CSc 346 Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 6 – Part 1

Implementing Interfaces and Inheriting Classes

- This chapter is about deriving new types from existing ones using **object-oriented programming** (**OOP**). You will:
- learn how to use operators as an alternative method to implement simple functionality
- learn how to use generics to make your code safer and more performant
- learn about delegates and events to exchange messages between types
- see the differences between reference and value types
- implement interfaces for common functionality
- create a derived class to inherit from a base class to reuse functionality, override an inherited type member, and use polymorphism
- learn how to create extension methods and cast between classes in an inheritance hierarchy

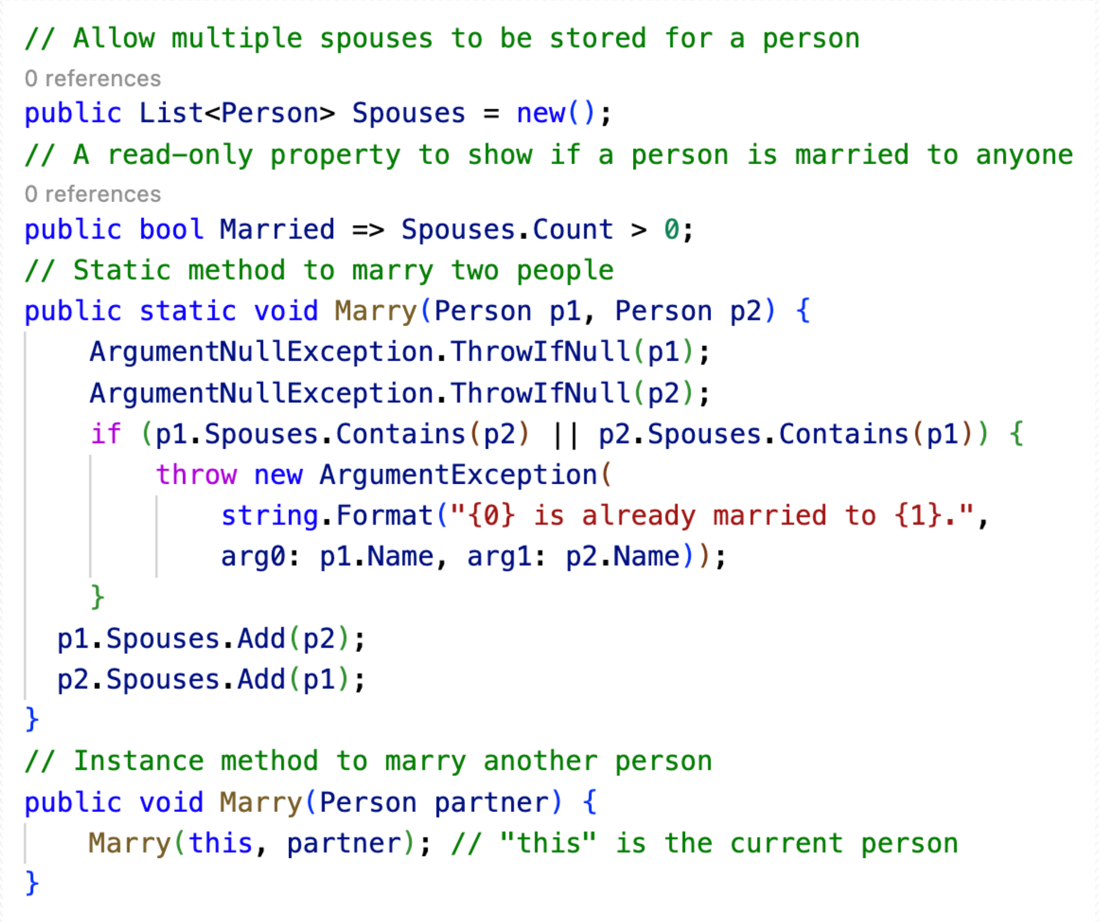
Static methods and overloading operators

- This section is specifically about methods that apply to two instances of the same type
- It is not about the more general case of methods that apply to zero, one, or more than two instances
- I wanted to think of some methods that would apply to two Person instances that could also become binary operators, like + and *
- What would adding two people together represent?
- What would multiplying two people represent?
- Possible answers are getting married and having babies

Static methods and overloading operators

- We might want two instances of Person to be able to
  - Marry
  - Procreate
- We can implement this by writing methods and overriding operators
  - Instance methods are actions that an object does to itself
  - static methods are actions the type does

