Chapter 4
Object-Oriented Programming

Dating back to the 1960s, a succession of innovative languages, including Simula, Smalltalk, Eiffel, and C++, introduced object-oriented programming concepts to the software world. Primarily due to their innate ability to effectively manage complexity, object-oriented languages have been embraced by the programming community. As pure object-oriented languages, Java and C# continue this tradition and further the evolution of object-oriented programming. Though their object-oriented features are similar in many respects, there are some stark contrasts in the manner in which Java and C# implement object-oriented constructs. In this chapter, the following concepts will be discussed in detail:

- Object-Oriented Programming Principles
- Classes
- Methods
- Interfaces
- Structs

Object-Oriented Programming Principles

Writing software is an inherently complex process in which code may be structured in an infinite number of ways. Many different programming methodologies have been proposed to help manage this complexity. Of these methodologies, object-oriented programming is currently the most widely followed. Object-oriented programming simplifies complex software systems by encapsulating program data and functions into discrete, self-contained components called objects. An *object* is a construct that stores data and the functions that perform operations on that data. The data stored in an object are referred to as its *properties* while the operations it performs are called *methods*. For example, a `Shape` object may expose properties that allow the programmer to indicate its `Type`, `Size`, and `Color`. Once these characteristics are specified, the programmer may then invoke the object's `Draw()` method in order to instruct the object to draw itself to the screen according to its current property settings.

Object-oriented programming techniques simplify complex software systems by encapsulating functional components into discrete objects that are easy to use and understand. To illustrate, imagine being tasked with writing an application to run your local library. This program would be responsible for signing up new patrons and managing the check-in/check-out process. Considering the innumerable ways that a library system may be implemented, this may seem like a very complex job. However, object-oriented programming simplifies the process by allowing the programmer to model the application after physical objects that exist in the real world. For instance, the programmer of a library system may begin by breaking the system into familiar objects such as `Book` and `Patron`. The `Book` object may contain properties such as `Title` and `Author` and methods like `IsReserved()`, `CheckIn()` and `CheckOut()`. On the other hand, the `Patron` object might expose `Name` and `PhoneNumber` properties as well as

`SignUp()` and `Suspend()` methods. Encapsulating data and methods inside of an object provides a logical grouping of information and functionality (e.g., information about a book is stored within the `Book` object) and allows method implementations to change without affecting client applications that use the object. As you can imagine, writing and testing the `Book` and `Patron` objects is typically much simpler than a procedural approach requiring development of an extensive function library that requires data to be stored as global variables or repeatedly passed around as parameters.

In addition to encapsulation, two other principles are primary to the object-oriented programming paradigm: inheritance and polymorphism. The following sections explore each of these principles.

Encapsulation

Encapsulation is a method of keeping together data structures and the methods that operate on them. A complex program can be significantly simplified by grouping related data and methods into a common object. Encapsulation also means that the internal workings of a method are hidden from clients that use the object. Since they are private to the object itself, method implementations can be altered and improved without adversely affecting client applications.

Inheritance

Objects are created from templates known as classes. A class defines the data members (a.k.a., properties) and methods that will be exposed by each object. A powerful feature of object-oriented programming is the ability to derive a new class from an existing class. *Inheritance* means that a derived class (a.k.a., subclass) inherits all non-private fields and methods from its parent class (a.k.a., base class). For example, assume that you are building an application to keep track of a fleet of heterogeneous vehicles. These vehicles include cars, planes, and boats. Since all of these vehicles have some characteristics in common (e.g., they all have a motor and a color), a single parent class called `Vehicle` can be defined that contains the properties and methods that all of these vehicles have in common. In turn, specific subclasses can be declared that inherit all of the characteristics (i.e., properties and methods) of the `Vehicle` class plus add additional characteristics that are specific to a particular type of vehicle. For instance, a `Vehicle` class might define properties like `Year`, `Make`, `Model`, `Color`, and `MotorType` and methods such as `StartMotor()` and `TurnOnLights()`. Once the `Vehicle` class is defined, the specific subclasses `Car`, `Plane`, and `Boat` can be derived (or extend) from the `Vehicle` parent class. In this way, each specific vehicle type inherits characteristics (i.e., properties and methods) that are common to all vehicles. Subclasses can then differentiate themselves from the parent class as well as other subclasses by adding vehicle-specific properties and methods. For instance, the `Car` class may have a `NumberOfDoors` property and a `Drive()` method while the `Plane` class may have a `WingSpan` property and a `Fly()` method. As you can see, inheritance helps maximize code reuse by allowing all common properties and methods to be contained within a single base class and inherited by numerous subclasses without having to copy or re-write the common code for each new class.

Polymorphism

As we will discuss later in this chapter, interfaces are another important aspect of object-oriented programming. An interface is a collection of method signatures without actual implementations. In code, a class can be declared to "implement" an interface meaning that it provides a concrete implementation of all methods declared in the interface. *Polymorphism* refers to the condition

where two classes that implement the same interface behave differently. For instance, consider an interface named `Flyable` (or `IFly` if you're using .NET naming conventions) that specifies a single method called `Fly()`. Now imagine two different objects, `Airplane` and `HotAirBalloon`, that both implement the `Flyable` interface. Though each object implements a method named `Fly()`, individual implementations may differ greatly. For instance, the `Airplane` object's `Fly()` method may start the plane's engine while the `HotAirBalloon` object's `Fly()` method might ignite the balloon's burner. Even they are both `Flyable` objects, they each implement the `Fly()` method in their own way. Polymorphism allows objects that appear similar (i.e., subscribe to the same interface) to behave differently.

Classes

Classes form the foundation of object-oriented programming. A *class* is a template from which similar objects are created. That is, objects created from the same class share common attributes and behaviors. These attributes and behaviors are implemented by the properties and methods defined by the class. To illustrate, consider this analogy. A class is like a cookie cutter while an object is like a cookie. The class is used to construct each object and every object created from that class is similar in form and function. Since classes are abstract templates from which concrete objects are created, an individual object is often referred to as an instance of a class. In other words, the terms object and instance are interchangeable.

Basic classes are created similarly in Java and C#. For example, consider the following Java and C# implementations of a simple `Person` class.

Listing 4.1 ***Person class (Java)***

```
public class Person
{
  public String name;
  public int age;
  public String gender;
}
```

Listing 4.2 ***Person class (C#)***

```
public class Person
{
  public string Name;
  public int Age;
  public string Gender;
}
```

As you can see, the basic elements of Java and C# classes are practically identical. In both languages, classes are declared with the `class` keyword and class elements are defined within braces following the declaration. Though simple Java and C# classes are nearly identical, their similarities diminish as classes become more complex.

Inheritance

As discussed earlier, inheritance allows a new class to assume the attributes and behaviors of an existing class. The syntax for inheritance differs between Java and C#. For instance, imagine a

class called `Employee` that extends (or inherits from) the `Person` class. The Java version of `Employee` uses the `extends` keyword like this:

```
public class Employee extends Person
{
  public String title;
  public float salary;
}
```

Since it extends `Person`, the `Employee` class exposes the `name`, `age`, and `gender` attributes (inherited from `Person`) in addition to the `title` and `salary` attributes that it defines. In contrast to the Java syntax, C# declares inheritance using the colon (":") operator like this:

```
public class Employee : Person
{
  public string Title;
  public float Salary;
}
```

In this case, the `extends` keyword in Java is directly equivalent to the colon operator in C#. However, later in this chapter we will see how C# uses the colon operator to declare more than just inheritance (see the *Interfaces* section later in this chapter).

Unlike C++, Java and C# support only *single inheritance*. This means that classes can only extend (and inherit from) a single class. Java and C# designers chose to restrict their languages to single inheritance in order to avoid the complexity inherent with multiple inheritance.

Modifiers

Class modifiers define a class's accessibility or some other characteristic. For example, the `public` modifier shown in Listings 4.1 and 4.2 indicates that the class is globally accessible from any other class. In addition to `public`, Java and C# define many more class modifiers. Table 4.1 describes the modifiers supported by Java and C#.

Table 4.1 ***Java and C# Class Modifiers***

Java Modifier	C# Modifier	Description
`private`	`private`	Class is accessible only from its containing class. Used for inner classes only (inner classes are discussed later in this chapter).
none	`internal`	Class is accessible from any class within its package/assembly.
`--`	`protected`	Class is accessible from its containing class or classes derived from the containing class. Used for inner classes only.
`protected`	`protected internal`	Class is accessible from any subclass or any class within its package/assembly.
`public`	`public`	Class is accessible from any other class.
`abstract`	`abstract`	Class cannot be instantiated. Subclasses of an abstract class can be instantiated if they implement all methods that are marked `abstract`.
`final`	`sealed`	Class cannot be extended. That is, subclasses cannot be created from a class marked `final`/`sealed`. This modifier cannot be used in conjunction with the `abstract` modifier.

Notice that there exists a Java equivalent for each C# class modifier with the exception of the C# `protected` modifier. Though Java defines a modifier named `protected`, Java's version is functionally equivalent to C#'s `protected internal` modifier. The C# `protected` modifier is a bit more restrictive than the Java version. While both Java and C# use the `protected`

modifier to indicate that a class is accessible from its subclasses, Java interprets this modifier a little more liberally allowing access from any class within the same package. Again, C# provides the `protected internal` modifier to achieve this level of accessibility.

