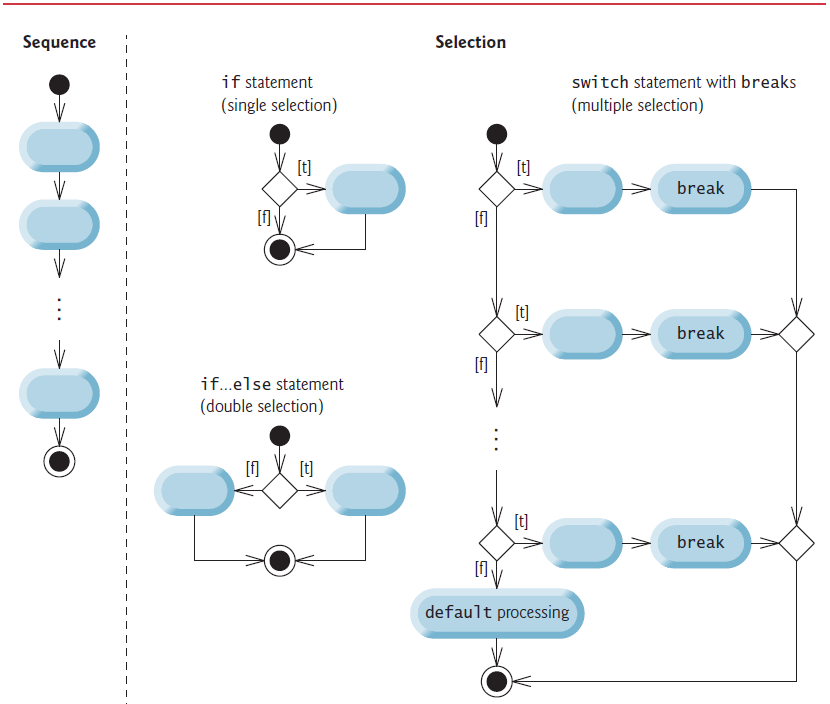
UML activity diagram for the do…while statement. This diagram makes it clear that the loop-continuation condition is not evaluated until after the loop performs the action state at least once.

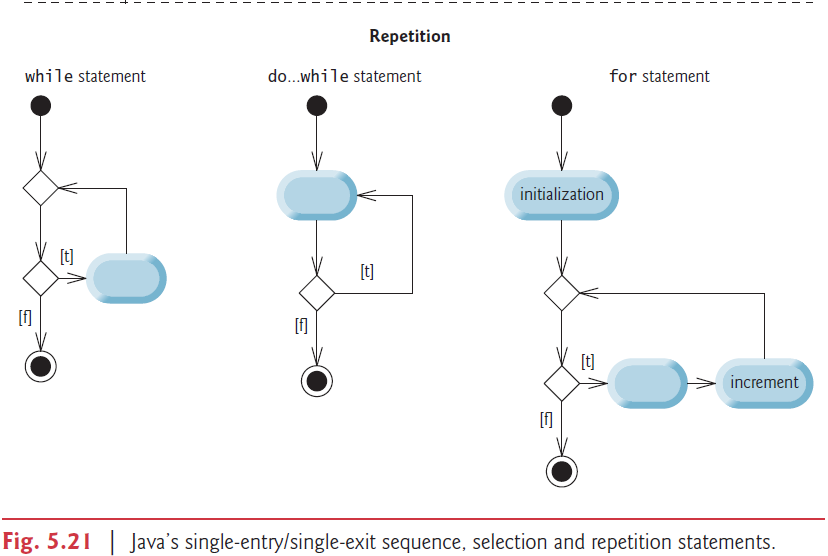


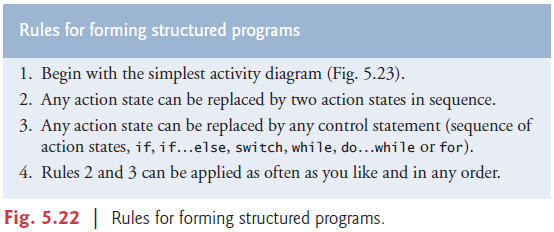
UML activity diagram for the general switch statement. Most switch statements use a break in each case to terminate the switch statement after processing the case. Figure emphasizes this by including break statements in the activity diagram. The diagram makes it clear that the break statement at the end of a case causes control to exit the switch statement immediately.



