Inheritance and Polymorphism

Objectives
Pre-Test Questions
Introduction
Inheritance
Inheriting Attributes
Inheriting Methods
Extending Components with Inheritance
Polymorphism
Abstract Classes
Multiple Inheritance
Summary
Post-Test Questions
OBJECTIVES

At the completion of this chapter, you will be able to:

- Identify the benefits of inheritance.
- Identify the benefits of polymorphism.
- Define the abstract class.
- Identify the benefits of multiple inheritance.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. How does inheritance enable software reuse?

2. How does polymorphism overcome some of the limits of inheritance?
INTRODUCTION

This chapter examines two powerful components of object oriented programming, inheritance and polymorphism. Inheritance takes advantage of the similarities that exist between various classes; its benefits include data generalization and code reduction. Inheritance can be compared to scientific classification, because science begins with a large collection of objects which are then split into refined classes (this is called encapsulation). These classes are then organized hierarchically into more abstract classes (inheritance). In software development, the design of classes and of the inheritance hierarchies occurs simultaneously. Polymorphism uses inheritance to gain system extensibility. For example, a parent class may contain a virtual function, which can be redefined by its children. Using polymorphism, new objects can be defined without introducing new code.

INHERITANCE

In the previous lesson, you learned that objects can relate to each other with either a has a or uses a relationship. Inheritance is a third type of relationship between objects: the is a relationship.

For example, a library lends more than just books; it also lends magazines, audiocassettes, and microfilm. On some level, all of these items can be treated in the same manner, since all four types represent lendable assets. However, also consider the differences: A book has an ISBN and a magazine does not; audiocassettes have a play length; and microfilm cannot be checked out overnight.

Each of these assets should be represented by its own class definition. However, without inheritance, each class must independently implement the characteristics that are common to all lendable assets. All assets are either checked out, or available for checkout. All assets have a title, a date of acquisition, and a replacement cost. Rather than duplicate functionality, inheritance allows you to inherit functionality from another class, called a superclass.
Rather than duplicate functionality, object-oriented programming allows you to define a base class, such as LendableAsset, which maintains the characteristics common to all lendable assets. Figure 3-1 illustrates the relationship between the LendableAsset class and the classes that inherit from it.

Figure 3-1: Inheritance
INHERITING ATTRIBUTES

LendableAsset is a superclass that maintains only the data and methods that are common to all lendable assets. Book, Magazine, Audiocassette, and Microfilm are subclasses of the LendableAsset class, and so they inherit these characteristics. The inheritance relationship is called the "is a" relationship. A Book is a LendableAsset, as are a Magazine, an Audiocassette, and Microfilm. Following are the class definitions for the LendableAsset, Book, and Audiocassette classes using Java:

class LendableAsset
{
    String title;
    Date acquisitionDate;
    float replacementCost;
    boolean checkedOut;
}

class Book extends LendableAsset
{
    String author;
    String isbn;
}

class Audiocassette extends LendableAsset
{
    int playLength;
}
In the previous example, the `LendableAsset` class defines properties relevant to all lendable assets. The `Book` and `Audiocassette` classes inherit these properties. In Java, the `extends` keyword is used to indicate inheritance. When these subclasses are instantiated, the properties they inherit from the `LendableAsset` class can be accessed using dot notation. For example:

```java
Book catcherInTheRye = new Book();
catcherInTheRye.title = "Catcher in the Rye";
catcherInTheRye.author = "J. D. Salinger";
catcherInTheRye.isbn = "0-316-76948-7";
catcherInTheRye.checkedOut = false;
```

**INHERITING METHODS**

Methods can also be inherited. Figure 3-2 illustrates the relationships between the `Patron`, `LendableAsset`, `Book`, and `Audiocassette` classes.

![Figure 3-2: Inheritance relationships](image-url)
A Patron can check out a LendableAsset. "Check out" is an example of a uses a relationship, which often translate into class methods. The following code is modified to include a checkOut method in the LendableAsset class:

```java
class LendableAsset
{
    String title;
    Date acquisitionDate;
    float replacementCost;
    boolean checkedOut;
    public void checkOut()
    {
        checkedOut = true;
    }
}
```

Using this new LendableAsset class definition, the Book and Audiocassette classes automatically inherit the ability to be checked out. The following Java code illustrates this fact:

```java
Book bookInPrint = new Book();
Audiocassette bookOnTape = new Audiocassette();

bookInPrint.checkOut();
bookOnTape.checkOut();
```
EXTENDING COMPONENTS WITH INHERITANCE

One of the most powerful features of inheritance is the ability to extend components, without any knowledge of class implementation. The following code is a simplified class definition for the Date class in the Java API:

```java
public class Date extends Object {
    // Public Constructors
    public Date();
    public Date(long date);
    // Methods
    public boolean after(Date when);
    public int getDate();
    public int getDay();
    public int getHours();
    public int getMinutes();
    public int getMonth();
    public int getSeconds();
    public int getYear();
    public void setDate(int date);
    public void setHours(int hours);
    public void setMinutes(int minutes);
    public void setMonth(int month);
    public void setSeconds(int seconds);
    public void setYear(int year);
}
```
This class definition depicts only the publicly defined methods of the Date class. No implementation details are known. The problem is that the Java Date class provides no mechanism for determining the date three weeks in the future (when a LendableAsset is due). Using inheritance, this functionality can be added in a derived class, called DueDate. The following code defines the new DueDate class:

class DueDate extends Date
{
    public void addDays(int days)
    {
        setDate(getDate() + days);
    }

    public void addHours(int hours)
    {
        setHours(getHours() + hours);
    }
}
The new DueDate class inherits the functionality of the Java API Date class, and adds two new methods to allow calculation of a due date. Figure 3-3 illustrates the relationship between the LendableAsset class and the new DueDate class.