Creating Objects

As previously discussed, classes are the templates from which objects are created. In Java and C#, objects are created using the `new` operator and the name of a class like this:

```
Person p = new Person();
```

This line declares `p` to be a variable that points to objects of type `Person` and uses the `new` operator to assign an instance (i.e., object) of the `Person` class to it. This assignment could have been performed in two steps like this:

```
Person p; //declare variable p as type Person
p = new Person(); //assign new instance of Person to p
```

The previous examples demonstrated creating objects without any initialization arguments (i.e., the class name is followed by empty parentheses). However, it is possible to specify initialization values at the time an object is created. For instance, if the `Person` class was constructed to accept name and age initialization values, a properly initialized `Person` object could be created like this:

```
Person p = new Person("Reagan", 4);
```

The number, types, and order of initialization arguments that can be specified when an object is created is defined by each class's constructor.

Constructors

Constructors are special methods that are called whenever a class is instantiated. These methods are most commonly used for initialization tasks. In both Java and C#, constructors look like methods that are named the same as the class and have no return value. For example, Listings 4.3 and 4.4 demonstrate simple Java and C# classes that each have a single constructor that accepts no parameters. Since it takes no parameters, this type of constructor is often referred to as the *empty* or *parameterless* constructor. The empty constructor initializes the `age` variable to `40` when the class is instantiated.

Listing 4.3 ***Person class with simple constructor (Java)***

```
public class Person
{
  public String name;
  public int age;
  public String gender;


  public Person()
  {
    age = 40;
    System.out.println("Person class constructor called!");
  }
}
```

And here is the same class in C#:

Listing 4.4 ***Person class with simple constructor (C#)***

```
public class Person
{
  public string Name;
  public int Age;
  public string Gender;


  public Person()
  {
    age = 40;
    System.Console.WriteLine("Person class constructor called!");
  }
}
```

As you can see, the Java and C# classes are virtually identical. To verify that the constructor is working, you can run this Java program:

```
public class PersonTest
{
  public static void main(String[] args)
  {
    Person person = new Person();

    System.out.println("Name: " + person.name);
    System.out.println("Age: " + person.age);
    System.out.println("Gender: " + person.gender);
  }
}
```

Or this C# program:

```
public class PersonTest
{
  public static void Main()
  {
    Person person = new Person();

    System.Console.WriteLine("Name: {0}", person.Name);
    System.Console.WriteLine("Age: {0}", person.Age);
    System.Console.WriteLine("Gender: {0}", person.Gender);
    System.Console.ReadLine();
  }
}
```

When executed, the Java version of `PersonTest` class produces the following output:

```
Person class constructor called!
Name: null
Age: 40
Gender: null
```

And the C# version generates this output:

```
Person class constructor called!
Name:
```

```
Age: 40
Gender:
```

Notice that Java's `System.out.println()` method actually prints the word `null` for variables having a null value while the `System.Console.WriteLine()` method in C# prints null values as empty strings.

Constructor Parameters

Like standard methods, constructors can accept any type and number of parameters. By adding parameters to the `Person` class's constructor, object initialization can be simplified and accelerated. For example, Listing 4.5 demonstrates a `Person` class that initializes its instance variables according to the parameter values passed to its constructor.

Listing 4.3 ***Person class that uses constructor parameters (Java)***

```
public class Person
{
  public String name;
  public int age;
  public String gender;


  public Person(String name, int age, String gender)
  {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }
}
```

These constructor parameters can then be used to initialize `Person` objects like this in Java (and similarly in C#):

```
public class PersonTest
{
  public static void main(String[] args)
  {
    Person person1 = new Person("Tyler", 9, "Male");
    Person person2 = new Person("Madison", 7, "Female");

    System.out.println("Name: " + person1.name);
    System.out.println("Age: " + person1.age);
    System.out.println("Gender: " + person1.gender);

    System.out.println();

    System.out.println("Name: " + person2.name);
    System.out.println("Age: " + person2.age);
    System.out.println("Gender: " + person2.gender);
  }
}
```

The previous Java class produces the following output:

```
Name: Tyler
Age: 9
Gender: Male
```

```
Name: Madison
Age: 7
Gender: Female
```

Default Constructor

If no constructor is explicitly defined, the Java and C# compilers automatically create an empty (i.e., parameterless) constructor, known as the *default constructor*, so that the class can be instantiated using an empty parameter list like this:

```
Person person = new Person();
```

The compiler implicitly creates the default constructor only if no other constructors are defined by the programmer. This rule is valid even if the programmer creates a constructor that employs a different signature than the default constructor. Therefore, in order to include both a parameterless and parametered constructor within a class, they must both be explicitly defined.

Constructor Overloading

A class can include multiple constructors as long as their signatures vary. Multiple constructors provide flexibility by allowing classes to be created in different ways. When multiple constructors are defined, it is often useful to call one constructor from within another. This technique is commonly used to reduce duplication of code since functionality implemented in one constructor can be invoked from another. The syntax for constructor-to-constructor calls is different in Java and C#. In Java, a constructor can call another constructor using the `this` keyword and a parameter list that matches the target constructor. This call can only be made from the first line of the constructor that issues the call (enforced by the compiler). To demonstrate, Listing 4.4 adds a second constructor to the `Person` class that accepts only two parameters. By moving the code to process these two parameters into a new constructor and calling this constructor from the original constructor, code can be efficiently reused.

Listing 4.4 ***Person class that calls one constructor from another (Java)***

```
public class Person
{
  public String name;
  public int age;
  public String gender;


  public Person(String name, int age)
  {
    this.name = name;
    this.age = age;
  }


  public Person(String name, int age, String gender)
  {
    this(name, age); //must be on first line of constructor
    this.gender = gender;
  }
}
```

Though it also employs the `this` keyword, C# uses a slightly different syntax for constructor-to-constructor calls. The difference lies in the fact that the C# constructor call is defined within the constructor declaration itself rather than in the first line of its body. These two

approaches are functionally equivalent since, in both cases, the target constructor is called before the original constructor is executed. C# uses a colon (“:”) to separate the target constructor call from the definition of the constructor itself. For instance, Listing 4.5 demonstrates how one constructor can be invoked from another in C# (equivalent to Listing 4.4).

Listing 4.5 ***Person class that calls one constructor from another (C#)***

```
public class Person
{
  public string Name;
  public int Age;
  public string Gender;


  public Person(string name, int age)
  {
    Name = name;
    Age = age;
  }


  public Person(string name, int age, string g) : this(name, age)
  {
    Gender = g;
  }
}
```

Constructor Inheritance

Unlike standard methods, constructors are not inherited by subclasses. Each subclass must either define its own constructor(s) or rely on the parameterless default constructor. Fortunately however, it is possible to call a parent class’s constructor from within a subclass’s constructor. In both Java and C#, the syntax for calling a parent’s constructor is nearly identical to that of calling a constructor within the same class. For example, in Java, simply use the `super` keyword (a reference to the parent or super class) in place of the `this` keyword. Like the call to `this`, `super` can only be invoked from the first line of the constructor. Listing 4.6 shows how a parent class’s constructor can be called from a subclass in Java.

Listing 4.6 ***Employee class that calls its parent’s constructor (Java)***

```
public class Employee extends Person
{
  public String title;
  public float salary;


  public Employee(String name, int age, String gender,
    String title, float salary)
  {
    super(name, age, gender); //must be on first line
    this.title = title;
    this.salary = salary;
  }
}
```

In C#, the syntax for calling a parent’s constructor is identical to calling another constructor within the same class with the exception that the `base` keyword is used instead of `this`. For

example, Listing 4.7 demonstrates how a parent class's constructor can be called from a subclass in C#.

Listing 4.7 ***Employee class that calls its parent's constructor (C#)***

```
public class Employee : Person
{
  public string Title;
  public float Salary;


  public Employee(string name, int age, string gender,
    string title, float salary) : base(name, age, gender)
  {
    Title = title;
    Salary = salary;
  }
}
```

Private Constructors

Declaring its constructor as private prevents a class from being instantiated with the `new` operator. Typically used when implementing the Singleton design pattern, a private constructor allows the creator of the class to have complete control over how and when the class is instantiated. For example, Listing 4.8 demonstrates a class that shares a single instance among all clients. Clients that wish to acquire an instance of the class must call the `getInstance()` method. Since the only constructor is private, the class cannot be instantiated using the `new` operator.

Listing 4.8 ***Singleton class that uses a private constructor (Java and C#)***

```
public class MathSingleton
{
  private static MathSingleton ms;


  private MathSingleton()
  {
  }


  public MathSingleton getInstance()
  {
    if (ms == null)
      ms = new MathSingleton();

    return ms;
  }


  public int add(int x, int y)
  {
    return x + y;
  }


  public int subtract(int x, int y)
  {
    return x - y;
  }
```

```
}
```

Instance Initializers (Java)

Not supported by C#, an *instance initializer* is a somewhat curious Java construct that behaves similarly to a parameterless constructor. This construct is primarily intended for use with anonymous classes (discussed later in this chapter) but can be used with any class. An instance initializer is a block of code that resides within a class but outside of any methods. This block of code is executed after the base class's constructor but before its own constructor each time the class is instantiated. Multiple instance initializers are permitted and are executed in the order in which they appear. The following code demonstrates a class that defines two instance initializers:

```
public class InstanceInitializer
{
  {
    System.out.println("First instance initializer called!");
  }


  {
    System.out.println("Second instance initializer called!");
  }


  public static void main(String[] args)
  {
    InstanceInitializer ii = new InstanceInitializer();
  }
}
```

The `InstanceInitializer` class sends the following output to the console when executed:

```
First instance initializer called!
Second instance initializer called!
```

Remember that instance initializers are executed in order after the base class's constructor is called but before its current class's constructor is run.