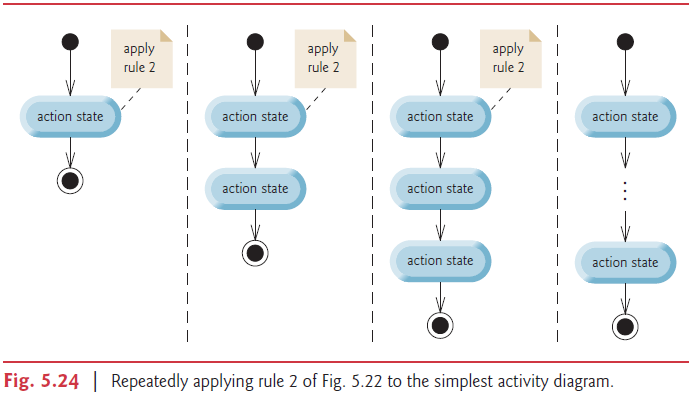
**-Structured Programming Summary**



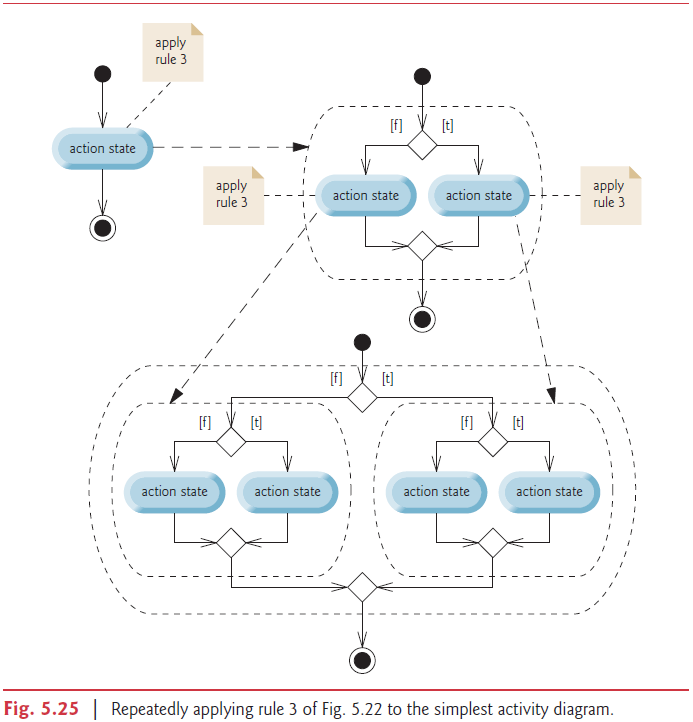




Repeatedly applying rule 2 to the simplest activity diagram results in an activity diagram containing many action states in sequence. Rule 2 generates a *stack* of control statements, so let’s call rule 2 the stacking rule.



Rule 3 is called the nesting rule. Repeatedly applying rule 3 to the simplest activity diagram results in one with neatly *nested* control statements.



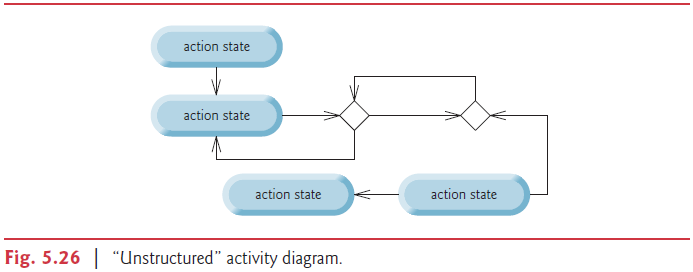
Rule 4 generates larger, more involved, and more deeply nested statements. The diagrams that emerge from applying the rules constitute the set of all possible structured activity diagrams and hence the set of all possible structured programs. The beauty of the structured approach is that we use *only seven* simple single-entry/single-exit control statements and assemble them in *only two* simple way.

