COMP 3602

C# Application Development

Week Four

Some Notes

In principle: For *compiled languages,* declare variables *as close as possible* to where they are used

- Not all at the top
- Easier to understand what it is if it is in context
- No confusion around the scope of the variable being too large

Split() and similar methods – what's wrong with this?:

```
string firstName = lineData.Split(',')[0];
string lastName = lineData.Split(',')[1];
string age = lineData.Split(',')[2];
```

Don't do this: Console.WriteLine("----------");

Or this: Console.WriteLine(" {0} {1}", firstName, lastName);

Some Notes

Pay attention to details:

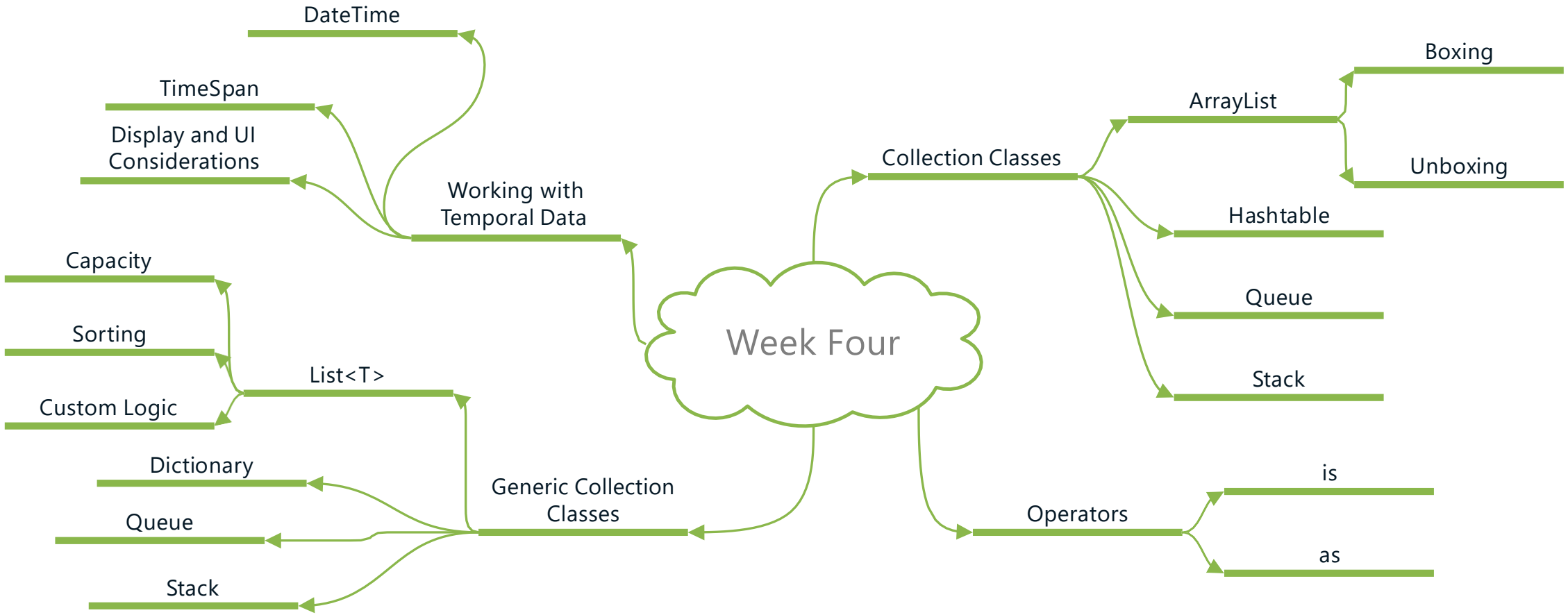
5) Override the ToString method to return `LastName, FirstName` (use an interpolated string)

6) Prompt and collect data from the user into string variables.

- It takes time and experience to build up a good intuition of what is important and what is not/what assumptions we can reasonably make and when to ask for clarifications instead.