Destructors/Finalizers

A *destructor* is a special method that is called when the memory occupied by an object is reclaimed. Though destructors in Java and C# are somewhat analogous to destructors in C++, there are some significant differences that make Java and C# destructors far less useful. For instance, unlike C++, Java and C# destructors are non-deterministic. That is, there is no way to know when a particular object's destructor will be called or the order in which multiple objects' destructors will be invoked. This is due to the fact that Java and C# destructors are called by the garbage collector at a time that it deems appropriate. Therefore, when a Java or C# object is no longer referenced, its destructor will eventually be called by the garbage collector (immediately prior to collecting the object) but there is no guarantee when this call will occur. For this reason, programmers should not rely on Java and C# destructors to release critical system resources.

> **NOTE**
> It is possible to request the garbage collector to run and execute the finalizers for any classes ready for collection. Java uses the `Runtime.runFinalization()` and `Runtime.gc()` methods to

explicitly request the execution of finalizers and the garbage collector, respectively. In C#, the `System.GC.Collect()` method explicitly requests object finalization and collection.

Though similar in many respects, there are a few important differences between Java and C# destructors. For instance, destructors in Java are known as *finalizers* (Java does not use the term destructor) and the destructor method is called `finalize()`. C#, on the other hand, supports both C++-style destructor syntax as well as Java-style finalizers. To illustrate, the following code presents a Java class that defines a finalizer method that is called by the garbage collector when the object is reclaimed. In this case, the `finalize()` method is used to close a database connection:

```
import java.sql.Connection;
import java.sql.SQLException;

public class MyDatabaseClass
{
  Connection dbConn;


  public MyDatabaseClass()
  {
    dbConn = getConnection();
  }


  private Connection getConnection()
  {
    //create and return database connection
  }


  protected void finalize() throws SQLException
  {
    if (dbConn != null)
      dbConn.close();
  }
}
```

As previously discussed, C# supports both destructor-style and finalizer-style syntax for performing clean-up operations. A C# destructor is defined the same as in C++. That is, it is named after its class and prefixed with a tilde character ("~") like this:

```
using System.Data.SqlClient;

public class MyDatabaseClass
{
  SqlConnection dbConn;


  public MyDatabaseClass()
  {
    dbConn = getConnection();
  }


  private SqlConnection getConnection()
  {
```

```
    //create and return database connection
  }


  ~MyDatabaseClass()
  {
    if (dbConn != null)
      dbConn.Close();
  }
}
```

Notice that the destructor method is defined without parameters, return type, or access modifier. After completing execution, C# destructors automatically call their parents' destructors. Implementing a C# destructor is functionally equivalent to implementing a `Finalize()` method as follows:

```
using System.Data.SqlClient;

public class MyDatabaseClass
{
  SqlConnection dbConn;


  public MyDatabaseClass()
  {
    dbConn = getConnection();
  }


  private SqlConnection getConnection()
  {
    //create and return database connection
  }


  protected override void Finalize()
  {
    if (dbConn != null)
      dbConn.Close();

    base.Finalize(); //calls parent class's Finalize()
  }
}
```

The C# destructor (or `Finalize()` method) is useful for performing clean-up tasks (i.e., finalization) when an object is garbage collected. However, at times it may be preferable to execute clean-up code explicitly rather than having to wait for the garbage collector. Though the destructor cannot be called directly, it is possible to finalize an object on demand using the `IDisposable` interface. This interface defines a single method named `Dispose()` that accepts no parameters and has a `void` return type. For classes that hold critical resources, it is a good practice to implement both the `IDisposable` interface as well as a destructor. In this way, the object can be finalized either immediately by the client or later by the garbage collector. Also, since the garbage collector does not need to finalize the object if the `Dispose()` method is called, the `System.GC.SuppressFinalize()` should be called within the `Dispose()` method to prevent the garbage collector from attempting to finalize the object a second time. The following code demonstrates the `Dispose()` method:

```
using System;
using System.Data.SqlClient;

public class MyDatabaseClass : IDisposable
{
  SqlConnection dbConn;


  public MyDatabaseClass()
  {
    dbConn = getConnection();
  }


  private SqlConnection getConnection()
  {
    //create and return database connection
  }


  ~MyDatabaseClass()
  {
    if (dbConn != null)
      dbConn.Close();
  }


  public void Dispose()
  {
    if (dbConn != null)
      dbConn.Close();

    GC.SuppressFinalize(this);
  }
}
```

For some objects, terms other than "dispose" may make more sense for finalization. For example, many of the file and network classes provided by .NET use the `Close()` method for finalization. Though this may appear to be a departure from the finalization approach employed by other classes, the `Close()` method actually just calls the `Dispose()` method defined by the `IDisposable` interface. In essence, the `Close()` method is simply provided as an alternative to `Dispose()` for classes where the term "close" is semantically more appropriate.

Explicit Type Conversions (Casting)

Casting is the process of converting one type to another. Like C++, Java and C# perform casts by prefixing a variable with the desired type within parentheses like this:

```
int i = 65;
char c = (char) i;  //casts an int to a char
```

In addition to primitives, objects can be converted between class and interface types using explicit type casts. An object can be cast to its class type or any of its class's parent classes. Additionally, objects can be cast to any interface types implemented by the class from which they were derived. For instance, consider the following Java class:

```
public class Car extends java.lang.Object implements Drivable
```

```
{
  public int numberOfDoors;
  public String color;
}
```

> **NOTE**
> If no base class is specified, all Java classes implicitly extend from the `java.lang.Object` class. The `java.lang.Object` base class was only specified in this example for the purpose of clarity.

Since it extends `Object` and implements `Drivable`, the `Car` class can be cast to either of these types. However, an explicit cast is not necessary in either of these cases. Since every instance of `Car` is an instance of `Object` and every instance of `Car` is of type `Drivable`, these casts can be made either explicitly or implicitly as shown here:

```
Car c = new Car();

//explicit casts
Drivable d = (Drivable) c;
Object o = (Object) c;

//implicit casts
Drivable d2 = c;
Object o2 = c;
```

On the other hand, since every `Object` instance or `Drivable` type is not necessarily an instance of `Car`, an explicit cast is necessary when converting from these more general types to the more specific `Car` type. For example, consider the following code:

```
Car c = new Car();

//explicit casts optional
Drivable d = c;
Object o = c;

//explicit casts required
Car c2 = (Car) d;
Car c3 = (Car) o;
```

Static Initializers/Constructors

In contrast to normal constructors, Java and C# both define constructs that are used for initializing a class as opposed to an instance of the class. In Java and C#, these constructs are known as *static initializers* and *static constructors*, respectively. These static methods are called at the time the class is loaded prior to calling any normal constructor. Since static initializers and static constructors operate on the class itself (as opposed to a particular instance), they do not have access to any instance methods or variables. For this reason, they are primarily used to initialize static class variables.

In Java, static initializers are defined similarly to instance initializers except that they use the `static` keyword. Like instance initializers, static initializers consist of a block of code that resides inside of a class but outside of any methods. That is a static initializer consists of the `static` keyword followed by any amount of code contained within braces like this:

```
static
{
  System.out.println("Static initializer called!");
}
```

Static initializers are called prior to any constructors and immediately after static variables are initialized. In contrast to constructors, initializers are called only once when the class is first loaded rather than being called each time a new instance is created. The only time a static initializer may be called a second time is if the virtual machine unloaded and reloaded the class for some reason (perhaps to save memory). Typically, the virtual machine will maintain the class in memory as long as the application is running. Listing 4.9 demonstrates a Java static initializer.

Listing 4.9 ***Static initializer demonstration (Java)***

```
public class StaticInitializerTest
{
  private static String var = "Hello";


  static
  {
    System.out.println("Static initializer called!");
    System.out.println("Value of static variable var: " + var);

    var = "Hello World";
  }


  public StaticInitializerTest()
  {
    System.out.println("Constructor called!");
  }


  public void instanceMethod()
  {
    System.out.println("Instance method called!");
  }


  public static void classMethod()
  {
    System.out.println("Class method called!");
  }


  public static void main(String[] args)
  {
    System.out.println("main() method called!");
    System.out.println("Value of static variable var: " + var);

    classMethod(); //call class method

    //instantiate this class
    StaticInitializerTest sit = new StaticInitializerTest();

    sit.instanceMethod(); //call instance method
  }
}
```

Notice that the static initializer in Listing 4.9 is defined after the static variable that it initializes. This order is important. If the static initializer is defined prior to any static variable that it references, the compiler will generate an illegal forward reference exception. When executed, the output generated by Listing 4.9 looks like this:

```
Static initializer called!
Value of static variable var: Hello
main() method called!
Value of static variable var: Hello World
Class method called!
Constructor called!
Instance method called!
```

Note the order of events shown in this output. We can see that the static variable `var` is initialized to "Hello" prior to the static initializer being called. After the static variable is initialized, the static initializer is called and the value of `var` is changed to "Hello World". After the static initializer is called, the `main()` method is invoked. Notice that the class's constructor is not called until the class is actually instantiated.

Listing 4.10 is the C# equivalent of Listing 4.9. Other than the fact that C# uses a static constructor rather than a static initializer, these code samples are virtually identical and they produce identical output.

