Week 1: Notes

This week's topics are covered in Essential C#: see ch. 1 Introducing C#, ch. 2 Data Types, ch. 3 More with Data Types (Arrays), ch. 4 Operators and Control Flow.

Here is a summary of the C# elements we discussed:

hello, world

```
using System;

class Hello {
  static void Main() {
    Console.WriteLine("hello, world");
  }
}
```

local variable declarations

A local variable declaration declares one or more local variables and optionally gives them initial values:

```
    int i = 3, j = 4;
```

Implicitly typed variables

When you declare a local variable, you can use the `var` keyword to tell the compiler to infer its type. For example:

```
    var s = "distant hill";
```

This is equivalent to

```
    string s = "distant hill";
```

comments

A single-line comment begins with `//` and extends to the end of the line:

```
x += 1;    // increment x
```

Comments delimited with `/*` and `*/` can extend over multiple lines:

```
/*
  this is a comment with
  multiple lines
*/
```

integer types

- `sbyte` – 8 bits, signed (-128 to 127)
- `byte` – 8 bits, unsigned (0 to 255)
- `short` – 16 bits, signed (-32,768 to 32,767)
- `ushort` – 16 bits, unsigned (0 to 65,535)
- `int` – 32 bits, signed (-2^{31} to $2^{31} - 1$)

- `uint` – 32 bits, unsigned (0 to 2^{32} – 1)
- `long` – 64 bits, signed (- 2^{63} to 2^{63} – 1)
- `ulong` – 64 bits, unsigned (0 to 2^{64} - 1)

A literal integer (or floating-point value) may contain embedded underscores for readability:

```
int i = 1_000_000_000;    // 1 billion
```

The inferred type of an integer literal is the first of these types which can hold it: int, uint, long, ulong.

floating-point types

- `float` – 32 bits, 7 significant digits
- `double` – 64 bits, 15-16 significant digits
- `decimal` – 128 bits, 28 significant digits, base-10 representation

The inferred type of a floating-point literal (e.g. `3.4`) is `double` by default. To create a literal of type `float`, append the character 'f', e.g. `3.4f`.

numeric conversions

C# will perform an implicit conversion between two numeric types if every possible value of the source type is valid in the destination type. For example:

```
short s;
...
int i = s;   // implicit conversion
```

You can also implicitly convert a `char` to any numeric type that can hold a 16-bit unsigned value:

```
char c = 'ř';
int i = c;   // now i holds the Unicode value for 'ř', i.e. 345
```

You can use an explicit conversion to force a conversion between any two numeric types. This is accomplished with a type cast:

```
int i = 1000;
short s = (short) i;
```

If the destination type cannot hold the source value, it will wrap around or be truncated.

You can explicitly convert any numeric type to a `char`:

```
int i = 100;
char c = (char) i;  // now c is 'd', which is ASCII/Unicode character 100
```

bool

The `bool` type represents a Boolean value, namely either `true` or `false`.

char

A `char` is a 16-bit Unicode character.

A character constant is enclosed in single quotes, e.g. `'x'` or `'ř'`. A character constant containing a backslash is an escape sequence. Here are some common escape sequences:

- `\n` – newline
- `\r` – carriage return
- `\'` - single quote
- `\"` - double quote
- `\\` - backslash

To create a character constant representing a single quote or backslash, use one of the sequences above:

```
WriteLine('\\');  // writes a backslash
WriteLine('\'');  // writes a single quote
```

string

A `string` is an immutable sequence of characters. A string literal is enclosed in double quotes, e.g. `"hello"`. Ordinary string literals may contain escape sequences.

You may access individual characters of a string using square brackets:

```
string s = "spire";

char c = s[2];    // now c = 'i'
```

Note that characters are indexed starting from 0.

interpolated values

A string beginning with `$` may contain interpolated values enclosed in braces:

```
int a = 3, b = 4;

WriteLine($"{a} plus {b} equals {a + b}");
```

A interpolated value may be followed by a format specifier (preceded by a colon).

There are many predefined format specifiers. Many format specifiers can be followed by an integer called the precision, whose meaning varies among specifiers. Here are two useful specifiers:

- `f` – prints a number in fixed-point decimal (i.e. not using scientific notation). The precision is the number of the digits after the decimal point.
- `n` – prints a number using thousands separators. The precision is the number of digits after the decimal point.

For example,

```
const double pi = 3.14159;
const int i = 1357;

WriteLine($"{pi:f2} {i:n0}");
```

writes

```
3.14 1,357
```

verbatim strings

A literal string preceded with @ is a verbatim string, and differs from an ordinary string literal in two ways:

- A backslash (\) in a verbatim string is an ordinary backslash; it does not introduce an escape sequence.
- A verbatim string may extend over multiple lines.

To include a double quote character (") in a verbatim string, type it twice. For example:

```
string s = @"He said, ""Where is she?"""

Console.WriteLine(s)
```

prints

```
He said, "Where is she?"
```

arrays

You can allocate an array like this:

```
    int[] i = new int[10];
```

With this form of allocation all elements are initialized to the element type's default value. For example, the default value for integers is 0.

Alternatively, you can allocate an array and initialize its elements to specific values:

```
    int[] i = { 3, 4, 5 };
```

Arrays are indexed from 0. The `Length` property returns the length of an array.

arithmetic operators

- `+` : addition

  The `+` operator can also be used to concatenate strings:

  ```
      WriteLine("good" + " " + "bread");
  ```

  It can also concatenate strings and other kinds of values:

  ```
      int i = 4;

      WriteLine("i = " + i);
  ```

- `-` : subtraction
- `*` : multiplication
- `/` : division

  The `/` operator performs integer division if both its arguments have integer types:

  ```
      WriteLine(7 / 3);    // will write 2
  ```

  Otherwise, floating-point division is performed:

  ```
      WriteLine(7 / 3.0);  // will write 2.3333
  ```

  The result of integer division is truncated toward zero. For example, -17 / 5 is -3. (This differs from some other languages such as Python, which truncates toward -∞.)

- `%` : remainder

  Be aware that if a < 0 and b > 0, then (a % b) will be negative or zero, never positive. For example, (-17 % 5) is -2. This is consistent with the fact that division truncates toward zero. (This differs from some other languages such as Python. In Python, if b > 0 then (a mod b) is always a value in the range 0 ≤ x < b.)

boolean operators

- ! : not
- | : or
- && : and

equality operators

- `==` : equals
- `!=` : does not equal

relational operators

These operators are defined on numbers and characters, but not on strings.

- `<` : less than
- `<=` : less than or equal to
- `>` : greater than
- `>=` : greater than or equal to

compound assignment operators

- `+=` : addition
- `-=` : subtraction
- `*=` : multiplication
- `/=` : division
- `%=` : remainder

if

```
if (i > 0) {
  WriteLine("positive");
  WriteLine("definitely positive");
} else if (i == 0)
  WriteLine("zero");
else
  WriteLine("negative");
```

An `if` statement executes a statement (or block) if the given value is true. If the statement has an `else` clause, it is executed if the given value is false.

while

```
while (i < 10) {
  sum = sum + i;
  i += 1;
}
```

A `while` loop loops as long as the given condition is true.

do / while

```
do {
  s = ReadLine();
} while (s != "yes" && s != "no");
```

A `do`/`while` loop is like a `while` loop, but checks the loop condition at the bottom of the loop body.

for

```
for (int i = 0 ; i < 10 ; i += 1)
  WriteLine(i);
```

A `for` statement contains three clauses, separated by semicolons, plus a loop body.

- The initializer (e.g. `int i = 0`) executes once at the beginning of the loop. It contains either a variable declaration, or one or more statements separated by commas.
- The condition (e.g. `i < 10`) is evaluated before every loop iteration. If it is false, the loop terminates.
- The iterator (e.g. `i += 1`) executes at the end of every loop iteration. It contains one or more statements, separated by commas.

Week 2: Notes

This week's topics are covered in Essential C#: see ch. 3 More with Data Types (Tuples, Arrays), ch. 5 Methods and Parameters, ch. 6 Classes.

Here is a summary of the C# elements we discussed:

conditional operator

The conditional operator (`? :`) is similar to the `if` statement, but it is an expression, not a statement. It takes a Boolean value and two extra values. It returns the first of these values if the boolean is true, or the second if it is false. For example, the following statement computes the greater of `i` and `j` and assigns it to `k`:

```
int i, j;
...
int k = i > j ? i : j;
```

switch

```
switch (i) {
  case 1:
    WriteLine("one");
    break;

  case 2:

  case 3:
    WriteLine("two or three");
    return;

  default:
    WriteLine("other");
    break;
}
```

The `switch` statement is a more compact (and possibly more efficient) alternative to a series of if statements. The `default` section is optional. Each case in a `switch` statement must end with a `break` or `return` statement.

foreach

```
foreach (char c in "hello")
  Write(c + " ");
```

`foreach` iterates over each element of a collection. The only kinds of collections we have learned about so far are strings and arrays (but we will soon see others).

A `foreach` statement always declares an iteration variable whose scope is the body of the statement.

tuples

A tuple holds two or more values. Unlike arrays, tuples have fixed length and can hold values of varying types.

A tuple type is written as a list of element types in parentheses. For example:

```
(int, string) p = (3, "hello");
```

We can access a tuple's elements using the property names Item1, Item2 and so on:

```
WriteLine(p.Item1);    // writes 3

WriteLine(p.Item2);    // writes "hello"
```

We can use multiple assignment to unpack a tuple into a set of variables:

```
(int x, string s) = p;   // now x is 3, s = "hello"
```

named tuples

The generic element names `Item1` and `Item2` are not very informative. Sometimes it is clearer to use a named tuple, which specifies names for the elements. For example:

```
(int x, int y) p = (10, 20);
```

Now we can access the tuple elements by name:

```
WriteLine($"x = {p.x}, y = {p.y}");
```

Be sure to note the difference between the syntax for multiple assignment and the syntax for creating a named tuple – these may look similar at first. The declaration above creates a single variable p with fields p.x and p.y. By contrast, the line

```
(int x, int y) = (10, 20);
```

is a multiple assigment that creates local variables x and y.

multidimensional arrays

There are two kinds of multidimensional arrays: rectangular arrays and jagged arrays.

You can dynamically allocate a rectangular array using the `new` operator:

```
int[,] a = new int[3, 4];
```

Or you can allocate an array and initialize it with a set of values:

```
int[,] a = { {1, 4, 5}, {2, 3, 6} };
```

You can access an element of a rectangular array using an expression such as `a[2,5]`.

An rectangular array may even have 3 or more dimensions, e.g.

```
int[,,] a = new int[5, 10, 15];    // a 3-dimensional array
```

The `Length` property returns the total number of elements in a rectangular array. This is the product of the lengths of all dimensions. The `Rank` property returns the number of dimensions, and the method `int GetLength(int d)` returns the length of dimension d.

A jagged array is an array of arrays. Unlike a rectangular multidimensional arrays, each subarray in a jagged array can have a different length.

You can allocate a jagged array like this:

```
int[][] i = new int[3][];
i[0] = new int[2];
i[1] = new int[4];
i[2] = new int[5];
```

If you like, you can initialize a jagged array as you allocate it:

```
    int[][] i = {
      new int[] { 2, 3},
      new int[] { 3, 4, 5},
      new int[] { 1, 3, 5, 7, 9}
    };
```

You can access an element of a jagged array using an expression such as `a[2][5]`.

classes

A class is a user-defined data type that can contain fields, constructors, methods, and other kinds of members.

Here is a definition for a simple class implementing a fixed-size stack:

```
class Stack {

    // fields

    int[] a;
    int count;

    // constructor

    public Stack(int maxSize) {
        a = new int[maxSize];
        count = 0;
    }

    // methods

    public bool isEmpty() {
        return count == 0;
    }

    public void push(int i) {
        a[count] = i;
        count += 1;
    }

    public int pop() {
        count -= 1;
        return a[count];
    }
}
```

access levels

Every member in a class can be either `public` or `private`. `public` members are accessible everywhere, and `private` members are accessible only within the class.

The default access level is `private`.

fields

Each instance of a class contains a set of `fields`.

A field may be declared with an initial value:

```
class Foo {
  int x = 3, y = 4;
  …

}
```

If you don't specify an initial value, a field will intially be set to its type's default value (e.g 0 for an `int`).

Code in a constructor or method may refer to fields by name and may modify their values.

constructors

A constructor makes a new instance of a class. It always has the same name as its containing class.

A constructor will often intialize fields. For example:

```
class Point {
  double x, y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
…
}
```

To call a constructor, use the `new` operator and pass any required arguments:

```
    Point p = new Point(3.0, 4.0);
```

If a class definition includes no constructors, then C# provides a default constructor that is public and takes no arguments.

Constructors may be overloaded, meaning that a class may contain several constructors with different numbers and/or types of arguments.

A constructor may call another constructor; this is called constructor chaining. For example, we could add a second constructor to the Point class that takes no arguments, and constructs a Point at the origin (0,0):

```
    public Point() : this(0.0, 0.0) {  }
```

this

`this` is a special value that refers to the object on which a constructor or method was invoked.

methods

A method is like a function, but belongs to a class. In a method declaration, the return type precedes the method name:

```
int mul(int x, int y) {
  return x * y;
}
```

overloaded methods

You may declare multiple methods that have the same name but have different numbers and/or types of parameters. This is called method overloading. For example:

```
int abc(int i) {
  return i + 1;
}

int abc(int i, int j) {
  return i + j;
}

string abc(string s) {
  return s + s;
}
```

expression-bodied methods

If a method's body is a simple expression, the method can be defined using a compact syntax:

```
int mul(int x, int y) => x * y;
```

static members

A field or method may be static. Static members are shared by all instances of a class.

```
class Foo {
  static int x = 1;
  static int y;

  static void inc() { x += 1; }
}
```

A static method is invoked on a class, not on an instance:

```
Foo.inc();
```

Static methods may access private members of any instance of their class. For example, we can add a static method to the `Point` class that compares two `Point` objects and returns true if they are equal:

```
public static bool eq(Point p, Point q) =>
  p.x == q.x && p.y == q.y;
```

value and reference types

Every type in C# is either a value type or a reference type.

- value types: all numeric types, `bool`, `char`
- reference types: `string`, arrays, classes

When a variable's type is a value type, the variable holds a value. An assignment between variables of value type copies the value from one variable to another.

When a variable's type is a reference type, the variable holds a reference to an object. An assignment between variables of reference type makes them refer to the same object.

For example:

```
    int[] a = { 4, 5, 6 };
    int[] b = a;       // now b and a refer to the same array
    a[1] = 7;
    WriteLine(b[1]);   // writes 7
```

using static

At the top of a source file, you can use a `using static` declaration to import all static methods from a given class, so that you can use these methods without prefixing with them with the class name. For example, if you write

```
    using static System.Math;
```

then your code will be able to call `Sqrt()` and `Exp()` directly without having to write `Math.Sqrt()` and `Math.Exp()`. This is often convenient.

tutorial

In the tutorial, we solved a couple of exercises. Here is the dynamic array class we wrote:

```
using static System.Math;

class DynArray {
    int[] a;
    int length;

    public DynArray(int length, int val) {
        a = new int[length];
        this.length = length;
        for (int i = 0 ; i < length ; ++i)
            a[i] = val;
    }

    public DynArray(int length) : this(length, 0) { }

    public DynArray() : this(0) { }

    int get(int index) => a[index];

    void set(int index, int val) {
        a[index] = val;
    }

    int getLength() => length;

    void setLength(int len) {
        if (len > a.Length) {   // must grow array
            int[] b = new int[Max(len, 2 * a.Length)];
            for (int i = 0 ; i < length ; ++i)
```

```
                b[i] = a[i];
            a = b;
        }
        for (int i = length ; i < len ; ++i)
            a[i] = 0;
        length = len;
    }

    void append(int i) {
        setLength(length + 1);
        a[length - 1] = i;
    }

    int[] toArray() {
        int[] b = new int[length];
        for (int i = 0 ; i < length ; ++i)
            b[i] = a[i];
        return b;
    }
}
```

Week 3: Notes

This week's topics are covered in Essential C#: see ch. 3 More with Data Types (Nullable Modifier), ch. 5 Methods and Parameters (Advanced Method Parameters, Optional Parameters), ch. 6 Classes (Properties), ch. 9 Value Types (Enums), ch. 10 Well-Formed Types (Operator Overloading), ch. 17 Building Custom Collections (Providing an Indexer).

Here is a summary of the C# elements we discussed:

enums

An enum type holds one of a fixed set of constant values:

```
enum Suit {
  Club, Diamond, Heart, Spade
}
```

To refer to one of these values, prefix it with the type name:

```
Suit s = Suit.Diamond;
```

If you'd like to be able to access an enum's values without the type name prefix, include a `using static` declaration at the top of your source file:

```
using static Suit;
```

Then you will be able to write, for example:

```
Suit s = Diamond;
```

Internally, an enumerated value is stored as an integer. Each constant in an enum is assigned an integer value, starting from 0. For example, in the enumeration above, Diamond is assigned the value 1.