To try things out, try https://dotnetfiddle.net/

Tonight's Learning Outcomes



Collection Classes

The System.Collections namespace includes several collection classes that are more flexible than a simple array:

Class Name	Description
ArrayList	has an index, just like an array
Hashtable	has a key, no index
Queue	implements a queue data structure (FIFO)
Stack	implements a stack data structure (LIFO)

ArrayList

ArrayList whose purpose is to hold Student objects

```
26
27      students.Add(new Cat { Name = "Felix" });
28
```

Adding a Cat object to the Student ArrayList. This will compile and run, unfortunately

- Stores all types as *object*
- No type safety
- Must *cast* to retrieve element
- Value types must by *boxed* and *unboxed*

Getting a reference to the first Student with an explicit cast (Student)

```
22
23      tempStudent = (Student)students[0];
24
25      Console.WriteLine(tempStudent.Id);
26
```

Use the Student object
Without the cast, we could not access the Student's Id Property

Boxing

Boxing an integer in a variable of type object

```
15
16      int age = 23;
17      object obj = age;
18
```

28, anotherAge, and oneMoreAge are automatically boxed before being added to the ArrayList

```
19
20      ArrayList ages = new ArrayList();
21
22      int anotherAge = 38;
23      int oneMoreAge = 5;
24
25      ages.Add(28);
26      ages.Add(anotherAge);
27      ages.Add(oneMoreAge);
28
```

- Value types are stored as type Object in an ArrayList
- Must be wrapped (boxed) as a reference type to be stored as Object
- This operation incurs runtime overhead

Unboxing

Unboxing operation

```
19
20          age = (int)obj;
21
```

Explicit Cast Required

- Need to explicitly cast back to original type
- We get a net new value created on the stack

Unboxing operation

```
32
33          anotherAge = (int)ages[1];
34
```

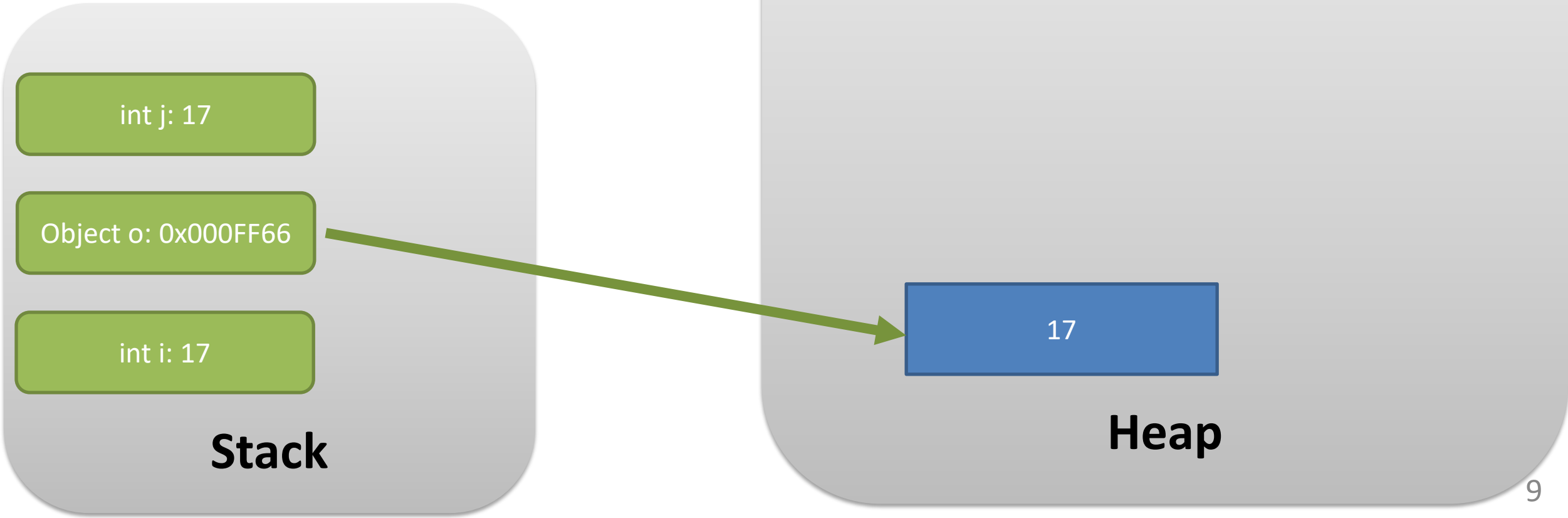
Explicit Cast Required

Boxing/Unboxing

```
int i = 17;         //value type

Object o = i;       //Boxing

int j = (int)o;     //Unboxing
```

