Introduction to Object-Oriented Concepts

Objectives

In this appendix you will learn:

- The concepts associated with object orientation:
  - abstraction, encapsulation, and information hiding;
  - objects and attributes;
  - object identity;
  - methods and messages;
  - classes, subclasses, superclasses, and inheritance;
  - overloading;
  - polymorphism and dynamic binding.

In this appendix we discuss the main concepts that occur in object orientation. We start with a brief review of the underlying themes of abstraction, encapsulation, and information hiding.

K.1 Abstraction, Encapsulation, and Information Hiding

Abstraction is the process of identifying the essential aspects of an entity and ignoring the unimportant properties. In software engineering, this means that we concentrate on what an object is and what it does before we decide how it should be implemented. In this way we delay implementation details for as long as possible, thereby avoiding commitments that we may find restrictive at a later stage. There are two fundamental aspects of abstraction: encapsulation and information hiding.

The concept of encapsulation means that an object contains both the data structure and the set of operations that can be used to manipulate it. The concept of information hiding means that we separate the external aspects of an object from its internal details, which are hidden from the outside world. In this way the internal
details of an object can be changed without affecting the applications that use it, provided that the external details remain the same. This prevents an application from becoming so interdependent that a small change has enormous ripple effects. In other words, information hiding provides a form of \textit{data independence}.

These concepts simplify the construction and maintenance of applications through \textbf{modularization}. An object is a “black box” that can be constructed and modified independently of the rest of the system, provided that the external interface is not changed. In some systems, such as Smalltalk, the ideas of encapsulation and information hiding are brought together. In Smalltalk the object structure is always hidden and only the operation interface can ever be visible. In this way the object structure can be changed without affecting any applications that use the object.

There are two views of encapsulation: the object-oriented programming language (OOPL) view and the database adaptation of that view. In some OOPLs, encapsulation is achieved through \textbf{abstract data types} (ADTs). In this view an object has an interface part and an implementation part. The interface provides a specification of the operations that can be performed on the object; the implementation part consists of the data structure for the ADT and the functions that realize the interface. Only the interface part is visible to other objects or users. In the database view, proper encapsulation is achieved by ensuring that programmers have access to only the interface part. In this way encapsulation provides a form of \textit{logical data independence}: we can change the internal implementation of an ADT without changing any of the applications using that ADT (Atkinson \textit{et al.}, 1989).

\section*{K.2 Objects and Attributes}

Many of the important object-oriented concepts stem from the Simula programming language developed in Norway in the mid-1960s to support simulation of “real world” processes (Dahl and Nygaard, 1966), although object-oriented programming did not emerge as a new programming paradigm until the development of the Smalltalk language (Goldberg and Robson, 1983). Modules in Simula are not based on procedures as they are in conventional programming languages, but instead are based on the physical objects being modeled in the simulation. This seemed a sensible approach, as the objects are the key to the simulation: each object has to maintain some information about its current \textbf{state}, and additionally has actions (\textbf{behavior}) that have to be modeled. From Simula, we have the following definition of an object.

\begin{quote}
\textbf{Object} \hspace{1cm} A uniquely identifiable entity that contains both the attributes that describe the state of a “real-world” object and the actions that are associated with it.
\end{quote}

In the \textit{DreamHome} case study, a branch office, a member of staff, and a property are examples of objects that we wish to model. The concept of an object is simple, but at the same time, very powerful: each object can be defined and maintained independently of the others. This definition of an object is very similar to the definition of an entity given in Section 12.1.1. However, an object encapsulates both state and behavior; an entity models only state.
The current state of an object is described by one or more attributes (instance variables). For example, the branch office at 163 Main St may have the attributes shown in Table K.1. Attributes can be classified as simple or complex. A simple attribute can be a primitive type such as integer, string, real, and so on, which takes on literal values; for example, `branchNo` in Table K.1 is a simple attribute with the literal value “B003”. A complex attribute can contain collections and/or references. For example, the attribute `SalesStaff` is a collection of `Staff` objects. A reference attribute represents a relationship between objects and contains a value, or collection of values, which are themselves objects (for example, `SalesStaff` is, more precisely, a collection of references to `Staff` objects). A reference attribute is conceptually similar to a foreign key in the relational data model or a pointer in a programming language. An object that contains one or more complex attributes is called a complex object (see Section K.9).

Attributes are generally referenced using dot notation. For example, the street attribute of a branch object is referenced as:

```
branchObject.street
```

### K.3 Object Identity

A key part of the definition of an object is unique identity. In an object-oriented system, each object is assigned an **Object Identifier (OID)** when it is created that is:

- system-generated;
- unique to that object;
- invariant, in the sense that it cannot be altered during its lifetime. Once the object is created, this OID will not be reused for any other object, even after the object has been deleted;
- independent of the values of its attributes (that is, its state). Two objects could have the same state but would have different identities;
- invisible to the user (ideally).