Explicit conversions exist between each enum type and `int` in both directions:

```
Suit s = Suit.Diamond;
int i = (int) s;    // convert a Suit to an int
Suit t = (Suit) i;  // convert an int to a Suit
```

nullable types

Value types such as `int` and `bool` cannot hold `null`. However, each such type can be made nullable by appending a ? character to its name. A nullable type holds either an instance of its base type, or the value `null`. For example:

```
int? x = 7;
if (x > 3)
   x = null;
```

A value of the base type can be converted to a nullable type implicitly:

```
int y = abc();
int? z = y;
```

To convert from a nullable type to its base type, use an explicit cast or access the `Value` property. These are equivalent, and will fail with an exception at run time if the value is `null`.

```
int? a = xyz();
int b = (int) a;
int c = a.Value;  // equivalent
```

optional parameters

You can make a method parameter optional by giving it a default value:

```
void fill(int[] a, int start = 0, int val = -1) {
  for (int i = start ; i < a.Length ; ++i)
    a[i] = val;
}
```

The caller can omit any optional parameters:

```
int[] a = new int[10];
fill(a);  // same as fill(a, 0, -1)
fill(a, 5);  // same as fill(a, 5, -1)
fill(a, 7, 3);
```

Optional parameters must appear after any non-optional parameters in a parameter list.

named arguments

When you invoke any method, you may precede any argument with a parameter name, followed by a colon.

For example, the `Substring` method in the `string` class is defined as

```
public string Substring (int startIndex, int length);
```

We may invoke it in any of these ways:

```
string s = "bowling ball";
string t = s.Substring(1, 3);  // "owl"
string u = s.Substring(startIndex: 1, length: 3);  // identical
string v = s.Substring(length: 3, startIndex: 1);  // identical
```

Notice that named arguments may appear in any order, and must appear after any non-named arguments in an argument list.

If some method parameters are optional, you may specify any subset of them you like using named arguments. For example, we may invoke the `fill` method from the previous section as

```
fill(a, val: 4)  // same as fill(a, 0, 4)
```

ref and out parameters

A parameter marked with `ref` is passed by reference: the method can modify the variable that is passed. Here's a method that swaps two variables:

```
static void swap(ref int a, ref int b) {
  int t = a;
  a = b;
  b = t;
}
```

You must include the keyword `ref` when passing an argument by reference:

```
int i = 3, j = 4;
swap(ref i, ref j);
```

A parameter marked with `out` returns a value to the caller. This method takes two integers and returns both their sum and product:

```
    static void sumAndProduct(int a, int b, out int sum, out int product) {
      sum = a + b;
      product = a * b;
    }
```

You must include the keyword `out` when passing a variable to an `out` parameter:

```
    int s, p;
    sumAndProduct(3, 4, out s, out p);
```

You can declare a new variable as you invoke a method with an `out` parameter. The following is equivalent to the two preceding lines:

```
    sumAndProduct(3, 4, out int s, out int p);
```

parameter arrays (params)

If the last parameter to a method is marked with `params` and has type T[], then in its place the method can receive any number of arguments of type T. The caller may pass individual values separated by commas, or may pass an array of type T[].

For example, this method receives a parameter array of integers and returns their sum:

```
    static int sum(params int[] a) {
      int s = 0;
      foreach (int i in a)
        s += i;
      return s;
    }
```

It can be invoked as

```
    int s = sum(4, 5, 6);
```

Alternatively, the caller can pass an array:

```
    int[] a = { 4, 5, 6 };
    int s = sum(a);
```

properties

A property is syntactically like a field, but contains a getter and/or a setter, which are methods that run when the caller retrieves or updates the property's value. Inside the setter, the keyword `value` refers to the value that is being set.

Here is a partial listing of a class `Vector` that includes a property `length`:

```
class Vector {
  double[] a;
  ...

  public int length {
    get {
      return a.Length;
    }
    set {
      a = new double[value];
    }
  }
}
```

You can use expression syntax to define getters or setters. The `length` property above could be written as

```
public int length {
  get => a.Length;

  set => a = new double[value];
}
```

indexers

An indexer allows you to define custom getter and/or setter methods that run when an instance of your class is accessed using the array-like syntax `a[i]`. For example, we can extend the Vector class above with an indexer that retrieves and sets elements of the underlying array v:

```
public double this[int i] {
    get { return a[i]; }
    set { a[i] = value; }
}
```

A caller can now invoke this indexer as if v itself were an array:

```
Vector v = new Vector(...);
v[3] = 77.2;
v[4] = v[5] + 1.0;
```

The indexer defined above has return type `double` and uses an index parameter of type `int`. In general, an indexer may have any return type and any index parameter type.

overloaded operators

You may defined overloaded operators for a class, which redefine the meaning of built-in operators such as + and * when invoked on instances of the class. For example:

```
class Vector {
  double[] a;

  public Vector(params double[] a) { this.a = a; }

  public static Vector operator + (Vector v, Vector w) {
    double[] b = new double[v.a.Length];
    for (int i = 0 ; i < v.a.Length ; ++i)
      b[i] = v.a[i] + w.a[i];
    return new Vector(b);
  }
}
```

The operator above can be invoked like this:

```
Vector v = new Vector(2.0, 5.0, 10.0);
Vector w = new Vector(1.0, 3,0, 9.9);
Vector x = v + w;
```

An overloaded operator must be `public` and `static`.

You may overload most of the built-in operators available in C#, including

- unary operators: `+`, `-`, `!`, `++`, `–`
- binary operators: `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`

Week 4: Notes

There was no lecture or tutorial this week.

Our topics for this week are interfaces and generics. They are discussed in our textbook Essential C# 7.0 (Chapter 8. Interfaces; Chapter 12. Generics: C# without Generics; Introducing Generic Types; Constraints; Generic Methods). These topics are also covered in Programming C# 8.0 (Chapter 3. Types: Interfaces; Chapter 4. Generics).

Here are some additional notes:

interfaces

In Introduction to Algorithms we learned about various abstract data types, such as stacks and queues. For each abstract data type, we described the methods that it provides. For example, a stack has methods `push()`, `pop()` and `isEmpty()`. We saw that we can implement each abstract data type in various ways. For example, we can implement a stack either using a linked list, or an array. Different implementations of an abstract data type will have equivalent functionality, though they might have different performance characteristics.

C# interfaces formalize this concept. An interface defines a set of methods or other members in an abstract data type. For example, here is an interface for a stack of integers:

```
interface Stack {
  bool isEmpty();
  void push(int i);
  int pop();
}
```

There is no implementation yet - just a set of methods to be implemented. We can write a concrete class `LinkedStack` that implements a stack using a linked list. It might begin like this:

```
class LinkedStack : Stack {
  Node head;

  public bool isEmpty() { return (head == null); }

  public void push(int i) {
    head = new Node(i, head);
  }

  public int pop() { … }
}
```

Above, ”`class LinkedStack : Stack`” means that we are defining a class `LinkedStack` that implements the `Stack` interface. The compiler will check that our class provides an implementation of every method that was declared in the interface. If any implementations are missing, we will get a compiler error.

We might then write a second class `ArrayStack` that implements a stack using a fixed-size array. It could begin like this:

```
class ArrayStack : Stack {
  int[] a;
  int num;  // number of elements currently on the stack

  public ArrayStack(int maxSize) {
    a = new int[maxSize];
  }

  public bool isEmpty() { return num == 0; }

  …
}
```

We now have two different concrete implementations of a `Stack`. We might create two `Stack` objects as follows:

```
Stack s = new LinkedStack();
Stack t = new ArrayStack(100);
```

Notice that a variable can have an interface type, such as the variables `s` and `t` above. A variable with an interface type can hold any object that implements that interface.

Method parameters can also have an interface type. For example, here is a method that will empty a `Stack`, printing all values as it does so:

```
static void empty(Stack s) {
  while (!s.isEmpty()) {
    int i = s.pop();
    WriteLine(i);
  }
}
```

We may pass either a `LinkedStack` or an `ArrayStack` to this method. For example:

```
ArrayStack as = new ArrayStack(100);
for (int i = 0 ; i < 5 ; ++i)
  as.push(i);
empty(as);
```

Notice that an interface may not contain a constructor. Any class that implements an interface may have its own constructor(s), but those are not part of the interface.

An interface may, however, contain properties and indexers. For example, let's modify the `Stack` interface so that `isEmpty` is a read-only property, not a method:

```
interface Stack {
  bool isEmpty { get; }
  void push(int i);
  int pop();
}
```

All members in an interface are implicitly public, so you should not write the keyword `public` in an interface declaration. However, when you write a class that implements an interface, all implementations of the interface's members must be declared `public` (as you can see in the examples above).

Note that a class may implement more than one interface. For example, let's write an interface `Countable` for any type that has a current element count:

```
interface Countable {

  int count { get; }
}
```

Now we can modify our `ArrayStack` class so that it implements both `Stack` and `Countable`:

```
class ArrayStack : Stack, Countable {
  int[] a;
  int num;  // number of elements currently on the stack

  public ArrayStack(int maxSize) {
    a = new int[maxSize];
  }

  public bool isEmpty => (num == 0);  // implementation of property from Stack

  public int count => num;  // implementation of property from Countable

  …
}
```

generic classes

We have now learned enough C# that we can implement data structures such as stacks, queues, and binary trees. However, in the subset of C# that we know so far, each of our implementations will be tied to some specific type. For example, we can write a stack class that holds integers:

```
class IntStack {
  int[] a;
  int num;  // number of elements currently on the stack

  public IntStack(int maxSize) {
    a = new int[maxSize];
  }
}
```

Similarly, we can write a stack that holds strings:

```
class StringStack {
  string[] a;
  int num;  // number of elements currently on the stack

  public StringStack(int maxSize) {
    a = new string[maxSize];
  }
}
```

But this is a significant limitation - we certainly don't want to have to rewrite our data structure classes once for every possible element type!

In Python last semester, we had no such issue because Python is dynamically typed. When we wrote a stack class in Python, we could add anything at all to it:

```
# Python code
s = new Stack()

s.push(14)

s.push('hello')

s.push([1, 2, 3])
```

C# is different: it is a statically typed language, meaning that every variable has a type. And so C# provides a different mechanism called generic classes. Generic classes let us write a container class only once and reuse its implementation for every possible element type.

A generic class has one or more type parameters. For example, we can implement a class called `FixedStack<T>`. This means ”a fixed-size stack of objects of type T”. Our class might look like this:

```
class FixedStack<T> {
  T[] a;
  int num;  // number of elements currently on the stack

  public FixedStack(int maxSize) {
    a = new T[maxSize];
  }

  public void push(T val) {
    a[num] = val;
    num += 1;
  }

  public T pop() {
    num -= 1;
    T ret = a[num];
    return ret;
  }
}
```

In the class above, T is a type parameter. When the caller creates a `FixedStack`, they will specify the type T:

```
FixedStack<int> s = new FixedStack<int>(100);  // create a stack of ints
s.push(15);
s.push(25);

FixedStack<string> t = new FixedStack<string>(100);  // create a stack of strings
t.push("hello");
t.push("goodbye");
```

Notice that inside the class `FixedStack<T>`, the type T can be used like any other type: we can create an array of objects of type T, or use T as a method parameter type, a method return type or a variable type.

Observer that a `FixedStack` is not quite like our stack implementation in Python, where a single stack could hold objects of various types. In C#, by contrast, each `FixedStack` is tied to some particular element type. That is a good thing: if we create a `FixedStack<int>`, intending to hold ints, and then attempt to push a string to it by mistake, we will get a compile-time error.

By the way, the word T above is arbitrary: a type parameter can have any name you like. It is traditional, however, in C# (and other languages with generics) to use ”T” for classes that take a single type parameter.

multiple type parameters

A single class may take multiple type parameters. For example, suppose that we want to write a `Dictionary` class that maps keys to values using a hash table. It might look like this:

```
class Dictionary<K, V> {
  …
  public void add(K key, V val) { … }

  public bool contains(K key) { … }
}
```

Here, K and V are two type parameters. When the caller creates a `Dictionary`, they will specify types for K and V:

```
Dictionary<int, string> d = new Dictionary<int, string>();   // maps int → string
d.add(10, "sky");
d.add(20, "purple");

Dictionary<string, double> e = new Dictionary<string, double>();  // maps string → double
e.add("purple", 55.2);
```

Let's consider how we could implement such a class. As we learned in Intro to Algorithms, a hash table contains an array of hash chains, each of which consists of a linked list of nodes. So we will need a Node class that is itself generic:

```
class Node<K, V> {
  public K key;
  public V val;
  public Node<K, V> next;

  public Node(K key, V val) {
    this.key = key; this.val = val;
  }
}
```

Notice that inside a generic class such as `Node`, we must specify the type parameters <K, V> when we refer to the class itself (e.g in the field declaration `Node<K, V> next`), but the constructor declaration does not include the type parameters.

Now, inside the `Dictionary` class we could have a field that holds an array of hash chains of appropriate type:

```
class Dictionary<K, V> {

  Node<K, V> a[];

}
```

generic interfaces

We discussed generic classes above. An interface may also be generic. For example, let's revisit the `Stack` interface we wrote before, which described a stack of ints:

```
interface Stack {
  bool isEmpty { get; }
  void push(int i);
  int pop();
}
```

We will now modify it to be a generic interface `Stack<T>`, which is a stack of elements of type T:

```
interface Stack<T> {
  bool isEmpty { get; }
  void push(T i);
  T pop();
}
```

Above, we wrote a class `FixedStack<T>`, implementing a fixed-size stack of elements of type T. We can now modify that class so that it implements the `Stack<T>` interface:

```
class FixedStack<T> : Stack<T> {
  T[] a;
  int num;  // number of elements currently on the stack
…
}
```

Once again, a variable may have an interface type:

```
Stack<double> s = new FixedStack<double>(100);

s.push(2.0);
s.push(4.0);
```

IComparable

Many types are naturally ordered: for any two values, we can say whether one is greater or less than the other. For example, we can compare integers or doubles in the obvious fashion, or can compare strings alphabetically. For other types, an ordering may not be so obvious: for example, does it make sense to say that one stack is greater or less than another? Perhaps not.

The C# standard library contains a generic interface `IComparable<T>` that is implemented by all ordered built-in types. `IComparable<T>` means "comparable with objects of type T". For example, the built-in type `int` implements `IComparable<int>`, since integers are comparable with integers. Similarly, `string` implements `IComparable<string>`.

`IComparable<T>` is defined like this:

```
interface IComparable<T> {

  int CompareTo (T other);

}
```

The `CompareTo` method returns

- a negative number if this object is less than `other`
- 0 if this object equals `other`
- a positive number if this object is greater than `other`

For example:

```
WriteLine(4.CompareTo(7));  // writes -1, since 4 < 7
```

`IComparable<T>` is useful because it enables us to write generic classes that compare their elements, as we will see in the next section.

generic class constraints

The generic class `FixedStack<T>` that we wrote above treats its elements as opaque values: it stores them, but does not know anything about their behavior.

Now suppose that we want to write a class `MaxStack<T>` which holds a stack of values of type T, and also provides a method `T max()` that returns the largest value currently on the stack. You might think that we could implement the class like this:

```
class MaxStack<T> {
  T[] a;
  int num;

  …

  T max() {
    T v = a[0];

    // Scan the stack, looking for the largest element.
    for (int i = 1 ; i < num ; ++i)
       if (a[i] > v)  // we found a larger element
          v = a[i];  // remember it
    return v;
  }
}
```

This code will not compile. The problem is that we can't use the > operator to compare two values of type T, because T might be some type that is not ordered – for example, T might be an array type, and arrays cannot be compared in C#.

So we will need to modify `MaxStack` so that it's only possible to create a `MaxStack<T>` if T is an ordered type. C# provides constraints for this purpose. As we saw in the previous section, any ordered type implements the `IComparable<T>` interface. So we can add a constraint to `MaxStack<T>` that says that T must implement `IComparable<T>`:

```
class MaxStack<T> where T : IComparable<T> {
  …
}
```

And now in the class we can implement the max() method like this:

```
  T max() {
    T v = a[0];

    // Scan the stack, looking for the largest element.
    for (int i = 1 ; i < num ; ++i)
       if (a[i].CompareTo(v) > 0)  // we found a larger element
          v = a[i];  // remember it
    return v;
  }
```

Notice that even if T is constrained to implement `IComparable<T>`, we still cannot use the < operator to compare two elements of type T – instead, we must call `CompareTo()`. In fact, in C# you may not ever use comparison operators to compare values of generic types, even the == operator! However, if T is known to implement `IComparable<T>` then you can test whether two values of type T are identical by calling `CompareTo()` and checking for the return value 0. It would be nice if C# allowed comparison operators in this context (and some other languages do, e.g. Kotlin), but it does not.

generic methods

So far we have used generics only to write entire classes. C# also allows us to write generic methods that take one or more type parameters, even outside a generic class. Occasionally this is useful. For example, suppose that we want to write a static method that swaps two values of any type T. We can write this as as a generic method:

```
public static void swap<T>(ref T a, ref T b) {
    T t = a;
    a = b;
    b = t;
  }
```

default

Suppose that we want to write a generic array class `ExtArray<T>` that holds elements of type T and has a special behavior: if the caller attempts to read an element that is out of bounds, then instead of an exception they will receive a default value in return. We would like this default value to be the ”natural” default for the element type; for example, an `ExtArray<int>` should return 0 if an access is out of bounds, but an `ExtArray<bool>` should return false.