Figure 3-3: DueDate Class
In the previous section, you learned that classes can inherit methods from other classes. The Book and Audiocassette classes inherit the checkOut method from the LendableAsset class. However, behavior may need to be modified depending on the subclass. For example, all lendable assets have a due date. Books and audiocassettes can be checked out for 21 days, but microfilm is due back within three hours. Polymorphism allows subclasses to override methods defined in their base class. Because assets are generally due in 21 days, the calculateDueDate method adds 21 days to the current date to calculate a due date. Study the following code:

```java
class LendableAsset {
    String title;
    Date acquisitionDate;
    float replacementCost;
    boolean checkedOut;
    DueDate dueDate;

    public void checkout() {
        checkedOut = true;
    }

    public void calculateDueDate() {
        dueDate = new DueDate();
        dueDate.addDays(21);
    }
}
```
The Microfilm class can use polymorphism to override the calculateDueDate method defined in the LendableAsset class, as shown:

```java
class Microfilm extends LendableAsset
{
    public void calculateDueDate()
    {
        dueDate = new DueDate();
        dueDate.addHours(3);
    }
}
```

When calculateDueDate is called on a LendableAsset object, a decision is made. If the object is of type Book or Audiocassette, the calculateDueDate method defined in the base class LendableAsset is called. If the object is of type Microfilm, the calculateDueDate method defined in the Microfilm class is called. This process is called late binding.

**Abstract Classes**

In the preceding section, you learned that subclasses can override methods defined in the classes from which they are derived. In the example, the Microfilm class defines a method, CalculateDueDate, that overrides the same method defined in the base class LendableAsset.

From a design perspective, this relationship seems incorrect. The LendableAsset class defines the CalculateDueDate method that assumes the due date to be three weeks in the future. While all instances of LendableAsset have a due date, not all calculate that due date in the same way. It becomes the responsibility of the derived class to identify itself as a special case and override this method to calculate a correct due date.
The answer to this design issue is an abstract class. An abstract class declares methods without providing any implementation. When a subclass inherits from an abstract base class, it must override these methods to provide the requisite functionality. The abstract class cannot be instantiated, it can only be extended. The following code is modified to define LendableAsset as an abstract class:

```java
abstract class LendableAsset
{
    String title;
    Date acquisitionDate;
    float replacementCost;
    boolean checkedOut;
    DueDate dueDate;

    public void checkout()
    {
        checkedOut = true;
    }
    public void calculateDueDate();
}

class Book extends LendableAsset
{
    String author;
    String isbn;

    public void calculateDueDate()
    {
        dueDate = new DueDate();
        dueDate.addDays(21);
    }
}
```
class Audiocassette extends LendableAsset
{
    int playLength;

    public void calculateDueDate()
    {
        dueDate = new DueDate();
        dueDate.addDays(21);
    }
}
class Microfilm extends LendableAsset
{
    public void calculateDueDate()
    {
        dueDate = new DueDate();
        dueDate.addHours(3);
    }
}

Notice that the calculateDueDate method is declared in the LendableAsset class, but is only implemented by the derived classes Book, Audiocassette, and Microfilm. LendableAsset is an abstract class, and because it is missing functionality, it cannot be instantiated. The abstract class implements functionality in derived classes; it defines a common interface to all derived classes without making unwarranted assumptions about implementation.
MULTIPLE INHERITANCE

Early in this chapter, you learned that inheritance allows the Book, Magazine, Audiocassette, and Microfilm classes to inherit common characteristics from a base class, LenadableAsset. Occasionally, however, single inheritance is not enough.

For example, a magazine is a lendable asset, and it is also a subscribable asset. The library system must be able to track magazine subscriptions and automate subscription renewal. If Magazine were the only class that ever exhibited these characteristics, you could build the subscribable functionality into the Magazine subclass. However, newspapers can be added to the library system. Recent newspapers are available for reading in the library, but they are not lent out. Since newspapers are subscribable assets, the library system must also be able to track its newspaper subscriptions.

Multiple inheritance allows the Magazine class to inherit characteristics from both the LenadableAsset class and the SubscribableAsset class. The following C++ code demonstrates multiple inheritance using the Magazine class:

```cpp
class SubscribableAsset {
public:
    float subscriptionPrice;
    Date renewDate;

    void renewSubscription();
};

class Newspaper : SubscribableAsset {
public:
    Date date;
};
```
class Magazine : SubscribableAsset, LendableAsset
{
    public:
        int volume;
        int issue;
}

The Newspaper class extends the SubscribableAsset class, and the Magazine class extends both the SubscribableAsset class and the LendableAsset class. Figure 3-4 illustrates the relationship between the Magazine class and both of its parent classes.

Figure 3-4: Multiple inheritance
**Exercise 3-1: Using inheritance and polymorphism**

1. Consider the grocery store inventory system from the previous chapter’s exercise. Identify classes that could benefit from multiple inheritance.
2. Develop a revised class diagram to add multiple inheritance to the grocery store inventory system.

**SUMMARY**

Inheritance is a third type of object relationship, known as the is-a relationship. Inheritance allows a class to inherit the attributes and functionality of a superclass, or base class. Inheritance can be used to extend components even if very little is known about a class’s implementation. Polymorphism allows subclasses to override methods defined higher in the class hierarchy. Abstract classes allow you to avoid making presumptions about implementation before the design requires it. Multiple inheritance is available with some object-oriented programming languages; it allows a class to inherit from two or more superclasses.

**POST-TEST QUESTIONS**

The answers to these questions are in Appendix A at the end of this manual.

1. What is the responsibility of all subclasses that inherit from an abstract class?

2. Is intricate knowledge of the implementation of a class necessary in order to inherit from it?