Listing 4.10 ***Static constructor demonstration (C#)***

```
using System;

public class StaticConstructorTest
{
  private static string var = "Hello";


  static StaticConstructorTest()
  {
    Console.WriteLine("Static initializer called!");
    Console.WriteLine("Value of static variable var: {0}", var);

    var = "Hello World";
  }


  public StaticConstructorTest()
  {
    Console.WriteLine("Constructor called!");
  }


  public void instanceMethod()
  {
    Console.WriteLine("Instance method called!");
  }


  public static void classMethod()
  {
    Console.WriteLine("Class method called!");
  }


  public static void Main()
```

```
  {
    Console.WriteLine("main() method called!");
    Console.WriteLine("Value of static variable var: {0}", var);

    classMethod(); //call class method

    //instantiate this class
    StaticConstructorTest sct = new StaticConstructorTest();

    sct.instanceMethod(); //call instance method
  }
}
```

Abstract Classes

An *abstract class* is a class that can be extended (i.e., subclassed) but not instantiated. Abstract classes play an important role in object-oriented programming. These classes allow the base class to implement general functionality common to all subclasses while declaring specific methods that must be implemented by each subclass. A class is declared abstract using the `abstract` keyword like this:

```
public abstract class Shape
```

Similarly, any methods that must be implemented by subclasses must also be declared abstract using the `abstract` keyword. Abstract methods consist only of a method declaration ending with a semi-colon (“`;`”) like this:

```
public abstract void ShowArea();
```

Abstract classes adhere to these rules:

- A class declared as `abstract` cannot be instantiated.
- An abstract class may or may not include abstract methods. However, if a single method is declared `abstract`, then its containing class must be declared `abstract` as well.
- If an abstract class is extended, the subclass can be instantiated only if it implements all methods that are marked `abstract` in the base class. If the subclass does not implement all abstract methods, then it must also be declared as `abstract`.

The rules governing abstract classes are the same in Java and C#. Listing 4.11 and 4.12 demonstrate an abstract class implemented in Java and C#, respectively.

Listing 4.11 ***Abstract Shape class (Java)***

```
public abstract class Shape
{
  private String color;


  public Shape(String color)
  {
    this.color = color;
  }


  public void ShowColor()
  {
```

```
    System.out.println("The color of this shape is " + color);
  }


  public void ShowArea()
  {
    double area = CalculateArea();
    System.out.println("The area of this shape is " + area);
  }


  public abstract double CalculateArea();
}
```

Listing 4.12 ***Abstract Shape class (C#)***

```
using System;

public abstract class Shape
{
  private string color;


  public Shape(String color)
  {
    this.color = color;
  }


  public void ShowColor()
  {
    Console.WriteLine("The color of this shape is {0}", color);
  }


  public void ShowArea()
  {
    double area = CalculateArea();
    Console.WriteLine("The area of this shape is {0}", area);
  }


  public abstract double CalculateArea();
}
```

The `Shape` class presented in Listing 4.11 and 4.12 is abstract. It implements the methods `ShowColor()` and `ShowArea()` but, because area calculations vary according to the shape, it defers the implementation of the `CalculateArea()` method to its shape-specific subclasses. In this way, the base class is able to implement functionality that is common to all subclasses while requiring each subclass to implement methods that are specific to it.

Now that we have seen how to create an abstract class, let's create a couple of subclasses that inherit its concrete (i.e., non-abstract) methods and implement its abstract methods. Though these examples are presented in C#, they are virtually identical in Java. Listing 4.13 shows two concrete classes that extend from `Shape`.

Listing 4.13 ***Concrete classes Rectangle and Circle (C#)***

```
public class Rectangle : Shape
{
```

```
  private int height;
  private int width;


  public Rectangle(string color, int height, int width) :
    base(color)
  {
    this.height = height;
    this.width = width;
  }


  public override double CalculateArea()
  {
    return height * width;
  }
}


public class Circle : Shape
{
  private const double PI = 3.14159;
  private int radius;


  public Circle(string color, int radius) : base(color)
  {
    this.radius = radius;
  }


  public override double CalculateArea()
  {
    return PI * (double)(radius * radius);
  }
}
```

Now that we have two concrete classes, `Rectangle` and `Circle`, that implement the abstract methods defined by `Shape`, let's see how these classes are used. Listing 4.14 instantiates each class and calls their inherited and implemented methods.

Listing 4.14 ***Program that exercises the Rectangle and Circle classes (C#)***

```
public class AbstractClassExercise
{
  public static void Main()
  {
    Rectangle rec = new Rectangle("Red", 2, 4);
    rec.ShowColor();
    rec.ShowArea();

    Circle cir = new Circle("Blue", 3);
    cir.ShowColor();
    cir.ShowArea();
  }
}
```

The output generated by Listing 4.14 is as follows:

```
The color of this shape is Red
```

```
The area of this shape is 8
The color of this shape is Blue
The area of this shape is 28.27431
```

Inner/Nested Classes

An *inner* or *nested class* is pretty much as its name describes. It is a class that is defined within another class. In Java, there are four different types of inner or nested classes: top-level nested classes, member classes, local classes, and anonymous classes. C# supports only a single type of inner class that is equivalent to Java's top-level nested class. The following sections describe each of these different classes.

Top-Level Nested Classes

A *top-level nested class* is essentially a static member of the class in which it is defined. This type of class is often used to implement a helper class that is of interest only to its enclosing class. For example, consider a class that implements a database connection pool. The purpose of this class is to create and store database connections and distribute them to clients in an orderly fashion. Though this class distributes only standard database connections, internally it may manage each connection by placing it within a `PooledConnection` nested class that contains additional information of interest to the enclosing connection pool class. This information might include whether or not the connection is currently being used by a client and how long the connection has been idle. Since clients are only interested in standard database connections, the internal `PooledConnection` class is an implementation detail that should not be publicly exposed. To briefly illustrate using pseudo-code, a private inner class is implemented like this in Java:

```
import java.sql.Connection;
import java.util.ArrayList;

public class ConnectionPool
{
  private ArrayList pooledConnections = new ArrayList();


  public ConnectionPool()
  {
    /*Create database connections and populate array list with
      pooled connections.*/

    Connection c = createConnection();
    PooledConnection pc = new PooledConnection(c);
    pooledConnections.add(pc);
  }


  private Connection createConnection()
  {
    //create database connection here
  }


  public Connection getConnection()
  {
    /*Search through array list of pooled connections until a
      free connection is found (i.e., inUse is false). Set the
      inUse variable to true and the lastUsed variable to the
```

```
      current time. Return the database connection contained in
      the pooled connection object.*/
  }


  public void returnConnection(Connection dbConn)
  {
    /*Search through the array list of pooled connections until
      the pooled connection that contains the connection dbConn
      is found. Mark the pooled connection as free by setting
      its inUse variable to false.*/
  }


  private static class PooledConnection
  {
    Connection connection;
    boolean inUse = false;
    long lastUsed;

    public PooledConnection(Connection connection)
    {
      this.connection = connection;
    }
  }
}
```

Notice that top-level nested classes in Java are declared using the `static` modifier. For this reason, in Java, these classes are often referred to as *static classes*. Though syntactically similar to Java, C# does not use the `static` modifier for top-level nested classes. Regardless, nested classes in C# are implicitly static. And since top-level classes in Java and C# are static, they cannot access instance variables or instance methods defined within their enclosing classes. Rather, only static variables and methods are accessible to top-level nested classes. C# nested classes are implemented like this:

```
using System.Collections;
using System.Data.SqlClient;

public class ConnectionPool
{
  private ArrayList pooledConnections = new ArrayList();


  public ConnectionPool()
  {
    /*Create database connections and populate array list with
      pooled connections.*/

    SqlConnection c = createConnection();
    PooledConnection pc = new PooledConnection(c);
    pooledConnections.Add(pc);
  }


  private SqlConnection createConnection()
  {
    //create database connection here
  }
```

```
  public SqlConnection getConnection()
  {
    /*Search through array list of pooled connections until a
      free connection is found (i.e., inUse is false). Set the
      inUse variable to true and the lastUsed variable to the
      current time. Return the database connection contained in
      the pooled connection object.*/
  }


  public void returnConnection(SqlConnection dbConn)
  {
    /*Search through the array list of pooled connections until
      the pooled connection that contains the connection dbConn
      is found. Mark the pooled connection as free by setting
      its inUse variable to false.*/
  }


  private class PooledConnection
  {
    SqlConnection connection;
    bool inUse = false;
    long lastUsed;

    public PooledConnection(SqlConnection connection)
    {
      this.connection = connection;
    }
  }
}
```

Notice that the nested class `PooledConnection` is declared `private` so that it is only visible to its enclosing class. In this way, nested classes improve encapsulation by allowing implementation details to be hidden.

Though we have only discussed private nested classes so far, top-level nested classes can also be declared `public`. Public nested classes are often used for organizational purposes. For instance, consider two classes that are closely related and both of interest to other classes. Typically, these two classes would reside in the same package or namespace. However, creating a separate package or namespace is not always convenient or desired. Alternatively, using a public nested class, it is possible to place one of the classes (typically the subordinate or "helper" class) inside of the other without sacrificing accessibility. That is, since the nested class is declared `public`, it is as accessible as its enclosing class.

To illustrate the concept of public nested classes, imagine that the `ConnectionPool` class presented previously resided in a package or namespace called `database`. Now imagine that another class needed to use the `PooledConnection` class nested inside of `ConnectionPool`. In this case, we have two obvious options. First, we could extract the `PooledConnection` class from `ConnectionPool` and move both of these classes to the `database.pool` package/namespace (in order to retain their close relationship to each other). Or second, we could leave the `ConnectionPool` class in the `database` package/namespace and simply make the `PooledConnection` nested class public. By declaring it `public`, the nested class can be instantiated by outside classes by prefixing it with the name of its enclosing class like this:

```
ConnectionPool.PooledConnection pc =
```

```
  new ConnectionPool.PooledConnection(dbConn);
```

Since the public nested class is static, it is possible to instantiate it without instantiating its containing class.

