Inheritance and Polymorphism

CSIS 3540

Client Server Systems

Class 02

Topics

- Inheritance Basics
- Class Diagrams
- Inheritance Details
- Containment/Delegation
- Polymorphism
- Base/Derived Class Casting
- Object Class

Basic Inheritance

- Building classes from classes
- Start with
    - Parent, superclass, base class
    - The foundation of an "is-a" relationship
- Build a
    - Derived, child, subclass class
    - Child is-a Parent, but builds on or adds functionality to the parent.
- Child is more specific than parent.
- Syntax
    - class Child : Parent {}
- Child inherits all public members (fields, constructors, methods, properties) from parent.
    - Users of the child class get access to all public members of the parent.
- See **BasicInheritance**
- Note in this example that MiniVan is a child of Car with NO additional members, so when we use Car or MiniVan, the same members are used.

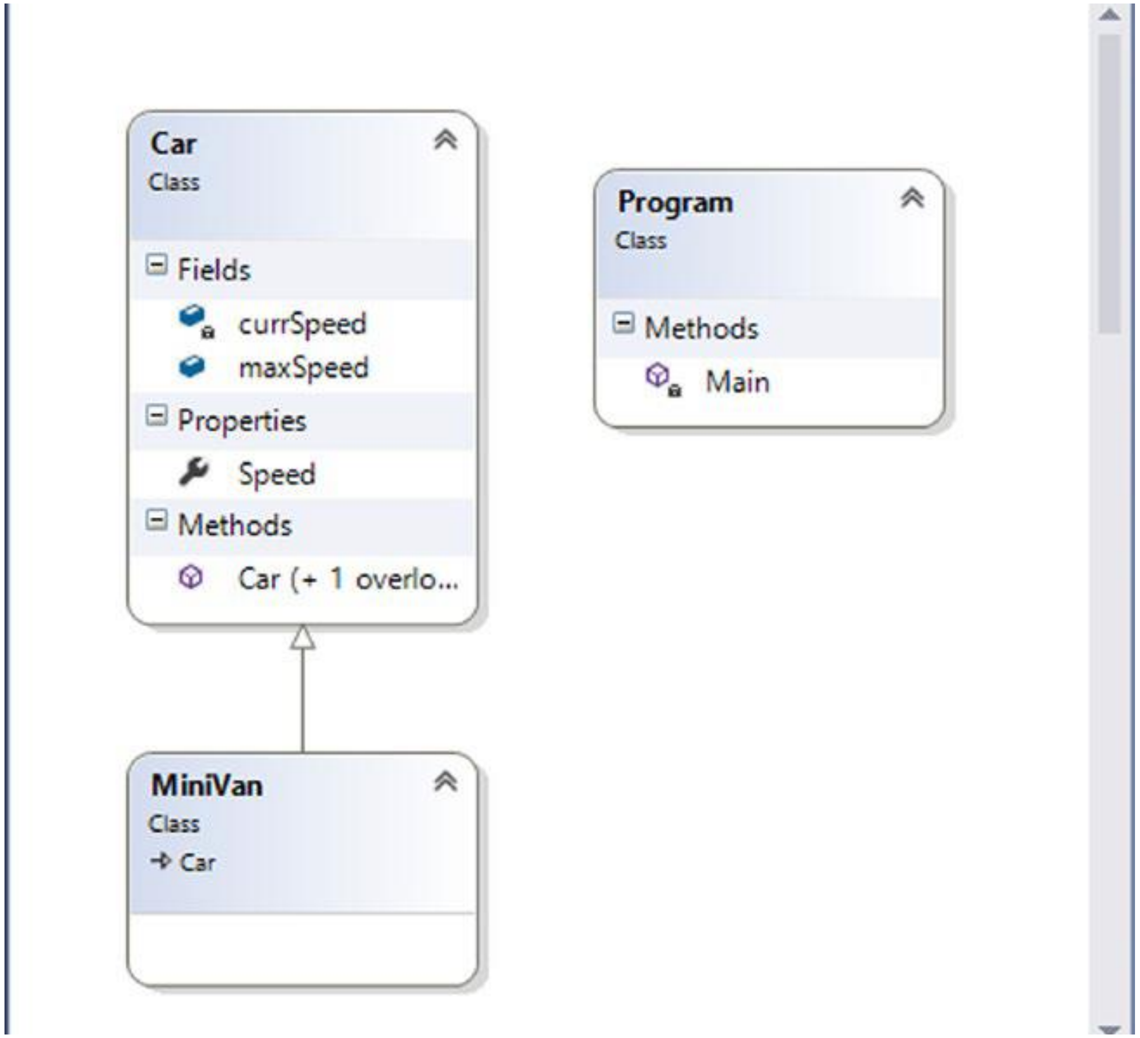
```
Console.WriteLine("***** Basic Inheritance *****\n");
// Make a Car object and set max speed.
Car myCar = new Car(80);

// Set the current speed, and print it.
myCar.Speed = 50;
Console.WriteLine("My car is going {0} MPH", myCar.Speed);

// Now make a MiniVan object.
MiniVan myVan = new MiniVan();
myVan.Speed = 10;
Console.WriteLine("My van is going {0} MPH",
  myVan.Speed);
```