C# provides an operator `default` that provides this. For any type T, default(T) returns C#’s default value for that type:

```
    WriteLine(default(int));   // writes 0

    WriteLine(default(bool));   // writes false
```

Now we can implement `ExtArray<T>` as follows:

```
class ExtArray<T> {
  T[] a;
  int count;

  …

  public T this[int index] {
    // return default value if out of bounds
    get => index < count ? a[index] : default(T);

    set => …
  }
}
```

Week 5: Notes

There was no lecture or tutorial this week.

Our topics for this week are inheritance and the C# collection classes. They are discussed in our textbook Essential C# 7.0 (Chapter 7 ”Inheritance”; Chapter 17 ”Building Custom Collections”: Primary Collection Classes) and are also covered in Programming C# 8.0 (Chapter 5 ”Collections”; Chapter 6 ”Inheritance”).

Here are some additional notes.

inheritance

C# (along with most other object-oriented languages) supports inheritance, a mechanism that allows a class to extend another class and to change its behavior. Interfaces may be inherited as well. We have already seen inheritance in Programming 1, where we briefly discussed inheritance between classes in Python. However we will emphasize inheritance more in Programming 2, for several reasons. Inheritance plays a major role in the C# collection class library and in other popular class libraries such as graphical interface toolkits. Furthermore, in Programming 2 we will start to write larger programs in which we may have our own uses for inheritance.

Let’s begin looking at inheritance in C# using an example that you may recall from our discussion of inheritance in Python. Suppose that we have a simple C# class that implements a fixed-size stack of values of type `double`:

```
class Stack {
  double[] a;
  int count;

  public Stack(int maxSize) {
    a = new double[maxSize];
  }

  public void push(double d) {
    a[count] = d;
    count += 1;
  }

  public double pop() {
    count -= 1;
    return a[count];
  }

  public bool isEmpty => count == 0;
}
```

We’d now like to write a class `AvgStack` that is like `Stack`, but has an additional method `average()` that can report the average value of the numbers that are currently on the stack. We would like `average()` to run in O(1). To achieve that, `AvgStack` will keep track of the current total of all numbers on the stack, so that it can compute the average instantly by dividing the total by the count.

We can write `AvgStack` using inheritance. Here is an implementation:

```
class AvgStack : Stack {
  double total;
  public AvgStack(int maxSize) : base(maxSize) {
  }

  public override void push(double d) {
    base.push(d);
    total += d;
  }

  public override double pop() {
    double d = base.pop();
    total -= d;
    return d;
  }

  public double average() => total / count;
}
```

Let's look at this code in some detail. The notation `class AvgStack : Stack` means that the class `AvgStack` inherits from `Stack`. (You will notice that this is just like the syntax for indicating that a class implements an interface.) In this situation, we say that `Stack` is the superclass (or parent class or base class), and `AvgStack` is a subclass (or derived class).

`AvgStack` has a public constructor that takes a single argument `maxSize`, just like the parent class. The notation `: base(maxSize)` means that the `AvgStack` constructor chains to the base class constructor – in other words, it calls the base class constructor, passing it the same value of `maxSize` that it itself received. (You will recall that we previously saw similar syntax for chaining to a different constructor in the same class. When chaining to a constructor in the same class, we write `: this()`; when chaining to a constructor in the base class, we write `: base()`.)

The `AvgStack` constructor does not need to initialize the `total` field to 0.0, since 0.0 is the default value for fields of type `double`.

`AvgStack` overrides the `push` and `pop` methods from its parent class. That means that `AvgStack` provides its own implementation of these methods. In the `push()` method, AvgStack calls `base.push(d)` to call the same-named method in the base class. It then runs `total += d` to update the running total. `pop()` is similar.

Note that if you attempt to build the `Stack` and `AvgStack` classes as written above, the code will not compile! We need to make two changes to the base class `Stack`:

1. We must modify the `push()` and `pop()` methods to be `virtual`:

```
  public virtual void push(double d) {
    a[count] = d;
    count += 1;
  }

  public virtual double pop() {
    count -= 1;
    return a[count];
  }
```

In C#, a subclass may only override base class methods if they are `virtual`. (This is a significant difference from Java, where any method may be overridden.)

2. We must modify the `count` field in the base class to be `protected`:

```
  protected int count;    // in Stack class
```

That is because the `average()` method in `AvgStack` accesses `count`:

```
  public double average() => total / count;   // in AvgStack class
```

In C#, the default access level for any member (e.g. field, method) is `private`. So if we use the simple declaration "`int count;`", then `count` will be `private`, which means that it will accessible only within its own class. A `protected` member, by constract, is accessible within its own class and within any subclasses. Finally, as we have seen before, a member marked as `public` is accessible from anywhere.

using base and derived types

Now that we have written a parent class `Stack` and subclass `AvgStack`, let's use them. We can create instances of each of these classes as follows:

```
Stack s = new Stack(100);         // create a Stack
AvgStack t = new AvgStack(100);   // create an AvgStack
```

How about the following – will this work?

```
AvgStack t = new Stack(100);    // does not compile
```

We are creating a `Stack`, and attempting to store it in a variable of type `AvgStack`. The code will not compile. If `t` has type `AvgStack`, then it cannot hold this object because a `Stack` is not an `AvgStack`. In general, an instance of a superclass (`Stack`) is not an instance of a subclass (`AvgStack`) because it doesn't have the extended capabilities that the subclass provides.

On the other hand, consider this:

```
Stack s = new AvgStack(100);   // works OK
```

This code will compile: a variable of type `Stack` can hold an `AvgStack`, because an `AvgStack` is a kind of `Stack`. In general, an instance of a subclass counts as an instance of its superclass (or of any ancestor class).

Now, however: what happens if we invoke a method through a variable whose type is the superclass? In other words, how will the following code behave?

```
Stack s = new AvgStack(100);
for (double d = 0.0 ; d < 10.0 ; d += 1.0)
  s.push(d);   // which push() will this call?
```

Which implementation of `push()` will be invoked by the code above – the implementation in `Stack`, or in the subclass `AvgStack`?

The code will call the implementation in `AvgStack`. Even though the object is held in a variable of type `Stack`, it is actually an `AvgStack`. A call to a virtual method is resolved based on an object's actual class – the type of the variable holding the object is irrelevant.

The assigment `Stack s = new AvgStack(100)` works because the type `AvgStack` is implicitly convertible to `Stack`. Because of this phenomenon, we may write a method that works on `Stack` objects and may pass it instances of the `AvgStack` subclass. For example:

```
static void empty(Stack s) {
  while (!s.isEmpty)
    Console.WriteLine(s.pop());
}
```

We may pass an `AvgStack` to this method:

```
AvgStack s = new AvgStack(100);
for (double d = 0.0 ; d < 10.0 ; d += 1.0)
  s.push(d);
empty(s);
```

We have seen that C# will implicitly convert from a subclass to a superclass type. In the other direction, C# provides an explicit conversion that we can access via a type cast. For example:

```
Stack s = new AvgStack(100);
AvgStack t = (AvgStack) s;    // explicit conversion
```

We can use an explicit conversion to a subclass type when we are sure that a variable holds a object of that subclass. On the other hand, if the object does not actually belong to the subclass then the program will exit with an exception:

```
Stack s = new Stack(100);
AvgStack t = (AvgStack) s;  // compiles, but will throw an exception when run
```

abstract classes and methods

Suppose that we'd like to write generic classes that implement several kinds of collections:

- `Stack<T>`, a LIFO stack
- `Queue<T>`, a FIFO queue
- `PriorityQueue<T>`, a priority queue

Furthemore we would like to use these classes somewhat interchangeably, so we'd like them all to provide a common set of members:

- a constructor that creates an empty collection
- a constructor that takes an array of type `T[]` and uses it to initialize the collection
- a method void `add(T val)`
- a method `T remove()`
- a property `count` that returns the current number of items in the collection
- a property `isEmpty` that is true if the collection is empty

The `add()` and `remove()` methods will work differently in the different classes. For example, in a `Queue<T>` the `remove()` method will remove the element that was added first, whereas in a `PriorityQueue<T>` it will remove the smallest value.

We could write all these classes independently. But since they share some common functionality, it might be nicer to use inheritance so that we can write the common code only once. However, none of these classes can reasonably inherit from each other. (Perhaps a priority queue sounds like a particular kind of queue, but the data structure for implementing a priority queue (e.g. a binary heap) will be completely different from the data structure for implementing a FIFO queue (e.g. a linked list), and so `PriorityQueue<T>` should really not inherit from `Queue<T>`.)

Instead, a better idea is to write a top-level class `Collection` that can serve as a superclass for all of these classes. Then we can implement the common functionality in the `Collection` class. Of course, it would make no sense to create an object of class `Collection`; the only purpose of `Collection` is for other classes to derive from this. So we will mark `Collection` as an abstract class. An abstract class cannot be instantiated – that is to say, you cannot create an instance of it. Furthermore, an abstract class can have abstract methods, which have no implementation: a concrete derived class must provide an implementation of these methods.

Here is the base class `Collection<T>`:

```
abstract class Collection<T> {
  protected int _count;

  protected Collection() { }

  protected Collection(T[] a) {
    foreach (T val in a)
      add(val);
  }

  public int count => _count;

  public bool isEmpty => _count == 0;

  protected abstract void _add(T val);
  protected abstract T _remove();

  public void add(T a) {
    _add(a);
    _count += 1;
  }

  public T remove() {
    T val = _remove();
    _count -= 1;
    return val;
  }
}
```

The field `_count` holds the number of elements in a collection. Its name is preceded with an underscore to distinguish it from the public read-only `count` property. The class has two constructors: one creates an empty `Collection` and another initializes a `Collection` with elements from an array. The constructors are protected, not public, since they will only be called by subclasses.

The public methods `add()` and `remove()` call the abstract methods `_add()` and `_remove()` to perform the actual work of adding and removing elements from a collection. Notice that these methods have no implementation; they will be implemented by each subclass.

Here is a subclass `Queue<T>` that implements a FIFO queue using a linked list:

```
class Node<T> {
  public T val;
  public Node<T> next;

  public Node(T val) { this.val = val; }
}

class Queue<T> : Collection<T> {
  Node<T> head, tail;

  public Queue() { }

  public Queue(T[] a) : base(a) { }

  protected override void _add(T val) {
    if (head == null)
      head = tail = new Node<T>(val);
    else {
      tail.next = new Node<T>(val);
      tail = tail.next;
    }
  }

  protected override T _remove() {
    T val = head.val;
    head = head.next;
    return val;
  }
}
```

Notice that we must write `override` when implementing an abstract method.

I will not provide implementations of the other subclasses. Hopefully it is clear enough, however, that the functionality in the top-level `Collection<T>` class could reasonably be shared by all subclasses.

You may notice some similarities between abstract classes and interfaces. In particular, they can each specify methods which need to be implemented by concrete classes. There are some differences, however: abstract classes may contain fields and constructors, and interfaces may not. An interface is something like an abstract class that only declares a set of abstract methods.

Object class

All C# class automatically inherit from a top-level class called `Object` which contains a modest number of methods that are shared by all objects. One of these methods is `ToString`:

```
virtual string ToString ();
```

C# automatically calls this method when it needs to convert any object to a string, for example when writing an object to the console:

```
Window w = new Window();

Console.WriteLine(w);   // will automatically call w.ToString()
```

The default implementation of ToString simply returns the name of the class, so the above code will write

```
Window
```

You can override `ToString()` in your own classes if you'd like to change their printed representation. (`ToString()` is like the `__repr` magic method in Python, which you may remember.)

Object contains a couple of other methods (`Equals`, `GetHashCode`) that we will consider when we discuss the C# collection classes below.

interface inheritance

Up to now we have only discussed inheritance between classes. C# also allows an interface to inherit from another interface. An inheriting interface has all the members of its parent interface, and can add additional members of its own. Here is a somewhat abstract example:

```
interface Shape {
  double perimeter();
  double area();
}

interface Polygon : Shape {
  int num_vertices { get; }
}

class Circle : Shape { … }

class Rectangle : Polygon { … }

class Triangle : Polygon { … }
```

In this example, a `Shape` represents a shape in two dimensions and a `Polygon` is a `Shape` that has a finite number of vertices. Because the `Triangle` class implements `Polygon`, it must provide an implementation of `num_vertices` as well as of the `perimeter()` and `area()` methods.

single and multiple inheritance

C# provides

- single class inheritance: a class may inherit from only one other class
- multiple interface inheritance: an interface may inherit from multiple interfaces

This particular combination was popularized by Java. Many object-oriented languages (including Python) have a more general mechanism: they have multiple class inheritance, so that one class may have several immediate superclasses. That is more powerful, but creates various complexities that C# avoids through its simpler model.

The C# collection classes

LIke many object-oriented languages, C# contains a set of collection classes in its standard library. These classes implement common data structures such as stacks, queues and hash tables. They are generic and make use of inheritance.

Here is a picture of the class hierarchy of some of the major collection classes and interfaces:

%3

All of these classes and interfaces live in the namespace `System.Collections.Generic.`

In the picture, an arrow from A to B means that B implements or inherits from A.

All entities above whose names begin with the letter I (e..g `ICollection`, `IDictionary`) are interfaces. This is a standard naming convention in the C# standard library.

The picture above includes the following classes:

- `List<T>` is a dynamic array, similar to a list in Python. List<T> contains an `add()` method for appending an element to the end of a `List`. It also contains an indexer so that you can get or set elements by index.
- `Stack<T>` implements a stack using a dynamic array. It includes methods `Push()` and `Pop()`.
- `Queue<T>` implements a queue using a dynamic array, and includes methods `Enqueue()` and `Dequeue()`.

- The classes `HashSet<T>` and `SortedSet<T>` implement sets, with methods `Add()`, `Contains()` and `Remove()`. They differ in their underlying data structure: `HashSet<>` uses a hash table and `SortedSet<T>` uses a balanced binary tree.
- Finally, the classes `Dictionary<K, V>` and `SortedDictionary<K, V>` implement dictionaries, with an indexer that can get/set values as well as a `ContainsKey` method. They differ in their underlying data structure: `Dictionary<K, V>` uses a hash table and `SortedDictionary<K, V>` uses a balanced binary tree.

You can read more details about these classes and interfaces in our Quick Library Reference. Note that you will need to follow the class hierarchy in order to find all the members available in each of these classes. For example, the methods available in `List<T>` include not only the ones listed under `List<T>` in the library reference, but also those listed under `IList<T>` and `ICollection<T>`, since `List<T>` implements those interfaces.

You can read even more details about the collection classes (including various classes not depicted above) in the official .NET API documentation.

Notice that all collection classes ultimately inherit from an interface `IEnumerable<T>`. This interface contains various methods that allow the `foreach` statement to iterate over a collection's values. We will not discuss these methods at this point, but you should certainly be aware that you can use `foreach` to iterate over any collection.

You will want to become fluent in using these collection classes. With them, you finally have easy access to the same useful data structures that we often used in Python, i.e. lists, sets and dictionaries.

Dictionaries in particular are quite useful, so let's look at the `Dictionary<K, V>` class a bit more. As you can see in the documentation, `Dictionary<K, V>` implements the interface `IDictionary<K, V>`, which in turn implements the interface `ICollection<KeyValuePair<K, V>>`. This means that a dictionary is a collection of key-value pairs. You can use `foreach` to iterate over a dictionary, and each element you get back will be a `KeyValuePair<K, V>`. Each KeyValuePair in turn has properties `Key` and `Value`. For example:

```
Dictionary<string, int> d = new Dictionary<string, int>();
d["yellow"] = 10;
d["red"] = 15;
d["blue"] = 20;

foreach (KeyValuePair<string, int> p in d)
    WriteLine($"key = {p.Key}, value = {p.Value}");
```

overriding Equals and GetHashCode

Suppose that we have a class `Vector` representing a vector in an arbitrary number of dimensions:

```
class Vector {
  double[] a;

  public Vector(params double[] a) { this.a = a; }
}
```

We could certainly add more methods to obtain the length of a vector, add two vectors, and so on, but those are not important for our discussion here.

Let's create a `HashSet<Vector>` and add a couple of vectors to it:

```
HashSet<Vector> s = new HashSet<Vector>();
s.Add(new Vector(1.0, 2.0, 3.0));
```

```
s.Add(new Vector(3.0, 4.0, 5.0));
```

OK – so far so good. Now let's test whether the vector (1.0 2.0 3.0) is in the set:

```
WriteLine(s.Contains(new Vector(1.0, 2.0, 3.0)));
==> False
```

`Contains` has reported that the set does not contain that vector! The problem is that the `HashSet` does not consider the first vector we added to be equal to the vector we passed to `Contains()`, despite the fact that they contain the same numbers. So how can we tell `HashSet` that they should be equal?