Static methods and overloading operators

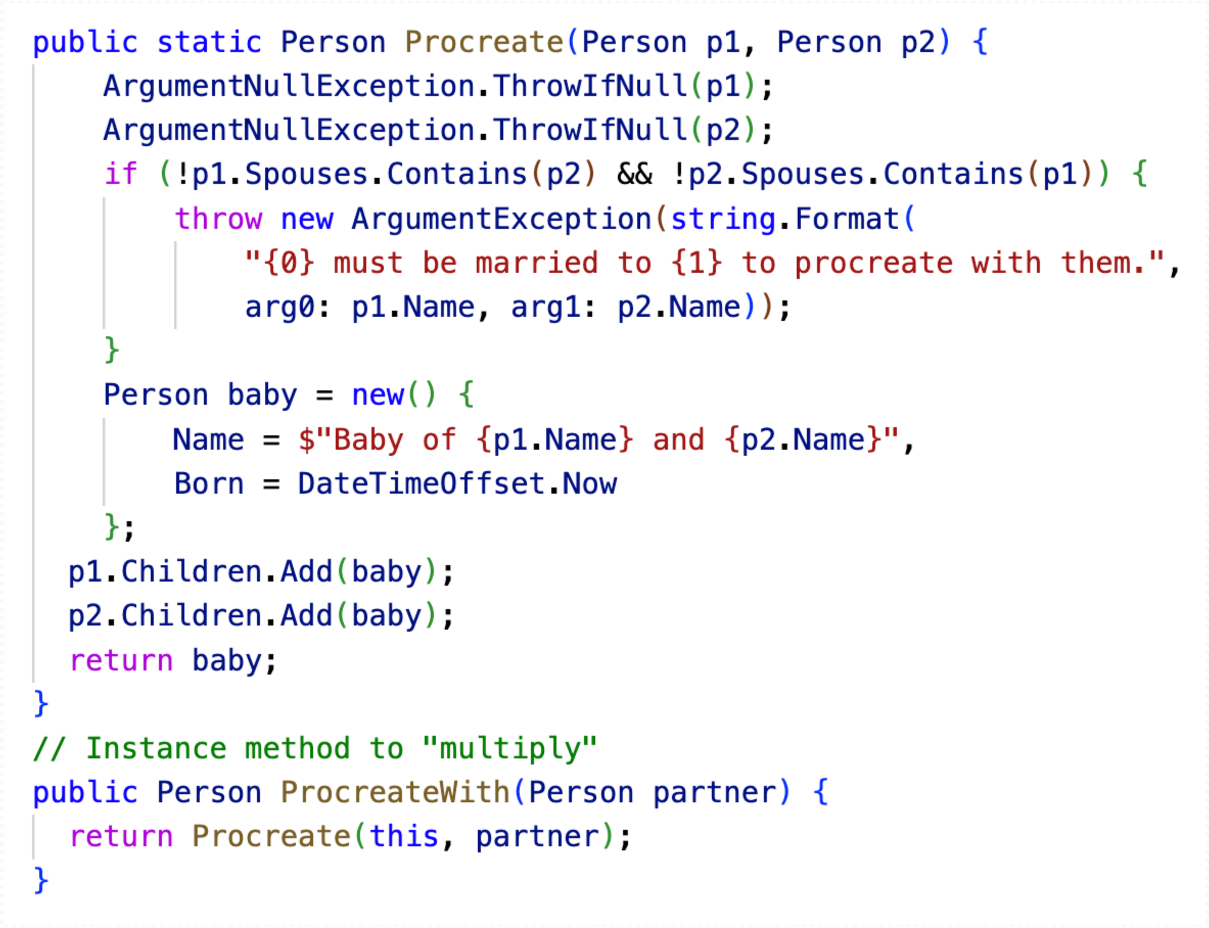


```
// Allow multiple spouses to be stored for a person
0 references
public List<Person> Spouses = new();
// A read-only property to show if a person is married to anyone
0 references
public bool Married => Spouses.Count > 0;
// Static method to marry two people
public static void Marry(Person p1, Person p2) {
    ArgumentNullException.ThrowIfNull(p1);
    ArgumentNullException.ThrowIfNull(p2);
    if (p1.Spouses.Contains(p2) || p2.Spouses.Contains(p1)) {
        throw new ArgumentException(
            string.Format("{0} is already married to {1}.",
            arg0: p1.Name, arg1: p2.Name));
    }
  p1.Spouses.Add(p2);
  p2.Spouses.Add(p1);
}
// Instance method to marry another person
public void Marry(Person partner) {
    Marry(this, partner); // "this" is the current person
}
```

Static methods and overloading operators

- In the static method, the Person objects are passed as parameters named p1 and p2, and guard clauses are used to check for null values
- If either is already married to the other an exception is thrown; otherwise, they are each added to each other's list of spouses
- You can model this differently if you want to allow two people to have multiple marriage ceremonies
- In that case, you might choose to not throw an exception and instead do nothing
- Their state of marriage would remain the same
- Additional calls to Marry would not change if they are married or not
- In this scenario, I want you to see that the code recognizes that they are already married by throwing an exception
- In the instance method, a call is made to the static method, passing the current person (this) and the partner they want to marry

Static methods and overloading operators



```
public static Person Procreate(Person p1, Person p2) {
    ArgumentNullException.ThrowIfNull(p1);
    ArgumentNullException.ThrowIfNull(p2);
    if (!p1.Spouses.Contains(p2) && !p2.Spouses.Contains(p1)) {
        throw new ArgumentException(string.Format(
            "{0} must be married to {1} to procreate with them.",
            arg0: p1.Name, arg1: p2.Name));
    }
    Person baby = new() {
        Name = $"Baby of {p1.Name} and {p2.Name}",
        Born = DateTimeOffset.Now
    };
  p1.Children.Add(baby);
  p2.Children.Add(baby);
  return baby;
}
// Instance method to "multiply"
public Person ProcreateWith(Person partner) {
  return Procreate(this, partner);
}
```

Static methods and overloading operators

- In the static method named Procreate, the Person objects that will procreate are passed as parameters named p1 and p2
- A new Person class named baby is created with a name composed of a combination of the two people who have procreated
- This could be changed later by setting the returned baby variable's Name property
- Although we could add a third parameter to the Procreate method for the baby name, we will define a binary operator later, and they cannot have third parameters, so for consistency, we will just return the baby reference and let the calling code set the name of it

Static methods and overloading operators

- The baby object is added to the Children collection of both parents and then returned
- Classes are reference types, meaning a reference to the baby object stored in memory is added, not a clone of the baby object
- In the instance method named ProcreateWith, the Person object to procreate with is passed as a parameter named partner, and that, along with this, is passed to the static Procreate method to reuse the method implementation
- this is a keyword that references the current instance of the class
- It is a convention to use a different method name for related static and instance methods
  - Compare(x, y) for the static method name
  - CompareTo(y) for the instance method name