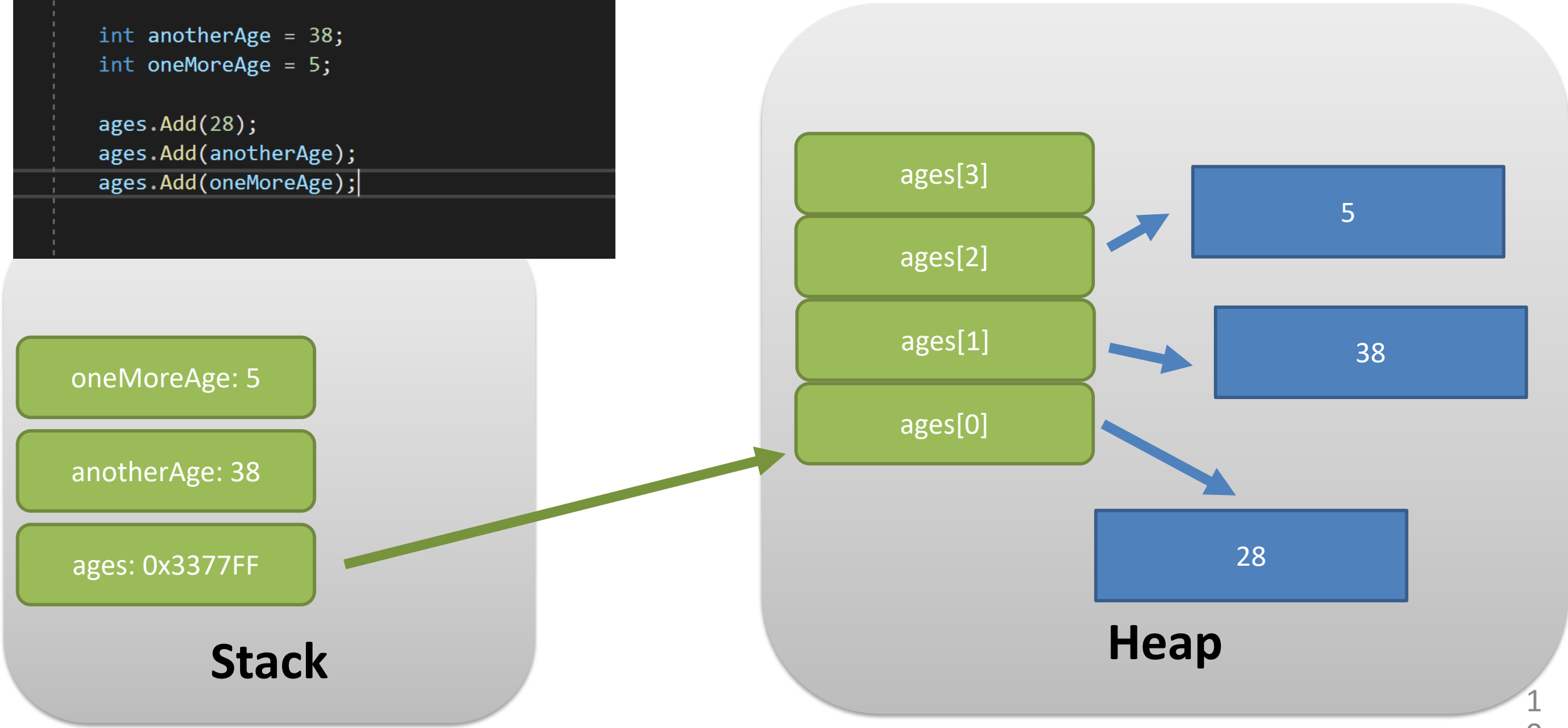


Boxing/Unboxing - Arraylist

```
ArrayList ages = new ArrayList();

int anotherAge = 38;
int oneMoreAge = 5;

ages.Add(28);
ages.Add(anotherAge);
ages.Add(oneMoreAge);
```