As we briefly mentioned above, the top-level `Object` class contains a method `Equals`:

```
virtual bool Equals (object obj);  // return true if this object equals (obj)
```

`HashSet` calls this method to determine whether two objects are equal. By default, `Equals` just tests for reference equality, i.e. it returns true only if two objects are actually one and the same object. But we can override `Equals` in our class to specify a different notion of equality:

```
public override bool Equals(object o) {
    if (!(o is Vector))
      return false;

    Vector w = (Vector) o;

    if (a.Length != w.a.Length)
      return false;

    for (int i = 0 ; i < a.Length ; ++i)
      if (a[i] != w.a[i])
        return false;

    return true;
  }
```

Our comparison method first returns false if `o` is not actually a `Vector`. (This is a sneak preview of the `is` operator, which will we will study next week). Assuming that it is, we use a type cast to convert to a `Vector`, and then compare the vectors element by element.

Now the `HashSet` class should consider any two vectors with the same components to be identical. However overriding `Equals` is not sufficient. In fact, it we build the code above then the compiler warns us that we have another task to complete:

```
prog.cs(4,7): warning CS0659: 'Vector' overrides Object.Equals(object o) but does not override
```

As the compiler has suggested, we should also override the `GetHashCode()` method, which also belongs to the top-level `Object` class, and has this signature:

```
virtual int GetHashCode ();
```

Why must we override `GetHashCode()` whenever we override `Equals()`? The default implementation of `GetHashCode()` computes hash codes that might be different for any two different objects. But if two objects are considered equal by `Equals()`, such as two different `Vector` objects that have identical components, then they must have the same hash code so that a hash table implementation (such as `HashSet`) will look for them in the same hash bucket. This should make sense to you based on our study of hash tables in Intro to Algorithms. (And in fact you may remember that we encountered this same phenomenon in Python, where we needed to implement a magic method __hash() if we implemented a custom equality operator for a class.)

So let's expand our `Equals` class with a `GetHashCode()` method. For an aggregate object such as a `Vector`, the easiest approach is to combine the hash codes of the underlying values in some way. For example:

```
  public override int GetHashCode() {
    long hash = 0;

    foreach (double d in a)
      hash = hash * 1000003 + d.GetHashCode();
    return (int) hash;
  }
```

(You might recognize the prime constant 1,000,003 above from our discussion of hash functions in Introduction to Algorithms last semester.)

Now there are no compiler warnings, and looking up a `Vector` in a `HashSet` has the expected result:

```
HashSet<Vector> s = new HashSet<Vector>();
s.Add(new Vector(1.0, 2.0, 3.0));
s.Add(new Vector(3.0, 4.0, 5.0));

WriteLine(s.Contains(new Vector(1.0, 2.0, 3.0)));

==> True
```

Week 6: Notes

There was no lecture or tutorial this week.

Our topics this week are the `is`/`as` operators, exception handling, and nested classes. You can read about them in Essential C# 7.0:

- Chapter 6 ”Classes”: Nested Classes
- Chapter 7 ”Inheritance”: Verifying the Underlying Type with the is Operator, Pattern Matching with the is Operator, Conversion Using the as Operator
- Chapter 11 ”Exception Handling”

They are also covered in Programming C# 8.0:

- Chapter 3 ”Types”: Nested Types
- Chapter 6 ”Inheritance”: Inheritance and Conversions
- Chapter 8 ”Exceptions”

Here are a few notes briefly summarizing these topics.

is

The `is` operator returns true if a value is not null and belongs to a given type. It works with both reference types and nullable types:

```
class LinkedStack : Stack { … }
…

Stack s = getStack();
if (s is LinkedStack)
  WriteLine("it is a LinkedStack");
int? i = abc();
if (i is int)  // true if i != null
  WriteLine(i.Value);
```

The `is` operator can optionally bind a variable. The examples above can be rewritten as follows:

```
Stack s = getStack();
if (s is LinkedStack ls)
    WriteLine("it is a LinkedStack");

if (abc() is int i)
    WriteLine(i);
```

This variable binding works not only in `if` statements, but also in `while` loops. This is often convenient when we want to loop as long as a function returns a non-null value. For example, instead of

```
while (true) {
    s = ReadLine();
    if (s == null)
        break;
    WriteLine(s);
}
```

we can use `is`:

```
    while (ReadLine() is string s)
        WriteLine(s);
```

In this loop, when `ReadLine()` returns a non-null value, the string variable `s` receives that value and the loop continues. When `ReadLine()` returns null, the loop terminates.

as

The `as` operator checks whether a value belongs to a type. If so, it returns the value; otherwise it returns null:

```
    Stack s = getStack();
    LinkedStack ls = s as LinkedStack;  // if s was not a LinkedStack, ls will be null
```

throw

The `throw` statement throws an exception, which can be any object belonging to the `System.Exception` class or any of its subclasses. The exception will pass up the call stack, aborting the execution of any methods in progress until it is caught with a `try...catch` block at some point. If the exception is not caught, the program will terminate.

try/catch/finally

The `try` statement attempts to execute a block of code. It may have a set of `catch` clauses and/or a `finally` clause.

A `catch` clause catches all exceptions of a certain type. For example:

```
  static void Main() {
    StreamReader reader;
    try {
      reader = new StreamReader("numbers");
    } catch (FileNotFoundException e) {
      WriteLine("can't find input file: " + e.FileName);
      return;
    }
    …
```

The code above will catch an exception of class `FileNotFoundException`, or of any subclass of it. When an exception is caught, the `catch` block (called an exception handler) executes. The catch block may itself rethrow the given exception, or even a different exception. If the catch block does not throw an exception, execution resumes below the `try` statement.

A `finally` clause will always execute, even if an exception is thrown inside the body of the `try` statement. For example:

```
  StreamReader reader = …;
  StreamWriter writer = …;
  try {
    while (reader.ReadLine() is string s)
      writer.WriteLine(transform(s));
  } finally {
    reader.Close();
    writer.Close();
  }
```

In this close, `reader` and `writer` will be closed even if an exception occurs within the `try` body (for example, within the `transform` method). Note that a `finally` clause does not itself catch an exception, which will continue to pass up the call stack.

nested classes

In C# a class may be nested inside another class. You may reasonably want to use a nested class when you write a helper class that is useful only inside another class. For example:

```
class LinkedList {
  class Node {
    public int i;
    public Node next;
    public Node(int i, Node next) { this.i = i; this.next = next; }
  }

  Node head;

  public void prepend(int i) {
    head = new Node(i, head);
  }
}
```

The `Node` class is nested inside `LinkedList`. It is visible inside that class, but since the `Node` class is not explicitly marked as public, it is not accessible from outside `LinkedList`.

A class that is nested inside a generic class may use all the type variables of the containing class. This can be quite convenient. For example, let's make the previous example generic:

```
class LinkedList<T> {
  class Node {
    public T val;
    public Node next;
    public Node(T val, Node next) { this.val = val; this.next = next; }
  }

  Node head;

  public void prepend(T val) {
    head = new Node(val, head);
  }
}
```

Notice that we do not need to declare `Node` as a generic class `Node<T>`. If `Node` were not nested, we would have to declare it as generic since it would be outside the scope of the type variable T. And then inside the `LinkedList` class we would have to write `Node<T>` whenever we referred to that class.

Week 7: Notes

There was no lecture or tutorial this week.

Our topics this week are local functions, delegates, lambda expressions, and extension methods. You can read about them in Essential C# 7.0:

- Chapter 6 ”Classes”: Extension Methods
- Chapter 13 ”Delegates and Lambda Expressions”

They are also covered in Programming C# 8.0:

- Chapter 3 ”Types”: Members / Methods / Extension methods
- Chapter 9 ”Delegates, Lambdas, and Events”: Delegate Types, Anonymous Functions

Here are a few notes briefly summarizing these topics.

local functions

A local function is a function that is defined inside a method. (Sometimes local functions are called ”nested methods”, but strictly speaking they are functions, not methods, since they are not invoked on an object.) For example:

```
static void xyz(string[] args) {
  WriteLine("hello");

  double arg(int n) {
      return double.Parse(args[n]);
  }

  double d = arg(0);
  double e = arg(1);
  WriteLine(d + e);
}
```

A local function may access parameters and local variables in its containing method. For example, the local function `arg` above accesses the `args` parameter.

delegates

A delegate is a value that represents a function or method. It is similar to to a function object in Python, or a function pointer in languages such as C.

The `delegate` keyword declares a new delegate type. For example:

```
delegate bool IntCondition(int i);
```

With this declaration, an `IntCondition` is a type of delegate that takes an integer argument and returns a boolean. We can now declare a variable of type `IntCondition`, and use it to refer to a method of corresponding type:

```
static bool isOdd(int i) {
  return (i % 2 == 1);
}

static void Main() {
  IntCondition c = isOdd;
  …
```

We can invoke the delegate using function call syntax:

```
    WriteLine(c(4));    //  writes False
```

In the example above, the delegate c refers to a static method odd. A delegate may also refer to an instance method, in which case it actually references a particular object on which the method will be invoked. For example:

```
  class Interval {
    public int low, high;
    public Interval(int low, int high) { this.low = low; this.high = high; }

    public bool contains(int i) {
      return (low <= i && i <= high);
    }
  }

  static void Main() {
    IntCondition c = new Interval(1, 5).contains;
    IntCondition d = new Interval(3, 7).contains;
    WriteLine(c(2));  // writes True
    WriteLine(d(2));  // writes False
  }
```

Here is a method that counts how many elements in an array of integers satisfy an arbitrary condition:

```
  static int count(int[] a, IntCondition cond) {
    int n = 0;
    foreach (int i in a)
      if (cond(i))
        ++n;
    return n;
  }
```

We can invoke this method as follows:

```
  static bool isEven(int i) {
    return (i % 2 == 0);
  }

  int[] a = { 3, 4, 5, 6, 7 };
  WriteLine(count(a, isEven));  // writes 2
```

Delegates may be generic:

```
  delegate bool Condition<T>(T t);  // maps type T to bool
```

Here is the count method from above, rewritten to work on an array of any type T. Notice that the method itself must also be generic (indicated by the "<T>" after the method name).

```
  static int count<T>(T[] a, Condition<T> cond) {
    int n = 0;
    foreach (T val in a)
      if (cond(val))
        ++n;
    return n;
  }
```

lambda expressions

A lambda expression is an anonymous function that can appear inside another expression.

For example, here is a delegate type for a function from integers to integers:

```
delegate int IntFun(int i);
```

And here is a method that applies a given function to every element of an array of integers:

```
  static int[] map(int[] a, IntFun f) {
    int[] b = new int[a.Length];
    for (int i = 0 ; i < a.Length ; ++i)
      b[i] = f(a[i]);
    return b;
  }
```

We can define a named method and pass it to `map`:

```
  static int plus2(int i) {
    return i + 2;
  }

  static int[] add2(int[] a) {
    return map(a, plus2);
  }
```

Alternatively, we can invoke `map` using a lambda expression:

```
  static int[] add2(int[] a) {
    return map(a, i => i + 2);
  }
```

Here, `i => i + 2` is a lambda expression. It is an anonymous function that takes an integer parameter i and returns the value i + 2.

Like a local function, a lambda expression may refer to parameters or local variables in its containing method. For example, suppose that we want to write a method that adds a given value k to each element in an array. We could write a local function and pass it to `map`:

```
  static int[] add_k(int[] a, int k) {
    int f(int i) {
      return i + k;
    }
    return map(a, f);
  }
```

Or we can use a lambda expression that adds k directly:

```
  static int[] add_k(int[] a, int k) {
    return map(a, i => i + k);
  }
```

The lambda expressions in the examples above are expression lambdas, writen using a compact syntax similar to the expression syntax for methods. Alternatively, a lambda expression can be written as a statement lambda, which can include one or more statements and can use the `return` statement to return a value. For example, we can rewrite the last example above like this:

```
  static int[] add_k(int[] a, int k) {
    return map(a, i => { return i + k; } );
  }
```

functions as return values

In the examples above we've seen that a method can take a delegate (i.e. a function) as an argument. A method can also return a delegate constructed using a lambda expression.

Here is a simple example:

```
delegate bool IntCondition(int i);
static IntCondition divisibleBy(int k) {
  return i => (i % k == 0);
}
```

Now we can invoke this method and use the delegate that it returns:

```
IntCondition div3 = divisibleBy(3);
WriteLine(div3(6));  // writes 'True'
WriteLine(div3(7));  // writes 'False'
```

In this example, note that the delegate returned by `divisibleBy` can refer to the parameter k even after the method `divisibleBy` has returned! To put it differently, the lambda expression `i => (i % k == 0)` has captured the parameter k. Local variables may also be captured by a lambda expression.

Using lambda expressions we can write functions that transform other functions. This is a powerful technique that you will explore further in more advanced courses (e.g. Non-Procedural Programming). Here are a couple of examples. First, here is a function that composes two functions f and g, returning the function $(f \circ g)$, which is defined as $(f \circ g)(x) = f(g(x))$:

```
static IntFun compose(IntFun f, IntFun g) {
  return i => f(g(i));
}
```

We can call `compose` as follows:

```
IntFun f = compose(i => i * 2, i => i + 1);  // now f(x) = 2 * (x + 1)
WriteLine(f(4));   // writes 10
```

Second, here's a function that computes the nth power of a function f, defined as

```
static IntFun power(IntFun f, int n) {
  return i => {
    for (int j = 0 ; j < n ; ++j)
      i = f(i);
    return i;
  };
}
```

Addition to a power is multiplication:

```
IntFun f = power(i => i + 10, 4);
WriteLine(f(2));  // writes 42
```

extension methods

You can add extension methods to an existing class. An extension method can be used syntactically as if belonged to a class, even though it is written outside the class.

For example, suppose that we are using a C# library that provides this type:

```
class Vector {
  public double dx, dy;
  public Vector(double dx, double dy) { this.dx = dx; this.dy = dy; }
}
```

We wish that the author of the class had provided a `length` method that calculates the length of a `Vector`. Since we cannot modify the class, we can write an extension method:

```
static class Util {
  public static double length(this Vector v) =>
      Sqrt(v.dx * v.dx + v.dy * v.dy);
}
```

The keyword `this` before the argument ”`Vector v`” indicates that this is an extension method for the `Vector` class.

Now we can call the method as if it had been defined inside the `Vector` class itself!

```
Vector v = new Vector(3.0, 4.0);
WriteLine(v.length());  // writes 5.0
```

Note that an extension method must be `static`. The containing class can have any name, but must itself also be declared as `static`.

Week 8: Notes

There was no lecture or tutorial this week.

Our C# language topics this week are the using statement and events. In addition, this week we will begin to learn to write graphical interfaces in C# using the Windows Forms or GTK toolkits.

You can read about our C# topics in Essential C# 7.0:

- Chapter 10 ”Well-Formed Types”: Resource Cleanup
- Chapter 14 ”Events”

They are also covered in Programming C# 8.0:

- Chapter 7 ”Object Lifetime”: IDisposable
- Chapter 9 ”Delegates, Lambdas, and Events”: Events

You will need to choose one graphical interface toolkit to learn: either Windows Forms or GTK. You certainly do not need to learn both. Your choice will probably depend on the operating system you are running:

- On Linux, GTK is the native toolkit (i.e. many default applications in popular distributions such as Ubuntu are written using GTK). However, Windows Forms under Mono also works well. If you are unsure, I recommend learning Windows Forms since its API is a bit easier to use.
- On macOS, Windows Forms will not run, so you must learn GTK.
- On Windows, either library can run, but it can sometimes be tricky to get GTK to work correctly. I recommend Windows Forms, which is native to Windows and works well there.

Here are some more notes on these topics.

using

Certain classes in the standard C# library implement the interface `IDisposable`, which has a single method:

```
void Dispose ();
```

This method frees any external resources associated with an object. You should call it when you are finished using an object.

Fort example, the `StreamReader` and `StreamWriter` classes implement `IDisposable`. The Dispose() method in these classes performs the same task as the Close() method: it closes a file or other stream. It is especially important to call Close() or Dispose() when writing to a file - if you do not, some output may not be written! So to be sure that the file is closed in any case, you can write

```
StreamWriter w = new StreamWriter("output");
try {
  … write to w …
} finally {
  w.Dispose();
}
```

This pattern is so common that C# includes a statement called `using` that can be used for the same task. `using` takes a object that implements `IDisposable` and executes a block of code, disposing the object when the block exits:

```
StreamWriter w = new StreamWriter("output");
using (w) {
  … write to w …
}
```

Alternatively, you can bind a new variable inside a `using` statement:

```
using (StreamWriter w = new StreamWriter("output")) {
  … write to w …
}
```

events

An event is a class member that lets callers register event handlers that will receive notifications. Each event handler is a delegate. When an event is raised (i.e. fires), a notification is sent to each registered event handler. Each notification includes arguments matching the event's delegate type.

Events are useful for implementing the observer pattern, in which one or more observers may want to hear about changes to an object. A common example of this pattern is a model-view architecture, in which the view observes the model and displays the model's data. In such an architecture we want the model to be unaware of the view. Using an event, a view can register to find out when the model has changed, without giving the model specific knowledge of the view class.

Here is an array class including an event that is raised whenever any array element changes:

```
delegate void Notify(int index, int old, int now);

class WatchableArray {
  int[] a;

  public WatchableArray(int n) {
    a = new int[n];
  }

  public event Notify changed;

  public int this[int i] {
    get => a[i];
    set {
      int prev = a[i];
      a[i] = value;
      changed(i, prev, a[i]);  // fire the event to notify observers
    }
  }
}
```

Notice that the event declaration includes a delegate type, and that we can raise an event using method call syntax.