Member Classes (Java)

Like a top-level nested class, a *member class* is defined as a member of another class. However, unlike top-level nested classes, member classes are not declared `static`. Since they are not static, member classes behave much like other class members (e.g., fields and methods). Similar to other class members, member classes have access to all fields and methods of their enclosing class, even private fields and methods. Since they are literally class members, member classes are considered to be inner classes rather than nested classes. The following abbreviated code demonstrates how the `PooledConnection` nested class presented earlier can be converted to a member class by simply removing the `static` declaration:

```
import java.sql.Connection;
import java.util.ArrayList;

public class ConnectionPool
{
  private ArrayList pooledConnections = new ArrayList();


  //...additional methods here...


  private class PooledConnection
  {
    Connection connection;
    boolean inUse = false;
    long lastUsed;

    public PooledConnection(Connection connection)
    {
      this.connection = connection;
    }
  }
}
```

Similar to other class members, a member class is associated with an instance of its enclosing class. Therefore, unlike top-level nested classes, the enclosing class must be instantiated before a member class can be accessed. Instantiating member classes requires a slightly unusual syntax. For example, the following code first instantiates the `ConnectionPool` class and then uses the `ConnectionPool` object to instantiate its `PooledConnection` member class:

```
ConnectionPool cp = new ConnectionPool();

ConnectionPool.PooledConnection pc =
  cp.new PooledConnection(null);
```

Take a close look at the previous line of code. The `new` operator is prefixed with the name of the `ConnectionPool` instance so that it reads `cp.new`. This is a special syntax recognized by the Java compiler in order to support inner classes. The `cp.new PooledConnection(null)` command indicates that the `PooledConnection` class being instantiated is a member class of the `cp` object.

In addition to the `new` operator, a special syntax is also required when explicitly referencing variables from an inner class. Typically, members of the enclosing class can be accessed by name without further qualification. However, what if the inner class and enclosing class have variables of the same name? In this case, the Java compiler requires that you explicitly qualify both variables in order to avoid all ambiguity. Inner class variables can be qualified with the `this` keyword while members of the enclosing class can be explicitly referenced using the *`className`*`.this` syntax as follows:

```
this.count = 1; //refers to member class variable
ConnectionPool.this.count = 100; //refers to the enclosing class
```

As you can see in the previous line of code, the `ConnectionPool.this` syntax refers to the object that contains the inner class.

There are a few important points to be aware of when working with inner classes. First, Like other class members, inner classes may be declared using the `private`, `protected`, `public`, or default (package level) access modifiers to alter their visibility. Second, inner classes cannot contain any static members. Static members can be used only in top-level classes. Third, though inner classes can be nested infinitely deep (i.e., inner classes can contain inner classes themselves), they cannot have the same name as any containing class or package. Even though it is permitted, deeply nesting inner classes is not recommended (to avoid excess complexity).

Local Classes (Java)

A *local class* is a class that is defined within a block of code. Similar to local variables, local classes are narrowly scoped and are visible only within the block of code in which they reside. Local classes are typically defined within methods but they may also be defined within instance initializers or static initializers. Local classes are usually employed when implementing event listeners (discussed in a later chapter) or other simple classes that are unlikely to be reused. Listing 4.15 demonstrates how a local class is defined and instantiated.

Listing 4.15 ***Program that defines and instantiates a local class (Java)***

```
interface Messageable  //interface implemented by local class
{
  public void printMessage(String message);
}


public class LocalClassExample
{
  //This method defines a local class and returns an instance.
  public static Messageable getMessageable()
  {
    class MyMessage implements Messageable
    {
      public void printMessage(String message)
      {
        System.out.println(message);
      }
    }

    return new MyMessage();
  }


  public static void main(String[] args)
  {
```

```
    Messageable m = getMessageable();

    m.printMessage("Hello World");
  }
}
```

Notice that the `getMessageable()` method defines and instantiates the `MyMessage` local class and returns the class as a `Messageable` type. `Messageable` is an interface that is implemented by the `MyMessage` local class. Interfaces are commonly used in conjunction with local classes.

When working with local classes, there are a couple of restrictions to keep in mind. First, unlike member classes, local classes cannot be declared using any access modifiers (e.g., `private`, `protected`, `public`). Second, like member classes, local classes cannot contain any static members. As previously discussed, static members are only valid in top-level classes.

Anonymous Classes (Java)

An *anonymous class* is the same as a local class except that it is not given a name. In addition, unlike local classes, an anonymous class can be defined and instantiated within a single expression. Anonymous classes are typically used to create simple adapter classes (discussed in a later chapter). Listing 4.16 is identical to Listing 4.15 except that it uses an anonymous class rather than a local class.

Listing 4.16 ***Program that defines and instantiates an anonymous class (Java)***

```
interface Messageable  //interface used by anonymous class
{
  public void printMessage(String message);
}


public class AnonymousClassExample
{
  //This method defines and instantiates an anonymous class.
  public static Messageable getMessageable()
  {
    return new Messageable() {
      public void printMessage(String message)
      {
        System.out.println(message);
      }
    };
  }


  public static void main(String[] args)
  {
    Messageable m = getMessageable();

    m.printMessage("Hello World");
  }
}
```

As you can see, the body of the anonymous class is defined immediately following the `new` statement in the `getMessageable()` method. Also notice that the anonymous class is instantiated as a `Messageable` interface type. Since they do not have a name that can be referenced by clients, anonymous classes must implement an interface that defines the type by which they can be instantiated. Since anonymous classes are just a special type of local class,

they share many of the same characteristics. For instance, anonymous classes cannot be declared `public`, `private`, or `protected` and they cannot contain static members.

Checking Object Types

Java and C# both provide operators that indicate whether an object is an instance of a particular class or interface. An object is said to be an instance of an interface if the class from which it is derived implements that interface. For example, if a class named `Car` implements the `Drivable` interface, objects created from this class are instances of both `Car` and `Drivable`. Though every object of type `Car` is also of type `Drivable` (since `Car` implements the `Drivable` interface), not every object of type `Drivable` is of type `Car`. This is because the `Drivable` interface may be implemented by any number of classes. For instance, a class named `Truck` may also implement the `Drivable` interface. Therefore, instances of both `Car` and `Truck` are of type `Drivable`.

Java uses the `instanceof` operator to determine whether an object is an instance of a particular class or interface. The `instanceof` operator returns a boolean value and is used like this with classes:

```
if (objectName instanceof className)
{
  //object is an instance of specified class
}
```

and like this with interfaces:

```
if (objectName instanceof interfaceName)
{
  //object is an instance of specified interface
}
```

To demonstrate the Java `instanceof` operator in action, consider the interface and classes defined in Listing 4.17. Note that the interface and classes presented in this listing must be saved in separate files that are named according to their respective interface or class name in order to be compiled.

Listing 4.17 ***Interface and class definitions for instanceof example (Java)***

```
public interface Drivable
{
  public void drive();
}


public class Car implements Drivable
{
  public int numberOfDoors;


  public Car(int numberOfDoors)
  {
    this.numberOfDoors = numberOfDoors;
  }


  public void drive()
  {
```

```
    System.out.println("Driving the car...");
  }
}


public class Truck implements Drivable
{
  public int bedLength;


  public Truck(int bedLength)
  {
    this.bedLength = bedLength;
  }


  public void drive()
  {
    System.out.println("Driving the truck...");
  }
}
```

Listing 4.18 demonstrates how the `instanceof` operator is used to distinguish between the `Drivable` types `Car` and `Truck`.

Listing 4.18 ***Demonstrates the instanceof operator (Java)***

```
public class InstanceofExample
{
  public static void main(String[] args)
  {
    Drivable[] vehicles = new Drivable[5];

    vehicles[0] = new Truck(5);
    vehicles[1] = new Car(2);
    vehicles[2] = new Truck(6);
    vehicles[3] = new Car(4);
    vehicles[4] = new Truck(8);

    for (int x = 0; x < vehicles.length; x++)
    {
      Drivable vehicle = vehicles[x];

      vehicle.drive();

      if (vehicle instanceof Car)
      {
        Car car = (Car) vehicle; //cast to Car
        System.out.println("# of doors: " + car.numberOfDoors);
      }
      else if (vehicle instanceof Truck)
      {
        Truck truck = (Truck) vehicle; //cast to Truck
        System.out.println("Bed length: " + truck.bedLength);
      }
    }
  }
}
```

The output generated by Listing 4.18 looks like this:

```
Driving the truck...
Bed length: 5
Driving the car...
# of doors: 2
Driving the truck...
Bed length: 6
Driving the car...
# of doors: 4
Driving the truck...
Bed length: 8
```

Similar to Java's `instanceof` operator, C# provides the `is` operator to determine whether an object is an instance of a particular class or interface. Listing 4.19 is the C# version of Listing 4.18 using a similar interface and classes as those defined in Listing 4.17.

Listing 4.19 ***Demonstrates the is operator (C#)***

```
public class IsExample
{
  public static void Main()
  {
    Drivable[] vehicles = new Drivable[5];

    vehicles[0] = new Truck(5);
    vehicles[1] = new Car(2);
    vehicles[2] = new Truck(6);
    vehicles[3] = new Car(4);
    vehicles[4] = new Truck(8);

    for (int x = 0; x < vehicles.Length; x++)
    {
      Drivable vehicle = vehicles[x];

      vehicle.Drive();

      if (vehicle is Car)
      {
        Car car = (Car) vehicle; //cast to Car
        Console.WriteLine("# of doors: " + car.NumberOfDoors);
      }
      else if (vehicle is Truck)
      {
        Truck truck = (Truck) vehicle; //cast to Truck
        Console.WriteLine("Bed length: " + truck.BedLength);
      }
    }
  }
}
```

Listing 4.19 produces identical output to that created by Listing 4.18.