Thus, object identity ensures that an object can always be uniquely identified, thereby automatically providing entity integrity (see Section 4.3.2). In fact, as object identity ensures uniqueness systemwide, it provides a stronger constraint than the relational data model’s entity integrity, which requires only uniqueness.

<table>
<thead>
<tr>
<th>ATTRIBUTE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchNo</td>
<td>B003</td>
</tr>
<tr>
<td>street</td>
<td>163 Main St</td>
</tr>
<tr>
<td>city</td>
<td>Glasgow</td>
</tr>
<tr>
<td>postcode</td>
<td>G11 9QX</td>
</tr>
<tr>
<td>SalesStaff</td>
<td>Ann Beech; David Ford</td>
</tr>
<tr>
<td>Manager</td>
<td>Susan Brand</td>
</tr>
</tbody>
</table>
within a relation. In addition, objects can contain, or refer to, other objects using object identity. However, for each referenced OID in the system, there should always be an object present that corresponds to the OID; that is, there should be no **dangling references**. For example, in the *DreamHome* case study, we have the relationship `Branch Has Staff`. If we embed each branch object in the related staff object, then we encounter the problems of information redundancy and update anomalies discussed in Section 14.2. However, if we instead embed the OID of the branch object in the related staff object, then there continues to be only one instance of each branch object in the system and consistency can be maintained more easily. In this way, objects can be *shared* and OIDs can be used to maintain referential integrity (see Section 4.3.3). We discussed referential integrity in OODBMSs in Section 27.9.2.

There are several ways in which object identity can be implemented. In an RDBMS, object identity is **value-based**: the primary key is used to provide uniqueness of each tuple in a relation. Primary keys do not provide the type of object identity that is required in object-oriented systems. First, as already noted, the primary key is only unique within a relation, not across the entire system. Second, the primary key is generally chosen from the attributes of the relation, making it dependent on object state. If a potential key is subject to change, identity has to be simulated by unique identifiers, such as the branch number `branchNo`, but as these are not under system control, there is no guarantee of protection against violations of identity. Furthermore, simulated keys such as B001, B002, or B003, have little semantic meaning to the user.

Other techniques that are frequently used in programming languages to support identity are variable names and pointers (or virtual memory addresses), but these approaches also compromise object identity (Khoshafian and Abnous, 1990). For example, in C and C++ an OID is a physical address in the process memory space. For most database purposes, this address space is too small: scalability requires that OIDs be valid across storage volumes, possibly across different computers for distributed DBMSs. Further, when an object is deleted, the memory formerly occupied by it should be reused, and so a new object may be created and allocated to the same space as the deleted object occupied. All references to the old object, which became invalid after the deletion, now become valid again, but unfortunately referencing the wrong object. In a similar way, moving an object from one address to another invalidates the object’s identity. What is required is a **logical object identifier** that is independent of both state and location. We discussed logical and physical OIDs in Section 27.6.

There are several advantages to using OIDs as the mechanism for object identity:

- **They are efficient.** OIDs require minimal storage within a complex object. Typically, they are smaller than textual names, foreign keys, or other semantic-based references.

- **They are fast.** OIDs point to an actual address or to a location within a table that gives the address of the referenced object. This means that objects can be located quickly whether they are currently stored in local memory or on disk.

- **They cannot be modified by the user.** If the OIDs are system-generated and kept invisible, or at least read-only, the system can ensure entity and referential integrity more easily. Further, this avoids the user having to maintain integrity.
- They are independent of content. OIDs do not depend upon the data contained in the object in any way. This allows the value of every attribute of an object to change, but for the object to remain the same object with the same OID.

Note the potential for ambiguity that can arise from this last property: two objects can appear to be the same to the user (all attribute values are the same), yet have different OIDs and so be different objects. If the OIDs are invisible, how does the user distinguish between these two objects? From this we may conclude that primary keys are still required to allow users to distinguish objects. With this approach to designating an object, we can distinguish between object identity (sometimes called object equivalence) and object equality. Two objects are identical (equivalent) if and only if they are the same object (denoted by “=”), that is, their OIDs are the same. Two objects are equal if their states are the same (denoted by “==”). We can also distinguish between shallow and deep equality: objects have shallow equality if their states contain the same values when we exclude references to other objects; objects have deep equality if their states contain the same values and if related objects also contain the same values.