Use the += operator to register a handler with an event. For example, we can create an instance of the `WatchableArray` class and register an event handler:

```
  void onChange(int index, int old, int now) {
    WriteLine($"a[{index}] changed from {old} to {now}");
  }
  public void Main() {
    WatchableArray a = new WatchableArray(5);
    a.changed += onChange;
    …

  }
```

If some method later calls

```
  a[3] = 4;
```

then the above event handler will run, and will print a message such as

```
  a[3] changed from 0 to 4
```

events with no handlers

Be warned: if you attempt to raise an event that has no registered handlers, you will get a `NullPointerException`. In my opinion this is a weakness in the C# event system: if would certainly be nicer if raising such an event did nothing. However, this is how it works. So in the example above, instead of

```
changed(i, prev, a[i]);  // fire the event to notify observers
```

really we should write

```
if (changed != null)
    changed(i, prev, a[i]);  // fire the event to notify observers
```

which will guard against this condition.

It's worth noting that C# has a null-conditional operator `?.` that can make this slightly easier. This operator invokes a method if the value is non-null. If the value is null, the operator returns null. For example:

```
    string s = abc();
```

```
string t = s?.Trim();
```

We can use the null-conditional operator when firing an event, though we must add the special method name `Invoke` to make it work with events:

```
changed?.Invoke(i, prev, a[i]);  // fire the event to notify observers
```

This syntax is more compact than the if statement above, thoughs arguably harder to read. You may use whichever syntax you prefer.

graphical interfaces

Last semester we learned how to write graphical programs in Python using the pygame library. In those programs we had to draw all the graphical elements we wanted, since that's how pygame works. But real-world graphical programs contain many common user interface elements: push buttons, check boxes, scrollbars, dialog boxes and so on. In this course we will learn to write programs in C# that contain these kinds of elements, which are typically called widgets in the Unix programming world and controls in the Windows world.

There are many user interface toolkits that let you write graphical programs using a variety of widgets/controls. Some of these toolkits are specific to one particular operating system, and others are cross-platform. Also, some of them are specific to a particular programming language, and others have bindings to many different languages. As mentioned above, in this course we will study Windows Forms or GTK, and your choice of toolkit will probably depend on which operating system you run.

Windows Forms

Windows Forms is a popular toolkit for writing graphical programs in C#. You can learn about it by reading the documentation on Microsoft's web site. The same site also includes APi reference documentation for the System.Drawing and System.Windows.Forms namespaces that contain the various Windows Forms classes.

Windows Forms is enormous: it contains dozens of controls that you can use to build an application, and some of them are complex. We can certainly not study all of these in our course. I have written a quick reference page that lists the most important classes and methods that we will be using, and I recommend using this quick reference page rather than the full documentation whenever possible. (You can also find a link to it from our course's home page.)

Microsoft's Visual Studio IDE includes a tool called the Windows Forms Designer that let you build forms visually by dragging and dropping their elements. In this course we will not study this tool: we will construct graphical interfaces manually in code. (However if you want to learn about this Visual Studio capability on your own, you are welcome to use it for your work in this class.)

Hello, Forms

Here is a "hello, world" program in Windows Forms:

```
using System.Windows.Forms;

class Hello {
  [STAThread]
  static void Main() {
    Form form = new Form();
    form.Text = "hello, world";

    Application.Run(form);
  }
}
```

The program displays a blank window with title "hello, world".

In Windows Forms, a form means a window, so the call to `new Form()` creates a new top-level window.

Make sure that you can build and run this program on your computer. Here are some tips:

- On Linux, you cannot build a Windows Forms application using .NET Core. You must use Mono.
- On Windows, if you are using .NET Core, create your project using the command "`dotnet new winforms`". ("`dotnet new console`" will not work.) After you do this, you may wish to edit the .csproj file and change the <OutputType> setting from "WinExe" to "Exe". That will allow Console.Write() to produce output, which is useful for debugging.

The attribute `[STAThread]` above Main() is related to synchronization and multithreading, which are subjects we might discuss later. If you are using Mono on Linux, the attribute is unnecessary. If you are running Windows, be sure to include the `[STAThread]` attribute before Main() since without it some operations may hang (e.g. running an open file dialog).

drawing graphics

Like most application toolkits, Windows Forms is an event-driven system. Typically in your Main() method you will call the Application.Run() method, which runs the main event loop. Then, when actions occur such as the user clicking a mouse button, the system fires an event that your code can respond to.

There are two ways to enable your code to run in response to a Windows Forms event. One way is to add a handler to a C# event object. For example, `Form` is a subclass of `Control`, and `Control` has a event called `MouseDown`. So if you write an event handler method such as

```
void mouseHandler(object sender, MouseEventArgs e) { … }
```

then you can register it to run in response to the event, like this:

```
form.MouseDown += mouseHandler;
```

However it is often more convenient to write a subclass of a Form or other type of control. In addition to C# events, each Windows Forms class also contains virtual methods that will run automatically in response to each event type. You can override these methods to add code that will run in response to events, and that is a little easier than hooking up event handlers using +=.

Here is a Windows Forms program that draws a circle. Each time the user clicks the window, the circle toggles on and off:

```
using System;
using System.Drawing;
using System.Windows.Forms;

class MyForm : Form {
  bool draw = true;

  public MyForm() {
    Text = "circle";
    ClientSize = new Size(500, 500);
    StartPosition = FormStartPosition.CenterScreen;
  }

  protected override void OnMouseDown(MouseEventArgs args) {
    draw = !draw;
    Invalidate();
  }

  protected override void OnPaint(PaintEventArgs args) {
    if (draw) {
      Graphics g = args.Graphics;
      g.FillEllipse(Brushes.Olive, 100, 100, 300, 300);
    }
  }
}

class Hello {
  [STAThread]
  static void Main() {
    Form form = new MyForm();

    Application.Run(form);
  }
}
```

When the user clicks the mouse, the OnMouseDown() handler toggles the boolean field 'draw' and then calls the important method Invalidate(). This method tells the system that it is time to redraw the window's contents, and causes the system to call OnPaint() in the near future. This is how a Windows Forms program typically works. Do not attempt to paint anywhere outside the OnPaint() method – instead, whenever the underlying data (i.e. model) changes, call Invalidate() to cause OnPaint() to redraw the view.

In OnPaint(), the program retrieves a `Graphics` object from the `args` parameter. It then calls FillEllipse() to draw a circle. In the quick reference documentation you will find other methods you can use for drawing.

menus

Like most graphical toolkits, Windows Forms supports menus. Each menu contains a set of commands that the user can select, and application-specific code runs in response to each command.

The `ToolStripMenuItem` class represents a menu item. Each `ToolStripMenuItem` holds a instance of the `System.EventHandler` delegate, which is defined as follows:

delegate void EventHandler (object sender, EventArgs e);

> So when you write a method that will run in response to a menu item, it needs to take these two parameters of type `object` and `EventArgs`. (Usually you will just ignore the arguments that are passed.)

Here is a program with a menu that lets the user select a shape to be displayed:

```
using System;
using System.Drawing;
using System.Windows.Forms;

using static Shape;
enum Shape { Square, Circle, Triangle };

class MyForm : Form {
  Shape shape = Circle;

  public MyForm() {
    ClientSize = new Size(500, 500);
    StartPosition = FormStartPosition.CenterScreen;

    ToolStripMenuItem[] fileItems = {
        new ToolStripMenuItem("Square", null, onSquare),
        new ToolStripMenuItem("Circle", null, onCircle),
        new ToolStripMenuItem("Triangle", null, onTriangle),
        new ToolStripMenuItem("Quit", null, onQuit)
    };

    ToolStripMenuItem[] topItems = {
        new ToolStripMenuItem("File", null, fileItems)
    };
    MenuStrip strip = new MenuStrip();
    foreach (var item in topItems)
      strip.Items.Add(item);

    Controls.Add(strip);
  }

  void onSquare(object sender, EventArgs e) {
    shape = Square; Invalidate();
  }

  void onCircle(object sender, EventArgs e) {
    shape = Circle; Invalidate();
  }

  void onTriangle(object sender, EventArgs e) {
    shape = Triangle; Invalidate();
  }

  void onQuit(object sender, EventArgs e) {
    Application.Exit();
  }

  protected override void OnPaint(PaintEventArgs args) {
    Graphics g = args.Graphics;
    switch (shape) {
        case Square:
            g.FillRectangle(Brushes.Blue, 100, 100, 300, 300); break;
        case Circle:
            g.FillEllipse(Brushes.Blue, 100, 100, 300, 300); break;
        case Triangle:
            g.FillPolygon(Brushes.Blue, new Point[] {
              new Point(250, 100), new Point(100, 400), new Point(400, 400)
            }); break;
    }
  }
}

class Hello {
  [STAThread]
  static void Main() {
    Form form = new MyForm();

    Application.Run(form);
  }
}
```

Exercise: The methods onSquare(), onCircle() and onTriangle() above are all very similar. Figure out how to eliminate this code duplication.

GTK

First: Most graphical interface toolkits are fairly similar, at least at a high level. And so this introduction to GTK is quite similar to the Windows Forms introduction above – in fact I have copied much of the text, but with the code transformed to GTK!

GTK is a popular toolkit in the Linux world, where it forms the foundation of many desktop applications. For example, the popular Ubuntu distribution includes the GNOME desktop, all of whose applications are written using GTK. GTK was originally a Linux toolkit, but it has been ported to macOS and Windows and is sometimes used on those platforms as well.

GTK can be called from many languages, and the C# binding to GTK is called GTK#. The current version of GTK is GTK 3 but we will use Gtk# 2 since it is well supported by the MonoDevelop IDE and by Visual Studio for Mac. (Gtk# 3 does exist, but it is a bit tricky to get it working, especially on macOS.)

Here is a page that describes how to configure your system to build programs with GTK#. As you can see there, the easiest way is to create a Gtk# project using MonoDevelop or Visual Studio for Mac. If you want to use Visual Studio Code, I recommend that you first create your project using MonoDevelop or Visual Studio, and then you can edit it using Visual Studio Code. In this situation you will want to build your project from the command line using `msbuild`, not `dotnet build`.

You can learn about Gtk# by reading the documentation on the Mono project site. There are some tutorials there, though their scope is pretty limited. The Mono site also includes extensive API reference documentation for GTK and many other related libraries.

GTK is enormous: it contains dozens of controls that you can use to build an application, and some of them are complex. We can certainly not study all of these in our course. I have written a quick reference page that lists the most important classes and methods that we will be using, and I recommend using this quick reference page rather than the full documentation whenever possible. (You can also find a link to it from our course's home page.)

MonoDevelop includes a visual tool called Stetic that let you build Gtk# interfaces visually by dragging and dropping their elements. In this course we will not study this tool: we will construct graphical interfaces manually in code. (However if you want to learn about this MonoDevelop feature on your own, you are welcome to use it for your work in this class.)

Hello, GTK

Here is a ”hello, world” program in GTK:

```
using Gdk;
using Gtk;
using Window = Gtk.Window;

class MyWindow : Window {
    public MyWindow() : base("hello, world") {
    }

    protected override bool OnDeleteEvent(Event ev) {
        Application.Quit();
        return true;
     }
}

class Hello {
    static void Main() {
        Application.Init();
        MyWindow w = new MyWindow();
        w.ShowAll();
        Application.Run();
    }
}
```

The program displays a blank window with title ”hello, world”.

Make sure that you can build and run this program on your computer. Here is a page describing how to install Gtk# and build Gtk# programs on various operating systems.

At the beginning, the program imports the `Gdk` and `Gtk` namespaces. GDK is a lower-level graphics library that underlies GTK. Some of the classes we will use live in the `Gdk` namespace, so you should always import both `Gdk` and `Gtk`.

The next line ”`using Window = Gtk.Window`” is necessary because the namespaces `Gdk` and `Gtk` both contain classes called `Window`, so we must specify which one we want to use.

We next see that our class `MyWindow` is a subclass of `Window`. In this subclass, we have an event handling method `OnDeleteEvent`. This method runs automatically when the user closes the window, which causes a Delete event to occur. In response to this event, we quit the application.

drawing graphics

Like most application toolkits, GTK is an event-driven system. Typically in your Main() method you will call the Application.Run() method, which runs the main event loop. Then, when actions occur such as the user clicking a mouse button, the system fires an event that your code can respond to.

There are two ways to enable your code to run in response to a GTK event. One way is to add a handler to a C# event object. For example, `Window` is a subclass of `Widget`, and `Widget` has a event called `ButtonPressEvent` that fires when the user presses a mouse button. So if you write an event handler method such as

```
void buttonHandler(object o, ButtonPressEventArgs args) { … }
```

then you can register it to run in response to the event, like this:

```
w.ButtonPressEvent += buttonHandler;
```

However it is often more convenient to write a subclass of a Window or other type of widget. In addition to C# events, each Gtk# class also contains virtual methods that will run automatically in response to each event type. You can override these methods to add code that will run in response to events, and that is easier than hooking up event handlers using +=. This is the approach we took in the ”hello, world” program above, where `MyWindow` is a subclass of `Window`.

Here is a Gtk# program that draws a circle. Each time the user clicks the window, the circle toggles on and off:

```
using Cairo;
using Gdk;
using Gtk;
using System;
using Window = Gtk.Window;

class MyWindow : Window {
    bool draw = true;

    public MyWindow() : base("circle") {
        AddEvents((int) (EventMask.ButtonPressMask));
        Resize(500, 500);
    }
    protected override bool OnButtonPressEvent (EventButton e) {
        draw = !draw;
        QueueDraw();
        return true;
    }

    protected override bool OnExposeEvent (EventExpose e) {
        if (draw)
            using (Context c = CairoHelper.Create(GdkWindow)) {
                c.SetSourceRGB(0.5, 0.5, 0.0);  // olive color
                c.Arc(250, 250, 150, 0.0, 2 * Math.PI);
                c.Fill();
            }
        return true;
    }

    protected override bool OnDeleteEvent(Event ev) {
        Application.Quit();
        return true;
     }
}

class Hello {
    static void Main() {
        Application.Init();
        MyWindow w = new MyWindow();
        w.ShowAll();
        Application.Run();
    }
}
```

At the top of the program we import the `Cairo` namespace. Cairo is a library for drawing 2-dimensional graphics. GTK integrates closely with Cairo, and most GTK applications use Cairo for visual output.

In the MyWindow() constructor we call the important method AddEvents() to tell GTK which input events we wish to receive. In GTK you will not receive any input events unless you ask for them by calling AddEvents(), so don’t forget to make this call!

When the user clicks the mouse, the program toggles the boolean field ’draw’ and then calls the important method QueueDraw(). This method tells the system that it is time to redraw the window’s contents, and causes the system to call OnExposeEvent() in the near future. This is how a GTK program typically works. Do not attempt to paint anywhere outside the OnExposeEvent() method – instead, whenever the underlying data (i.e. model) changes, call QueueDraw() to cause OnExposeEvent() to redraw the view.

In OnExposeEvent(), the program retrieves a `Cairo.Context` object by calling CairoHelper.Create(). It then calls Arc() to draw a circle. By passing 0.0 and 2 * Math.PI, we tell Arc() that we want it to go all the way around the center point, drawing a complete circle. (If we had passed 0.0 and Math.PI, we would have gotten a half circle.) In the quick reference documentation you will find other methods you can use for drawing.

Be aware that Arc() and other drawing methods do not immediately draw - they merely add a geometric shape to the current path. After you call these methods, you must call either Stroke() or Fill() to draw to the output surface. Stroke() draws an outline of the current path, and Fill() fills it in.

menus

Like most graphical toolkits, GTK supports menus. Each menu contains a set of commands that the user can select, and application-specific code runs in response to each command.

The `MenuItem` class represents a menu item. Each `MenuItem` holds a instance of the `System.EventHandler` delegate, which is defined as follows:

delegate void EventHandler (object sender, EventArgs e);

So when you write a method that will run in response to a menu item, it needs to take these two parameters of type `object` and `EventArgs`. (Usually you will just ignore the arguments that are passed.)