Implementing functionality using operators

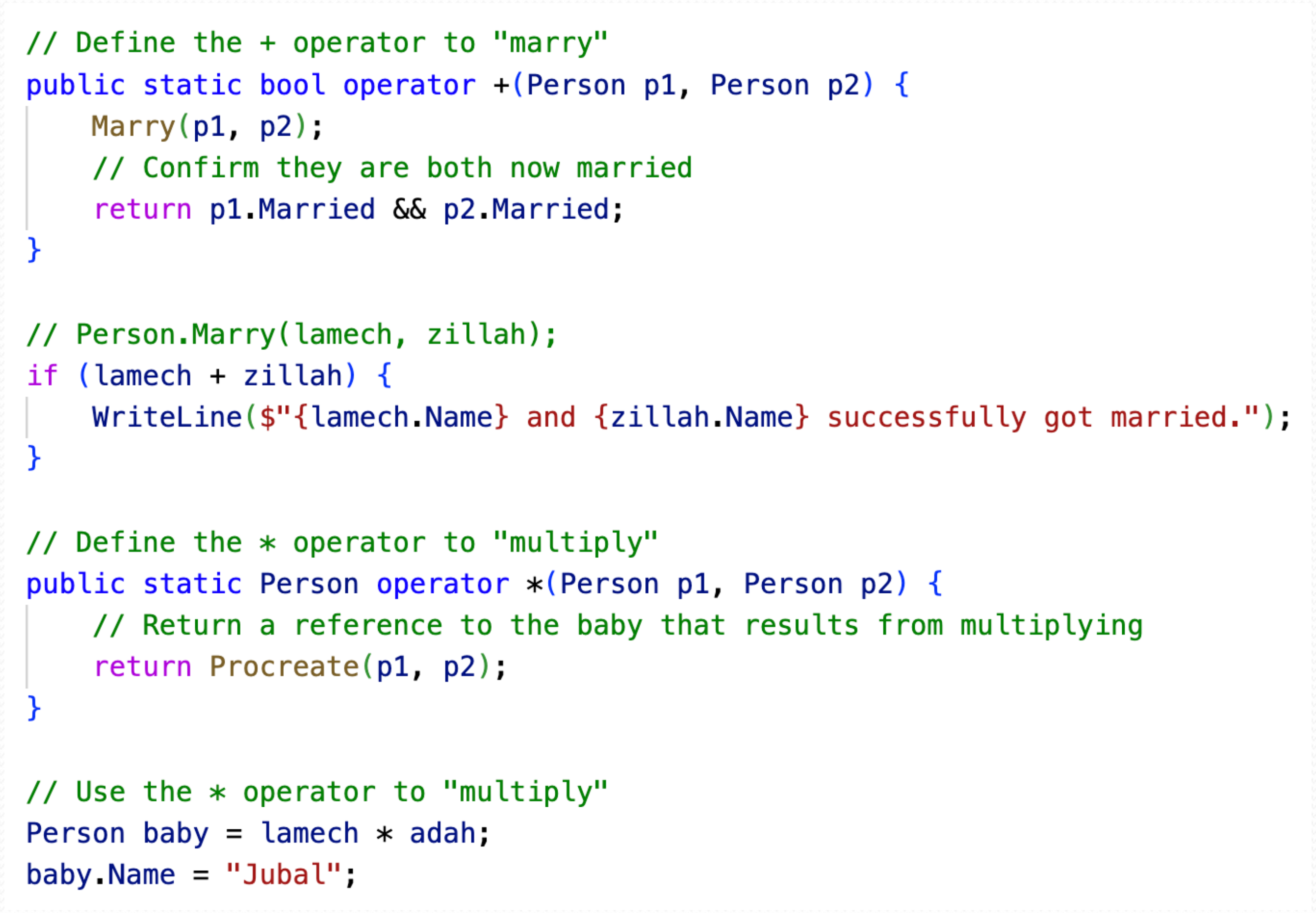
- The System.String class has a static method named Concat that concatenates two string values and returns the result

```
string s1 = "Hello ";
string s2 = "World!";
string s3 = string.Concat(s1, s2); WriteLine(s3); // Hello World!
```

- It might be more natural for a programmer to use the + symbol operator to "add" two string values together

```
string s3 = s1 + s2;
```

Implementing functionality using operators



```
// Define the + operator to "marry"
public static bool operator +(Person p1, Person p2) {
    Marry(p1, p2);
    // Confirm they are both now married
    return p1.Married && p2.Married;
}

// Person.Marry(lamech, zillah);
if (lamech + zillah) {
    WriteLine($"{lamech.Name} and {zillah.Name} successfully got married.");
}

// Define the * operator to "multiply"
public static Person operator *(Person p1, Person p2) {
    // Return a reference to the baby that results from multiplying
    return Procreate(p1, p2);
}

// Use the * operator to "multiply"
Person baby = lamech * adah;
baby.Name = "Jubal";
```

Making types safely reusable with generics

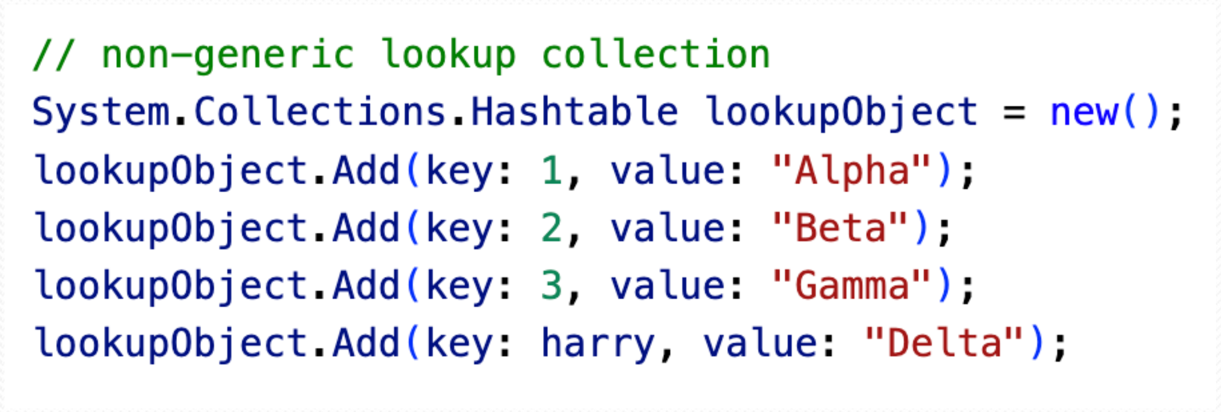
- With C# 2.0 and .NET Framework 2.0, Microsoft introduced a feature named **generics**
  - Enables your types to be more safely reusable and more efficient
- It does this by allowing a programmer to pass types as parameters
  - Similar to how you can pass objects as parameters

Working with non-generic types

- First, let's look at an example of working with a non-generic type so that you can understand the problem that generics are designed to solve
  - Weakly typed parameters and values
  - Performance problems caused by using **System.Object**
- **System.Collections.Hashtable** can be used to store multiple values each with a unique key that can later be used to quickly look up its value
  - Both the key and value can be any object because they are declared as **System.Object**
  - This provides flexibility when storing value types like integers
  - It is slow, and bugs are easier to introduce because **no type checks** are made when adding items