In addition to the `is` operator, C# provides the `as` operator. The `as` operator provides a shortcut for performing a safe type cast. A safe type cast is a type conversion that is guaranteed not to thrown an exception when the cast is performed. Typically, an exception is thrown if an object cannot be converted to the requested type. For instance, in the following code, if the object `o` is not of type `Car`, an exception will be thrown:

```
Car c = (Car) o;
```

A safe type cast can be performed by first verifying the object type using the `is` operator like this:

```
Car c = null;
if (o is Car)
  c = (Car) o;
```

This cast is safe because the `is` operator indicates that the object `o` is of type `Car` before attempting to cast it to that type. If `o` is not a `Car` type, then `c` is left `null`. The `as` operator performs this exact same function but in a single line of code as follows:

```
Car c = (o as Car);
```

In this example, the `as` operator attempts to cast the object `o` to the type `Car`. If the cast is illegal, rather than throwing an exception, the operation returns a `null`. So, as you can see, the three lines required to perform a safe type cast using the `is` operator can be condensed to a single line using the `as` operator.

Java has no equivalent to the `as` operator. Rather, use the `instanceof` operator prior to performing a cast to verify that the cast will succeed.

Methods

Analogous to procedures, routines, or functions in other programming methodologies, a *method* in object-oriented programming is an action that is defined by a class and exposed by all instances of that class. For example, a class named `Shape` may define the `draw()` and `calculateArea()` methods. These methods perform specific actions such as drawing the shape on the screen or calculating the shape's area.

Methods are implemented similarly in Java and C#. However, C# introduces a number of new modifiers that provide more fine-grained control over how and when a method is invoked or overridden. As we delve deeper into the definition and behavior of methods, we will discover some very significant differences between the two languages.

As we have previously demonstrated, methods are functions or procedures that are defined within a class. Methods are declared having a particular return type (or `void` if the method does not return a value) and may accept any number and type of parameters. For example, the following code demonstrates a basic class that contains two simple methods (`add` and `subtract`):

```
public class Math
{
  public int add(int x, int y)
  {
    return x + y;
  }


  public int subtract(int x, int y)
  {
    return x - y;
  }
}
```

With the exception of varying method naming conventions, this class is identical in Java and C#. Typically, the names of public methods (and any other public members) of a C# class begin with a capital letter. From this example, we can see that Java and C# methods are declared with an access modifier and return type followed by the name of the method and a comma-delimited parameter list indicating the type and local name for each parameter. Each language employs the `return` keyword to exit a method and return a specified value. For methods that do not return a value (i.e., methods with a `void` return type), the `return` keyword simply exits the method.

Method Modifiers

In Java and C#, a number of different modifiers can be applied to methods. Table 4.2 presents these modifiers.

Table 4.2 ***Java and C# Method Modifiers***

Java Modifier	C# Modifier	Description
`private`	`private`	Accessible from within the method's class.
`private`	*none*	Accessible from within the method's class (same as `private`).
`--`	`protected`	Accessible from within the method's class or any derived class.
none	`internal`	Accessible by any class within the same package or assembly.
`protected`	`protected internal`	Accessible from within the method's class or any derived class or from any class within the same package or assembly.
`public`	`public`	Accessible from any class.
`abstract`	`abstract`	Method must be implemented by a subclass. If one or more methods use the `abstract` modifier, the class must also be declared `abstract`.
none	`virtual`	Indicates that a method may be overridden by a subclass. In Java, all methods are virtual by default.
`--`	`override`	Indicates that a method is overriding a base class method that has the same signature (i.e., name and parameter list).
`final`	`sealed`	Indicates that the method cannot be overridden by subclasses.
`--`	`sealed override`	Same as `override` except that the method cannot be overridden again.
`--`	`new`	Indicates that a method having the same signature as a method in the base class is not intended to override the base class's method.
`static`	`static`	Indicates a static class-level method (as opposed to an instance method).
`synchronized`	`--`	Indicates that the method can only be accessed by one thread at a time.

The "`--`" shown in Table 4.2 indicates that there is no equivalent to the specified characteristics described in the Description column. The "*none*" modifiers indicate that the described characteristics are achieved by default without explicitly specifying a modifier. The virtual, override, and new modifiers will be discussed in greater detail in the *Overriding Methods* section later in this chapter.

Method Parameters

In Java and C#, class instances are passed by reference while value types (e.g., primitives in Java, simple types and Structs in C#) are passed by value. Passing by reference means that the actual object is not passed to a method, just a reference to the object. Passing by value indicates that the method receives its own copy of the variable that is being passed as a parameter. To illustrate, consider the C# class presented in Listing 4.20.

Listing 4.20 ***Characteristics of passing parameters by reference and value (C#)***

```
public class ParameterTest
{
  private static void modifyInt(int i)
  {
    i = 4;
  }


  private static void modifyObject(Car c)
  {
    c.NumberOfDoors = 4;
  }


  public static void Main()
  {
    int i = 2;
    Car c = new Car();
    c.NumberOfDoors = 2;

    modifyInt(i);
    modifyObject(c);

    Console.WriteLine("Int value: {0}", i);
    Console.WriteLine("Car.NumberOfDoors: {0}", c.NumberOfDoors);
  }
}
```

The output generated by Listing 4.20 looks like this:

```
Int value: 2
Car.NumberOfDoors: 4
```

As you can see, the integer value was not changed by the call to `modifyInt()` while the call to `modifyObject()` did change the value of the `Car` object's `NumberOfDoors` field. This is due to the fact that `int` parameters are passed by value while class instances are passed by reference. When an `int` is passed to a method, the method receives its own copy of the variable. Changing the method's copy does not alter the value of the original variable. On the other hand, when a class instance is passed as a parameter, it is passed by reference such that the method's copy is pointing to the same object as the original reference. Since they both point to the same object, changes made to the object within the method are visible to the original reference.

> **NOTE**
> Though we refer to class instances as being passed by reference, technically, they are passed by value. However, since Java and C# always manipulate objects by reference, when an object is passed as a parameter, the method receives its own copy of the object reference. In this way, we can say that the object reference is passed by value. In other words, the method receives its own copy of a reference pointing to the same object as the original reference. The method that receives a copy of the reference cannot alter the original reference, but it can change the object to which the original reference points. Because Java and C# always manipulate objects by reference, it is simpler to think of class instances as being passed by reference.

`ref` and `out` Parameters (C#)

C# provides two keywords that can be applied to the parameter list that alter the manner in which variables are passed. The `ref` keyword indicates that the specified parameter should be passed by reference rather than value. When working with simple types like `int` or `long`, the `ref` keyword causes a reference to the simple type (rather than a copy of it) to be passed to the method. In this way, the method can alter the value of the original variable. With class instances, the `ref` keyword causes a reference to the object reference to be passed to the method. In this way, a method can change the object to which the original reference points. Listing 4.21 demonstrates use of the `ref` keyword.

Listing 4.21 *Demonstration of the ref keyword (C#)*

```
public class ParameterTest
{
  private static void modifyInt(ref int i)
  {
    i = 4;
  }


  private static void modifyObject(ref Car c)
  {
    Car c2 = new Car();
    c2.NumberOfDoors = 4;

    c = c2;
  }


  public static void Main()
  {
    int i = 2;
    Car c = new Car();
    c.NumberOfDoors = 2;

    modifyInt(ref i);
    modifyObject(ref c);

    Console.WriteLine("Int value: {0}", i);
    Console.WriteLine("Car.NumberOfDoors: {0}", c.NumberOfDoors);
  }
}
```

The output created by Listing 4.21 looks like this:

```
Int value: 4
Car.NumberOfDoors: 4
```

Notice that the call to `modifyInt()` actually altered the value of the `int` variable `i`. In addition, the original reference `c` was modified by `modifyObject()` to point to the same object as `c2`.

There are a couple of caveats to remember when working with the ref keyword. First, `ref` must be declared in the method definition as well as the method call. Though this may seem redundant, this repetition is intended to reduce errors by ensuring that clients are well aware that the parameter is being passed by reference. Second, parameters using the `ref` keyword must be initialized prior to being passed. For instance, the following code would generate a compilation error since the variable `i` is not assigned an initial value:

```
int i;
changeInt(ref i);
```

The `out` keyword behaves identically to the `ref` except that it is not restricted to initialized variables. Rather, any declared variable (initialized or not) can be passed as an `out` parameter and initialized or assigned a new value within the target method. Listing 4.22 demonstrates the `out` keyword.

Listing 4.22 ***Demonstration of the out keyword (C#)***

```
public class ParameterTest
{
  private static void modifyInt(out int i)
  {
    i = 4;
  }


  private static void modifyCar(out Car c)
  {
    c = new Car();
    c.NumberOfDoors = 4;
  }


  public static void Main()
  {
    int i;
    Car c;

    modifyInt(out i);
    modifyCar(out c);

    Console.WriteLine("Int value: {0}", i);
    Console.WriteLine("Car.NumberOfDoors: {0}", c.NumberOfDoors);
  }
}
```

When uninitialized variables are passed as `out` parameters, they must be initialized before use. For example, the `modifyCar()` method in Listing 4.22 assigns a new instance of `Car` to the reference `c` before assigning a value to its `NumberOfDoors` field.

Variable Length Parameter Lists (C#)

C# supports variable length parameter lists using the `params` keyword. The term "variable length parameter list" refers to a method that accepts any number of parameters of a certain type. For instance, the `System.Console.WriteLine()` method uses a variable length parameter list to support placeholders (e.g., `{0}`, `{1}`, `{2}`, etc.) like this:

```
System.Console.WriteLine("Hello {0}", "Larry");

System.Console.WriteLine("Hello {0} and {1}", "Larry", "Curly");

System.Console.WriteLine("Hello {0}, {1}, and {2}", "Larry",
  "Curly", "Moe");
```

The first parameter defined by the `WriteLine()` method represents the text to be written to the console while the second parameter is defined as a variable length list of `object` parameters. In this example, each method call adds another `string` parameter. This pattern could continue

indefinitely since there is no limit to the number of variables supported by a variable length parameter list.