Here is a program with a menu that lets the user select a shape to be displayed:

```
using Cairo;
using Gdk;
using Gtk;
using System;
using Window = Gtk.Window;

using static Shape;
enum Shape { Square, Circle, Triangle };

class MyWindow : Window {
    Shape shape = Circle;

    public MyWindow() : base("shapes") {
        Resize(500, 500);

        MenuItem makeItem(string name, EventHandler handler) {
            MenuItem i = new MenuItem(name);
            i.Activated += handler;
            return i;
        }

        MenuItem[] items = {
            makeItem("Square", onSquare),
            makeItem("Circle", onCircle),
            makeItem("Triangle", onTriangle),
            makeItem("Quit", onQuit)
        };

        Menu fileMenu = new Menu();
        foreach (MenuItem i in items)
            fileMenu.Append(i);
        MenuItem fileItem = new MenuItem("File");
        fileItem.Submenu = fileMenu;

        MenuBar bar = new MenuBar();
        bar.Append(fileItem);

        VBox vbox = new VBox();
        vbox.PackStart(bar, false, false, 0);
        Add(vbox);
    }

    void onSquare(object sender, EventArgs e) {
        shape = Square; QueueDraw();
    }

    void onCircle(object sender, EventArgs e) {
        shape = Circle; QueueDraw();
    }

    void onTriangle(object sender, EventArgs e) {
        shape = Triangle; QueueDraw();
    }

    void onQuit(object sender, EventArgs e) {
        Application.Quit();
    }

    protected override bool OnExposeEvent (EventExpose e) {
        using (Context c = CairoHelper.Create(GdkWindow)) {
            c.SetSourceRGB(0.1, 0.1, 0.9);
            switch (shape) {
                case Square:
                    c.Rectangle(100, 100, 300, 300); break;
                case Circle:
                    c.Arc(250, 250, 150, 0.0, 2 * Math.PI); break;
                case Triangle:
                    c.MoveTo(250, 100); c.LineTo(100, 400); c.LineTo(400, 400);
                    c.ClosePath(); break;
                }
            c.Fill();
        }
        return true;
    }

    protected override bool OnDeleteEvent(Event ev) {
        Application.Quit();
        return true;
     }
}

class Shapes {
    static void Main() {
        Application.Init();
        MyWindow w = new MyWindow();
        w.ShowAll();
        Application.Run();
    }
}
```

Notice how the MyWindow() constructor builds the menu. The top-level `MenuBar` contains a `MenuItem` representing the `File` menu. That `MenuItem` contains a `Menu` representing a submenu, which itself contains a series of `MenuItem` objects (named ”Square”, ”Circle” and so on).

To place the menu at the top of the window, the program places the `MenuBar` into a `VBox`, which is a container widget that arranges one or more child widgets in a vertical column. In this case the `VBox` contains only a single child widget, namely the `MenuBar`. We add the `MenuBar` to the `VBox` by calling PackStart() which places it at the beginning, i.e. at the top. The two false arguments to PackStart() indicate that the `MenuBar` should not grow to occupy the available space below it, which exists because we don't add any more widgets to the `VBox`. Finally, the constructor calls Add() to add the `VBox` to the window itself.

Exercise: The methods onSquare(), onCircle() and onTriangle() above are all very similar. Figure out how to eliminate this code duplication.

Week 9: Notes

There was no lecture or tutorial this week.

Our topics this week are covariance and contravariance and iterators.

You can read about our C# topics in Essential C# 7.0:

- Chapter 12 ”Generics”: Covariance and Contravariance
- Chapter 17 ”Building Custom Collections”: Iterators

They are also covered in Programming C# 8.0:

- Chapter 5 ”Collections”: Implementing IEnumerable<T> with Iterators
- Chapter 6 ”Inheritance”: Covariance and Contravariance

Here are some more notes on these topics.

covariance

In C# we have studied both generics and inheritance. Now we will examine the phenomenon of covariance, which arises from the combination of generic and inheritance in C# (and also in related languages such as Java).

The fundamental question in covariance is this: Suppose that `Foo<T>` is a generic class, and that B is a subclass of A. Should `Foo<B>` be considered a subtype of `Foo<A>`? You might think that the answer is obviously yes, but the question is not so simple.

For example, suppose that we've written a generic class `DynArray` representing a dynamic array:

```
class DynArray<T> {
  T[] a = new T[1];
  int count;

  public int length { get { … } set { … } }

  public void add(T t) { … }

  public T this[int index] { get { … } set { … } }

  …
}
```

And suppose that we have an abstract class `Shape` with subclasses `Rectangle`, `Circle` and `Triangle`:

```
abstract class Shape {
  public abstract double area { get; }
}

class Rectangle : Shape {
  public double width, height;
  public Rectangle(double width, double height) { … }

  public override double area { get => width * height; }
}
```

```
class Circle : Shape { … }

class Triangle : Shape { … }
```

Finally, suppose that we have a method `areaSum` that adds the area of all Shapes in a dynamic array:

```
public static double areaSum(DynArray<Shape> a) {
  double sum = 0;
  for (int i = 0 ; i < a.length ; ++i)
    sum += a[i].area;
  return sum;
}
```

We might like to use the `areaSum` method to add up the areas of a set of rectangles:

```
DynArray<Rectangle> r = new DynArray<Rectangle>();
r.add(new Rectangle(10, 2));
r.add(new Rectangle(20, 4));
double a = areaSum(r);   // is this allowed?
```

Is the call to areaSum() above legal? That is to say, is `DynArray<Rectangle>` a subtype of `DynArray<Shape>`, so that we can implicitly convert it to that type?

Actually this call to areaSum will not compile because `DynArray<Rectangle>` is not a subtype of `DynArray<Shape>`. To put it differently, the class `DynArray<T>` is not covariant.

Again, this might surprise you. Why does the compiler disallow this? Suppose that `DynArray<Rectangle>` were convertible to `DynArray<Shape>`. Then we could make the following call:

```
public static void addCircle(DynArray<Shape> a) {
  a.add(new Circle(5.0));
}

DynArray<Rectangle> r = new DynArray<Rectangle>();
addCircle(r);  // ???
```

This call would add a circle to a dynamic array of rectangles, which makes no sense. In other words, the compiler must disallow this conversion because it would be unsafe.

In fact in C# class types such as `DynArray<T>` can never be covariant. However, C# allows generic interface types to be covariant. When you declare a generic interface, you can mark a type parameter T using the keyword `out` to indicate that the interface is covariant in T. For example, here is an interface with an covariant type parameter:

```
interface ReadOnlyArray<out T> {
  int length { get; }
  T this[int index] { get; }
}
```

An interface that is covariant in T may only produce values of type T, e.g. as return values from methods. It may never receive values of type T, e.g. as values passed to methods. The compiler will check this. This constraint allows covariant type conversions to be safe.

Suppose that the `DynArray` class above implements this interface:

```
class DynArray<T> : ReadOnlyArray<T> { … }
```

And suppose that we modify `areaSum` to take a `ReadOnlyArray<Shape>` as its argument:

```
public static double areaSum(ReadOnlyArray<Shape> a) {
  double sum = 0;
  for (int i = 0 ; i < a.length ; ++i)
    sum += a[i].area;
  return sum;
}
```

Now we may pass a `DynArray<Rectangle>` to `areaSum`:

```
DynArray<Rectangle> r = new DynArray<Rectangle>();
r.add(new Rectangle(10, 2));
r.add(new Rectangle(20, 4));
double a = areaSum(r);  // will now compile
```

This works because we can convert a `DynArray<Rectangle>` to a `ReadOnlyArray<Shape>`, which works because `ReadOnlyArray<T>` is covariant.

You will get a compile-time error if you attempt to mark the following interface's type parameter T as covariant, since the interface's methods and properties receive values of type T:

```
interface Arr<T> {
  int length { get; set; }
  void add (T t);
  T this[int index] { get; set; }
}
```

contravariance

Contravariance is a complement to covariance. You can mark an interface's type parameter T as contravariant using the `in` keyword, which means that the interface only receives (never produces) values of type T.

Covariance and contravariance work in opposite directions. Suppose that `Foo<T>` is a generic interface, and that B is a subtype (e.g. a subclass) of A. If Foo is covariant in T, then a Foo<B> is also a subtype of Foo<A>, so we can implicitly convert a Foo<B> to a Foo<A>. If Foo is contravariant in T, then this goes the other way: Foo<A> is a subtype of Foo<B>!

That might seem pretty abstract, so let's look at an example of a contravariant interface. Here is an interface type for a dictionary that maps values of type K to type V. The interface is contravariant in the type parameter K:

```
interface Dictionary<in K, V> {
    V get(K key);
    void set(K key, V val);
}
```

Continuing the example from above, suppose that I have a `Dictionary<Shape, Color>` that maps a bunch of `Shape` objects to their color. Because `Dictionary` is contravariant in K, I can pass this `Dictionary` to a method squareColors() that expects a `Dictionary<Square, Color>`. That is safe, because squareColors() will only pass `Square` objects as keys, which will work in this `Dictionary`. The type conversion is allowed because the interface is contravariant and `Square` is a subtype of `Shape`. Again, notice that this is the opposite direction from a covariance conversion.

By the way, the interface `IDictionary<K, V>` in the standard C# class library is not actually contravariant in K. That's because it has methods such as `Keys()` that allow the caller to receive values values of type K from a `IDictionary`.

array covariance

Surprisingly, arrays in C# are covariant. For example, the following code will compile:

```
Rectangle[] a = new Rectangle[5];
Shape[] b = a;   // Rectangle[] is convertible to Shape[]
b[0] = new Circle(4.0);
```

But the last statement above will fail at run time since b is actually a `Rectangle[]` and a `Circle` cannot be added to a `Rectangle[]`.

I (and many other people) believe that this array covariance is a design flaw in the C# language. It exists for historical reasons (largely because Java arrays are also covariant, and the early design of C# imitated Java). Array covariance has the following negative consequences:

- Every assignment to an array of a reference type (such as a `Shape[]` above) must check at run time that the value being assigned is compatible with the destination array. This may have a significant performance cost.
- Code that assigns an incompatible element to an array may fail at run time rather than at compile time. In other words, array covariance is not type safe: it allows code to compile that may yield a run-time type error.

iterators

C# supports a special type of method called iterators. An iterator produces a sequence of values by returning them one at a time. If that sounds familiar, that's because we have seen the same concept before, namely in Python, where these are called generators!

Here is an iterator method in C# that produces a sequence of squares of integers:

```
static IEnumerable<int> squares(int start, int end) {
for (int i = start ; i <= end ; ++i)
    yield return i * i;
}
```

Notice the `yield return` statement that returns a single value in the sequence. (In Python this statement was called `yield`.)

The method returns an `IEnumerable<int>`, representing a sequence of integers. We can iterate over this sequence with `foreach`:

```
static void Main() {
     foreach (int i in squares(1, 5))
         WriteLine(i);   // writes 1, 4, 9, 16, 25
}
```

The sequence of values returned by an iterator is lazy: they are computed only on demand. Each time the caller requests the next value in the sequence, an iterator's code runs until it calls the `yield return` statement, which yields the next value in the sequence. At that point the iterator's execution is suspended until the caller requests the next value, at which point its code continues executing until the next `yield return` statement, and so on. When execution reaches the end of the iterator method body, the sequence is complete.

An iterator must have return type `IEnumerable<T>` (or the related type `IEnumerator<T>`) for some (concrete or generic) type T.

We can use iterators to construct sequences of non-numeric values as well. For example, this iterator yields a sequence of strings representing all lines in a file:

```
  static IEnumerable<string> lines(string filename) {
      using (StreamReader r = new StreamReader(filename)) {
          while (r.ReadLine() is string s)
              yield return s;
      }
  }
```

Note that an iterator may even return an infinite sequence of values! Here is an iterator that returns the infinite Fibonacci sequence:

```
static IEnumerable<int> fibs() {
   int a = 1, b = 1;

   while (true) {
       yield return a;
       (a, b) = (b, a + b);
   }
}
```

Let's add up the first 10 Fibonacci numbers:

```
static void Main() {
    int count = 0, sum = 0;
    foreach (int k in fibs()) {
        sum += k;
        count += 1;
        if (count >= 10)
            break;
    }
    WriteLine(sum);
}
```

Making classes enumerable

Using iterators, we can easily make any class enumerable, so that the `foreach` statement will work on instances of the class. All we have to do is implement a method with the special name `GetEnumerator()`. The method should return a sequence of type `IEnumerable<T>`, and the easiest way to write it is using an iterator. For example, here is an enumerable linked list class:

```
class LinkedList<T> {
    class Node {
        public T val;
        public Node next;

        public Node(T val, Node next) {
            this.val = val; this.next = next;
        }
    }

    Node head;

    void prepend(T val) {
        head = new Node(val, head);
    }
    public IEnumerable<T> GetEnumerator() {
        for (Node n = head ; n != null ; n = n.next)
            yield return n.val;
    }
}
```

Week 10: Notes

There was no lecture or tutorial this week.

Our topic this week is generating sequences using recursion.

Generating sequences using recursion

In the last lecture of Introduction to Algorithms last semester, we briefly discussed how to use recursion to generate sequences such as permutations and combinations of a set. However we didn't have time to explore that topic much, and I did not include it on the Intro to Algorithms exam. In this course we will return to this important topic and will explore it more deeply.

As a first example, suppose that we'd like to generate all subsets of the set {1, 2, …, N}. As you know from studying discrete mathematics, there are 2^N such subsets. Let's actually try to write a slightly more general C# method `subsets` that takes integers a and b and returns a sequence of subsets of {a, …, b}. For now, we will represent each subset as a string, e.g. "1 3 4" for the set {1, 3, 4}.

When we are faced with any recursive problem like this, we need to find a way to first recursively solve a subproblem and then transform the subproblem solution into a solution to the entire problem. It may help to write down the entire solution and a subproblem solution for a specific example. For example, suppose that our goal is to generate subsets of {1, 2, 3}. There are 8 such subsets:

```
(empty set)
1
2
1 2
3
1 3
2 3
1 2 3
```

Suppose that we recursively generate subsets of {2, 3}. We will get 4 subsets:

```
(empty set)
2
3
2 3
```

We must find a way to transform these into the 8 subsets above. We can now make these key observations:

- Every subset of {2, 3} is itself a subset of {1, 2, 3}.
- In addition, we can prepend the original value (1) to each subset of {2, 3} to obtain another subset of {1, 2, 3}.

This is the recursive pattern we were seeking. With this understanding, we can now write the method subsets():

```
// Return all subsets of {a, a + 1, …, b}.
static IEnumerable<string> subsets(int a, int b) {
    if (a > b)
        yield return "";
    else {
        foreach (var s in subsets(a + 1, b)) {
            yield return s;
            yield return a + " " + s;
        }
    }
}
```

We can call the method like this:

```
foreach (string s in subsets(1, 4))
    WriteLine(s);
```

This will print

```
1
2
1 2
3
1 3
…
```

Let’s study this method in detail. First, if a > b then the set {a, a + 1, …, b} is the empty set. How many subsets of the empty set exist? Be careful: there actually is one, namely the empty set itself. And so if a > b we return the empty string, representing the empty set. That is the base case.

For the recursive case, we recursively generate all subsets of {a + 1, …, b}. Following the recursive pattern we discovered above, for each such subset s we both return s, and also a subset that contains ’a’ prepended to s:

```
yield return s;
yield return a + " " + s;
```

Generating subsets using linked lists

Let’s generalize the subsets() method above in two ways:

- We like to be able to generate subsets of any set of values, not just the integers {a .. b}.
- Instead of representing each subset as a string, we’d like to store it as some kind of set of values that we can manipulate programmatically.

We will write our generalized method using linked lists, which are a good choice of data structure here because we want to be able to prepend elements efficiently (i.e. in O(1) time). Here is our implementation:

```
class Node<T> {
    public T val;
    public Node<T> next;

    public Node(T val, Node<T> next) {
        this.val = val; this.next = next;
    }
}

// Return a sequence of subsets of the given list.
static IEnumerable<Node<T>> subsets<T>(Node<T> list) {
    if (list == null)
        yield return null;  // empty set
    else {
        foreach (Node<T> n in subsets(list.next)) {
            yield return n;
            yield return new Node<T>(list.val, n);
        }
    }
}
```

Notice the similarities between this method and the subsets() method on integers above.

Generating compositions of an integer

Consider the problem of generating all compositions of an integer. A composition of an integer N is a sequence of positive integers that add up to N. Order matters: two distinct compositions of N might contain the same set of integers in a different order. For example, the compositions of 4 are

```
4
3 + 1
2 + 2
2 + 1 + 1
1 + 3
1 + 2 + 1
1 + 1 + 2
1 + 1 + 1 + 1
```

An integer N has 2^{N-1} compositions. To see this, consider a rod of length N. If we want to cut the rod into pieces with integral lengths, then there are (N – 1) places we can cut the rod. In each of those places we can either cut or not cut, which are 2 possibilities. So there are 2^{N-1} possible sets of cuts, each of which corresponds to a unique composition.

Once again we must find a recursive pattern. Here it is: given any integer N, let's first choose some value K to be the first element of a composition that we want to generate. We can recursively generate all compositions of the what's left, i.e. of (N – K). For each such composition, we can prepend K to get a composition of N. We must repeat this process for every possible K.

To put it differently, we can rewrite the compositions of 4 above as follows:

```
4
3 + (compositions of 1)
2 + (compositions of 2)
1 + (compositions of 3)
```

With this pattern in mind, we can write the code. We represent each composition as a string:

```
static IEnumerable<string> compositions(int n) {
    if (n == 0)
        yield return "";
    else
        for (int i = 1 ; i <= n ; ++i)
            foreach (string s in compositions(n - i))
                yield return i + " " + s;
}
```

Once again note the base case carefully. How many compositions exist of N = 0? There is one – the empty set. That's because the empty set is (vacuously) a set of positive integers that add up to 0.

Also note the strategy that we used to find the recursive pattern here. We chose a first element of the target sequence, subtracted it out, then recursively dealt with the rest. This strategy is often useful for this sort of problem.

Generating partitions of an integer

Now consider the problem of generating partitions of an integer. A partition of an integer N is a set of positive integers that add up to N. Order does not matter: {2, 4} and {4, 2} are the same set, so they count as only a single partition of 6.