Inheritance and Sealed

- Only ONE base class allowed
  - No Multiple Inheritance like C++
- sealed keyword
  - Prevents extension
  - Used for utility classes

Class Diagrams

- Under Project, Add New Item, Class Diagram.
- From **BasicInheritance**
- Note that methods and props can be filled in detail window below.



Derived and Base classes

- Derived classes can use any public member of the base class (methods, fields, properties, constructors)
- Derived classes can directly invoke base class members using the **base** keyword
    - Functions like the **this** keyword
    - Parent member must be virtual to override and use base keyword

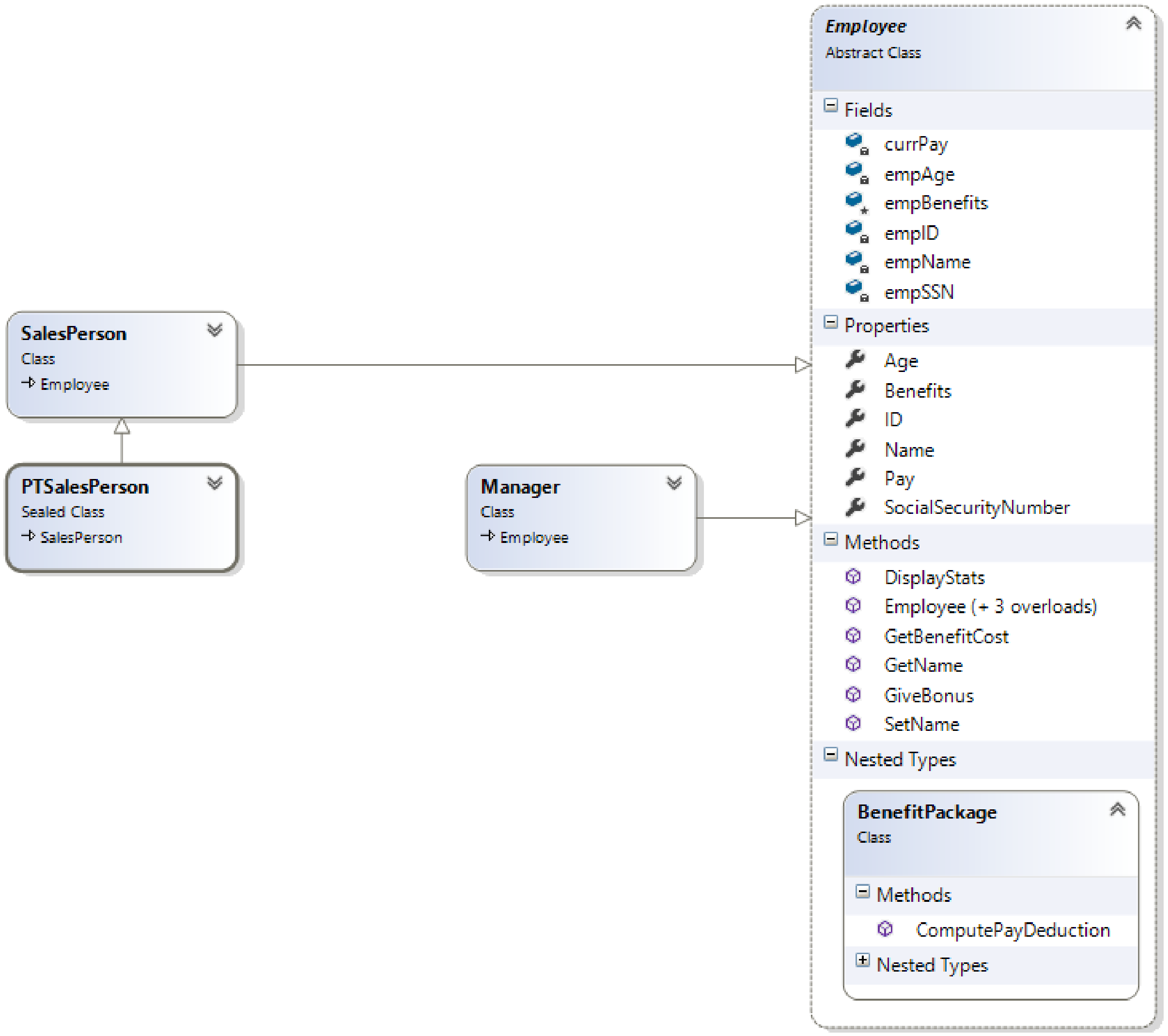
Example: BaseKeywordAndClasses

```
    class Program
    {
        static void Main(string[] args)
        {
            Child c = new Child();
            Parent p = new Parent();

            c.parentProp = 20;

            Console.WriteLine("Child {0} another {1} Parent {2} another {3}",
                c.parentProp, c.anotherParentProp, p.parentProp, p.anotherParentProp);
            Console.ReadLine();
        }
    }
    class Parent
    {
        public virtual int parentProp { get; set; } = 5;
        public int anotherParentProp { get; set; }

        public Parent()
        {
            anotherParentProp = 100;
        }

        public Parent(int n)
        {
            anotherParentProp = n;
        }
    }

    class Child : Parent
    {
        public override int parentProp { set { base.parentProp = value + 10; } }

        public Child() : base(1000)
        {

        }
    }
```

Protected and Sealed

- Protected keyword indicates that a member can be accessed only by derived classes
  - Allows access to internal data by children, which could be a problem
  - Violates encapsulation best practice
- Sealed keyword can be used to “cap off” a class
  - A child class may be the logical end to an inheritance hierarchy
    - Think of a family – these might be sealed!
      - Son: Person
      - Father: Person
      - Etc
    - Each “is-a” Person
    - Other relationships are “have-a”

Employees project



Containment/Delegation

- Containment
  - HAS-A relationship
  - Employee HAS-A BenefitPackage
- Delegation
  - Expose the functionality of the contained object to the outside
  - GetBenefitCost()

```
partial class Employee
  {
      // Contain a BenefitPackage object.
      protected BenefitPackage empBenefits = new BenefitPackage();

      // Expose certain benefit behaviors of object.
      public double GetBenefitCost()
      { return empBenefits.ComputePayDeduction(); }

      // Expose object through a custom property.
      public BenefitPackage Benefits
      {
          get { return empBenefits; }
          set { empBenefits = value; }
      }
  }
```