Variable length parameter lists are defined using the `params` keyword and an array of a particular type like this:

```
public static void WriteLine(string msg, params object[] args)
```

This code shows one of the signatures supported by the `System.Console.WriteLine()` method. It accepts a message to be written to the console followed by any number of `object` parameters. When a method containing a variable length parameter list is invoked, the variable length parameters are packaged into the array defined using the `params` keyword. For instance, the three string variables "`Larry`", "`Curly`", and "`Moe`" presented earlier would each be stored in the `WriteLine()` method's `args` array in positions `args[0]`, `args[1]`, and `args[2]`, respectively.

When working with variable length parameter lists, there are two important restrictions to keep in mind. First, each method can have only one variable length parameter. Second, the variable length parameter must be the last (i.e., rightmost) parameter in the parameter list. These restrictions are enforced for a good reason. Without them, there would be no way to positively determine where one parameter ended and the next began.

Listing 4.23 further illustrates how to use the `params` keyword to support variable length parameter lists. This example demonstrates two methods that utilize variable length `string` and `int` parameter lists.

Listing 4.23 ***Demonstration of the params keyword (C#)***

```
public class VariableParametersExample
{
  public static void Greet(string msg, params string[] names)
  {
    System.Console.Write(msg);
    System.Console.Write(" ");

    foreach (string name in names)
    {
      System.Console.Write(name);
      System.Console.Write(",");
    }

    System.Console.WriteLine();
  }


  public static int AddNumbers(params int[] numbers)
  {
    int sum = 0;

    foreach (int number in numbers)
      sum += number;

    return sum;
  }


  public static void Main()
  {
    Greet("Hello", "Larry", "Curly", "Moe");
```

```
    int total = AddNumbers(1, 2, 3, 4, 5);

    System.Console.WriteLine("Total: {0}", total);
  }
}
```

Overriding Methods

Method overriding is the process of replacing an inherited method with a new implementation. Subclasses often override generic methods inherited from their base class with more specific implementations. Java and C# handle method overriding very differently. Java defines default rules that implicitly establish how methods are overridden. In contrast, C# requires (or allows) the programmer to explicitly indicate how and when a method may be overridden. This section explores the differences between these two approaches.

Method overriding in Java is extremely simple and straight-forward. Simply put, a subclass can override any public or protected method in the base class by defining a new method of the same name and signature. In Java, when a subclass overrides a method, the overriding method is called for all instances of that subclass even if the subclass is cast to another type. Listing 4.24 demonstrates that even when a subclass is cast to its base class, the overriding methods are still called.

Listing 4.24 ***Method overriding (Java)***

```
class BaseClass
{
  public void someMethod()
  {
    System.out.println("BaseClass.someMethod() called.");
  }
}


class DerivedClass extends BaseClass
{
  public void someMethod()
  {
    System.out.println("DerivedClass.someMethod() called.");
  }
}


public class OverrideMethodExample
{
  public static void main(String[] args)
  {
    BaseClass bc = new BaseClass();
    DerivedClass dc = new DerivedClass();

    bc.someMethod();
    dc.someMethod();

    //explicit cast not required but specified for clarity
    bc = (BaseClass)dc;

    bc.someMethod();
  }
}
```

Listing 4.24 produces the following output:

```
BaseClass.someMethod() called.
DerivedClass.someMethod() called.
DerivedClass.someMethod() called.
```

Note that the derived class's overriding method was called even when the derived class was cast to its base class type. As you can see, Java methods are called according to the underlying class type rather than the type to which a class is currently cast.

Unlike Java, in C#, not all public and protected methods can be overridden. Rather, subclasses can override only those public and protected methods that are declared using the `virtual` keyword. Though subclasses can still implement a method that matches the name and signature of a non-virtual method in the base class, this new method is said to hide the base method rather than override it. The difference between a hidden method and an overridden one is the way in which the methods behave when their class is cast to a different type. To illustrate, Listing 4.25 is the C# version of the Java program presented in Listing 4.24. Since the `BaseClass` version of the `SomeMethod()` method is not declared using the `virtual` keyword, `SomeMethod()` in `DerivedClass` hides the `BaseClass` version. Typically, if one method purposely hides another, it should be declared using the `new` keyword as demonstrated in Listing 4.25. Otherwise, each time the class is compiled, the compiler will issue a warning like this:

```
warning: The keyword new is required on 'DerivedClass.SomeMethod()'
because it hides inherited member 'BaseClass.SomeMethod()'
```

Listing 4.25 ***Method hiding (C#)***

```
using System;

class BaseClass
{
  public void SomeMethod()
  {
    Console.WriteLine("BaseClass.SomeMethod() called.");
  }
}


class DerivedClass : BaseClass
{
  public new void SomeMethod()
  {
    Console.WriteLine("DerivedClass.SomeMethod() called.");
  }
}


public class OverrideMethodExample
{
  public static void Main()
  {
    BaseClass bc = new BaseClass();
    DerivedClass dc = new DerivedClass();

    bc.SomeMethod();
    dc.SomeMethod();

    //explicit cast not required but included for clarity
```

```
    bc = (BaseClass)dc;

    bc.SomeMethod();
  }
}
```

The output from Listing 4.25 looks like this:

```
BaseClass.SomeMethod() called.
DerivedClass.SomeMethod() called.
BaseClass.SomeMethod() called.
```

Notice how the output produced by Listing 4.25 differs from the output created by Listing 4.24. Since the `DerivedClass` version of `SomeMethod()` only hides (rather than overrides) the `BaseClass` version, when a `DerivedClass` object is cast to a `BaseClass` type, the `BaseClass` version of `SomeMethod()` is called. Since Java automatically overrides any method in the base class that shares the same name and signature as a subclass method, this behavior is not possible in Java.

A method is said to be *virtual* if it can be overridden (rather than just hidden). In Java, all methods are virtual by default. On the other hand, in C#, methods that are meant to be overridden must be declared virtual using the `virtual` keyword like this:

```
public virtual void SomeMethod()
```

In addition to declaring the base method as virtual, the overriding method must explicitly indicate that it is meant to override the base class's virtual method using the `override` keyword like this:

```
public override void SomeMethod()
```

If the `override` keyword is not specified, the new method will simply hide the old method rather than override it. Additionally, if the `override` keyword is not used, the compiler will generate a warning like this:

```
warning CS0114: 'DerivedClass.SomeMethod()' hides inherited member
'BaseClass.SomeMethod()'. To make the current member override that
implementation, add the override keyword. Otherwise add the new keyword.
```

Listing 4.26 is the C# equivalent of the Java code presented in Listing 4.24.

Listing 4.26 ***Method overriding (C#)***

```
using System;

class BaseClass
{
  public virtual void SomeMethod()
  {
    Console.WriteLine("BaseClass.myMethod() called.");
  }
}


class DerivedClass : BaseClass
{
  public override void SomeMethod()
  {
    Console.WriteLine("DerivedClass.myMethod() called.");
```

```
  }
}


public class OverrideMethodExample
{
  public static void Main()
  {
    BaseClass bc = new BaseClass();
    DerivedClass dc = new DerivedClass();

    bc.SomeMethod();
    dc.SomeMethod();

    //explicit cast not required but included for clarity
    bc = (BaseClass)dc;

    bc.SomeMethod();
  }
}
```

The output produced by Listing 4.26 is identical to that generated by Listing 4.24.

Overloading Methods

Java and C# methods can be *overloaded* by defining a new method that has the same name but a different signature than another method in the same class. A different signature means that the method accepts different parameters. Overloaded methods cannot differ by return type only.

Interfaces

An *interface* is an object-oriented programming construct that defines a collection of methods exposed by a class. An interface only defines method signatures; it does an implementation for these methods. For instance, the following interface defines two methods (identical in Java and C#):

```
public interface IBasicMath
{
  int add(int x, int y);
  int subtract(int x, int y);
}
```

Notice that `IBasicMath` ("`I`" stands for interface) only defines its methods; it does not implement them. Rather, classes that subscribe to (i.e., implement) the `BasicMath` interface must implement the methods that it defines. For example, the following Java class implements the `IBasicMath` interface:

```
public class Math implements IBasicMath
{
  public int add(int x, int y)
  {
    return x + y;
  }
```

```
  public int subtract(int x, int y)
  {
    return x - y;
  }
}
```

This same class in C# looks like this:

```
public class Math : IBasicMath
{
  public int add(int x, int y)
  {
    return x + y;
  }


  public int subtract(int x, int y)
  {
    return x - y;
  }
}
```

Though Java uses the `extends` keyword to derive a new class from an existing one, it uses the `implements` keyword to indicate that a class implements a particular interface. In contrast, C# uses the colon ("`:`") operator for class derivation as well as interface implementation.

> **NOTE**
> By convention, names of .NET interfaces begin with a capital "`I`" such as `IEnumerator` or `IDisposable`. In contrast, Java interfaces are often (but not always) given names that end with "`able`" such as `Runnable`, `Serializable`, or `Cloneable`.

