The Extended Entity Relationship Model and Object Model

The ER Model Revisited

The ER Modelling concepts that we discussed in previous semester are sufficient for representing many database schemas for traditional database applications, which include many data-processing applications in business and industry. Since the late 1970s, however, designers of database applications have tried to design more accurate database schemas that reflect the data properties and constraints more precisely. This was particularly important for newer applications of database technology, such as databases for engineering design and manufacturing (CAD/CAM), telecommunications, complex software systems, and Geographic Information Systems (GIS), among many other applications. These types of databases have more complex requirements than do the more traditional applications. This led to the development of additional semantic data modeling concepts that were incorporated into conceptual data models such as the ER model. Various semantic data models have been proposed in the literature. Many of these concepts were also developed independently in related areas of computer science, such as the knowledge representation area of artificial intelligence and the **object modelling** area in software engineering.

In this chapter, we describe features that have been proposed for semantic data models, and show how the ER model can be enhanced to include these concepts, leading to the **Enhanced ER (EER**) model.

EER Model

EER is a high-level data model that incorporates the extensions to the original ER model.

It is a diagrammatic technique for displaying the following concepts

- Sub Class and Super Class
- Specialization and Generalization
- Union or Category
- Aggregation

These concepts are used when it comes in EER schema and the resulting schema diagrams called as EER Diagrams.

Features of EER Model

- EER creates a design more accurate to database schemas.
- It reflects the data properties and constraints more precisely.
- It includes all modelling concepts of the ER model.
- Diagrammatic technique helps for displaying the EER schema.
- It includes the concept of specialization and generalization.
- It is used to represent a collection of objects that is union of objects of different of different entity types.

User defined data types

A user-defined data type (UDT) is a data type that derived from an existing data type. You can use UDTs to extend the built-in types already available and create your own customized data types.

There are six user-defined types:

- Distinct type
- Structured type
- Reference type
- Array type
- Row type
- Cursor type

Each of these types is described in the following sections.

Distinct type

A *distinct type* is a user-defined data type that shares its internal representation with an existing built-in data type (its "source" type).

Distinct types include qualified identifiers. If the schema name is not used to qualify the distinct type name when used in other than the CREATE TYPE (Distinct), DROP, or COMMENT statements, the SQL path is searched in sequence for the first schema with a distinct type that matches.

Distinct types that are sourced on LOB types are subject to the same restrictions as their source type.

A distinct type is defined to use either strong typing or weak typing rules. Strong typing rules are the default.

Strongly typed distinct type

A strongly typed distinct type is considered to be a separate and incompatible type for most operations. For example, you want to define a picture type, a text type, and an audio type. Each of these types has different semantics, but each uses the built-in data type BLOB for their internal representation.

The following example illustrates the creation of a distinct type named AUDIO:

CREATE TYPE AUDIO **AS BLOB** (1M)

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type; this consideration allows the creation of functions that are written specifically for AUDIO, and assures that these functions are not applied to values of any other data type (for example pictures or text).

Strongly typed distinct types support strong typing by ensuring that only those functions and operators that are explicitly defined on the distinct type can be applied to its instances. For this reason, a strongly typed distinct type does not automatically acquire the functions and operators of its source type, because these functions and operators might not be meaningful. For example, a LENGTH function could be defined to support a parameter with the data type AUDIO that returns length of the object in seconds instead of bytes.

Weakly typed distinct type

A weakly typed distinct type is considered to be the same as its source type for all operations, except when the weakly typed distinct type applies constraints on values during assignments or casts. This consideration also applies to function resolution.

The following example illustrates the creation of a distinct type named POSITIVEINTEGER:

CREATE TYPE POSITIVEINTEGER **AS INTEGER**

WITH WEAK TYPE RULES CHECK (VALUE>=0**)**

Weak typing means that except for accepting only positive integer values, POSITIVEINTEGER operates in the same way as its underlying data type of INTEGER.