Working with non-generic types

1. Create an instance of the non-generic collection, **System.Collections.Hashtable**, and then add four items to it



```
// non-generic lookup collection
System.Collections.Hashtable lookupObject = new();
lookupObject.Add(key: 1, value: "Alpha");
lookupObject.Add(key: 2, value: "Beta");
lookupObject.Add(key: 3, value: "Gamma");
lookupObject.Add(key: harry, value: "Delta");
```

Working with non-generic types

2. Add statements to define a key with the value of 2 and use it to look up its value in the hash table

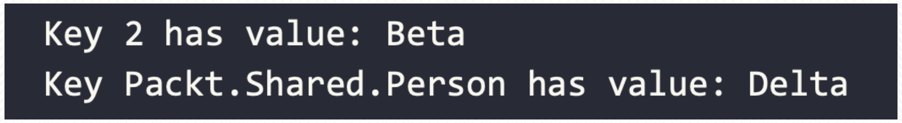
```
int key = 2;    // lookup the value that has 2 as its key
WriteLine(format: "Key {0} has value: {1}", arg0: key, arg1: lookupObject[key]);
```

3. Add statements to use the harry object to look up its value

```
// lookup the value that has harry as its key
WriteLine(format: "Key {0} has value: {1}", arg0: harry, arg1: lookupObject[harry]);
```

Working with non-generic types

4. Run the code and note that it works



```
Key 2 has value: Beta
Key Packt.Shared.Person has value: Delta
```

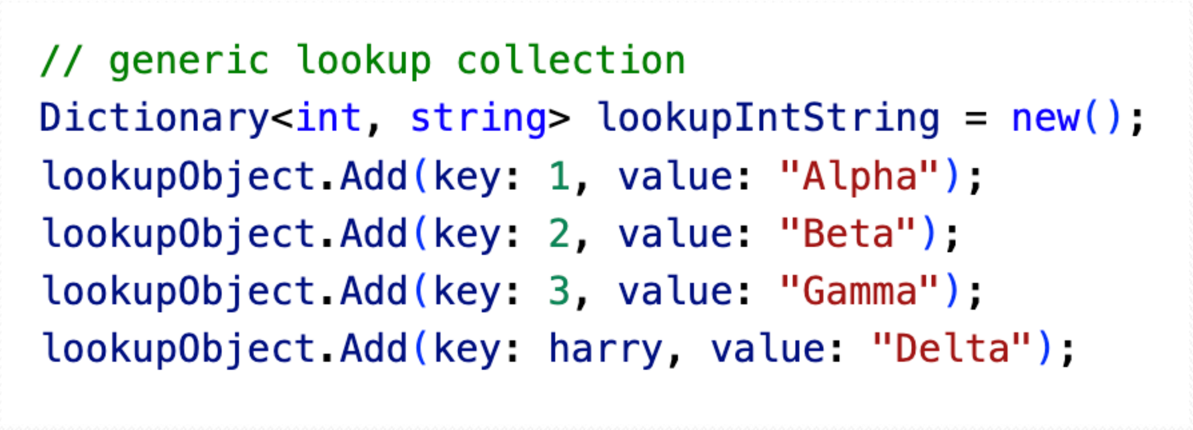
- Although the code works, there is potential for mistakes because literally any type can be used for the key or value
- If another developer used your lookup object and expected all the items to be a certain type, they might cast them to that type and get exceptions because some values might be a different type
- A lookup object with lots of items would also give poor performance

Working with generic types

- **System.Collections.Generic.Dictionary<TKey, TValue>** can be used to store multiple values each with a unique key that can later be used to quickly look up its value
- Both the key and value can be any object
  - You must tell the compiler what the types of the key and value will be when you first instantiate the collection
- You do this by specifying types for the **generic parameters** in angle brackets <>, **TKey**, and **TValue**

Working with generic types

1. Create an instance of the generic collection **Dictionary<Tkey, Tvalue>** and then add four items to it



```
// generic lookup collection
Dictionary<int, string> lookupIntString = new();
lookupObject.Add(key: 1, value: "Alpha");
lookupObject.Add(key: 2, value: "Beta");
lookupObject.Add(key: 3, value: "Gamma");
lookupObject.Add(key: harry, value: "Delta");
```

Working with generic types

2. Create an instance of the generic collection Dictionary<Tkey, Tvalue> and then add four items to it

```
/Users/markjprice/Code/Chapter06/PeopleApp/Program.cs(98,32): error
CS1503: Argument 1: cannot convert from 'Packt.Shared.Person' to 'int' [/
Users/markjprice/Code/Chapter06/PeopleApp/PeopleApp.csproj]
```

3. Replace harry with 4.

4. Add statements to set the key to 3 and use it to look up its value in the dictionary

```
key = 3;
WriteLine(format: "Key {0} has value: {1}", arg0: key, arg1: lookupIntString[key]);
```

Person class

```
public class Person : object {
    // fields
    public string? Name;   // ? allows null – nullable
    public DateTime DateOfBirth;
    public List<Person> Children = new( ); // C# 9 or later
    // methods
    public void WriteToConsole( ) { ... }
    public static Person Procreate(Person p1, Person p2) { ... }
    public Person ProcreateWith(Person partner) { ... }
}
```

Raising and handling events

- Methods are often described as **actions that an object can perform, either on itself or on related objects**
  - E.g., List<T> can add an item to itself or clear itself, and File can create or delete a file in the filesystem
- **Events** are often described as **actions that happen to an object**
  - E.g., in a user interface, Button has a **Click event**, a click being something that happens to a button, and **FileSystemWatcher listens** to the filesystem for change notifications and **raises events** like **Created** and **Deleted** that are triggered when a directory or file changes
- Another way of thinking of **events** is that they **provide a way of exchanging messages between two objects**
- Events are built on **delegates**, so let's start by having a look at what delegates are and how they work

Calling methods using delegates

- Another way to call or execute a method is to use a delegate
- A delegate contains the memory address of a method that matches the same signature as the delegate
- E.g., imagine there is a method that must have a string type passed as its only parameter, and it returns an int type

```
public int MethodIWantToCall(string input) {
    return input.Length;
}
```