If the rules are followed, an “unstructured’ activity diagram cannot be created. If you’re uncertain about whether a particular diagram is structured, apply the rules in reverse to reduce it to the simplest activity diagram. If you can reduce it, the original diagram is structured; otherwise, it’s not.

The first thing wrong with this program is that the arrow coming out of the second decision symbol goes in the middle of the do while statement. It should go to the beginning if we want two nested do while statements. Going to the middle of a do while statement directly can only be achieved with a goto statement.

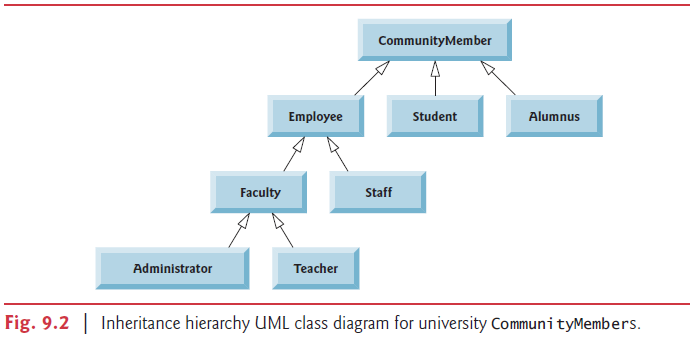
The second thing wrong with this program is, there is no way of reaching the second part of the program. There is not even a goto statement that jumps to the second part.

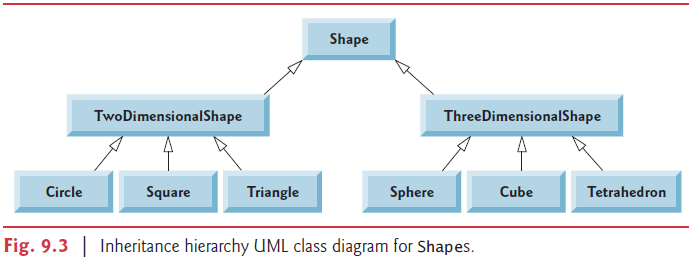
The last wrong thing is the fact that there are two out outgoing transition arrows from a statement.



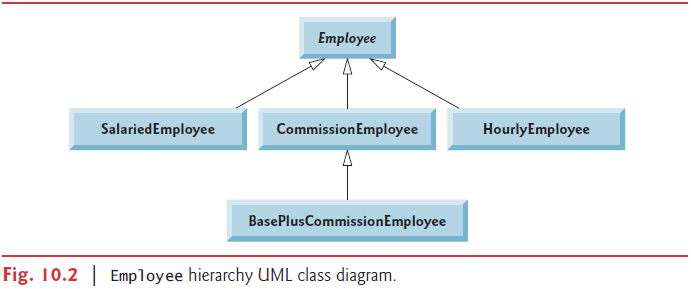
* **Chapter 9**

**-Inheritance:** Each arrow represents an is-a relationship. The top class is inherited from Object class. So we can say Employee is a CommunityMember and CommunityMember is an Object.

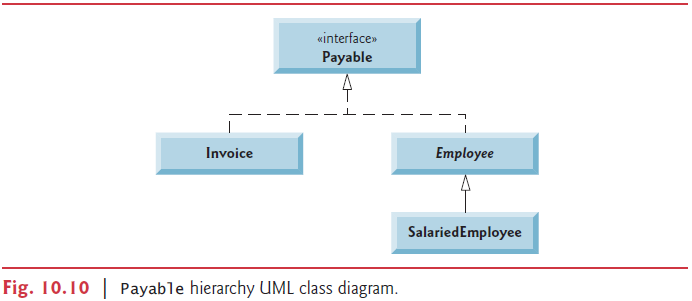




* **Chapter 10**

**-Abstract classes:** We use italic font to distinguish abstract classes. But since italic is hard to see, we tend to use <<abstract>> instead.  
  