A weakly typed distinct type can be used as an alternative method of referring to a built-in data type within application code. The ability to define constraints on the values that are associated with the distinct type provides a method for checking values during assignments and casts.

Using distinct types can provide benefits in the following categories:

Extensibility

By defining new data types, you increase the set of data types available to you to support your applications.

Flexibility

You can specify any semantics and behaviour for your new data type by using user-defined functions (UDFs) to augment the diversity of the data types available in the system.

Consistent and inherited behaviour

Strong typing guarantees that only functions defined on your distinct type can be applied to instances of the distinct type. Weak typing ensures that the distinct type behaves the same way as its underlying data type and so can use all the same functions and methods available to that underlying type.

Encapsulation

Using a weakly typed distinct type makes it possible to define data type constraints in one location for all usages within application code for that distinct type.

Performance

Distinct types are highly integrated into the database manager. Because distinct types are internally represented the same way as built-in data types, they share the same efficient code that is used to implement components such as built-in functions, comparison operators, and indexes for built-in data types.

Not all built-in data types can be used to define distinct types. The source data type cannot be XML, array, row, or cursor. For more information, see CREATE TYPE (distinct) statement.

Structured type

A *structured type* is a user-defined data type that has a structure that is defined in the database. It contains a sequence of named *attributes*, each of which has a data type. A structured type also includes a set of method specifications.

A structured type can be used as the type of a table, view, or column. When used as a type for a table or view, that table or view is known as a *typed table* or *typed view*. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of this typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type. When used as a data type for a column, the column contains values of that structured type (or values of any of the subtypes for that type, as defined later in this section). Methods are used to retrieve or manipulate attributes of a structured column object.

A *supertype* is a structured type for which other structured types, called *subtypes*, are defined. A subtype inherits all the attributes and methods of its supertype and can have additional attributes and methods defined. The set of structured types that is related to a common supertype is called a type *hierarchy* and the type that does not have any supertype is called the *root type* of the type hierarchy.

The term subtype applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. Therefore, a subtype of a structured type T is T and all structured types below T in the hierarchy. A *proper subtype* of a structured type T is a structured type below T in the type hierarchy.

There are restrictions on having recursive type definitions in a type hierarchy. For this reason, it is necessary to develop a shorthand way of referring to the specific type of recursive definitions that are allowed. The following definitions are used:

- *Directly uses*: A type **A** is said to directly use another type **B**, if and only if one of the following statements is true:
    1. Type **A** has an attribute of type **B**.
    2. Type **B** is a subtype of **A**or a supertype of **A**.
- *Indirectly uses*: A type **A** is said to indirectly use a type **B**, if one of the following statements is true:
    1. Type **A** directly uses type **B**.
    2. Type **A** directly uses some type **C**and type **C** indirectly uses type **B**.

A type cannot be defined so that one of its attribute types directly or indirectly uses itself. If it is necessary to have such a configuration, consider using a reference as the attribute. For example, with structured type attributes, there cannot be an instance of "employee" with an attribute of "manager" when "manager" is of type "employee". There can, however, be an attribute of "manager" with a type of REF(employee).

A type cannot be dropped if certain other objects use the type, either directly or indirectly. For example, a type cannot be dropped if a table or view column makes direct or indirect use of the type.

Reference type

A *reference type* is a companion type to a structured type. Similar to a distinct type, a reference type is a scalar type that shares a common representation with one of the built-in data types. This same representation is shared for all types in the type hierarchy. The reference type representation is defined when the root type of a type hierarchy is created. When using a reference type, a structured type is specified as a parameter of the type. This parameter is called the *target type* of the reference.

The target of a reference is always a row in a typed table or a typed view. When a reference type is used, it can have a *scope* defined. The scope identifies a table (called the *target table*) or view (called the *target view*) that contains the target row of a reference value. The target table or view must have the same type as the target type of the reference type. An instance of a scoped reference type uniquely identifies a row in a typed table or typed view, called the *target row*.