- This method may be called on an instance of the Person class named p1

```
int answer = p1.MethodIWantToCall("Frog");
```

Calling methods using delegates

- Alternatively, I can define a delegate with a matching signature to call the method indirectly
- The names of the parameters do not have to match
- The types of parameters and return values must match

```
delegate int DelegateWithMatchingSignature(string s);
```

- Create an instance of the delegate, point it at the method, and then call the delegate (which calls the method)

```
// create a delegate instance that points to the method
DelegateWithMatchingSignature d = new(p1.MethodIWantToCall);
// call the delegate, which calls the method
int answer2 = d("Frog");
```

Examples of delegate use

- "What's the point of that?" **It provides flexibility**
- E.g., use delegates to create a queue of methods that need to be called in order
  - Queuing actions that need to be performed is common in services to provide improved scalability
- E.g., allow multiple actions to perform in parallel
  - Delegates have built-in support for asynchronous operations that run on a different thread, and that can provide improved responsiveness
- E.g., delegates allow us to implement **events** for sending messages between different objects that do not need to know about each other
  - Events are an example of **loose coupling** between components
    - The components do not need to know about each other
    - They just need to know the event signature
  - Delegates and events are two of the most confusing features of C# and can take a few attempts to understand, so don't worry if you feel lost!

Status: It’s complicated

- Delegates and events are two of the most confusing features of C# and can take a few attempts to understand
  - Don’t worry if you feel lost as we walk through how they work!
- Move on to other topics and come back again another day when your brain has had the opportunity to process the concepts while you sleep

Defining and handling delegates

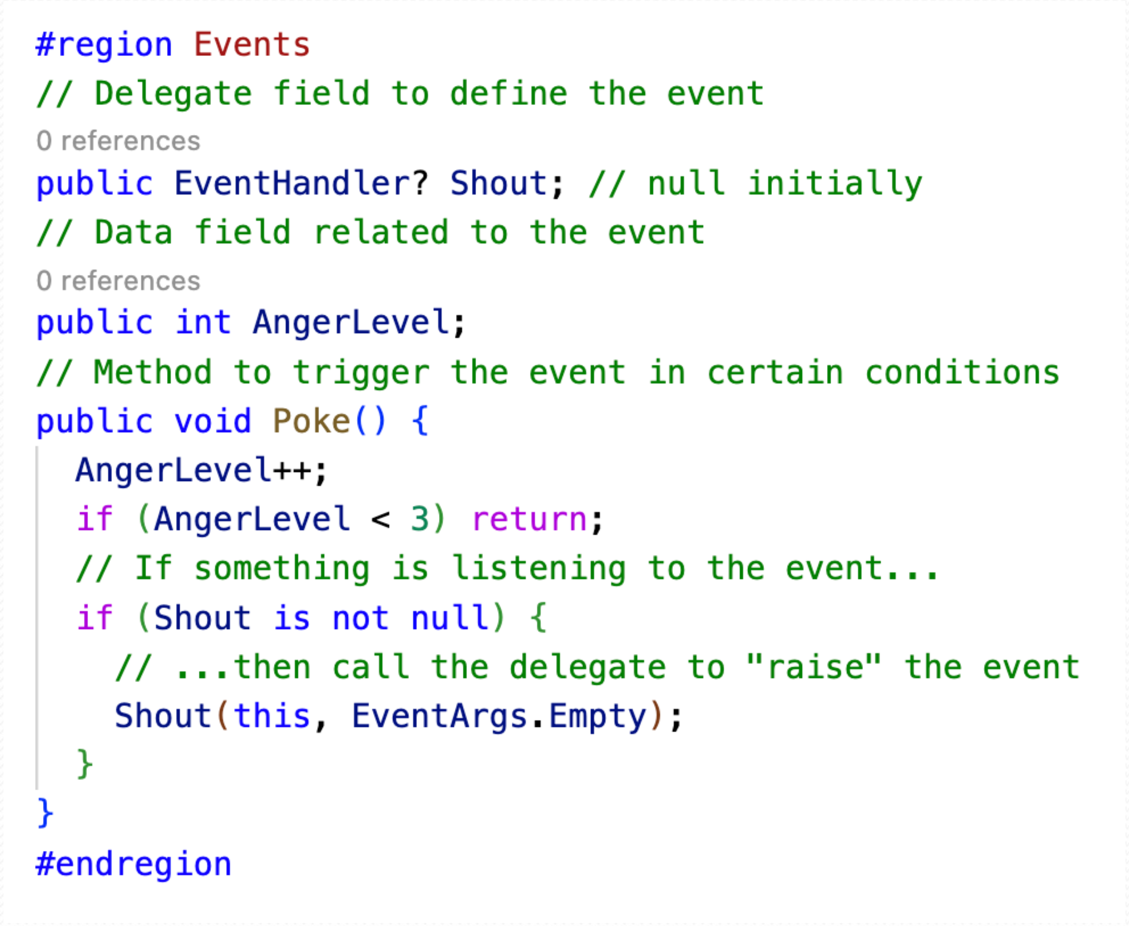
- Microsoft has two predefined delegates for use as events
  - object? sender:
    - reference to the object raising the event or sending the message
    - ? indicates that this reference could be null
  - EventArgs e or TEventArgs e:
    - contains additional relevant information about the event
    - E.g., a GUI app might define MouseMoveEventArgs, which has properties for the X and Y coordinates for the mouse pointer
    - E.g., a bank account might have a WithdrawEventArgs with a property for the Amount to withdraw

```
public delegate void EventHandler(object? sender, EventArgs e);
public delegate void EventHandler<TEventArgs>(object? sender, TEventArgs e);
```

Exploring delegates and events

1. Add statements to the Person class
   - Define an EventHandler delegate field named **Shout**
   - Define an int field to store **AngerLevel**
   - Define a method named **Poke**
   - Each time a person is poked, their AngerLevel increments
     - Once their AngerLevel reaches three, they raise the Shout event
       - Only if at least one event delegate pointing at a method defined somewhere else in the code
         - that is, it is not null