Nested types

- Complete control over the access level of the inner type
  - they may be declared privately
  - non-nested classes cannot be declared using the private keyword
- Because a nested type is a member of the containing class, it can access private members of the containing class.
- Useful only as a helper for the outer class and is not intended for use by the outside world.
- Note use of nested BenefitPackage class , but watch out for a duplicate external class of the same name. Fully qualified name can help determine which is used.

```
public class OuterClass
{
    // A public nested type can be used by anybody.
    public class PublicInnerClass {}
    // A private nested type can only be used by members
    // of the containing class.
    private class PrivateInnerClass {}
}
```

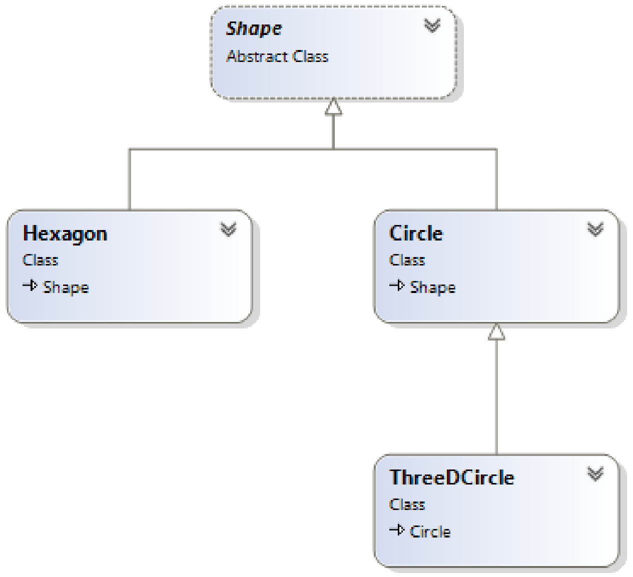
Polymorphism

- How can related types respond differently to the same request for information?
- Use of **virtual** members and **override**
- **virtual** members can be overridden for different responses to a request

```
 abstract partial class Employee
 {
            …
        public virtual void GiveBonus(float amount)
        { Pay += amount; }
            …
 }
 class SalesPerson : Employee
{
…
        public override sealed void GiveBonus(float amount)
        {
            int salesBonus = 0;
            if (SalesNumber >= 0 && SalesNumber <= 100)
                salesBonus = 10;
            else
            {
                if (SalesNumber >= 101 && SalesNumber <= 200)
                    salesBonus = 15;
                else
                    salesBonus = 20;
            }
            base.GiveBonus(amount * salesBonus);
        }

        public override void DisplayStats()
        {
            base.DisplayStats();
            Console.WriteLine("Number of sales: {0}", SalesNumber);
        }
}
```