Array type

A user-defined *array type* is a data type that is defined as an array with elements of another data type. Every ordinary array type has an index with the data type of INTEGER and has a defined maximum cardinality. Every associative array has an index with the data type of INTEGER or VARCHAR and does not have a defined maximum cardinality.

Row type

A *row type* is a data type that is defined as an ordered sequence of named fields, each with an associated data type, which effectively represents a row. A row type can be used as the data type for variables and parameters in SQL PL to provide simple manipulation of a row of data.

Cursor type

A user-defined *cursor type* is a user-defined data type defined with the keyword CURSOR and optionally with an associated row type. A user-defined cursor type with an associated row type is a *strongly typed cursor type*; otherwise, it is a *weakly typed cursor type*. A value of a user-defined cursor type represents a reference to an underlying cursor.

Subclasses,Superclasses and Inheritance

The Enhanced Entity Relationship Model contains all the features of the Entity Relationship model. In addition to all that, it also contains features of Subclasses, Superclasses and Inheritance.

All of these in detail are as follows −

The first Enhanced ER (EER) model concept we take up is that of a **subtype** or **subclass** of an entity type. As we discussed in last semester, an entity type is used to represent both a *type of entity* and the *entity set* or *collection of entities of that type* that exist in the database.

For example, the entity type EMPLOYEE describes the type (that is, the attributes and relationships) of each employee entity, and also refers to the current set of EMPLOYEE entities in the COMPANY database.

In many cases an entity type has numerous subgroupings or subtypes of its entities that are meaningful and need to be represented explicitly because of their significance to the database application.

For example, the entities that are members of the EMPLOYEE entity type may be distinguished further into SECRETARY, ENGINEER, MANAGER,TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on.

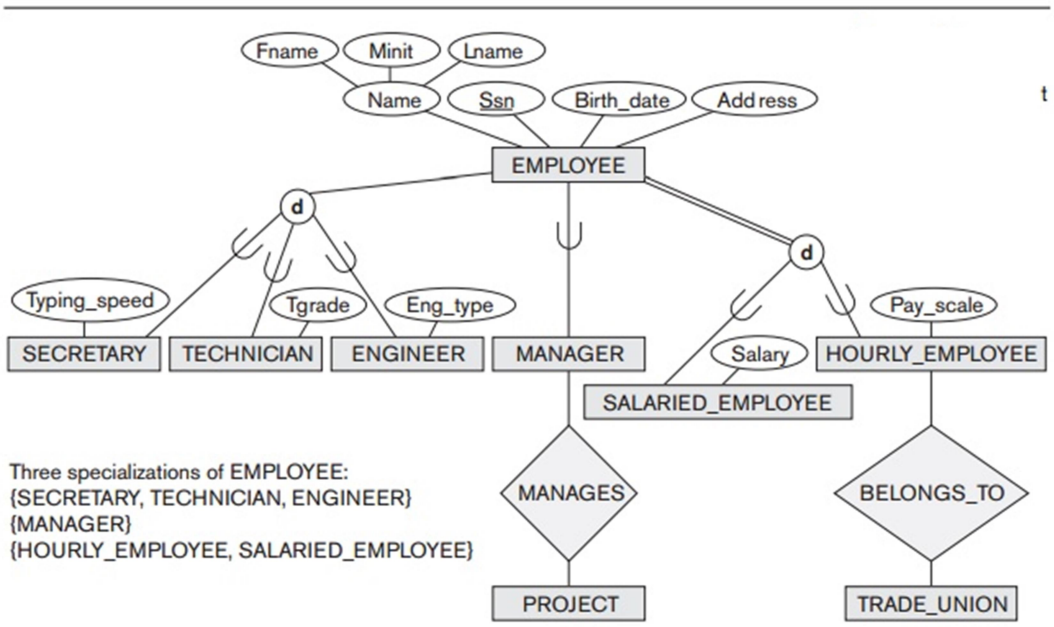
The set of entities in each of the latter groupings is a subset of the entities that belong to the EMPLOYEE entity set, meaning that every entity that is a member of one of these subgroupings is also an employee. We call each of these subgroupings a **subclass** or **subtype** of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the **superclass** or **supertype** for each of these subclasses, below Illustration-1 shows how to represent these concepts diagrammatically in EER diagrams.