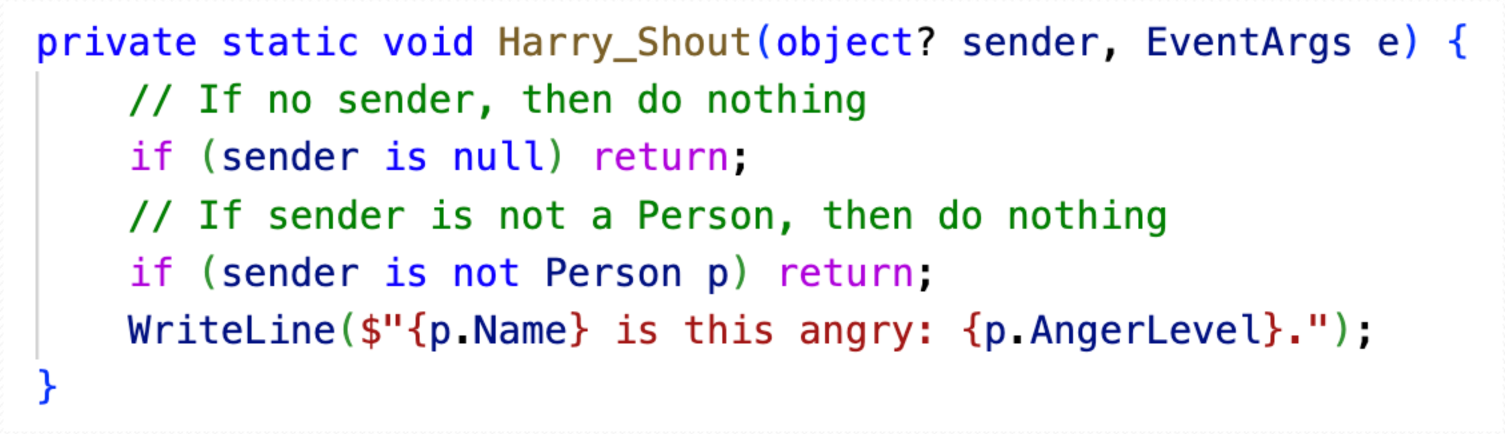
Exploring delegates and events



```
#region Events
// Delegate field to define the event
0 references
public EventHandler? Shout; // null initially
// Data field related to the event
0 references
public int AngerLevel;
// Method to trigger the event in certain conditions
public void Poke() {
  AngerLevel++;
  if (AngerLevel < 3) return;
  // If something is listening to the event...
  if (Shout is not null) {
    // ...then call the delegate to "raise" the event
    Shout(this, EventArgs.Empty);
  }
}
#endregion
```

Exploring delegates and events

2. Add a method with a matching signature that gets a reference to the Person object from the sender parameter and outputs some information about them



```
private static void Harry_Shout(object? sender, EventArgs e) {
    // If no sender, then do nothing
    if (sender is null) return;
    // If sender is not a Person, then do nothing
    if (sender is not Person p) return;
    WriteLine($"{p.Name} is this angry: {p.AngerLevel}.");
}
```

- Microsoft's convention for method names that handle events is ObjectName_EventName

Exploring delegates and events

3. Assign the method to the delegate field

```
harry.Shout = Harry_Shout;
```

4. Add statements to call the Poke method four times, after assigning the method to the Shout event

```
harry.Shout = Harry_Shout;
harry.Poke();
harry.Poke();
harry.Poke();
harry.Poke();
```

Exploring delegates and events

5. Run the code and view the result
   - Harry says nothing the first two times he is poked
   - Harry only gets angry enough to shout once he's been poked at least three times

```
Harry is this angry:  3.
Harry is this angry:  4.
```

Defining and handling events

- You've seen delegates implement the most important functionality of events:
  - The ability to define a signature for a method that can be implemented by a completely different piece of code, and then call that method and any others that are hooked up to the delegate field
- But what about events?
  - There is less to them than you might think
- When assigning a method to a delegate field, you should not use the simple assignment operator as we did
- Delegates are multicast, meaning multiple delegates can be assigned to a single delegate field
- Instead of the = operator, use the **+= operator** so more methods may be added to the same delegate field
- When the delegate is called, all assigned methods are called
  - The order in which they are called is not guaranteed

Defining and handling events

- If the Shout delegate field was already referencing one or more methods, by assigning a method, it would replace all the others
- With delegates that are used for events, we usually want to make sure that a programmer only ever uses either the
  - += **operator** to assign methods
  - -= **operator** to remove methods

Defining and handling events

1. To enforce this, in Person.cs, add the **event** keyword to the delegate field declaration

```
public event EventHandler? Shout;
```

2. Build the PeopleApp project and note the compiler error message

```
Program.cs(41,13): error CS0079: The event 'Person.Shout' can only appear
on the left hand side of += or -=
```

This is (almost) all that the event keyword does!

Defining and handling events

3. Modify the method assignment to use +=

```
harry.Shout += Harry_Shout;
```

4. Run the code and note that it has the same behavior as before

- If you will never have more than one method assigned to a delegate field, then technically you do not need "events," but it is still good practice to indicate your meaning and that you expect a delegate field to be used as an event

Implementing interfaces

- Interfaces are a way of connecting different types to make new things
- Think of them like the
  - studs on top of LEGO™ bricks, which allow them to "stick" together
  - electrical standards for plugs and sockets
- If a type implements an interface, then it is making a promise to the rest of .NET that it supports specific functionality
  - They are sometimes described as being contracts

