CHEAT SHEET

This cheat sheet is designed as a way for you to quickly study the key points of this chapter.

Inheritance

- C# does not enable multiple inheritance.
- Use the `base` keyword to make a constructor invoke a parent class constructor as in the following code:

```
public class Employee : Person
{
    public Employee(string firstName, string lastName)
        : base(firstName, lastName)
    {
        ...
    }
}
```

- Use the `this` keyword to make a constructor invoke another constructor in the same class as in the following code:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName)
    {
        FirstName = firstName;
    }

    public Person(string firstName, string lastName)
        : this(firstName)
    {
        LastName = lastName;
    }
}
```

- A constructor can invoke at most one base class constructor or one same class constructor.
- If a parent class has constructors, a child class's constructors must invoke them directly or indirectly.

Interfaces

- By convention, interface names begin with `I` as in `IComparable`.
- A class can inherit from at most one parent class but can implement any number of interfaces.
- Implementing an interface is sometimes called interface inheritance.
- If a class implements an interface explicitly, the code cannot use an object reference to access the interface's members. Instead it must use an interface instance.

- If a class implements an interface implicitly, the code can use a class instance or an interface instance to access the interface's members.
- An `IComparable` class provides a `CompareTo` method that determines the order of objects.
- An `IComparer` class provides a `Compare` method that compares two objects and determines their ordering.
- An `IEquatable` class provides an `Equals` method that determines whether an object is equal to another object.
- An `ICloneable` class provides a `Clone` method that returns a copy of an object.
- An `IEnumerable` class provides a `GetEnumerator` method that returns an `IEnumerator` object that has `MoveNext` and `Reset` methods for moving through a list of objects.
- A method can use the `yield return` statement to add objects to an `IEnumerator` result.

Destructors

- Destructors can be defined in classes only, not structures.
- A class can have at most one destructor.
- Destructors cannot be inherited or overloaded.
- Destructors cannot be called directly.
- Destructors cannot have modifiers or parameters.
- The destructor is converted into an override version of the `Finalize` method. You cannot override `Finalize` or call it directly.

Resource Management

- If a class contains no managed resources and no unmanaged resources, it doesn't need to implement `IDisposable` or have a destructor.
- If the class has only managed resources, it should implement `IDisposable` but it doesn't need a destructor. (When the destructor executes, you can't be sure managed objects still exist, so you can't call their `Dispose` methods anyway.)
- If the class has only unmanaged resources, it needs to implement `IDisposable` and it needs a destructor in case the program doesn't call `Dispose`.
- The `Dispose` method must be safe to run more than once. You can achieve that by keeping track of whether it has been run before.
- The `Dispose` method should free managed and unmanaged resources.
- The destructor should free only unmanaged resources. (When the destructor executes, you can't be sure managed objects still exist, so you can't call their `Dispose` methods anyway.)
- After freeing resources, the `Dispose` method should call `GC.SuppressFinalize` to prevent the GC from running the object's destructor and to keep the object out of the finalization queue.
- The `using` statement lets a program automatically call an object's `Dispose` method, so you can't forget to do it. If you declare and initialize the object in the `using` statement, this also limits the object's scope to the `using` block.

REVIEW OF KEY TERMS

ancestor class A class's parent, the parent's parent, and so on.

base class A class from which another class is derived through inheritance. Also known as a *parent class* or *superclass*.

child class A class derived from a parent class.

Common Language Runtime (CLR) A virtual machine that manages execution of C# (and other .NET) programs.

deep clone A copy of an object where reference fields refer to new instances of objects, not to the same objects referred to by the original object's fields.

derive To create one class based on another through inheritance.

derived class A child class derived from a parent class through inheritance.

descendant class A class's child classes, their child classes, and so on.

destructor A method with no return type and a name that includes the class's name prefixed by ~. The destructor is converted into a `Finalize` method that the GC executes before permanently destroying the object.

finalization The process of the GC calling an object's `Finalize` method.

finalization queue A queue through which objects with finalizers must pass before being destroyed. This takes some time, so you should not give a class a finalizer (destructor) unless it needs one.

garbage collection The process of running the GC to reclaim memory that is no longer accessible to the program.

garbage collector (GC) A process that executes periodically to reclaim memory that is no longer accessible to the program.

inherit A derived class inherits the properties, methods, events, and other code of its base class.

interface inheritance Using an interface to require a class to provide certain features much as inheritance does (except the interface doesn't provide an implementation).

managed resources Resources that are under the control of the CLR.

multiple inheritance Allowing a child class to have more than one parent class. C# does not allow multiple inheritance.

nondeterministic finalization Because you can't tell when the GC will call an object's `Finalize` method, the process is called *nondeterministic finalization*.

parent class A base class. Also known as a *superclass*.

reachable During garbage collection, an object is reachable if the program has a path of references that let it access the object.

shallow clone A copy of an object where reference fields refer to the same objects as the original object's fields.

sibling classes Classes that have the same parent class.

subclass A derived class.

subclassing The process of deriving a subclass from a base class through inheritance.

superclass A base class. Also known as a *parent class*.

unmanaged resources Resources that are not under the control of the CLR.

unreachable During garbage collection, an object is unreachable if the program has no path of references that let it access the object.

EXAM TIPS AND TRICKS

The Review of Key Terms and the Cheat Sheet for this chapter can be printed off to help you study. You can find these files in the ZIP file for this chapter at `www.wrox.com/remtitle.cgi?isbn=1118612094` on the Download Code tab.