We call the relationship between a superclass and any one of its subclasses a **superclass/subclass** or **supertype/subtype** or simply **class/subclass relationship**.

In our previous example, EMPLOYEE/SECRETARY and EMPLOYEE/TECHNICIAN are two class/subclass relationships. Notice that a member entity of the subclass repre-sents the *same real-world entity* as some member of the superclass; for example, a SECRETARY entity 'ANB CDE is also the EMPLOYEE 'ANB CDE.'

Hence, the subclass member is the same as the entity in the superclass, but in a distinct *specific role.* When we implement a superclass/subclass relationship in the database system, however, we may represent a member of the subclass as a distinct database object— say, a distinct record that is related via the key attribute to its superclass entity.

Illustration-1



An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. Such an entity can be included optionally as a member of any number of subclasses. For example, a salaried employee who is also an engineer belongs to the two subclasses ENGINEER and SALARIED_EMPLOYEE of the EMPLOYEE entity type. However, it is not necessary that every entity in a superclass is a member of some subclass.

An important concept associated with subclasses (subtypes) is that of **type inheritance**. Recall that the *type* of an entity is defined by the attributes it possesses and the relationship types in which it participates. Because an entity in the subclass rep-resents the same real-world entity from the superclass, it should possess values for its specific attributes *as well as* values of its attributes as a member of the superclass. We say that an entity that is a member of a subclass **inherits** all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates. Notice that a subclass, with its own specific (or local) attributes and relationships together with all the attributes and relationships it inherits from the superclass, can be considered an *entity type* in its own right.

Specialization and Generalization

1. Specialization

Specialization is the process of defining a *set of subclasses* of an entity type; this entity type is called the **superclass** of the specialization. The set of subclasses that forms a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass.

For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the *job type* of each employee entity. We may have several specializations of the same entity type based on different distinguishing characteristics. For example, another specialization of the EMPLOYEE entity type may yield the set of subclasses {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}; this specialization distinguishes among employees based on the *method of pay.*

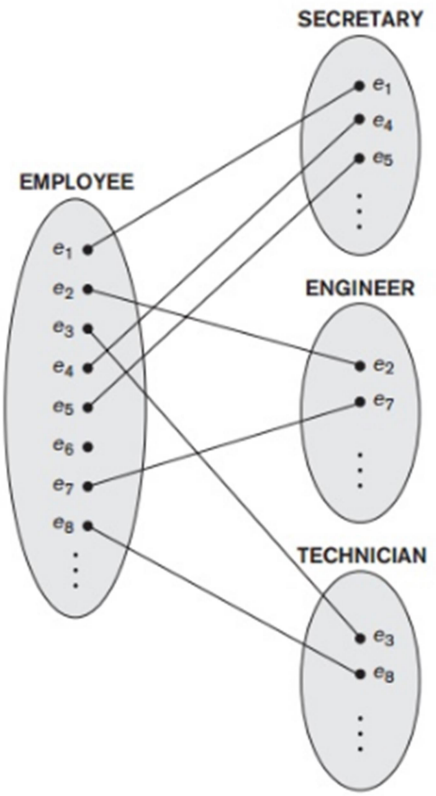
Illustration-1 shows how we represent a specialization diagrammatically in an EER diagram. The subclasses that define a specialization are attached by lines to a circle that represents the specialization, which is connected in turn to the superclass. The *subset symbol* on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship. Attributes that apply only to entities of a particular subclass—such as TypingSpeed of SECRETARY—are attached to the rec-tangle representing that subclass. These are called **specific attributes** (or **local attributes**) of the subclass. Similarly, a subclass can participate in **specific relation-ship types**, such as the HOURLY_EMPLOYEE subclass participating in the BELONGS_TO relationship in Illustration-1. We will explain the **d** symbol in the circles in Illustration-1 and additional EER diagram notation shortly.