Common interfaces

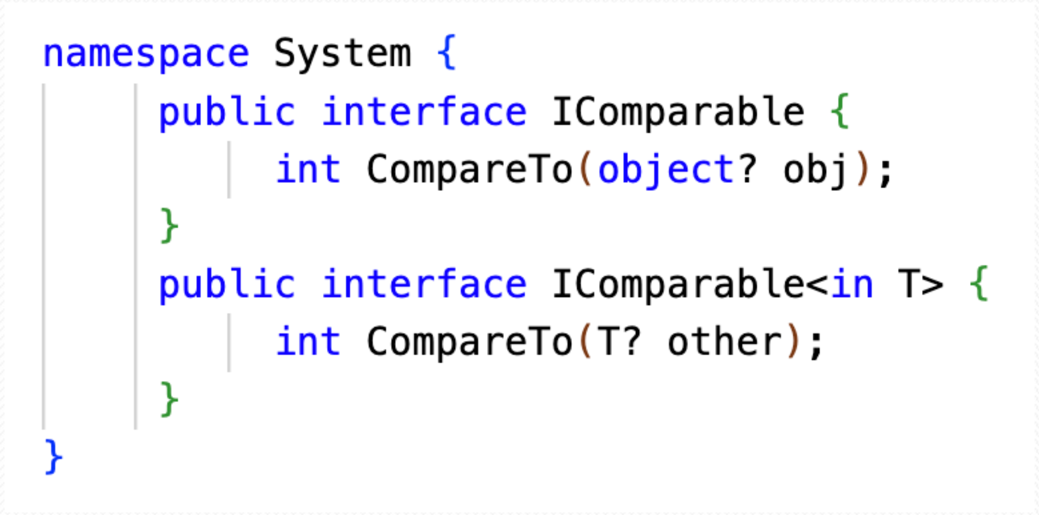
Interface	Method(s)	Description
`IComparable`	`CompareTo(other)`	This defines a comparison method that a type implements to order or sort its instances.
`IComparer`	`Compare(first, second)`	This defines a comparison method that a secondary type implements to order or sort instances of a primary type.
`IDisposable`	`Dispose()`	This defines a disposal method to release unmanaged resources more efficiently than waiting for a finalizer (see the *Releasing unmanaged resources* section later in this chapter for more details.
`IFormattable`	`ToString(format, culture)`	This defines a culture-aware method to format the value of an object into a string representation.

`IFormatter`	`Serialize(stream, object)`
	
	`Deserialize(stream)`
		This defines methods to convert an object to and from a stream of bytes for storage or transfer.

`IFormatProvider`	`GetFormat(type)`	This defines a method to format inputs based on a language and region.

Comparing objects when sorting

- One of the most common interfaces to implement is **IComparable**
- It has one method named **CompareTo**
- It has two variations
  - one that works with a nullable object type
  - one that works with a nullable generic type T



```
namespace System {
    public interface IComparable {
        int CompareTo(object? obj);
    }
    public interface IComparable<in T> {
        int CompareTo(T? other);
    }
}
```

Comparing objects when sorting

- The string type implements **IComparable** by returning
  - -1 if the string is less than the string being compared to
  - 1 if the string is greater than the string being compared to
- The int type implements **IComparable** by returning
  - -1 if the int is less than the int being compared
  - 1 if the int is greater than the int being compared
- If a type implements one of the IComparable interfaces, then arrays and collections can sort it
- Before we implement the IComparable interface and its CompareTo method for the Person class let's see what happens when we try to sort an array of Person instances

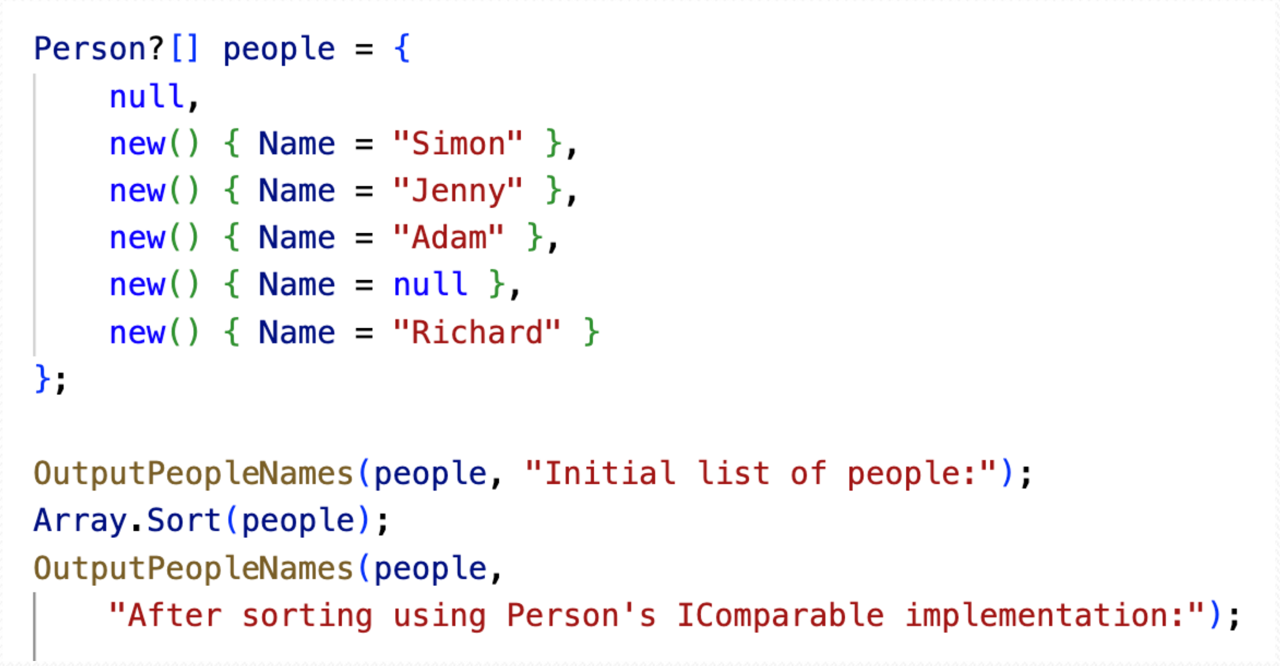
Comparing objects when sorting

1. Add a method to the Program class that will output all the names of a collection of people passed as a parameter with the title beforehand

```
private static void OutputPeopleNames(
    IEnumerable<Person?> people, string title) {
        WriteLine(title);
        foreach (Person? p in people) {
            WriteLine("  {0}",
            p is null ? "<null> Person" : p.Name ?? "<null> Name");
            /* if p is null then output: <null> Person
               else output: p.Name
               unless p.Name is null then output: <null> Name */
    }
}
```

Comparing objects when sorting

2. Add statements that create an array of Person instances and write the items to the console, and then attempt to sort the array and write the items to the console again



```
Person?[] people = {
    null,
    new() { Name = "Simon" },
    new() { Name = "Jenny" },
    new() { Name = "Adam" },
    new() { Name = null },
    new() { Name = "Richard" }
};

OutputPeopleNames(people, "Initial list of people:");
Array.Sort(people);
OutputPeopleNames(people,
    "After sorting using Person's IComparable implementation:");
```