Abstract classes

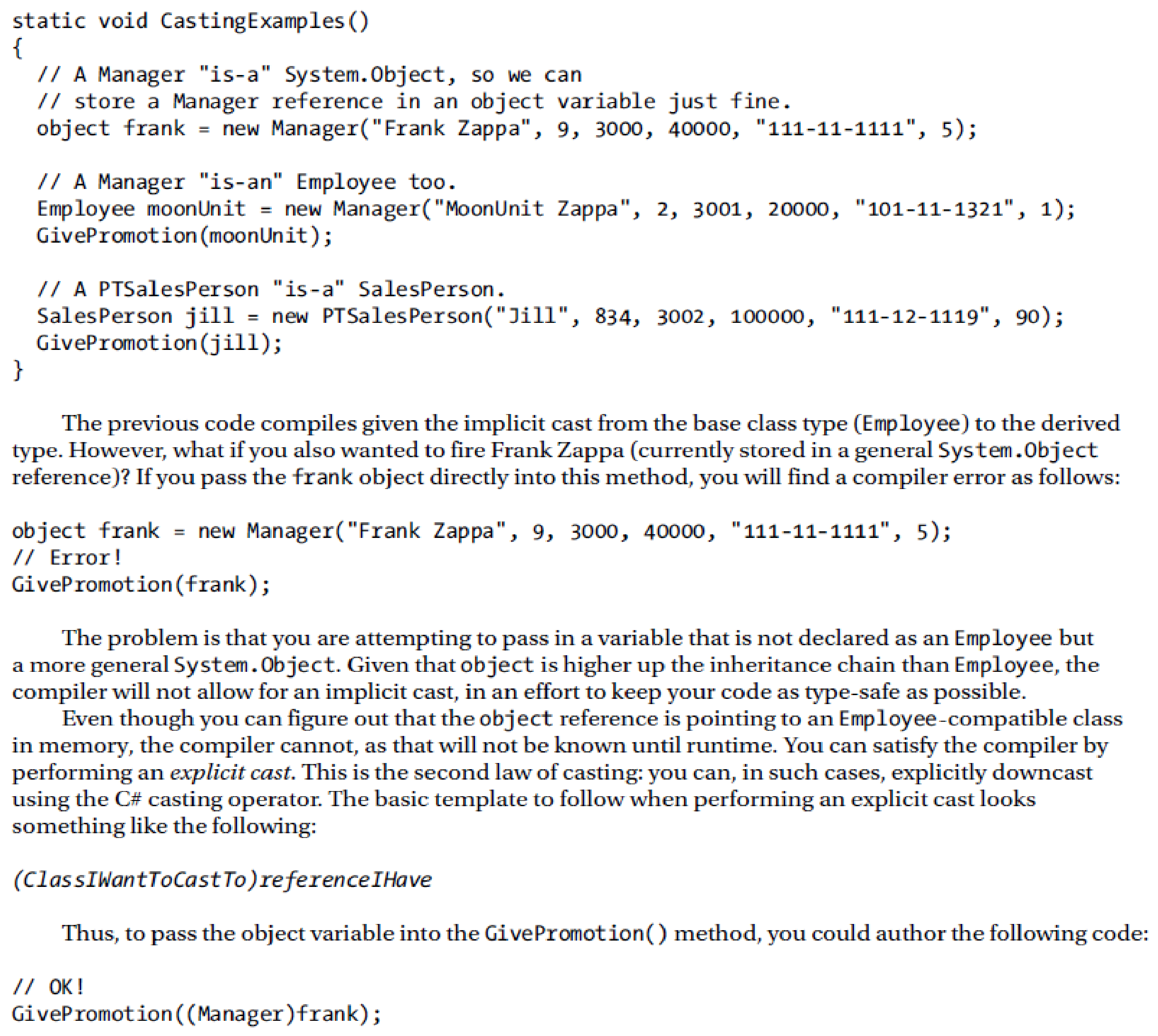
- Derived classes of an abstract class MUST provide instances of virtual members.
- When a class has been defined as an abstract base class (via the abstract keyword), it may define any number of *abstract members*.
- Abstract members can be used whenever you want to define a member **that does *not* supply a default implementation but *must* be accounted for by each derived class.**
- By doing so, you enforce a *polymorphic interface* on each descendant, leaving them to contend with the task of providing the details behind your abstract methods.
- An abstract base class's polymorphic interface simply refers to its set of virtual and abstract methods.
- If a class is abstract, it cannot be directly created! (see Employee)
- Virtual members are OPTIONALLY overridden. Abstract members MUST be overridden.
- Example
  - Shapes



```
abstract class Shape
   {
       public Shape(string name = "NoName")
       { PetName = name; }

       public string PetName { get; set; }

       // Force all child classes to
       // define how to be rendered.
       public abstract void Draw();
   }
```

Casting

- Any object can be downcasted to a base class without a problem.
- See Employees



```
static void CastingExamples()
{
  // A Manager "is-a" System.Object, so we can
  // store a Manager reference in an object variable just fine.
  object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);

  // A Manager "is-an" Employee too.
  Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000, "101-11-1321", 1);
  GivePromotion(moonUnit);

  // A PTSalesPerson "is-a" SalesPerson.
  SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000, "111-12-1119", 90);
  GivePromotion(jill);
}
```