K.4 Methods and Messages

An object encapsulates both data and functions into a self-contained package. In object technology, functions are usually called methods. Figure K.1 provides a conceptual representation of an object, with the attributes on the inside protected from the outside by the methods. Methods define the behavior of the object. They can be used to change the object’s state by modifying its attribute values, or to query the values of selected attributes. For example, we may have methods to add a new property for rent at a branch, to update a member of staff’s salary, or to print out a member of staff’s details.

A method consists of a name and a body that performs the behavior associated with the method name. In an object-oriented language, the body consists of a block of code that carries out the required functionality. For example, Figure K.2 represents the method to update a member of staff’s salary. The name of the method is updateSalary, with an input parameter increment, which is added to the instance variable salary to produce a new salary.
Messages are the means by which objects communicate. A message is simply a request from one object (the sender) to another object (the receiver) asking the second object to execute one of its methods. The sender and receiver may be the same object. Again, dot notation is generally used to access a method. For example, to execute the `updateSalary` method on a `Staff` object, `staffObject`, and pass the method an increment value of 1000, we write:

```
staffObject.updateSalary(1000)
```

In a traditional programming language, a message would be written as a function call:

```
updateSalary(staffObject, 1000)
```

K.5 Classes

In Simula, classes are blueprints for defining a set of similar objects. Thus, objects that have the same attributes and respond to the same messages can be grouped together to form a class. The attributes and associated methods are defined once for the class rather than separately for each object. For example, all branch objects would be described by a single `Branch` class. The objects in a class are called instances of the class. Each instance has its own value(s) for each attribute, but shares the same attribute names and methods with other instances of the class, as illustrated in Figure K.3.

In the literature, the terms “class” and “type” are often used interchangeably, although some authors make a distinction between the two terms as we now describe. A type corresponds to the notion of an abstract data type (Atkinson and Buneman, 1989). In programming languages, a variable is declared to be of a particular type. The compiler can use this type to check that the operations performed on the variable are compatible with its type, thus helping to ensure the correctness of the software. On the other hand, a class is a blueprint for creating objects and provides methods that can be applied on the objects. Thus, a class is referred to at runtime rather than compile time.

In some object-oriented systems, a class is also an object and has its own attributes and methods, referred to as class attributes and class methods, respectively. Class attributes describe the general characteristics of the class, such as totals or averages; for example, in the class `Branch` we may have a class attribute for the total number of branches. Class methods are used to change or query the state of class attributes. There are also special class methods to create new instances of the class and to destroy those that are no longer required. In an object-oriented language, a new instance is normally created by a method called `new`. Such methods are usually called constructors. Methods for destroying objects and reclaiming the space occupied are typically called destructors. Messages sent to a class method are

---

**Figure K.2**

Example of a method.

```java
method void updateSalary(float increment)
{
    salary = salary + increment;
}
```
sent to the class rather than an instance of a class, which implies that the class is an instance of a higher-level class, called a metaclass.

### K.6 Subclasses, Superclasses, and Inheritance

Some objects may have similar but not identical attributes and methods. If there is a large degree of similarity, it would be useful to be able to share the common properties (attributes and methods). Inheritance allows one class to be defined as a special case of a more general class. These special cases are known as subclasses, and the more general cases are known as superclasses. The process of forming a superclass is referred to as generalization and the process of forming a subclass is specialization. By default, a subclass inherits all the properties of its superclass(es) and, additionally, defines its own unique properties. However, as you will see shortly, a subclass can also redefine inherited properties. All instances of the subclass are also instances of the superclass. In addition, the principle of substitutability states that we can use an instance of the subclass whenever a method or a construct expects an instance of the superclass.

The concepts of superclass, subclass, and inheritance are similar to those discussed for the Enhanced Entity–Relationship (EER) model in Chapter 13, except that in the object-oriented paradigm, inheritance covers both state and behavior. The relationship between the subclass and superclass is sometimes referred to as
A KIND OF (AKO) relationship; for example, a Manager is AKO Staff. The relationship between an instance and its class is sometimes referred to as IS-A; for example, Susan Brand IS-A Manager.

There are several forms of inheritance: single inheritance, multiple inheritance, repeated inheritance, and selective inheritance. Figure K.4 shows an example of single inheritance, where the subclasses Manager and SalesStaff inherit the properties of the superclass Staff. The term “single inheritance” refers to the fact that the subclasses inherit from no more than one superclass. The superclass Staff could itself be a subclass of a superclass, Person, thus forming a class hierarchy.