Comparing objects when sorting

3. Run the code and an exception will be thrown. As the message explains, to fix the problem, our type must implement IComparable

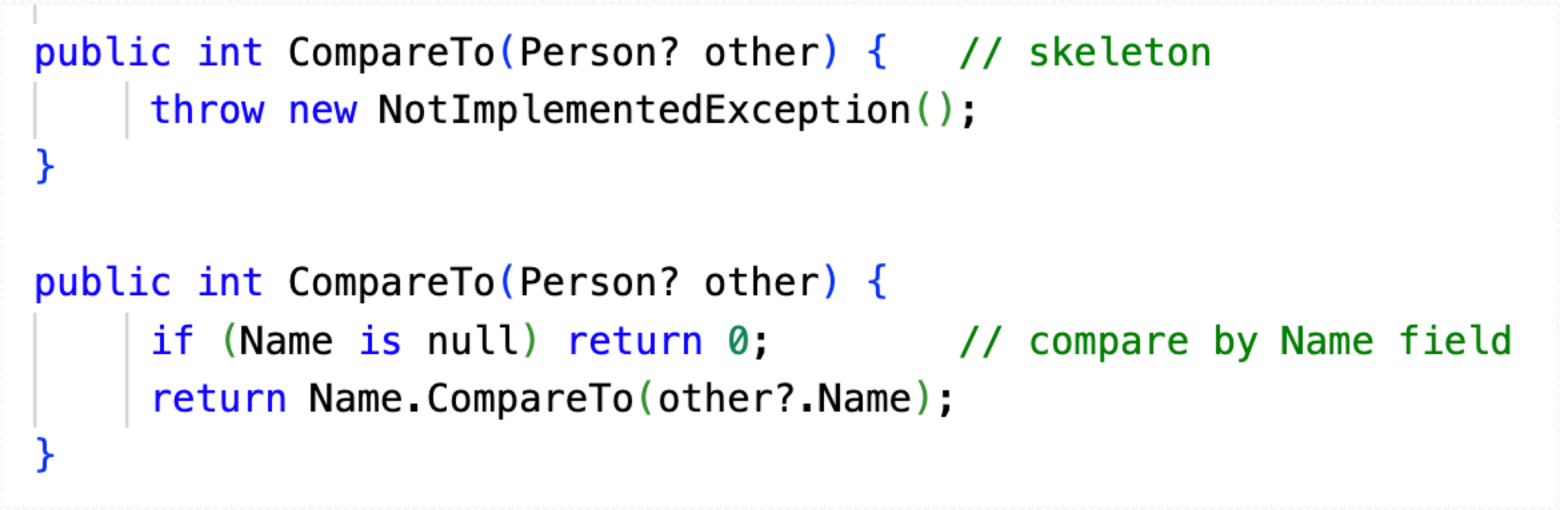
```
Unhandled Exception: System.InvalidOperationException: Failed to compare
two elements in the array. ---> System.ArgumentException: At least one
object must implement IComparable.
```

4. In Person.cs, after inheriting from object, add a comma and enter IComparable<Person?>

```
public class Person : object, IComparable<Person?>
```

Comparing objects when sorting

- Your code editor will draw a red squiggle under the new code to warn you that you have not yet implemented the method you have promised to
- Your code editor can write the skeleton implementation for you if you click on the light bulb and choose the **Implement interface** option



```
public int CompareTo(Person? other) {   // skeleton
    throw new NotImplementedException();
}

public int CompareTo(Person? other) {
    if (Name is null) return 0;         // compare by Name field
    return Name.CompareTo(other?.Name);
}
```

Implicit and explicit interface implementations

- Interfaces can be implemented implicitly and explicitly
    - Implicit implementations are simpler and more common
    - Explicit implementations are only necessary if a type must have multiple methods with the same name and signature

- E.g., both **IGamePlayer** and **IKeyHolder** might have a method named **Lose** with the same parameters because both a game and a key can be lost
    - In a type that must implement both interfaces, only one implementation of Lose can be the implicit method
    - If both interfaces can share the same implementation, that works
    - If both interfaces cannot share the same implementation, then the other Lose method will have to be implemented differently and called explicitly

Implicit and explicit interface implementations

```
public interface IGamePlayer { void Lose(); }
public interface IKeyHolder { void Lose(); }
public class Person : IGamePlayer, IKeyHolder {
    public void Lose()          // implicit { // implement losing a key }
    void IGamePlayer.Lose()     // explicit { // implement losing a game }
}

// calling implicit and explicit implementations of Lose
Person p = new();
p.Lose();                   // calls implicit implementation of losing a key
((IGamePlayer)p).Lose();    // calls explicit implementation of losing a game
// calls explicit implementation of losing a game
IGamePlayer player = p as IGamePlayer;
player.Lose();
```

Defining interfaces with default implementations

1. Define a public IPlayable interface with two methods to Play and Pause

```
public interface IPlayable { void Play(); void Pause(); }    // methods are public
```

2. Define a new class file named DvdPlayer.cs and have it implement the IPlayable interface

```
public class DvdPlayer : IPlayable {
    public void Pause() { WriteLine("DVD player is pausing."); }
    public void Play() { WriteLine("DVD player is playing."); }
}
```

Defining interfaces with default implementations

- What if we decide to add a third method named Stop?
- One of the main points of an interface is that it is a fixed contract
- C# 8.0 allows an interface to add new members after release as long as they have a default implementation
- C# purists do not like the idea, but for practical reasons, such as avoiding breaking changes or having to define a whole new interface, it is useful, and other languages such as Java and Swift enable similar techniques
- Support for default interface implementations requires some fundamental changes to the underlying platform, so they are only supported with C# if the target framework is .NET 5.0 or later, .NET Core 3.0 or later, or .NET Standard 2.1. They are therefore not supported by .NET Framework

Defining interfaces with default implementations

3. Modify the IPlayable interface to add a Stop method with a default implementation

```
public interface IPlayable {
    void Play();
    void Pause();
    void Stop() {                              // default interface implementation
        WriteLine("Default implementation of Stop.");
    }
}
```