The previous code compiles given the implicit cast from the base class type (Employee) to the derived type. However, what if you also wanted to fire Frank Zappa (currently stored in a general System.Object reference)? If you pass the frank object directly into this method, you will find a compiler error as follows:

```
object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
// Error!
GivePromotion(frank);
```

The problem is that you are attempting to pass in a variable that is not declared as an Employee but a more general System.Object. Given that object is higher up the inheritance chain than Employee, the compiler will not allow for an implicit cast, in an effort to keep your code as type-safe as possible.

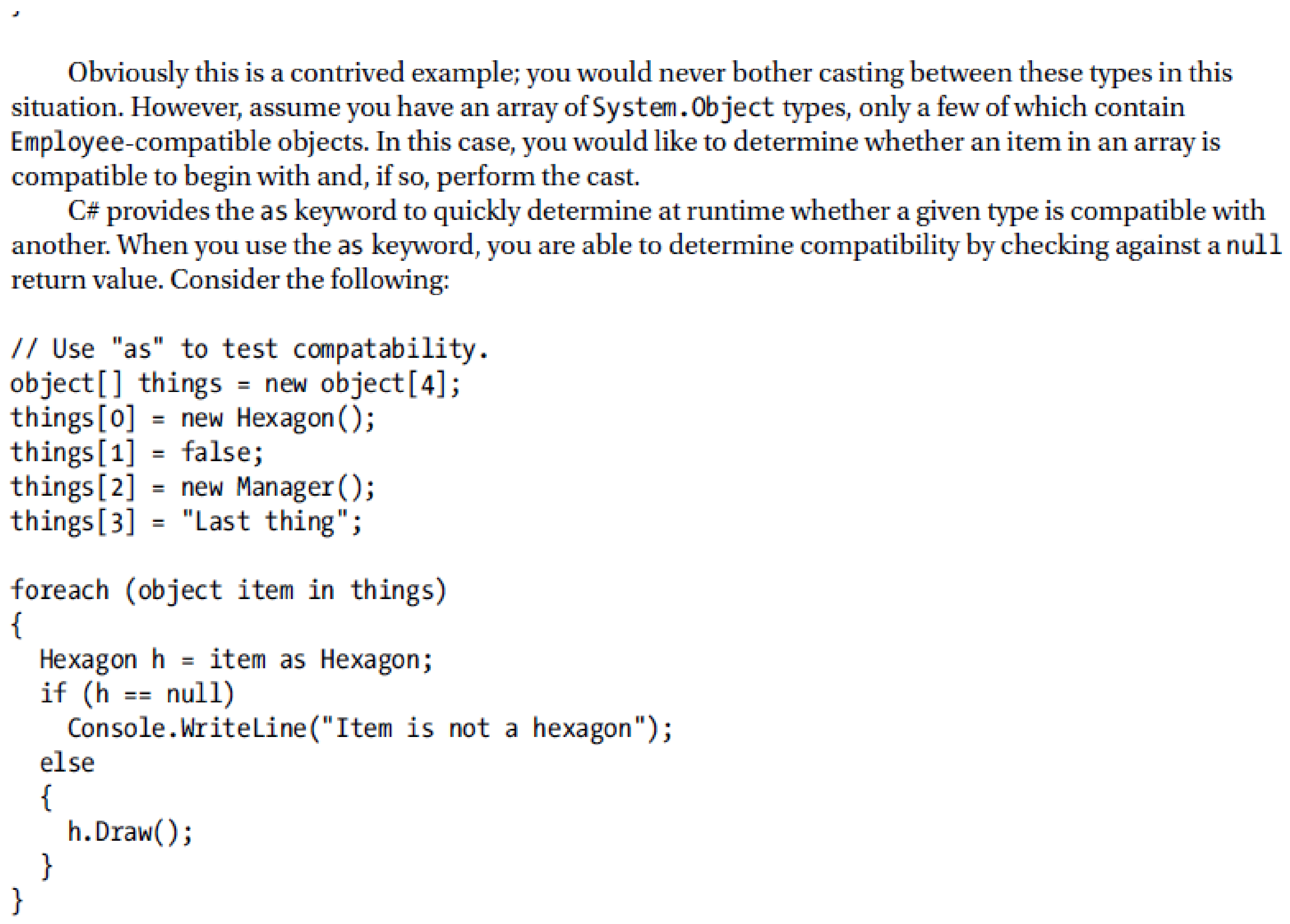
Even though you can figure out that the object reference is pointing to an Employee-compatible class in memory, the compiler cannot, as that will not be known until runtime. You can satisfy the compiler by performing an *explicit cast*. This is the second law of casting: you can, in such cases, explicitly downcast using the C# casting operator. The basic template to follow when performing an explicit cast looks something like the following:

```
(ClassIWantToCastTo)referenceIHave
```