Illustration-2 shows a few entity instances that belong to subclasses of the {SECRETARY, ENGINEER, TECHNICIAN} specialization. Again, notice that an entity that belongs to a subclass represents *the same real-world entity* as the entity connected to it in the EMPLOYEE superclass, even though the same entity is shown twice; for example, e_1 is shown in both EMPLOYEE and SECRETARY in Illustration-2. As the figure suggests, a superclass/subclass

relationship such as EMPLOYEE/ SECRETARY somewhat resembles a 1:1 relationship *at the instance level*

The main difference is that in a 1:1 relationship two *distinct entities* are related, whereas in a superclass/subclass relationship the entity in the subclass is the same real-world entity as the entity in the superclass but is playing a *specialized role*—for example, an EMPLOYEE specialized in the role of SECRETARY, or an EMPLOYEE specialized in the role of TECHNICIAN.

Illustration-2



There are two main reasons for including class/subclass relationships and specializations in a data model. The first is that certain attributes may apply to some but not all entities of the superclass. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of their attributes with the other members of the superclass. For example, in Illustration-2 the SECRETARY subclass has the specific attribute Typing_speed, whereas the ENGINEER subclass has the specific attribute Eng_type, but SECRETARY and ENGINEER share their other inherited attributes from the EMPLOYEE entity type.

The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass. For example, if only HOURLY_EMPLOYEES can belong to a trade union, we can represent that fact by creating the subclass HOURLY_EMPLOYEE of EMPLOYEE and relating the subclass to an entity type TRADE_UNION via the BELONGS_TO relationship type, as illustrated in Illustration-1.

In summary, the specialization process allows us to do the following:

Define a set of subclasses of an entity type

Establish additional specific attributes with each subclass

Establish additional specific relationship types between each subclass and other entity types or other subclasses