Figure K.5 shows an example of multiple inheritance in which the subclass SalesManager inherits properties from both the superclasses Manager and SalesStaff. The provision of a mechanism for multiple inheritance can be quite problematic, as it has to provide a way of dealing with conflicts that arise when the superclasses contain the same attributes or methods. Not all object-oriented languages and DBMSs support multiple inheritance as a matter of principle. Some authors claim that multiple inheritance introduces a level of complexity that is hard to manage safely and consistently. Others argue that it is required to model the real world, as in this example. Those languages that do support it handle conflict in a variety of ways, such as:

- Include both attribute/method names and use the name of the superclass as a qualifier. For example, if bonus is an attribute of both Manager and SalesStaff, the subclass SalesManager could inherit bonus from both superclasses and qualify the instance of bonus in SalesManager as either Manager.bonus or SalesStaff.bonus.
• Linearize the inheritance hierarchy and use single inheritance to avoid conflicts. With this approach, the inheritance hierarchy of Figure K.5 would be interpreted as:

SalesManager → Manager → SalesStaff

or

SalesManager → SalesStaff → Manager

With the previous example, SalesManager would inherit one instance of the attribute bonus, which would be from Manager in the first case, and SalesStaff in the second case.

• Require the user to redefine the conflicting attributes or methods.
• Raise an error and prohibit the definition until the conflict is resolved.

Repeated inheritance is a special case of multiple inheritance where the superclasses inherit from a common superclass. Extending the previous example, the classes Manager and SalesStaff may both inherit properties from a common superclass Staff, as illustrated in Figure K.6. In this case, the inheritance mechanism must ensure that the SalesManager class does not inherit properties from the Staff class twice. Conflicts can be handled as discussed for multiple inheritance.

Selective inheritance allows a subclass to inherit a limited number of properties from the superclass. This feature may provide similar functionality to the view mechanism discussed in Section 7.4 by restricting access to some details but not others.

K.7 Overriding and Overloading

As we have just mentioned, properties (attributes and methods) are automatically inherited by subclasses from their superclasses. However, it is possible to redefine a property in the subclass. In this case, the definition of the property in the subclass is the one used. This process is called overriding. For example, we might define a method in the Staff class to increment salary based on a commission:

```java
method void giveCommission(float branchProfit) {
    salary = salary + 0.02 * branchProfit;
}
```
Appendix K  Introduction to Object-Oriented Concepts

Figure K.7
Overloading print method: (a) for Branch object; (b) for Staff object.

```java
method void print() {
    print("Branch number: %s\n", branchNo);
    print("Street: %s\n", street);
    print("City: %s\n", city);
    print("Postcode: %s\n", postcode);
}

(a)

method void print() {
    print("Staff number: %s\n", staffNo);
    print("First name: %s\n", fName);
    print("Last name: %s\n", lName);
    print("Position: %s\n", position);
    print("Sex: %c\n", sex);
    print("Date of birth: %s\n", DOB);
    print("Salary: %f\n", salary);
}

(b)
```

However, we may wish to perform a different calculation for commission in the Manager subclass. We can do this by redefining, or overiding, the method giveCommission in the Manager subclass:

```java
method void giveCommission(float branchProfit) {
    salary = salary + 0.05 * branchProfit;
}
```

The ability to factor out common properties of several classes and form them into a superclass that can be shared with subclasses can greatly reduce redundancy within systems and is regarded as one of the main advantages of object orientation. Overriding is an important feature of inheritance, as it allows special cases to be handled easily with minimal impact on the rest of the system.

Overriding is a special case of the more general concept of overloading. Overloading allows the name of a method to be reused within a class definition or across class definitions. This means that a single message can perform different functions depending on which object receives it and, if appropriate, what parameters are passed to the method. For example, many classes will have a print method to print out the relevant details for an object, as shown in Figure K.7.

Overloading can greatly simplify applications, as it allows the same name to be used for the same operation irrespective of what class it appears in, thereby allowing context to determine which meaning is appropriate at any given moment. This avoids the necessity of having to provide unique names for methods such as printBranchDetails or printStaffDetails for what is in essence the same functional operation.