Implementing Multiple Interfaces

Java and C# classes may implement any number of interfaces. Multiple interfaces can be implemented by simply declaring them in a comma-delimited list following the `implements` keyword (in Java) or the colon operator (in C#). For example, consider the following interface:

```
public interface IAdvancedMath
{
  int multiply(int x, int y);
  float divide(int x, int y);
}
```

Now consider the following Java class that implements both the `IBasicMath` and `IAdvancedMath` interfaces:

```
public class NewMath implements IBasicMath, IAdvancedMath
{
  public int add(int x, int y)
  {
    return x + y;
  }
```

```
  public int subtract(int x, int y)
  {
    return x - y;
  }


  public int multiply(int x, int y)
  {
    return x * y;
  }


  public float divide(int x, int y)
  {
    return x / y;
  }
}
```

Note how the `NewMath` class implemented all of the methods defined by `IBasicMath` and `IAdvancedMath` since the class declaration indicates that it implements both of these interfaces. The `NewMath` class looks like this in C#:

```
public class NewMath : IBasicMath, IAdvancedMath
{
  public int add(int x, int y)
  {
    return x + y;
  }


  public int subtract(int x, int y)
  {
    return x - y;
  }


  public int multiply(int x, int y)
  {
    return x * y;
  }


  public float divide(int x, int y)
  {
    return x / y;
  }
}
```

Since C# uses the colon operator to designate class inheritance as well as interface implementation, you might be wondering how to tell whether each type following the colon represents a class or an interface. The answer is that C# mandates that the parent class must come first within the comma-delimited list following the colon. That is, in the class declaration, interfaces cannot precede the parent class. In addition, to further distinguish between classes and interfaces, .NET convention specifies that all interfaces should begin with a capital "`I`". For example, the following C# class declaration indicates that the `ScientificMath` class extends the `BaseMath` class and implements the `IBasicMath` and `IAdvancedMath` interfaces:

```
public class ScientificMath : BaseMath, IBasicMath, IAdvancedMath
```

Casting Objects to Interfaces

Interfaces do more than just force a class to implement certain methods. Much like a class, an interface represents a new type. In fact, it is possible to cast a class to any interface that it implements. For example, if a class called `Math` implements the `IBasicMath` interface, then objects created from this class are instances of both `Math` and `IBasicMath` and can be cast to either type. However, the object will only be able to access the methods defined by the type to which it was cast. For instance, if the object is cast to the `IBasicMath` type, then only the methods defined by that interface will be accessible. On the other hand, if cast to type `Math`, all of the `Math` class's methods (including those defined by `IBasicMath`) will be accessible. To illustrate, consider the following class:

```
public class Math implements IBasicMath
{
  public int add(int x, int y)
  {
    return x + y;
  }


  public int subtract(int x, int y)
  {
    return x - y;
  }


  public int multiply(int x, int y)
  {
    return x * y;
  }


  public float divide(int x, int y)
  {
    return x / y;
  }
}
```

This version of `Math` implements four methods (`add`, `subtract`, `multiply`, `divide`). If this class is instantiated as type `Math`, all four methods are available to the instance as shown here:

```
Math m = new Math();

int a = m.add(3, 2);
int s = m.subtract(3, 2);
int m = m.multiply(3, 2);
float d = m.divide(3, 2);
```

In contrast, if cast to the interface type `IBasicMath`, only the two methods defined by `IBasicMath` are available to the instance as shown here:

```
IBasicMath bm = new MathClass();

int a = bm.add(3, 2);
int s = bm.subtract(3, 2);
```

The `instanceof` and `is` operators supported by Java and C#, respectively, can be used with interfaces as well as classes. For instance, to determine if a Java class implements the `IBasicMath` interface, the `instanceof` operator can be used like this:

```
Math m = new Math();

if (m instanceof IBasicMath)
{
  //...do something...
}
```

Or the C# `is` operator can be used like this:

```
Math m = new Math();

if (m is IBasicMath)
{
  //...do something...
}
```

Interface Inheritance

Similar to class inheritance, an interface can be derived from another interface; thus inheriting all of the base interface's method definitions. As previously discussed, classes can only extend from a single base class. This limitation is known as single inheritance. Unlike classes, interfaces are not limited to single inheritance and, therefore, can inherit from any number of interfaces. For example, the following Java interface inherits method definitions from two existing interfaces and adds a method definition of its own:

```
public interface IMath extends IBasicMath, IAdvancedMath
{
  int squared(int x);
}
```

In C#, the `IMath` interface looks like this:

```
public interface IMath : IBasicMath, IAdvancedMath
{
  int squared(int x);
}
```

Since `IMath` inherits the method definitions from the interfaces that it extends, any class that implements the `IMath` interface must implement the methods defined in `IMath` in addition to those it inherited from `IBasicMath` and `IAdvancedMath` like this:

```
public class Math implements IMath
{
  public int add(int x, int y)
  {
    return x + y;
  }


  public int subtract(int x, int y)
  {
```

```
    return x - y;
  }


  public int multiply(int x, int y)
  {
    return x * y;
  }


  public float divide(int x, int y)
  {
    return x / y;
  }


  public int squared(int x)
  {
    return x * x;
  }
}
```

Storing Constants in Interfaces (Java)

In contrast to C#, Java allows constants to be defined within an interface. Constants defined within an interface are implicitly available to any class that implements the interface. For instance, consider the following Java interface:

```
public interface IMathConstants
{
  static final double PI = 3.14159;
  static final double LOG_BASE = 2.71828;
  static final double EULERS = 0.57721;
  static final double GOLDEN_RATIO = 1.61803;
}
```

Any class that implements the `IMathConstants` interface has implicit access to all of its constants. For example, the following class inherits the `PI` constant from the `IMathConstants` interface:

```
public class CircleMath implements IMathConstants
{
  public double area(int r)
  {
    return PI * r * r;
  }


  public double circumfrence(int r)
  {
    return PI * 2 * r;
  }
}
```

By implementing the `IMathConstants` interface, the `CircleMath` class was able to access all of its constants without additional qualification.

Though Java allows interfaces to contain constants, this practice is not recommended. An interface is meant to represent a type and define the actions implemented by a particular class. Using an interface simply to store constants violates this principal of object-oriented programming. Rather, while adhering to good object-oriented design principles, shared constants can be defined as static members of a class as demonstrated here:

```
public class MathConstants
{
  public static final double PI = 3.14159;
  public static final double LOG_BASE = 2.71828;
  public static final double EULERS = 0.57721;
  public static final double GOLDEN_RATIO = 1.61803;
}
```

The constants defined in the `MathConstants` class can be accessed by qualifying them as follows:

```
public class CircleMath
{
  public double area(int r)
  {
    return MathConstants.PI * r * r;
  }


  public double circumfrence(int r)
  {
    return MathConstants.PI * 2 * r;
  }
}
```

Though it may require a bit more typing, accessing constants through a class helps preserve the object-oriented integrity of interfaces.

Structs (C#)

As discussed in Chapter 3, a `struct` is very similar to a class. The primary difference between these two types is that a `struct` is a value type that is stored on the stack while a class instance is a reference type that is stored on the heap. The stack is a small portion of memory that stores temporary local variables. The heap, on the other hand, is a large pool of memory that is used to store objects that persist longer than a single method call. Remember that when a value type variable is copied or passed to a method, an entirely new copy of the variable is created. In contrast, when a reference type is copied or passed to a method, a new type is not created but only a new reference to the same object.

Outside of where they are stored and how they are passed, structs are very similar to classes. Both types may implement any number of interfaces and can define many of the same members such as constructors, methods, fields, properties, events, and indexers. In a few ways, however, structs are more limited than classes. First, structs cannot extend classes or other structs. Second, since they are not stored on the heap like classes, a `struct` cannot implement a destructor. Listing 4.27 presents a basic `struct`. Last, a `struct` cannot define a constructor that has no parameters. Any constructors implemented by a `struct` should accept one or more values that are used to initialize its internal members.

Listing 4.27 ***Struct that defines a constructor, two public fields, and a method (C#)***

```
public struct Rectangle
{
  public int width;
  public int height;


  public Rectangle(int width, int height)
  {
    this.width = width;
    this.height = height;
  }


  public int GetArea()
  {
    return width * height;
  }
}
```

Summary

This chapter introduced the concept of object-oriented programming. Object-oriented principles simplify programming by facilitating the decomposition of complex software systems into discrete objects that store and act upon data. In addition, the following topics were discussed:

- *Encapsulation* is a method of keeping together data structures and the methods that operate on them.
- A powerful feature of object-oriented programming is the ability to derive a new class from an existing class. *Inheritance* means that a derived class (a.k.a., subclass) inherits all non-private fields and methods from its parent class (a.k.a., base class).
- *Polymorphism* refers to the condition where two classes that implement the same interface behave differently.
- Classes form the foundation of object-oriented programming. A *class* is a template from which similar objects are created.
- *Constructors* are special methods that are called whenever a class is instantiated.
- A *destructor* is a special method that is called when the memory occupied by an object is reclaimed.
- In contrast to normal constructors, Java and C# both define constructs that are used for initializing a class as opposed to an instance of the class. In Java and C#, these constructs are known as *static initializers* and *static constructors*, respectively.
- An *abstract class* is a class that can be extended (i.e., subclassed) but not instantiated.
- An *inner* or *nested class* is a class that is defined within another class.
- Analogous to procedures, routines, or functions in other programming methodologies, a *method* in object-oriented programming is an action that is defined by a class and exposed by all instances of that class.
- *Method overriding* is the process of replacing an inherited method with a new implementation.

- An *interface* is an object-oriented programming construct that defines a collection of methods exposed by a class. An interface only defines method signatures; it does an implementation for these methods.
- A `struct` is very similar to a class. The primary difference between these two types is that a `struct` is a value type that is stored on the stack while a class instance is a reference type that is stored on the heap.

Now that we have covered the basics of object-oriented programming, the next chapter will introduce some common programming structures that build upon these techniques.