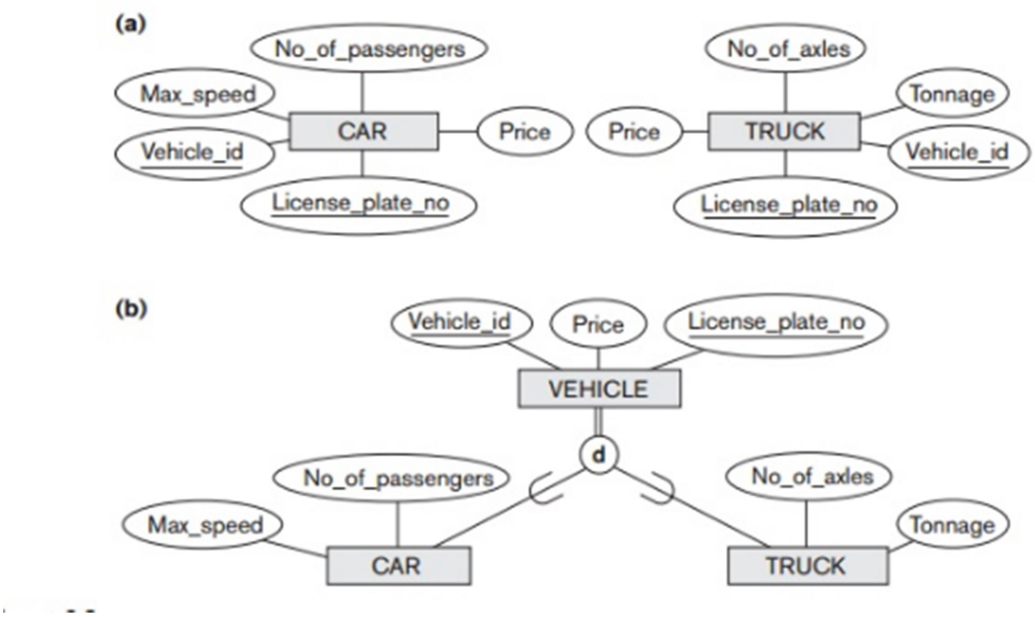
2. Generalization

We can think of a *reverse process* of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize** them into a single **superclass** of which the original entity types are special **subclasses**. For example, consider the entity types CAR and TRUCK shown in Illustration-3(a). Because they have several common attributes, they can be generalized into the entity type VEHICLE, as shown in Illustration-3(b).
Both CAR and TRUCK are now subclasses of the **generalized superclass** VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

Notice that the generalization process can be viewed as being functionally the inverse of the specialization process. Hence, in Illustration-3 we can view {CAR, TRUCK} as a specialization of VEHICLE, rather than viewing VEHICLE as a generalization of CAR and TRUCK. Similarly, in Illustration-1 we can view EMPLOYEE as a generalization of SECRETARY, TECHNICIAN, and ENGINEER. A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a specialization. We will *not* use this notation because the decision as to which process is followed in a particular situation is often subjective. Appendix A gives some of the suggested alternative dia-grammatic notations for schema diagrams and class diagrams.

So far we have introduced the concepts of subclasses and superclass/subclass relationships, as well as the specialization and generalization processes. In general, a superclass or subclass represents a collection of entities of the same type and hence also describes an *entity type*; that is why superclasses and subclasses are all shown in rectangles in EER diagrams, like entity types. Next, we discuss the properties of specializations and generalizations in more detail.

Illustration 3



Constraints and Characterisitics of Specialization and Generalization

Constraints on Specialization and Generalization

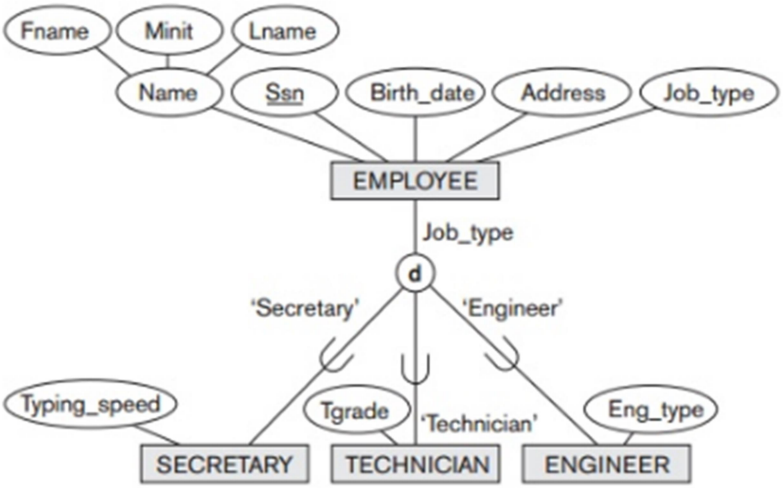
In general, we may have several specializations defined on the same entity type (or superclass), as shown in Illustration-1. In such a case, entities may belong to subclasses in each of the specializations. However, a specialization may also consist of a *single* subclass only, such as the {MANAGER} specialization in Illustration-1; in such a case, we do not use the circle notation.

In some specializations we can determine exactly the entities that will become members of each subclass by placing a condition on the value of some attribute of the superclass. Such subclasses are called **predicate-defined** (or **condition-defined**) **subclasses**. For example, if the EMPLOYEE entity type has an attribute Job_type, as shown Illustration-4, we can specify the condition of membership in the SECRETARY subclass by the condition (Job_type = 'Secretary'), which we call the **defining predicate** of the subclass. This condition is a *constraint* specifying that exactly those entities of the EMPLOYEE entity type whose attribute value for Job_type is 'Secretary' belong to the subclass. We display a predicate-defined subclass by writing the predicate condition next to the line that connects the subclass to the specialization circle.