**-Realization:** The UML expresses the relationship between a class and an interface through a relationship known as realization. A class is said to realize, or implement, the methods of an interface. A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface.



**-Generalization**

-Generalization is an everyday technique that we use to manage complexity.

-Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.

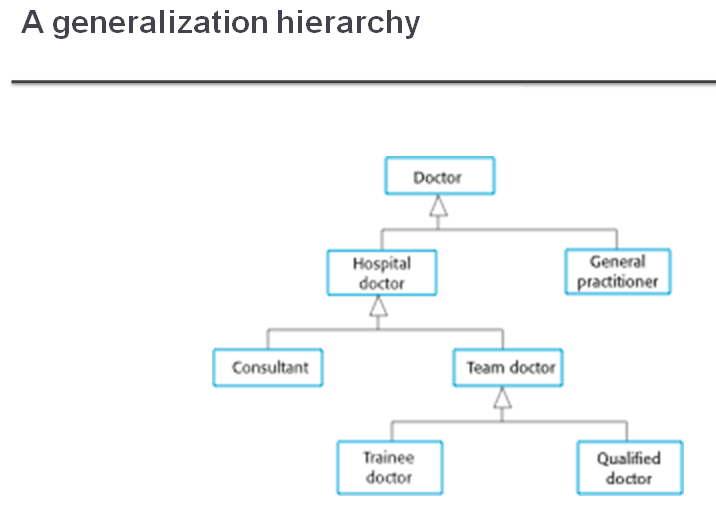
-This allows us to infer that different members of these classes have some common characteristics e.g. squirrels and rats are rodents.

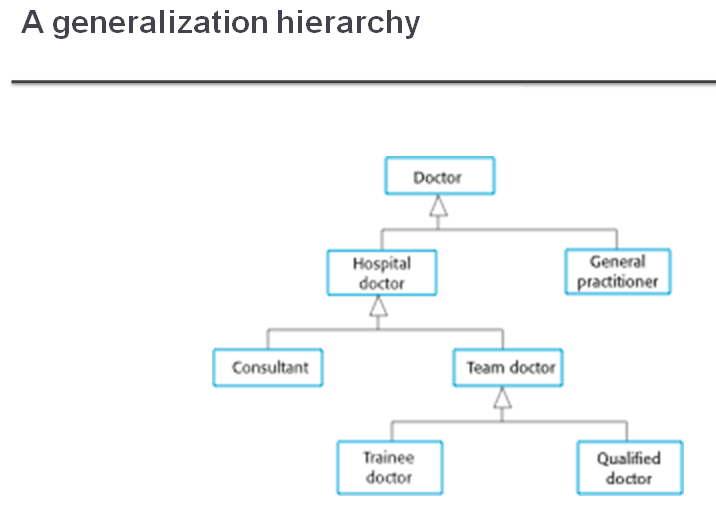
-In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.

-In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

-In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.

