A First Look At C#

Visual Studio is freely available to
download by IU students at: IUWare.iu.edu

Outline

- 13.2 Thinking about objects
- 13.3 Simple expressions and statements
- 13.4 Class definitions
- 13.5 About references and pointers
- 13.6 Getting started with a C# language system

C# borrowed from Java

Simple Java Program

```
public class ex1 {
  public static void main(string args[])
  {    System.out.println("Hello World");
  }
}
```

Simple C# Program

```
public class ex1{
  public static void Main(string [] args)    {
  System.Console.WriteLine("Hello, World!");
  }
}
```

Java/C#

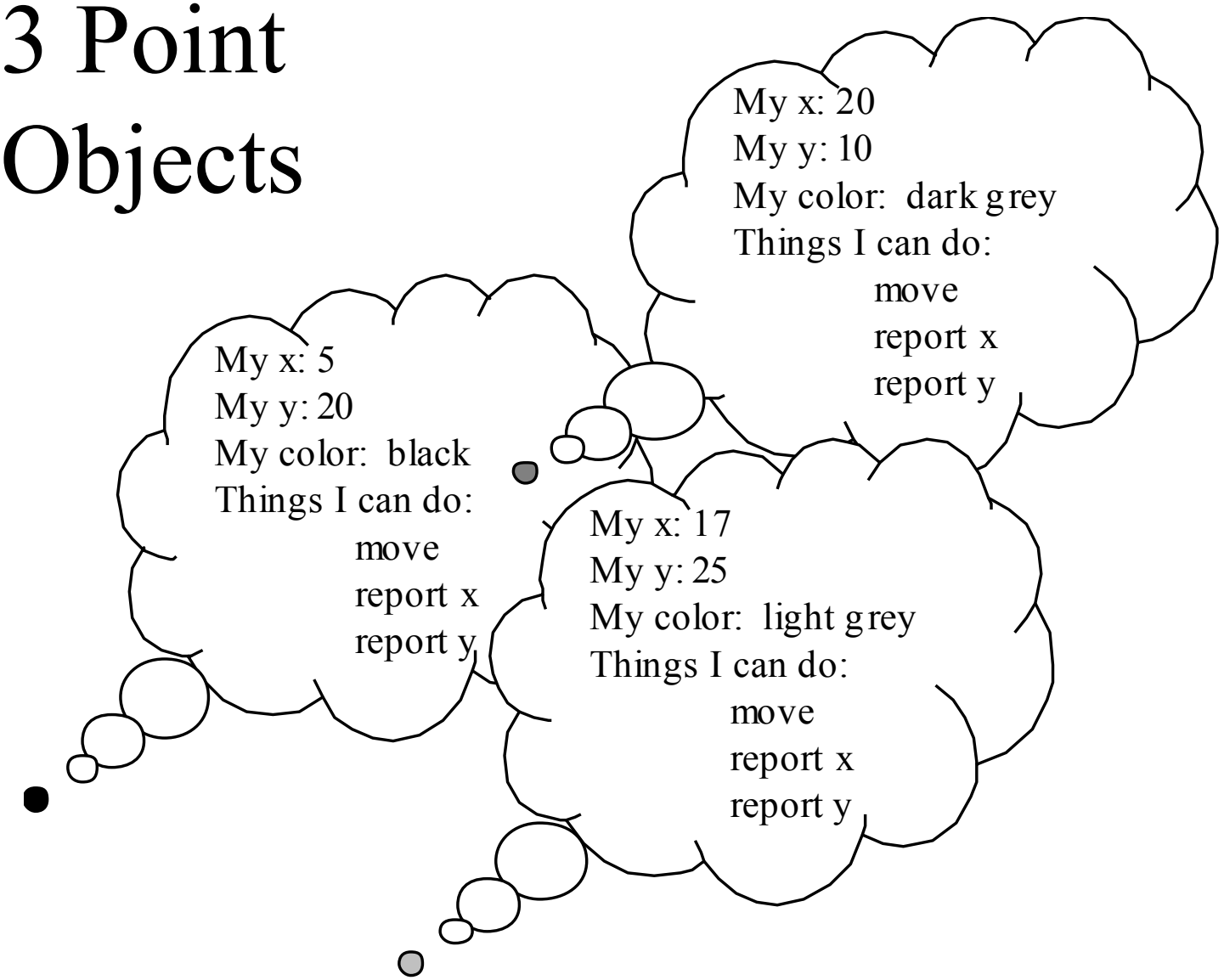
```
class Stack  {
  private double    s[ ] = new double[10];
  private int           top;

  public Stack() { top = -1; }
  public double pop() {
       return s[top--]; };
  public void push( double e ) {
       s[++top] = e; }

  public static void main(string a[])
  {  Stack s1 = new Stack();
     s1.push(3.14);
     s1.push(-13.0);
     System.out.print( s1.pop() );
  }
}
```

```
using System;
class Stack{
    private double [ ] s=new double[10];
    private int            top;

    public Stack() { top = -1; }
    public double pop() {
      return s[top--]; }
    public void push( double e ) {
      s[++top] = e; }

    static void Main(string[] a)
    { Stack s1 = new Stack();
       s1.push(3.14);
       s1.push(-13.0);
       Console.WriteLine(  s1.pop() );
    }
}
```