If *all* subclasses in a specialization have their membership condition on the *same* attribute of the superclass, the specialization itself is called an **attribute-defined specialization**, and the attribute is called the **defining attribute** of the specialization. In this case, all the entities with the same value for the attribute belong to the same sub-class. We display an attribute-defined specialization by placing the defining attribute name next to the arc from the circle to the superclass, as shown in Illustration4.

When we do not have a condition for determining membership in a subclass, the subclass is called **user-defined**. Membership in such a subclass is determined by the database users when they apply the operation to add an entity to the subclass; hence, membership is *specified individually for each entity by the user*, not by any condition that may be evaluated automatically…

Illustration 4



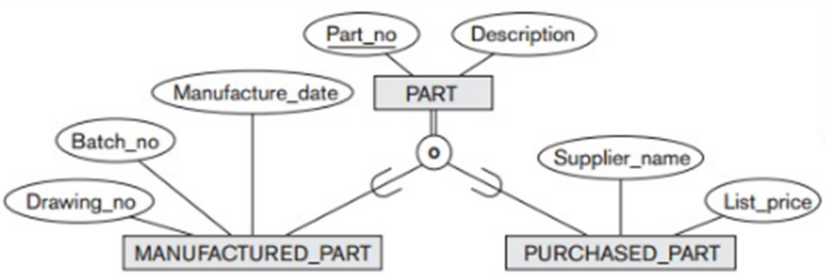
Two other constraints may apply to a specialization. The first is the **disjointness (or disjointedness) constraint**, which specifies that the subclasses of the specialization must be disjoint. This means that an entity can be a member of *at most* one of the subclasses of the specialization. A specialization that is attribute-defined implies the disjointness constraint (if the attribute used to define the membership predicate is single-valued). Illustration-1 illustrates this case, where the **d** in the circle stands for *disjoint*. The **d** notation also applies to user-defined subclasses of a specialization that must be disjoint, as illustrated by the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Illustration-1. If the subclasses are not constrained to be dis-joint, their sets of entities may be **overlapping**; that is, the same (real-world) entity may be a member of more than one subclass of the specialization. This case, which is the default, is displayed by placing an **o** in the circle, as shown in Illustration-5.

The second constraint on specialization is called the **completeness** (or **totalness**) **constraint**, which may be total or partial. A **total specialization** constraint specifies that *every* entity in the superclass must be a member of at least one subclass in the specialization. For example, if every EMPLOYEE must be either an HOURLY_EMPLOYEE or a SALARIED_EMPLOYEE, then the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Illustration-1 is a total specialization of EMPLOYEE. This is shown in EER diagrams by using a double line to connect the superclass to the circle. A single line is used to display a **partial specialization**, which allows an entity not to belong to any of the subclasses. For example, if some EMPLOYEE entities do not belong to any of the subclasses {SECRETARY, ENGINEER, TECHNICIAN} in Illustration-1 and Illustration-4, then that specialization is partial.

Notice that the disjointness and completeness constraints are *independent.* Hence, we have the following four possible constraints on specialization:

Disjoint, total
Disjoint, partial
Overlapping, total
Overlapping, partial

Illustration 5



Of course, the correct constraint is determined from the real-world meaning that applies to each specialization. In general, a superclass that was identified through the *generalization* process usually is **total**, because the superclass is *derived from* the subclasses and hence contains only the entities that are in the subclasses.

Certain insertion and deletion rules apply to specialization (and generalization) as a consequence of the constraints specified earlier. Some of these rules are as follows:

Deleting an entity from a superclass implies that it is automatically deleted from all the subclasses to which it belongs.

Inserting an entity in a superclass implies that the entity is mandatorily inserted in all *predicate-defined* (or *attribute-defined*) subclasses for which the entity satisfies the defining predicate.

Inserting an entity in a superclass of a *total specialization* implies that the entity is mandatorily inserted in at least one of the subclasses of the specialization.