The is Operator

The is operator provides a simpler syntax to determine data types at runtime:
(Applies to All Types)

```
if (object.GetType().Name == “MyType”) …
```

Can be rewritten as:

```
if (object is MyType) …
```

The as Operator

The as operator provides a safer way to downcast objects to their original type:
(Applies to Reference Types Only)

An unsuccessful cast will throw an exception:

```
MyType myObject = (MyType)object;
```

An unsuccessful cast will return null:

```
MyType myObject = object as MyType;
```

It is common to perform a null test to determine success of the cast:

```
if (myObject != null)
{
      ...;
}
```

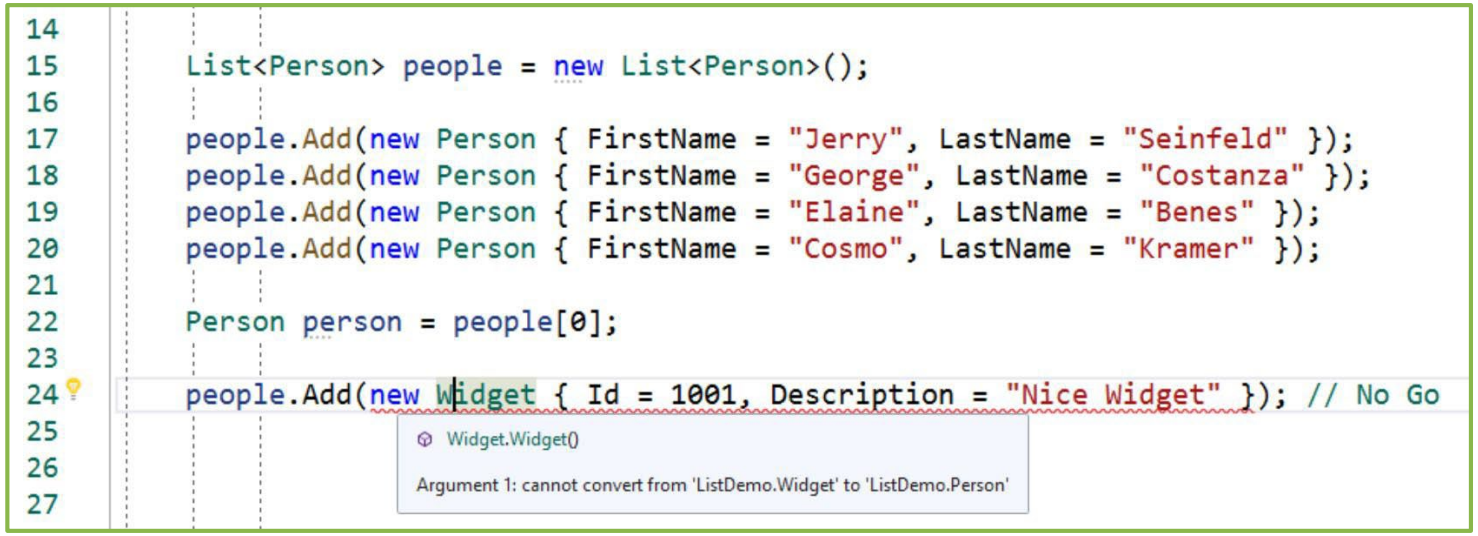
Generic Collection Classes

The System.Collections.Generic namespace includes several collection classes that use generics:

Class Name	Description
List	like an ArrayList but uses generics
Dictionary	like HashTable but uses generics
Queue	like queue but uses generics (note same class name)
Stack	like stack but uses generics (note same class name)

List<T>

List is similar to an ArrayList except it uses *generics*
To declare and instantiate a List object, you must specify the type it will hold



- List<Type>
- Type Safe
- Only accepts specified type or descendants
- Add(item): Adds passed item
- Remove(item): Removes passed item
- RemoveAt(index): Removes item at specified position

List Capacity

```
14
15      List<int> intList = new List<int>();
16
```

```
List Capacity Demo
 Elements  Capacity
--------------------
        0         0
        1         4
        2         4
        3         4
        4         4
        5         8
        6         8
        7         8
        8         8
        9        16
       10        16
       11        16
       12        16
```

```
14
15      List<int> intList = new List<int>(20);
16
```

```
List Capacity Demo
 Elements  Capacity
--------------------
        0        20
        1        20
        2        20
        3        20
        4        20
        5        20
        6        20
        7        20
        8        20
        9        20
       10        20
       11        20
       12        20
```

- The default capacity of a List is zero, increases to four after the first addition and then doubles every time the current capacity is exhausted.
- This incurs some runtime overhead. You can pass an int value to the constructor to pre-size the initial capacity to avoid this.