-The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

****



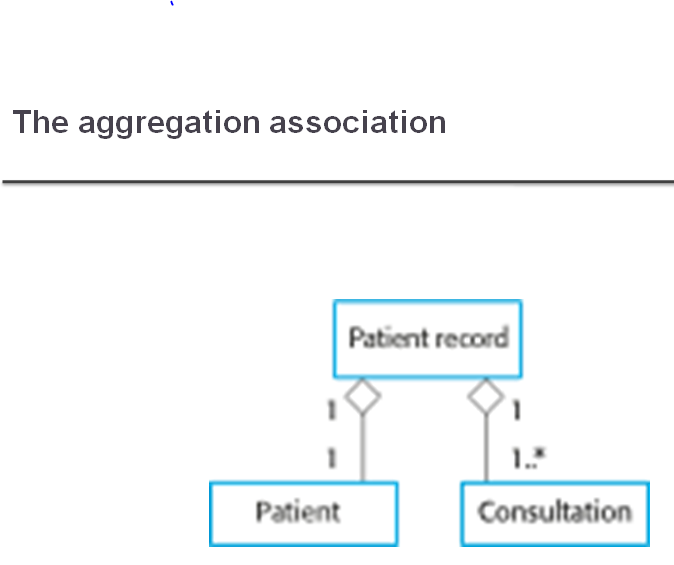
-To express the common actor behavior

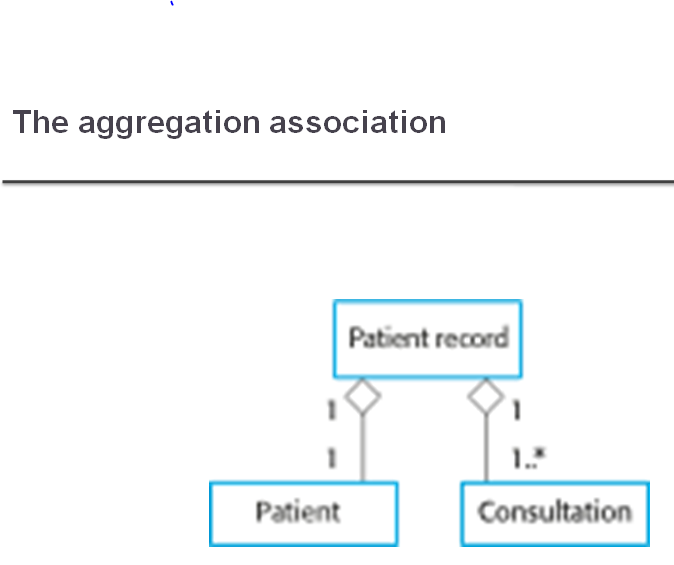
-Actors communicate with the same set of use cases

**-Object class aggregation models**

An aggregation model shows how classes that are collections are composed of other classes.

Aggregation models are similar to the part-of relationship in semantic data models.





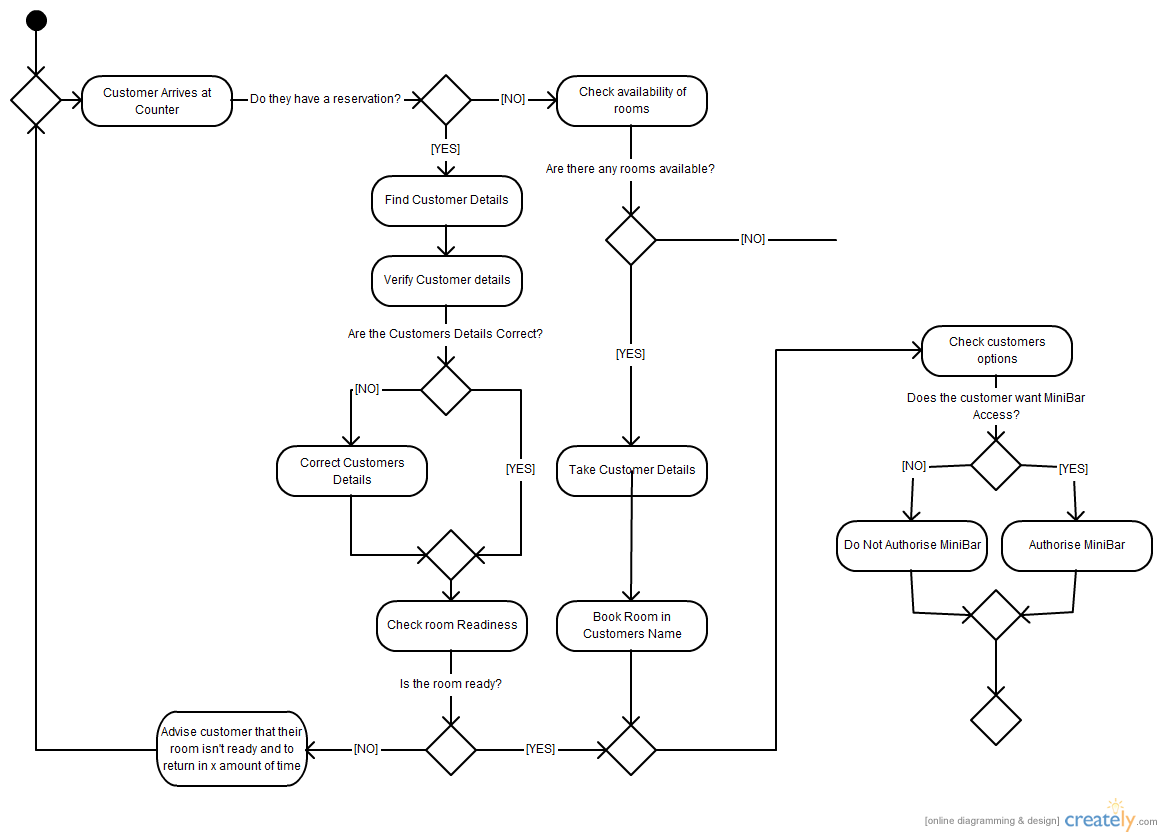
**Include:** Use if you have to do something before doing something else.