Object Oriented Example

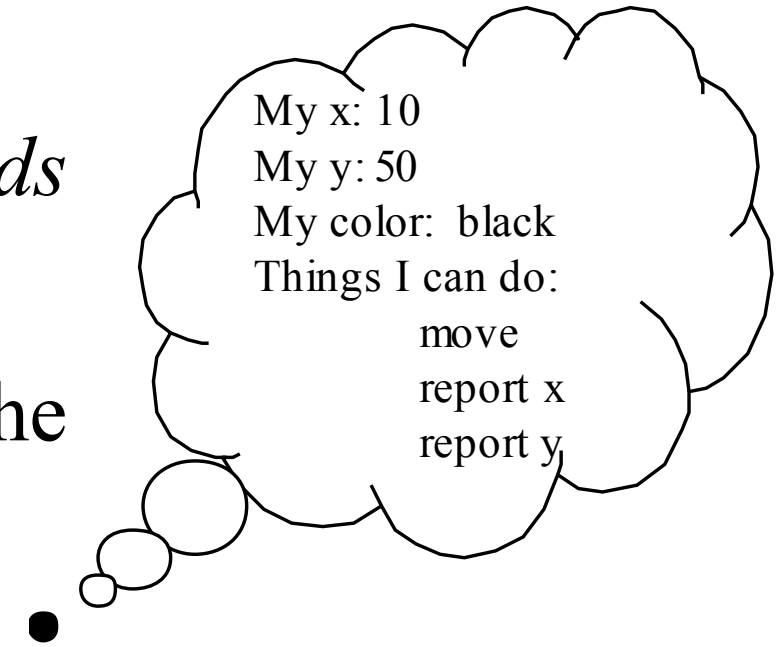
- Colored points on the screen
- Data?
  - Coordinates
  - Color
- Operations?
  - Move itself
  - Report its position

3 Point Objects



C# Terminology

- Each point is an *object*
- Each includes three *fields*
- Each has three *methods*
- Each is an *instance* of the same *class*



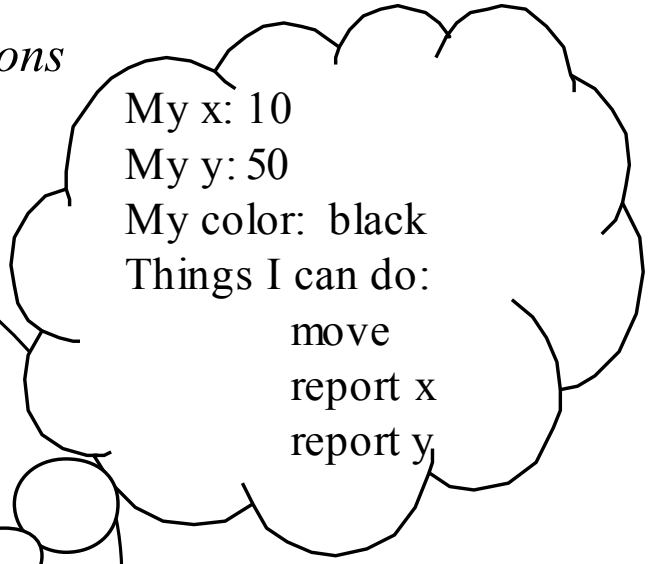
Object-Oriented Style

- Class serves as template for data fields and implements operations
- Object is instance of a class, essentially the memory storage allocated for field data
- Each object independent of others
- The class of an object determines the operations that can be performed on an object
- Object field data can be limited to access only within methods implemented in the object's class.
- Object-oriented languages support information-hiding and data encapsulation.