Thus, to pass the object variable into the GivePromotion() method, you could author the following code:

```
// OK!
GivePromotion((Manager)frank);
```

Using **as** keyword (See Employees)



Obviously this is a contrived example; you would never bother casting between these types in this situation. However, assume you have an array of `System.Object` types, only a few of which contain `Employee`-compatible objects. In this case, you would like to determine whether an item in an array is compatible to begin with and, if so, perform the cast.

C# provides the as keyword to quickly determine at runtime whether a given type is compatible with another. When you use the as keyword, you are able to determine compatibility by checking against a `null` return value. Consider the following:

```
// Use "as" to test compatability.
object[] things = new object[4];
things[0] = new Hexagon();
things[1] = false;
things[2] = new Manager();
things[3] = "Last thing";

foreach (object item in things)
{
  Hexagon h = item as Hexagon;
  if (h == null)
    Console.WriteLine("Item is not a hexagon");
  else
  {
    h.Draw();
  }
}
```

Everything derives from System.Object (object)

- Look at code in **Shapes** testing for equality of object references.
- Copy-on-write changes the reference.

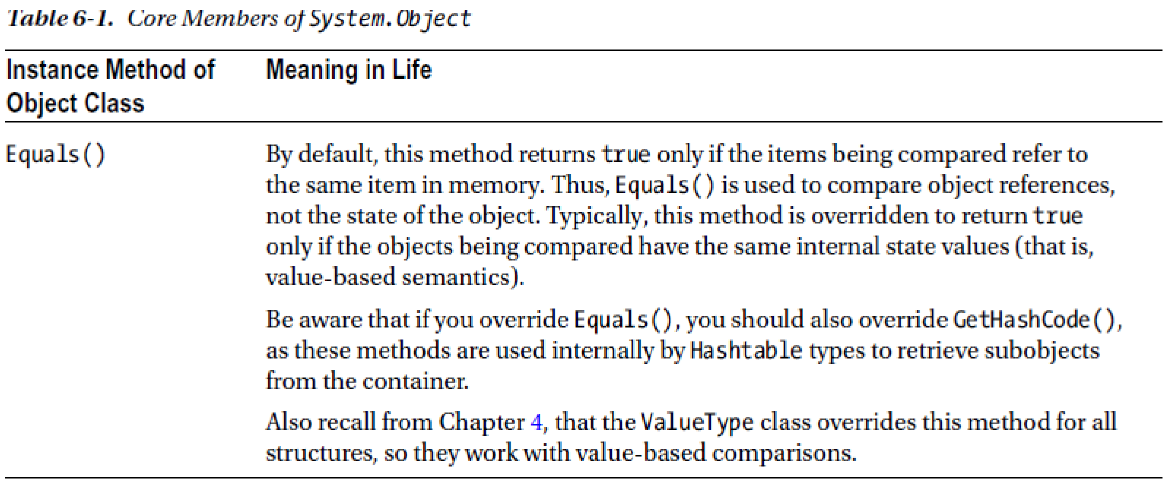
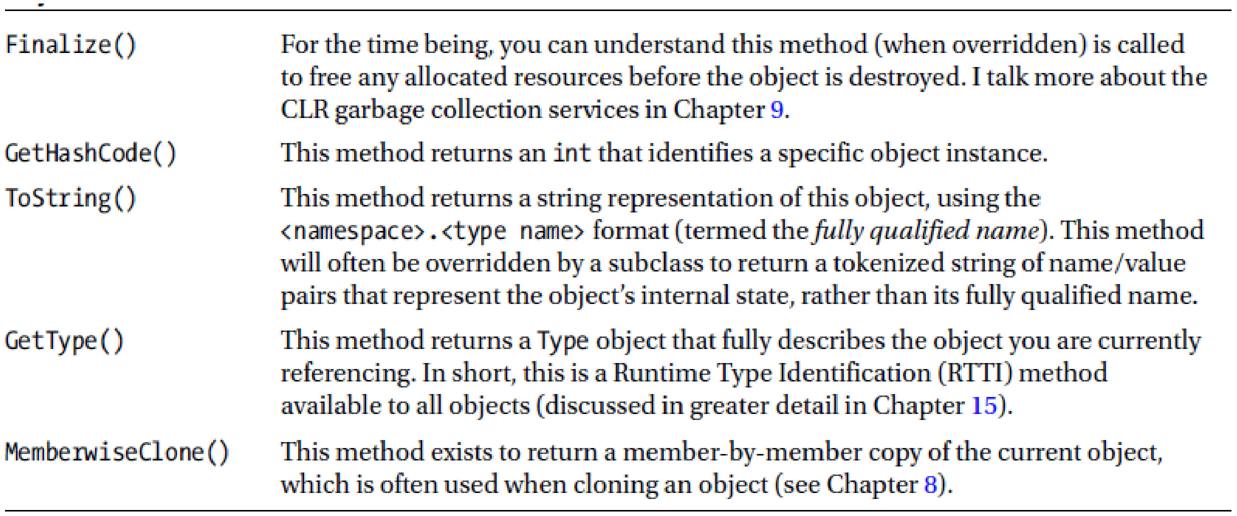


Table 6-1. Core Members of `System.Object`

Instance Method of Object Class	Meaning in Life