There are 11 partitions of the integer 6:

```
6
5 1
4 2
4 1 1
3 3
3 2 1
3 1 1 1
2 2 2
2 2 1 1
2 1 1 1 1
1 1 1 1 1 1
```

Unlike compositions, no closed formula is known for the number of partitions of a given integer N.

Let's try to write a function that will generate all partitions of an integer N. We will write the numbers in each partition in non-increasing order (so that each partition can be written in exactly one way). In theory, we could generate all compositions of N, and then select only those that are in non-increasing order. That would generate the partitions, but would be highly inefficient, so we would like to find a better approach.

A recursive pattern may not be immediately obvious. For example, it is certainly not true that we can prepend 2 to any non-increasing partition of 4 to yield a non-increasing partition of 6. For example, 3 1 is a non-increasing partition of 4, but "2 3 1" is not valid in our schema, since it is not non-increasing (and hence duplicates the partition "3 2 1").

In looking at the partitions of 6 above, we may notice the following pattern. In each partition that begins with 2, the rest of the numbers are a partition of 4 in which every number is at most 2.

This suggests that we need to generalize the problem in order to solve it recursively. Specifically, we will write a recursive function that takes integers N and K and generates the partitions of N that contain only values that are ≤ K, with all numbers in decreasing order.

```
static IEnumerable<string> partitions(int n, int k) {
    if (n == 0)
        yield return "";
    else if (n > 0)
        for (int i = k ; i >= 1 ; --i)
            foreach (string s in partitions(n - i, i))
                yield return i + " " + s;
}

static IEnumerable<string> partitions(int n) => partitions(n, n);
```

Note the following:

- When the function calls itself recursively, it passes i as the next value for k. That is so all subpartitions that it generates will contain only values that are ≤ i, which ensures that even after i is prepended, the resulting partition will contain numbers in non-increasing order.

- The check `if (n > 0)` is very important. That's because at the moment the function calls itself recursively, it's possible that i > n and so (n – i) is negative, so in the recursive call we will have n < 0. In that case the function must return no results. Without the check for (n > 0), it would recurse forever.

  Now, alternatively you could write the for loop like this:

  ```
      for (int i = Math.Min(k, n) ; i >= 1 ; --i)
  ```

As a final note, note that the situation we saw here is common: many problems can be solved recursively only when generalized. When you cannot solve a problem using direct recursion, look for a way to generalize it that will let you solve a useful subproblem recursively.

Week 9: Notes

There was no lecture or tutorial this week.

Our topics this week include searching using recursion and dynamic programming.

You can read about dynamic programming in chapter 15 ”Dynamic Programming” of Introduction to Algorithms.

Here are some more notes on these topics:

searching using recursion

In Programming 1, we learned how to search a graph using a depth-first or breadth-first search. We also noted that we can use these algorithms to search through other spaces with a graph-like structure, such as a chessboard, which we can think of as a graph where each square is a vertex with up to 8 neighbors.

In fact, we can use a similar approach to solve an enormous number of other problems. In some cases we have a physical structure that looks like a graph. And in some cases we are searching through a state space, where there is a starting state and from each state there are a set of possible transitions that can take us to other states. For example, if we are trying to unscramble a Rubik’s Cube, the start state is the intiial (scrambled) position of the cube, and each transition is a move that we can make that takes us to another state. The goal state is the position in which the cube is unscrambled.

For problems like this, we can use recursion to perform an exhaustive search of the state space. When we do this, we are really performing a depth-first search, since recursion naturally performs a depth-first search of a tree. Depending on the nature of the problem, we may sometimes want to keep a visited set, to prevent us from walking around in circles. For some problems, we may need to stop searching in a certain direction when we reach a contradiction or some other indication that no solution in that direction is possible. It is difficult to give a general set of rules for solving problems like this, since they vary in nature, and the best way to learn how to tackle them is to practice solving various examples. The key idea to keep in mind is that recursion is a powerful tool for exploring an exponential space. In some cases you may want to use a breadth-first search instead of recursion, such as when you need a shortest path through a state space, but in most cases recursion will give the most straightforward solution.

As an example, let’s look at a classic problem of this nature. Can we place N queens on an N-by-N chessboard such that none of them attack any other? (Recall that in chess queens can move and attack horizontally, vertically and diagonally, and can move by any number of squares).

Suppose that N = 8. As a first, naive approach to the problem, we could generate all possible positions of 8 queens on an 8-by-8 chessboard, and check each such position to see whether any two queens attack each other. Since no two queens can occupy the same square, the number of possible positions is 64 choose 8, or (64!)/(56!)(8!). That number equals about 4.4 billion. That’s a pretty big number, and especially for larger values of N this approach will be infeasible.

A far more efficient approach is possible. We know that each column can hold at most one queen. So we can first place a queen somewhere in column 1; there are 8 possible ways to do so. Then we can place a queen in column 2, and so on. With this approach, the number of ways to place queens is 8^8, or about 16.8 million, which is a much smaller number than with the previous approach where we were placing queens everywhere.

However we certainly should not generate all 16.8 million of these possibilities, and check each one individually to see if any queens attack each other. Instead, as we place queens, we will cut off the search at any point where two queens attack each other. To put it differently, we will never consider any path in which a queen is placed in a position that attacks another queen. As a trivial example, if we place the first queen at (1, 1), i.e. row 1 and column 1, then we will not even consider placing the next queen at (1, 2). This search pruning technique dramatically decreases the effective search space, and is widely applicable to many problems.

Below is a C# program that finds and prints a solution to the 8 queens problem. Notice that the queens() method is recursive, and uses an array to keep track of all the positions where queens have been placed so far. Study the program to understand how it works. When we run the program, it prints

```
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```

```
== queens.cs ==

using static System.Console;
using static System.Math;

class Queens {
    // Print the board.
    static void print(int[] a) {
        for (int r = 0 ; r < a.Length ; ++r) {
            for (int c = 0 ; c < a.Length ; ++c)
                Write(a[c] == r ? "Q " : ". ");
            WriteLine();
        }
    }

    // Return true if we can set a[k] = row without attacking any of the k queens
    // that have already been placed.
    static bool valid(int row, int k, int[] a) {
        for (int i = 0 ; i < k ; ++i)
            if (a[i] == row || Abs(a[i] - row) == k - i)
                return false;
        return true;
    }

    // Find a solution, given that k queens have already been placed.
    // a[i] holds the row number of the queen in column i.
    static bool queens(int k, int[] a) {
        if (k == a.Length) {
            print(a);    // found a solution
            return true;
        }
        for (int row = 0 ; row < a.Length ; ++row)
            if (valid(row, k, a)) {
                a[k] = row;
                if (queens(k + 1, a))
                    return true;
                }
        return false;
    }

    static void Main() {
        const int n = 8;

        queens(0, new int[n]);
    }
}
```

compositions, revisited

Last week we saw that we can use iterators in C# to recursively generate sequences of subsets, combinations, permutations, and so on. For example, here is a method we saw last week to generate all compositions of an integer. (Recall that a composition of an integer N is a sequences of positive integers whose sum is N.)

```
static IEnumerable<string> compositions(int n) {
    if (n == 0)
        yield return "";
    else
        for (int i = 1 ; i <= n ; ++i)
            foreach (string s in compositions(n - i))
                yield return i + " " + s;
}
static void printCompositions(int n) {
    foreach (string s in compositions(n))
        WriteLine(s);
}
```

In fact, we can generate and print any of these kinds of sequences without using iterators at all. Here is a similar recursive method that will print all compositions of an integer, without using an iterator:

```
  static void compositions(string prefix, int n) {
      if (n == 0)
          WriteLine(prefix.Trim());
      else
          for (int i = 1 ; i <= n ; ++i)
              compositions(prefix + " " + i, n - i);
  }

  static void printCompositions(int n) {
      compositions("", n);
  }
```

The method uses a string parameter ”prefix” to accumulate all integers in a composition that it has seen so far. (This is somewhat similar to the parameter `int[] a` in the queens() method in the previous example.) When it reaches the end of a composition, it prints out the string. (If you like, you can think of this as a state space search, where in the initial state we have no integers in the composition, and at each step we choose an integer i to add to the composition until the composition is complete.)

Which approach is better - using an iterator, or a string prefix? The approach with an iterator is more general, since it returns a sequence of compositions and the caller can do anything they like with that sequence. (This is a more ”functional” approach to the problem). But the string prefix approach is arguably simpler, since it gets the job done without using any fancy iterator machinery. Eiter approach might be appropriate depending on the problem you are trying to solve and on your programming style.

dynamic programming

You will learn about dynamic programming both in this class and also in Algorithms and Data Structures 1. As with some other topics, the discussion in ADS 1 will be a bit more theoretical, and in this class we will focus more on writing code to solve particular problems.

Dynamic programming is a general technique that we can use to solve a wide variety of problems. Many of these problems involve optimization, i.e. finding the shortest/longest/best solution to a certain problem.

Problems solvable using dynamic programming generally have the following characteristics:

- They have a recursive structure. In other words, the problem’s solution can be expressed recursively as a function of the solutions to one or more subproblems. A subproblem is a smaller instance of the same problem.
- They have overlapping subproblems. A direct recursive implemention solves the same subproblems over and over again, leading to exponential inefficiency.

Usually we can dramatically improve the running time by arranging so that each subproblem will be solved only once. There are two ways to do that:

- In a top-down implementation, we keep the same recursive code structure but add a cache of solved subproblems. This technique is called memoization.
- In a bottom-up implementation, we also use a data structure (typically an array) to hold subproblem solutions, but we build up these solutions iteratively.

Generally we prefer a bottom-up solution, because

- A bottom-up implementation is generally more efficient.
- The running time of the bottom-up implementation is usually more obvious.

This week we will study one-dimensional dynamic programming problems, which have a straightforward recursive structure. Next week we will look at two-dimensional problems, which are a bit more complex.

Fibonacci numbers

Computing the Fibonacci number F_n is a trivial example of dynamic programming. Recall that the Fibonacci numbers are defined as

- $F_1 = 1$
- $F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}$ $(n \geq 3)$

yielding the sequence 1, 1, 2, 3, 5, 8, 13, 21, …

Here is a recursive function that naively computes the n-th Fibonacci number:

```
  static int fib(int n) => n < 3 ? 1 : fib(n - 1) + fib(n – 2);
```

What is this function's running time? The running time T(n) obeys the recurrence

$T(n) = T(n - 1) + T(n - 2)$

This is the recurrence that defines the Fibonacci numbers themselves! In other words,

$T(n) = O(F_n)$

The Fibonacci numbers themselves increase exponentially. It can be shown mathematically that

$F_n = O(\varphi^n)$

where

$\varphi = (1 + \text{sqrt}(5)) / 2$

So `fib` runs in exponential time! That may come as a surprise, since the function looks so direct and simple. Fundamentally it is inefficient because it is solving the same subproblems repeatedly. For example, `fib(10)` will compute `fib(9) + fib(8)`. The recursive call to `fib(9)` will compute `fib(8) + fib(7)`, and so we see that we already have two independent calls to `fib(8)`. Each of those calls in turn will solve smaller subproblems over and over again. In a problem with a recursive structure such as this one, the repeated work multiplies exponentially, so that the smallest subproblems (e.g. `fib(3)`) are computed an enormous number of times.

top-down dynamic programming (memoization)   As mentioned above, one way we can eliminate the repeated work is to use a cache that stores answers to subproblems we have already solved. This technique is called memoization. Here is a memoized implementation of `fib`, using a local function:

```
static int fib(int n) {
      int[] cache = new int[n + 1];

      int f(int i) {
          if (cache[i] == 0)
              cache[i] = i < 3 ? 1 : f(i - 1) + f(i - 2);
          return cache[i];
      }

      return f(n);
  }
```

Above, the `cache` array holds all the Fibonacci numbers we have already computed. In other words, if we have already computed F_i, then `cache[i]` = F_i. Otherwise, `cache[i]` = 0.

This memoized version runs in linear time, because the line

runs only once for each value of i. This is a dramatic improvement!

bottom-up dynamic programming   In this particular recursive problem, the subproblem structure is quite straightforward: each Fibonacci number F_n depends only on the two Fibonacci numbers below it, i.e. F_{n-1} and F_{n-2}. Thus, to compute all the Fibonacci numbers up to F_n we may simply start at the bottom, i.e. the values F_1 and F_2, and work our way upward, first computing F_3 using F_1 and F_2, then F_4 and so on. Here is the implementation:

```
static int fib(int n) {
    int[] a = new int[n + 1];
    a[1] = a[2] = 1;
    for (int k = 3 ; k <= n ; ++n)
        a[k] = a[k - 1] + a[k - 2];
```

```
      return a[n];
  }
```

Clearly this will also run in linear time. Note that this implementation is not even a recursive function. This is typical: a bottom-up solution consists of one or more loops, without recursion.

This example may seem trivial. But the key idea is that we can efficiently solve a recursive problem by solving subproblem instances in a bottom-up fashion, and as we will see we can apply this idea to many other problems.

rod cutting

The rod cutting problem is a classic dynamic programming problem. Suppose that we have a rod that is n cm long. We may cut it into any number of pieces that we like, but each piece's length must be an integer. We will sell all the pieces, and we have a table of prices that tells us how much we will receive for a piece of any given length. The problem is to determine how to cut the rod so as to maximize our profit.

We can express an optimal solution recursively. Let prices[i] be the given price for selling a piece of size i. We want to compute profit(n), which is the maximum profit that we can attain by chopping a rod of length n into pieces and selling them. Any partition of the rod will begin with a piece of size i cm for some value 1 ≤ i ≤ n. Selling that piece will yield a profit of prices[i]. The maximum profit for dividing and selling the rest of the rod will be profit(n – i). So profit(n), i.e. the maximum profit for all possible partitions, will equal the maximum value for 1 ≤ i ≤ n of

prices[i] + profit(n – i)

naive recursive solution

Here is a naive recursive solution to the problem:

```
  // Return the best price for cutting a rod of length n, given a table
  // with the prices for pieces from lengths 1 .. n.
  static int profit(int n, int[] prices) {
      int best = 0;
      for (int i = 1 ; i <= n ; ++i)
          best = Max(best, prices[i] + profit(n - i, prices));
      return best;
  }
```

This solution runs in exponential time, because for each possible size k it will recompute `profit(k, prices)` many times.

bottom-up dynamic programming

This solution uses bottom-up dynamic programming:

```
  // Return the best price for cutting a rod of length n, given a table
  // with the prices for pieces from lengths 1 .. n.
  static int profit(int n, int[] prices) {
      int[] best = new int[n + 1];  // best possible price for each size

      for (int k = 1 ; k <= n ; ++k) {
          // Compute the best price best[k] for a rod of length k.
          best[k] = 0;
          for (int i = 1 ; i <= k ; ++i)
              best[k] = Max(best[k], prices[i] + best[k - i]);
      }

      return best[n];  // best price for a rod of length n
```

```
}
```

Once again, this bottom-up solution is not a recursive function. Notice its double loop structure. Each iteration of the outer loop computes best[k], i.e. the solution to the subproblem of finding the best price for a rod of size k. To compute that, the inner loop must loop over all smaller subproblems, which have already been solved, looking for a maximum possible profit.

This version runs in O(N^2).

bottom-up dynamic programming, extended   Often when we use dynamic programming to solve an optimization problem such as this one, we want not only the value of the optimal solution, but also the solution itself. For example, the function in the previous section tells us the best possible price that we can obtain for a rod of size n, but it doesn't tell us the sizes of the pieces in which the rod should be cut!

So we'd like to extend our solution to return that information. To do that, for each size k < n we must remember not only the best possible prices for a rod of size k, but also the size of the first piece that we should slice off from a rod of that size in order to obtain that price. With that information, we can reconstruct the solution at the end:

```
// Return the best price for cutting a rod of length n, given a table
// with the prices for pieces from lengths 1 .. n.
static int profit(int n, int[] prices, out int[] sizes) {
    int[] best = new int[n + 1];   // best possible price for a given rod size
    int[] cut = new int[n + 1];    // size of first piece to slice from a given rod size

    for (int k = 1 ; k <= n ; ++k) {
        best[k] = 0;
        for (int i = 1 ; i <= k ; ++i) {      // for every size <= k
            int p = prices[i] + best[k - i];
            if (p > best[k]) {
                best[k] = p;
                cut[k] = i;  // remember the best size to cut
            }
        }
    }

    List<int> l = new List<int>();
    for (int s = n ; s > 0 ; s -= cut[s])
        l.Add(cut[s]);
    sizes = l.ToArray();
    return best[n];
}
```

Study this function to understand how it works. Like the preceding version, it runs in O(N^2).

smallest number of coins

Here is another classic dynamic programming problem. We would like to write a method `int numCoins(int sum, int[] coins)` that takes a integer and an array of coin denominations. The method should determine the smallest number of coins needed to form the given sum. A sum can contain an arbitrary number of coins of each denomination. For example, if sum = 34 and coins = { 1, 15, 25 }, the method will return 6, since

34 = 15 + 15 + 1 + 1 + 1 + 1

and it is not possible to form this sum with a smaller number of coins.

If we cannot form the given sum using the given denominations of coins, our method will return `int.MaxValue`.

By the way, you will recall that in a recent homework exercise we wrote a recursive program that can generate all possible ways of forming a sum from a set of coins. So one way to find the smallest number of coins would be to generate all those possible ways, and see which one uses the smallest number. However that will be hopelessly inefficient, because there may be an exponential number of ways to form the sum. Dynamic programming will give us a more efficient solution.

naive recursive solution   First, here is a naive recursive solution to the problem:

```
// Return the smallest number of coins needed to form (sum).
static int numCoins(int sum, int[] coins) {
    if (sum == 0) return 0;

    int min = int.MaxValue;
    foreach (int c in coins)
        if (c <= sum)
            min = Min(min, numCoins(sum - c, coins));

    return min == int.MaxValue ? int.MaxValue : min + 1;
}
```

This will run in exponential time.

bottom-up dynamic programming   Here is a solution using bottom-up dynamic programming. Its structure is similar to the previous rod-cutting solution.