C# Class Definitions: A Peek

```
public class Point {
  private int x,y;
  private Color myColor;

  public int currentX() {
    return x;
  }

  public int currentY() {
    return y;
  }

  public void move(int newX, int newY) {
    x = newX;
    y = newY;
  }
}
```

field definitions

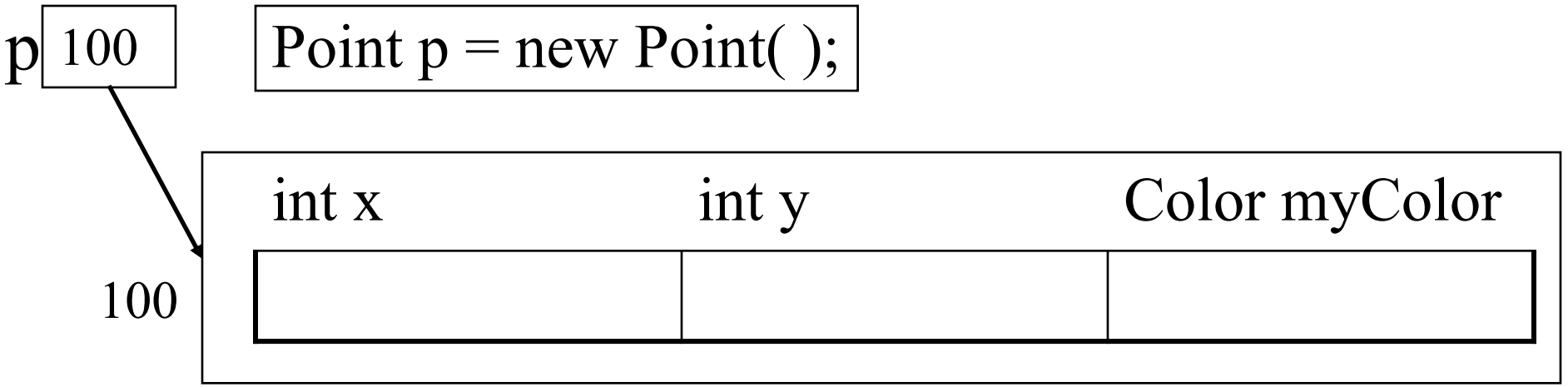


method definitions

Point Object - An Instance

- A Point object is an instance of the Point class definition.
- Each field of the object requires a memory allocation.
- A simple view of the memory allocated for a Point object instantiation.

```
class Point {
      int          x, y;
      Color        myColor;
```



Outline

- 13.2 Thinking about objects
- 13.3 Simple expressions and statements
- 13.4 Class definitions
- 13.5 About references and pointers
- 13.6 Getting started with a C# language system

Primitive Types We Will Use

- Not classes, C# not a pure OOL
- Scalars (i.e. simple, having only one value)
- **`int`**: $-2^{31}..2^{31}-1$, written the usual way
- **`char`**: $0..2^{16}-1$, written **`'a'`**, **`'\n'`**, etc., using the Unicode character set
- **`double`**: IEEE 64-bit standard, written in decimal (**`1.2`**) or scientific (**`1.2e-5`**, **`1e3`**)
- **`boolean`**: **`true`** and **`false`**
- Oddities: **`void`** and **`null`**

Primitive Types We Won't Use

- **`byte`**: $-2^7..2^7-1$
- **`short`**: $-2^{15}..2^{15}-1$
- **`long`**: $-2^{63}..2^{63}-1$, written with trailing **`L`**
- **`float`**: IEEE 32-bit standard, written with trailing **`F`** (**`1.2e-5`**, **`1e3`**)