**Extend:** Use if you have a more detailed way of doing one action.

Make payment -> make payment with cash

-> make payment with debit card

**-Activity diagrams**

****

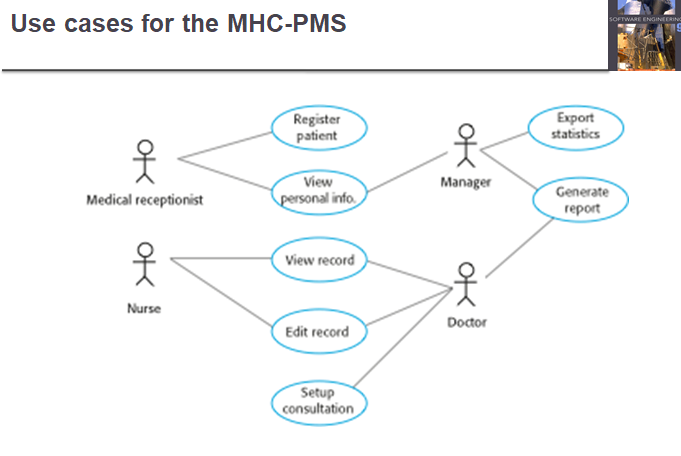
**-Use case diagrams**

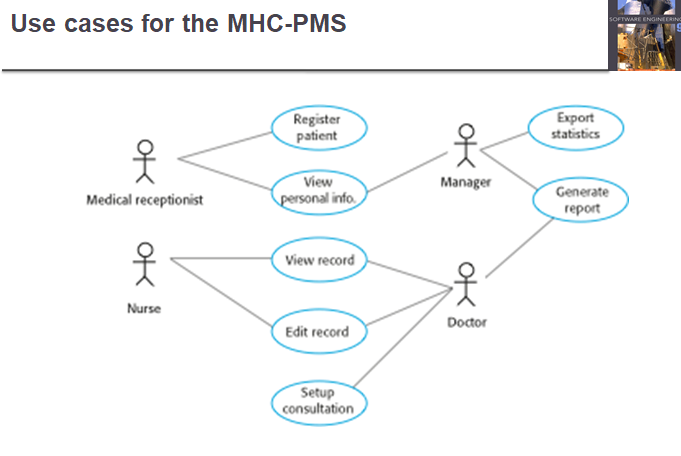
Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.

A set of use cases should describe all possible interactions with the system.

High-level graphical model supplemented by more detailed tabular description (see Chapter 5).

Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.