List Sorting – IComparable<T> Interface

The List<T> class has a Sort method which requires implementation of the IComparable<T> interface in the element class

```
11      12 references
        class Product : IComparable<Product>
12      {
            9 references
13          public int Id { get; set; }
            11 references
14          public string Description { get; set; }
            9 references
15          public decimal Price { get; set; }
16
            1 reference
17          public int CompareTo(Product other)
18          {
19              if (other == null)
20              {
21                  return 1;
22              }
23
24              return this.Description.CompareTo(other.Description);
25          }
26      }
27
```

If other object is null, this object is greater than other

Default order is ascending; reverse sign for descending

Determine the field or fields on which to base your sort criteria and invoke their CompareTo methods

- Basic types like int and string implement IComparable<T>
- You can implement IComparable<T> in your own classes when you wish to support sorting in a List
- This forces implementation of the CompareTo method which returns an int and takes one argument of the class type
- CompareTo Returns:
  - 1 this > other
  - 0 this == other
  - -1 this < other

List<T> - Aliases and Custom Logic

This class inheritance provides an alias for List<Employee>

```
5 references
class EmployeeList : List<Employee>
{
}
```

This collection employs custom logic to process its elements

```
5 references
class EmployeeList : List<Employee>
{
    1 reference
    public void GiveBonuses(decimal totalBonusFund)
    {
        decimal bonus = totalBonusFund / this.Count;

        foreach (Employee employee in this)
        {
            employee.GiveBonus(bonus);
        }
    }
```

- A very simple yet powerful technique is to inherit from one of the BCL generic classes to provide a custom list class that complements a business class
- Any logic that applies to a list of employees can be encapsulated within that class. Also, an EmployeeList object will only accept Employees

Dictionary

Dictionary stores elements as Key/Value pairs:

- Dictionary<key, value>
- Add(key, value): Adds passed Key with passed Value
- Keys Must Be Unique
- Remove(key): Removes element with passed Key

To declare and instantiate a Dictionary, you must specify the type of Key and the type of Value

A Dictionary's Add method takes two parameters, the Key and Value

```
14
15      Dictionary<string, Person> people = new Dictionary<string, Person>();
16
17      people.Add("JS", new Person { FirstName = "Jerry", LastName = "Seinfeld" });
18      people.Add("GC", new Person { FirstName = "George", LastName = "Costanza" });
19      people.Add("EB", new Person { FirstName = "Elaine", LastName = "Benes" });
20      people.Add("CK", new Person { FirstName = "Cosmo", LastName = "Kramer" });
21
22
```

Queue

```
13
14      Queue<string> queue = new Queue<string>();
15
16      queue.Enqueue("One");
17      queue.Enqueue("Two");
18      queue.Enqueue("Three");
19
20      while (queue.Count > 0)
21      {
22          Console.WriteLine(queue.Dequeue());
23      }
24
```

```
Queue Demo
One
Two
Three
```

FIFO Operation:

Enqueue: Adds an Item
Dequeue: Returns and Removes First Item
Peek: Returns First Item (No removal)
Count: Returns number of Items
Clear: Removes all Items

Stack

```
13
14      Stack<string> stack = new Stack<string>();
15
16      stack.Push("One");
17      stack.Push("Two");
18      stack.Push("Three");
19
20      while (stack.Count > 0)
21      {
22          Console.WriteLine(stack.Pop());
23      }
24
```

```
Stack Demo
Three
Two
One
```

LIFO Operation:

Push:	Adds an Item
Pop:	Returns and Removes Last Item
Peek:	Returns Last Item (No removal)
Count:	Returns number of Items
Clear:	Removes all Items