```
// Return the smallest number of coins needed to form (sum).
static int numCoins(int sum, int[] coins) {
    int[] min = new int[sum + 1];

    for (int k = 1 ; k <= sum ; ++k) {
        // Compute min[k], the minimum number of coins needed to add up to k.
        int m = int.MaxValue;
        foreach (int c in coins)
            if (c <= k)
                m = Min(m, min[k - c]);
        min[k] = m == int.MaxValue ? int.MaxValue : m + 1;
    }

    return min[sum];
}
```

This version runs in O(N^2), where N = `sum`.

bottom-up dynamic programming, extended   This version returns an array with the values of the coins that form the sum.

```
static int[] numCoins2(int sum, int[] coins) {
    int[] min = new int[sum + 1];
    int[] first = new int[sum + 1];

    for (int k = 1 ; k <= sum ; ++k) {
        // Compute min[k], the minimum number of coins needed to add up to k.
        // Also remember first[k], the first coin to use in forming the sum k.
        int m = int.MaxValue;
        foreach (int c in coins)
            if (c <= k && min[k - c] < m) {  // best so far
                m = min[k - c];
                first[k] = c;
            }
        min[k] = m == int.MaxValue ? int.MaxValue : m + 1;
    }

    List<int> l = new List<int>();
    for (int s = sum ; s > 0 ; s -= first[s])
        l.Add(first[s]);
    return l.ToArray();
}
```

Week 12: Notes

There was no lecture or tutorial this week.

Our topics this week is dynamic programming, continued.

You can read about dynamic programming in chapter 15 ”Dynamic Programming” of Introduction to Algorithms.

Here are some more notes on the subject:

2-dimensional dynamic programming problems

Last week we began our study of dynamic programming. The dynamic programming problems we saw last week had a 1-dimensional structure. That means that each of those problems could be solved by writing a recursive function with 1 argument. The naive recursive solution was exponentially inefficient, since it solved the same subproblems over and over again. We found that we could solve these problems much more efficiently by filling in a 1-dimensional array in a certain order. This array recorded the solution to each subproblem, so that it was immediately available to use in solving other, larger, subproblems.

For example, last week we studied the rod-cutting problem. The naive recursive solution looked like this:

```
// Return the best price for cutting a rod of length n, given a table
// with the prices for pieces from lengths 1 .. n.
static int profit(int n, int[] prices) {
    int best = 0;
    for (int i = 1 ; i <= n ; ++i)
        best = Max(best, prices[i] + profit(n - i, prices));
    return best;
}
```

Notice that profit() is a function of a single argument ”int n” (ignoring the constant array prices[], which could just be stored as a field outside the function). To compute the result more efficiently, we used a bottom-up approach that filled in a 1-dimensional array called best[], where best[i] held the best possible profit for a rod of size i, i.e. the value that would be returned by profit(i, prices).

This week we will study slightly more challenging dynamic programming problems that have a more general 2-dimensional structure. For these problems, a naive recursive solution will be a function with 2 arguments. Once again, this naive solution will run in exponential time. We can solve these problems more efficiently using bottom-up dynamic programming in which we fill in a 2-dimensional array or array-like data structure. We will need to be careful about the order in which we fill in the array elements. The solution to any subproblem instance will depend on the solutions to smaller instances, so we need to fill in the array in some order that guarantees that the dependent solutions will already be available when we compute any subproblem solution.

As always, some examples will make these ideas clearer. Let’s consider the problem of finding the longest palindromic subsequence of a given string. For example, consider the string ”watermelon stew”. The string ”wteretw” is a longest palindromic subsequence:

```
W A T E R M E L O N  S T E W
W   T E R   E          T   W
```

That’s because this subsequence is a palindrome (i.e. it reads the same forwards and backwards), and there is no longer palindromic subsequence in the string. Note that the longest palindromic subsequence is not necessarily unique: ”wtemetw” is another possibility.

We can solve this problem using two-dimensional dynamic programming. Suppose that the input string s has length N, with character indices from 0 to (N - 1). Let L(i, j) be the length of the longest palindromic subsequence of s[i .. j].

We must first find a recursive formulation of the problem that will allow us to compute L(i, j) recursively for any i and j. Our base case is straightforward: if i = j, then clearly L(i, j) = 1, since any 1-character string is a palindrome. In fact it will be useful to have an additional base case that is even smaller: if j < i, then s[i .. j] is empty, so L(i, j) = 0.

Now suppose that i < j, which will be the recursive case. We may consider two possibilities:

- First suppose that s[i] = s[j]. Then some longest palindromic subsequence of s[i .. j] must include both s[i] and s[j]. To see this, let c = s[i] = s[j], and consider any palindromic subsequence s[i .. j]. If it does not begin and end with c, then it is not as long as possible, since we could prepend and append c to it to make a longer palindromic subsequence beginning at s[i] and ending at s[j]. Now suppose that the subsequence begins and ends with c. Then certainly some version of it can begin at s[i] and end at s[j] (though there may also be other identical versions that are contained at inner positions). So in this case
- Now suppose that s[i] ≠ s[j]. Then the longest palindromic subsequence of s[i .. j] does not include both s[i] and s[j], since if it did, it would begin with s[i] and would end with s[j] and would not be a palindrome. Therefore the longest palindromic subsequence of s[i .. j] must either be a palindromic subsequence of s[i .. j – 1], or of s[i + 1 .. j] (or possibly of both). So in this case

Since we have found a recursive pattern, we may now write a recursive function to compute L(i, j):

```
// Compute the length of the longest palindromic subsequence of s[i .. j].
static int len(string s, int i, int j) {
    if (j < i) return 0;
    if (i == j) return 1;

    return s[i] == s[j] ? len(s, i + 1, j - 1) + 2
                        : Max(len(s, i, j - 1), len(s, i + 1, j));
}
```

It works:

```
string s = "watermelon stew";
WriteLine(lpsLength(s, 0, s.Length – 1));   // writes 7
```

However this function will run in exponential time.

To solve this problem more efficiently, we will use bottom-up dynamic programming to fill in a 2-dimensional array len[], where len[i, j] = L(i, j). As mentioned above, we need to be careful about the order in which we fill in the array. Specifically, when we compute len[i, j] we must already know len[i + 1, j] and len[i, j – 1] and len[i + 1, j – 1]. An illustration will help here. Supposed that s = ”india”. Here is a table showing the values of len[i, j]:

Notice the following:

- Every entry on the diagonal is 1, since len[i, i] = 1 for all i.
- Every entry below the diagonal is 0, since len[i, j] = 0 whenever j < i.
- len[0, 3] = 3, since the longest palindromic subsequence of s[0 .. 3] = ”indi” has length 3. When we recursively compute len[0, 3], we will need the values of len[0, 2], len[1, 2] and len[1, 3], as illustrated by the arrows above. I’ve only drawn these arrows for this one cell, but all cells in the table will have the same dependency pattern.
- len[0, 4] = 3. This is the top-level answer we are seeking.

Now, we certainly cannot fill in this rows in this table from top to bottom, because then as we computed each row we would not already have the values below it. One possible order for filling in the rows is bottom to top, left to right:

When we fill in this order, as we encounter each cell (i, j) we will already know the value of its neighbors below, to the left, and diagonally below and to the left. And so we will be able to compute len[i, j].

Now, there is really no need for us to iterate over all the cells below the diagonal in the table above, since they all have value 0 and will be initialized to that value when we create our array. So instead we can fill in the table from bottom to top, proceeding rightward from the diagonal position in each row:

We now have a strategy, so let's write code to fill in the array:

```
// Compute the length of the longest palindromic subsequence of s.
static int longestPalindromic(string s) {
    // len[i, j] is the length of the longest palindromic subsequence of
    // s[i .. j].
    int[,] len = new int[s.Length, s.Length];
    int i, j;
    for (i = s.Length - 1 ; i >= 0 ; --i) {
        len[i, i] = 1;
        for (j = i + 1 ; j < s.Length ; ++j)
            len[i, j] =
                s[i] == s[j] ? len[i + 1, j - 1] + 2
                             : Max(len[i, j - 1], len[i + 1, j]);
      }
    return len[0, s.Length - 1];
}
```

Our function will run in O(N^2), where N = s.Length. This is a huge improvement over the exponential version.

Of course, we may want to know not only the length of the longest palindromic subsequence, but also the subsequence itself! So we'd like to extend our code to return that string. Here is one possible approach: after we have filled in the table above we can use the values in it to reconstruct the string we want. To do that, we start at the upper right, i.e. the value len[0, s.Length – 1]. We can reconstruct a path through the table that explains how that value was derived, and that path will reveal the string itself. At any cell (i, j), if s[i] == s[j] then we know that s[i] and s[j] are included in the string we want, and we can record those characters and proceed to (i + 1, j – 1). Otherwise, we proceed either to (i, j – 1) or to (i + 1, j), choosing the cell with the larger value. Here is an extended version of the function above that can reconstruct the string in this way:

```
// Compute the longest palindromic subsequence of s.
static string longestPalindromic(string s) {
    // len[i, j] is the length of the longest palindromic subsequence of
    // s[i .. j].
    int[,] len = new int[s.Length, s.Length];

    … same code as above for filling in the table …

    // Now len[0, s.Length - 1] is the length of the longest palindromic
    // subsequence.  We now want the subsequence itself.  We need to build
    // the sequence from the outside inward, so we use two strings a and b
    // and will return (a + b).
```

```
        string a = "", b = "";
        i = 0;
        j = s.Length - 1;
        while (j >= i) {
            if (j == i) {
                a += s[i];
                break;
            }
            if (s[i] == s[j]) {
                a = a + s[i];
                b = s[j] + b;
                ++i;
                --j;
            } else if (len[i, j - 1] > len[i + 1, j])
                --j;
              else ++i;
        }

      return a + b;
    }
```

Study the function to understand how it works.

This code for constructing the longest palindromic subsequence may seem like a chore, and so you may be wondering if there is an easier way. Actually there is. When we build the table, instead of storing the length of the longest palindromic subsequence of s[i .. j] in each table cell, we can store the longest palindromic subsequence itself:

```
    // Compute the longest palindromic subsequence of s.
    static string longestPalindromic(string s) {
        // p[i, j] is the longest palindromic subsequence of s[i .. j].
        string[,] p = new string[s.Length, s.Length];
        int i, j;

        for (i = s.Length - 1 ; i >= 0 ; --i) {
            p[i, i] = s[i].ToString();
            for (j = i + 1 ; j < s.Length ; ++j)
                if (s[i] == s[j])
                    p[i, j] = s[i] + p[i + 1, j - 1] + s[j];
                else {
                    string t = p[i, j - 1], u = p[i + 1, j];
                    p[i, j] = t.Length > u.Length ? t : u;
                }
        }

        return p[0, s.Length - 1];
    }
```

That was easier! This seems like a nicer solution for this problem.

For some other dynamic programming problems, however, it may be easier or more efficient to store intermediate values in the table and the reconstruct the final solution at the end. (Actually we did that for a couple of one-dimensional dynamic programming problems last week.)

The subset sum problem

Let's look at another classic problem that we can solve using two-dimensional dynamic programming: the subset sum problem. Given a set of positive integers and an integer k, we wish to determine whether any subset of the given set has sum k.

As usual, we'd first like to come up with a recursive formulation of the problem. As a clue to how to do that, recall how a couple of weeks ago we wrote a function that generated all subsets of a given set. Here is the general approach we took. Given a set S, we can take any element x from the set, and let S' be the remaining elements in S. If we recursively generate all subsets of S', then each subset is itself a subset of S. In addition, we can add x to any of those subsets to form a subset of S.

So now suppose that we are given a set of positive integers S and an integer k, and we'd like to know whether any subset of S has the sum k. Take any integer x from S, and let S' be the remaining integers in S. If any subset of S' has sum k, then that is a subset of S which has sum k. In addition, if any subset of S' has sum (k – x), then we can add x to that subset to obtain a subset of S which has sum k.

We can use this idea to solve the problem recursively:

```
// Return true if any subset of a[0 .. (i - 1)] has sum k.
static bool hasSum(int[] a, int i, int k) {
    if (k == 0)
        return true;   // the empty set is a subset, and has sum k
    if (i == 0)
        return false;  // set is empty, cannot have non-zero sum
    return hasSum(a, i - 1, k)            // we can make k without a[i - 1]
        || a[i - 1] <= k &&
           hasSum(a, i - 1, k - a[i - 1]); // we can make k by adding a[i - 1]
}

static bool hasSum(int[] a, int k) => hasSum(a, a.Length, k);
```

As usual, this native recursive solution may take exponential time to run.

The function hasSum above takes two parameters i and k (in addition to the constant array a). That is a sign that we can solve this problem using two-dimensional dynamic programming. We will need a two-dimensional array that holds the boolean value `hasSum(a, i, k)` for every possible value of i and k. In our bottom-up implementation, we will also call this array hasSum. Specifically, `hasSum[i, k]` will be true if any subset of `a[0 .. (i – 1)]` has sum k, just like in the recursive function above.

Once again we must consider the order in which we will fill the array elements. When we compute `hasSum[i, k]`, we may need to know the value of `hasSum[i – 1, j]` for any value j in the range 0 <= j <= k. That shows that we should fill the array in increasing order of i. It does not matter whether we fill each array column in increasing or decreasing order of k, since the computation of hasSum[i, k] does not depend on any other values in the same column.

After we have filled in the array, if it turns out that there was some subset with sum k, we would like to generate that subset. As in other dynamic programming problems, we can use an iterative loop to reconstruct the solution from the array. Suppose that `hasSum[i, k]` is true for some i and k, so we know that some subset of `a[0 .. (i – 1)]` has sum k. We would like to generate that subset. If `hasSum[i – 1, k]` is true, we don't need to include `a[i - 1]` in the solution, so we can proceed to `hasSum[i – 1, k]`. If `hasSum[i – 1, k]` is false, then we know that `a[i – 1]` is in the solution set, so we can record it and proceed to `hasSum[i – 1, k – a[i – 1]]`.

Combining these ideas, here is our bottom-up solution:

```
// Does any subset of the integers in a have sum s?  If so, return the subset;
// otherwise return null.
static int[] subsetSum(int[] a, int s) {
    // hasSum[i, k] is true if some subset of a[0 .. (i - 1)] adds up to k.
    bool[,] hasSum = new bool[a.Length + 1, s + 1];
    hasSum[0, 0] = true;   // we can make 0 from the empty set

    for (int i = 1 ; i <= a.Length ; ++i)
        for (int k = 0; k <= s ; ++k)
            hasSum[i, k] = hasSum[i - 1, k]    // we can make k without a[i - 1]
                        || a[i - 1] <= k &&
                           hasSum[i - 1, k - a[i - 1]];  // we can make k by adding a[i - 1]

    if (!hasSum[a.Length, s])
        return null;  // sum is not possible

    // Now construct the integers in the set.
    List<int> result = new List<int>();
    for (int i = a.Length ; i > 0 ; --i) {
        if (!hasSum[i - 1, s]) {
            result.Add(a[i - 1]);
            s -= a[i - 1];
        }
    }

    return result.ToArray();
}
```

Once again, you may feel that having to reconstruct the solution at the end is a bother. Is there an easier way? Well, just as in the previous exercise we could store the solutions themselves in the array we are filling in. For example, instead of a two-dimensional array of booleans, we could store a two-dimensional array of `List<int>`:

```
List<int>[,] sumSet = new List<int>[a.Length + 1, s + 1];
```

And then for any i and k, if there is a subset of `a[0 .. i]` whose sum is k, then `sumSet[i, k]` could hold a list of integers containing that subset, and could otherwise be null.

However this solution would be relatively inefficient. If we use lists to represent sets, then every time we want to construct a list that contains all the elements in a previous set plus a new integer, we must make a copy of the previous list. If the problem instance was large, all of these list copies could take a significant amount of time and memory.

If, however, we represent sets using linked lists rather than List objects (which are really arrays), then no copying would be necessary, since we can prepend an element to a linked list (while leaving the existing list intact) in O(1). So that would be a reasonable solution, and you may want to try to code that up as an exercise. Of course, even that solution would be less efficient than our solution above, since it is hard to beat an array of booleans if you are trying to save space or time. :) So is it worth using linked lists to avoid the extra loop at the end of the method above? You can make your own judgment about that. :)