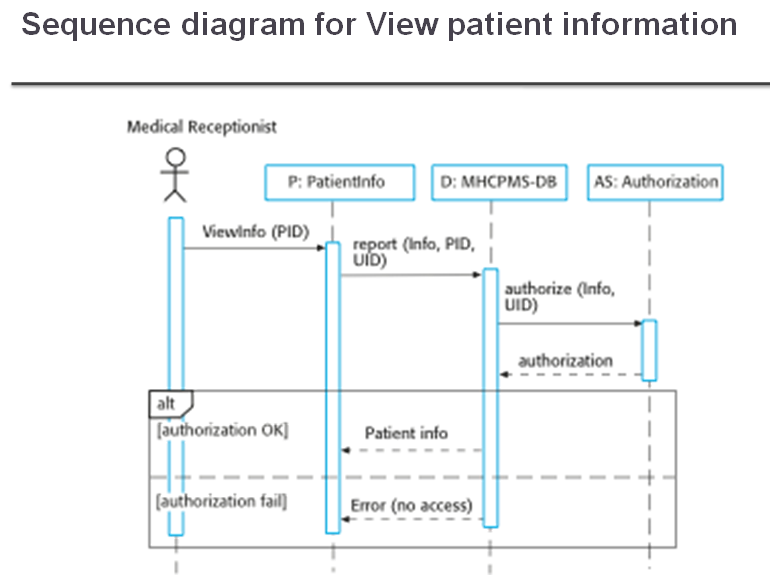
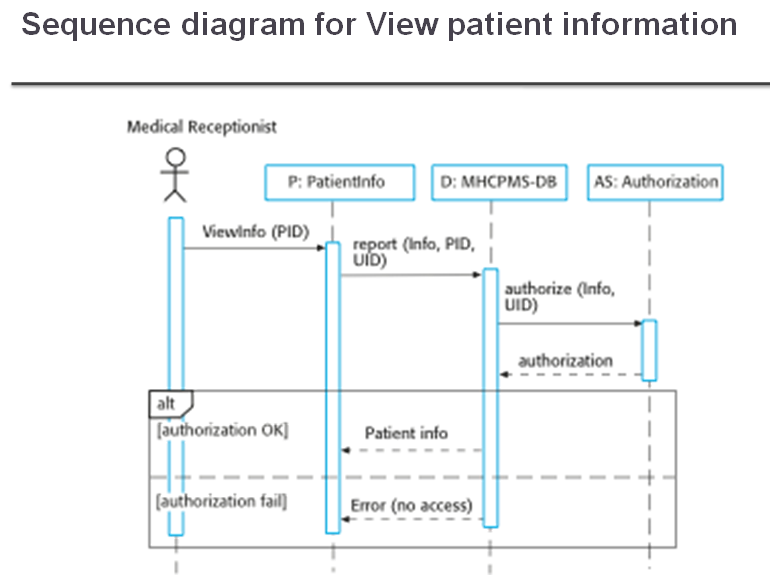
**-Sequence diagrams**

Sequence diagrams are part of the UML and are used to model the interactions between the actors and the objects within a system.

A sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.

The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.

Interactions between objects are indicated by annotated arrows.