`Equals()`	By default, this method returns `true` only if the items being compared refer to the same item in memory. Thus, `Equals()` is used to compare object references, not the state of the object. Typically, this method is overridden to return `true` only if the objects being compared have the same internal state values (that is, value-based semantics). Be aware that if you override `Equals()`, you should also override `GetHashCode()`, as these methods are used internally by `Hashtable` types to retrieve subobjects from the container. Also recall from Chapter 4, that the `ValueType` class overrides this method for all structures, so they work with value-based comparisons.



`Finalize()`	For the time being, you can understand this method (when overridden) is called to free any allocated resources before the object is destroyed. I talk more about the CLR garbage collection services in Chapter 9.
`GetHashCode()`	This method returns an `int` that identifies a specific object instance.
`ToString()`	This method returns a string representation of this object, using the `<namespace>.<type name>` format (termed the *fully qualified name*). This method will often be overridden by a subclass to return a tokenized string of name/value pairs that represent the object's internal state, rather than its fully qualified name.
`GetType()`	This method returns a Type object that fully describes the object you are currently referencing. In short, this is a Runtime Type Identification (RTTI) method available to all objects (discussed in greater detail in Chapter 15).
`MemberwiseClone()`	This method exists to return a member-by-member copy of the current object, which is often used when cloning an object (see Chapter 8).