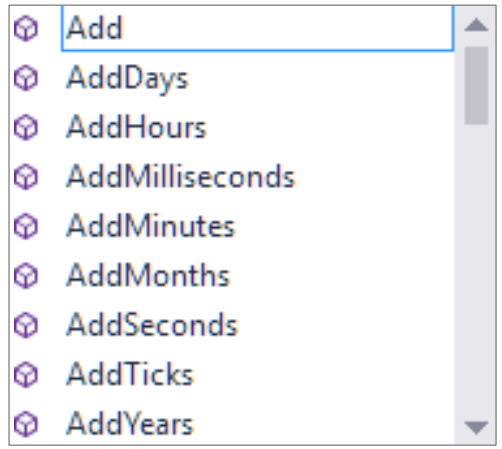
Working with Temporal Data – DateTime Data Type

One (of many) constructor overloads takes three ints representing year, month and day

```
15
16      DateTime today = new DateTime(2019, 10, 1);
17
18      int year = today.Year;
19      int month = today.Month;
20      int day = today.Day;
21
22      DateTime tomorrow = today.AddDays(1);
23      DateTime yesterday = today.AddDays(-1);
24
25      string todayString = today.ToString("MMM d, yyyy");
26      // returns Oct 1, 2019
27
28
```

Several properties expose component parts of the stored DateTime value

- DateTime Is a Struct.
- Internal storage is a long (64 bit int)
- Stores a unit called a "Tick"
- One Tick = 100 nanoseconds or 1/10,000,000 second




```
DateTime date = new DateTime(); // Jan 1, 0001
DateTime.Now // System Time
```

Working with Temporal Data – TimeSpan Data Type

- TimeSpan Is a Struct.
- Designed to store a duration of time.
- The difference between two DateTime values.

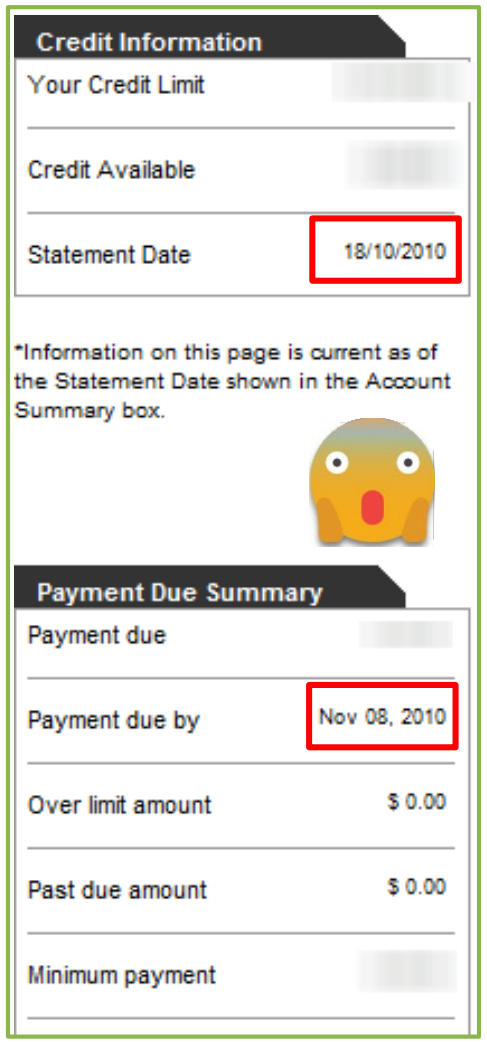
One (of many) constructor overloads takes three ints representing hours, minutes and seconds

```
14
15      TimeSpan span = new TimeSpan(0, 0, 3690);
16
17      Console.WriteLine($"Duration: {span.Hours:D2}:{span.Minutes:D2}:{span.Seconds:D2}");
18      // outputs Duration: 01:01:30
19
20      Console.WriteLine($"Total minutes: {span.TotalMinutes}\n");
21      // outputs Total minutes: 61.5
22
```

Several properties expose component parts of the stored TimeSpan value

Temporal Data – Display and UI Considerations

HSBC Mastercard



RBC ATM Receipt
Date: 06JAN2015

HSBC ATM Receipt
Date: 01/11/14
(Actual Date is Nov 1, 2014)

Always ensure your date output (screen and print) is consistent and unambiguous

hsbc.ca