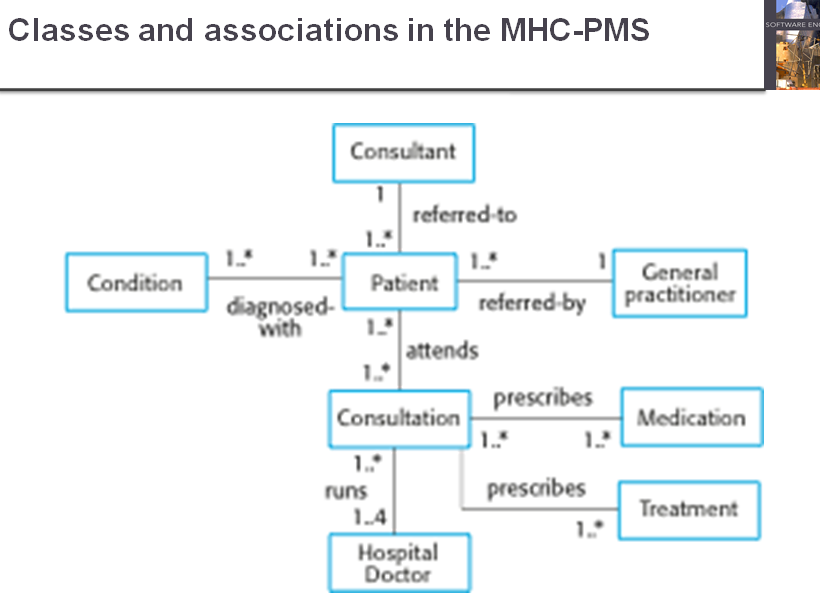
**-Class diagrams**

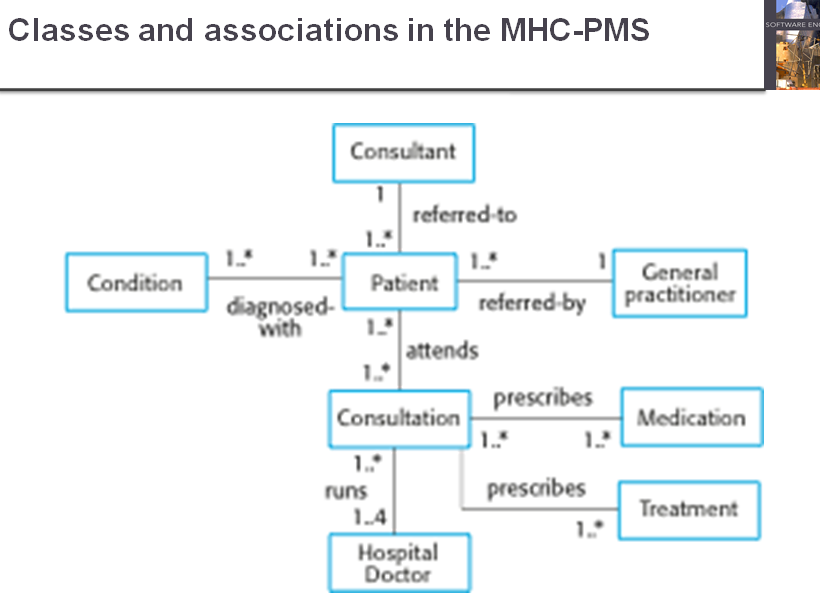
Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.

An object class can be thought of as a general definition of one kind of system object.

An association is a link between classes that indicates that there is some relationship between these classes.

When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.



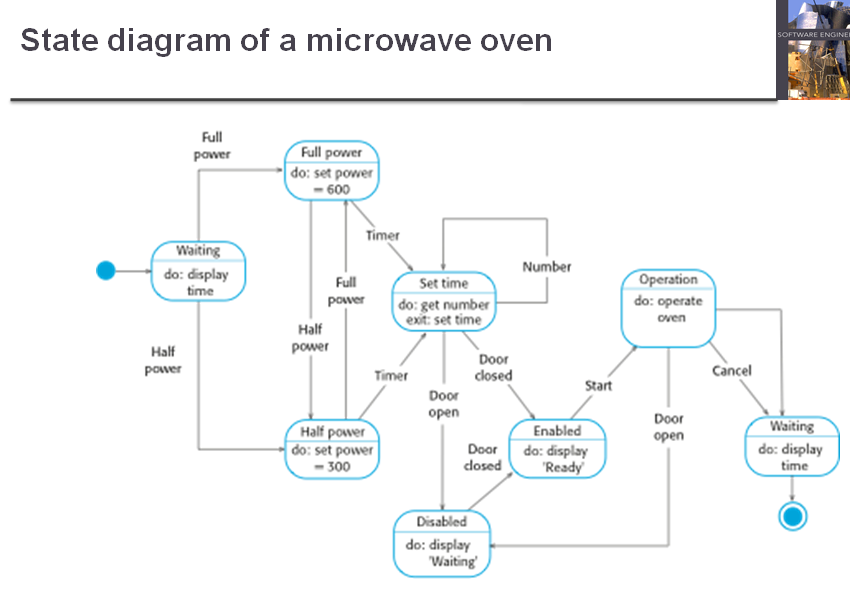


**-State diagrams**

Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.

Data-driven models show the sequence of actions involved in processing input data and generating an associated output.

They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

****