K.8  Polymorphism and Dynamic Binding

Overloading is a special case of the more general concept of polymorphism, from the Greek meaning “having many forms.” There are three types of polymorphism: operation, inclusion, and parametric (Cardelli and Wegner, 1985). Overloading, as in the previous example, is a type of operation (or ad hoc) polymorphism. A method defined in a superclass and inherited in its subclasses is an example of inclusion polymorphism. Parametric polymorphism, or genericity as it is sometimes called,
uses types as parameters in generic type, or class, declarations. For example, the following template definition:

```cpp
template <type T>
T max(x:T, y:T) {
    if (x > y) return x;
    else return y;
}
```

defines a generic function `max` that takes two parameters of type `T` and returns the maximum of the two values. This piece of code does not actually establish any methods. Rather, the generic description acts as a template for the later establishment of one or more different methods of different types. Actual methods are instantiated as:

```cpp
int max(int, int); // instantiate max function for two integer types
real max(real, real); // instantiate max function for two real types
```

The process of selecting the appropriate method based on an object’s type is called **binding**. If the determination of an object’s type can be deferred until runtime (rather than compile time), the selection is called **dynamic (late) binding**. For example, consider the class hierarchy of `Staff` with subclasses `Manager` and `SalesStaff` shown in Figure K.4, and assume that each class has its own `print` method to print out relevant details. Further assume that we have a list consisting of an arbitrary number of objects, say `n`, from this hierarchy. In a conventional programming language, we would need a `CASE` statement or a nested `IF` statement to print out the corresponding details:

```cpp
FOR i = 1 TO n DO
    SWITCH (list[i].type) {
        CASE staff:
            printStaffDetails(list[i].object); break;
        CASE manager:
            printManagerDetails(list[i].object); break;
        CASE salesPerson:
            printSalesStaffDetails(list[i].object); break;
    }
```

If a new type is added to the list, we have to extend the `CASE` statement to handle the new type, forcing recompilation of this piece of software. If the language supports dynamic binding and overloading, we can overload the `print` methods with the single name `print` and replace the `CASE` statement with the line:

```cpp
list[i].print()
```

Furthermore, with this approach we can add any number of new types to the list and, provided that we continue to overload the `print` method, no recompilation of this code is required. Thus, the concept of polymorphism is orthogonal to (that is, independent of) inheritance.

### K.9 Complex Objects

There are many situations in which an object consists of subobjects or components. A complex object is an item that is viewed as a single object in the real world but combines with other objects in a set of complex **A-PART-OF** relationships (**APO**).
The objects contained may themselves be complex objects, resulting in an A-PART-OF hierarchy. In an object-oriented system, a contained object can be handled in one of two ways. First, it can be encapsulated within the complex object and thus form part of the complex object. In this case, the structure of the contained object is part of the structure of the complex object and can be accessed only by the complex object’s methods. On the other hand, a contained object can be considered to have an independent existence from the complex object. In this case, the object is not stored directly in the parent object but only its OID. This is known as referential sharing (Khoshafian and Valduriez, 1987). The contained object has its own structure and methods, and can be owned by several parent objects.

These types of complex object are sometimes referred to as structured complex objects, as the system knows the composition. The term unstructured complex object is used to refer to a complex object whose structure can be interpreted only by the application program. In the database context, unstructured complex objects are sometimes known as Binary Large Objects (BLOBs), which we discussed in Section 27.2.

### Appendix Summary

- The concept of **encapsulation** means that an object contains both a data structure and the set of operations that can be used to manipulate it. The concept of **information hiding** means that the external aspects of an object are separated from its internal details, which are hidden from the outside world.

- An **object** is a uniquely identifiable entity that contains both the attributes that describe the state of a ‘real world’ object and the actions that are associated with it. Objects can contain other objects. A key part of the definition of an object is unique identity. In an object-oriented system, each object has a unique system-wide identifier that is independent of the values of its attributes and, ideally, invisible to the user.

- **Methods** define the behavior of the object. They can be used to change the object’s state by modifying its attribute values or to query the value of selected attributes. **Messages** are the means by which objects communicate. A message is simply a request from one object (the sender) to another object (the receiver) asking the second object to execute one of its methods. The sender and receiver may be the same object.

- Objects that have the same attributes and respond to the same messages can be grouped together to form a **class**. The attributes and associated methods can then be defined once for the class rather than separately for each object. A class is also an object and has its own attributes and methods, referred to as **class attributes** and **class methods**, respectively. Class attributes describe the general characteristics of the class, such as totals or averages.

- **Inheritance** allows one class to be defined as a special case of a more general class. These special cases are known as **subclasses** and the more general cases are known as **superclasses**. The process of forming a superclass is referred to as **generalization**, forming a subclass is **specialization**. A subclass inherits all the properties of its superclass and additionally defines its own unique properties. All instances of the subclass are also instances of the superclass. The principle of substitutability states that an instance of the subclass can be used whenever a method or a construct expects an instance of the superclass.

- **Overloading** allows the name of a method to be reused within a class definition or across definitions. **Overriding**, a special case of overloading, allows the name of a property to be redefined in a subclass. **Dynamic binding** allows the determination of an object’s type and methods to be deferred until runtime.