Reference Types

- Class types are all *reference* types: they are references to objects
  - Any class name, like **`Point`**
  - Any interface name (Chapter 15)
  - Any array type, like **`Point[]`** or **`int[]`** (Chapter 14)
  - Class parameters are allocated on the *heap*

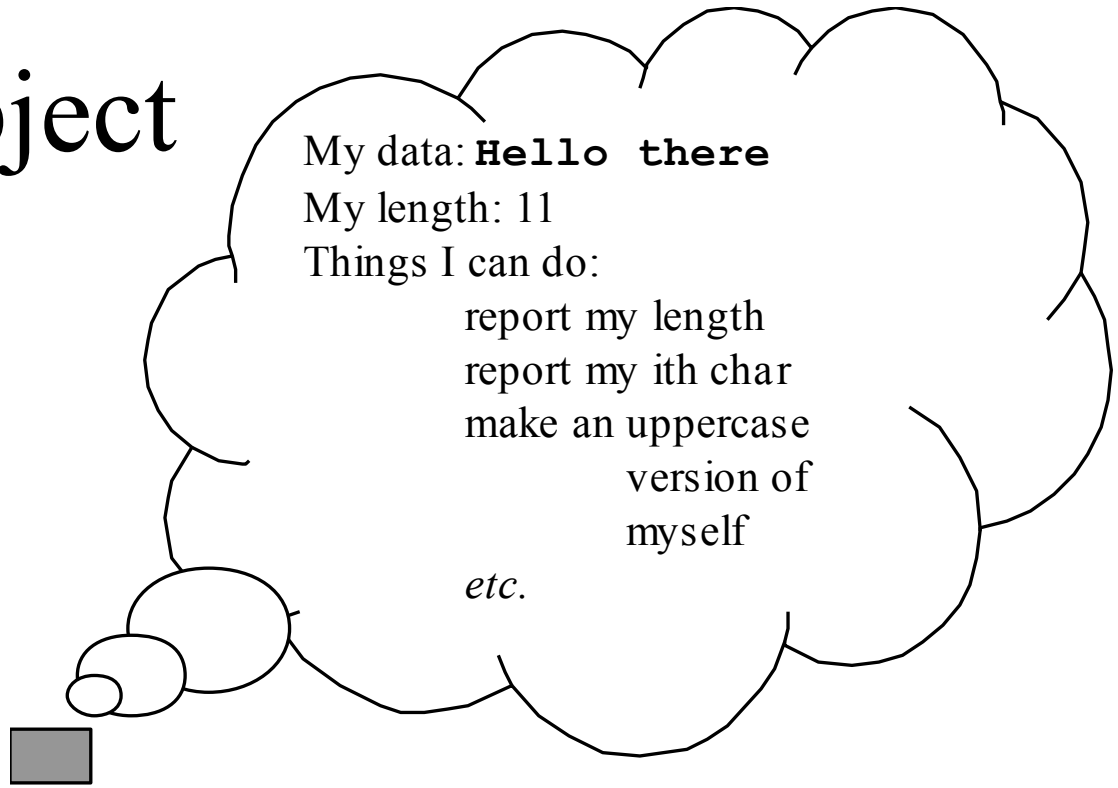
Value Types

- Struct types are *value* types: they are values of objects
  - Any struct name
  - Struct parameters are allocated on the *stack*

strings

- Predefined but not primitive: a class **`string`**
- A string of characters enclosed in double-quotes works like a string constant
- But it is actually an instance of the **`string`** class, and object containing the given string of characters

A string Object



"Hello there"

```
"Hello there".Length is 11
"Hello there".Substring(7,1) is h
"Hello there".ToUpper() is HELLO THERE
```

Numeric Operators

- **`int`**: **`+`**, **`-`**, **`*`**, **`/`**, **`%`**, unary **`-`**

C# Expression	Value
`1+2*3`	`7`
`15/7`	`2`
`15%7`	`1`
`-(5*5)`	`-25`

- **`double`**: **`+`**, **`-`**, **`*`**, **`/`**, unary **`-`**

C# Expression	Value
`13.0*2.0`	`26.0`
`15.0/7.0`	`2.142857142857143`

Concatenation

- The **+** operator has special overloading and coercion behavior for the class **`string`**

C# Expression	Value
`"123"+"456"`	`"123456"`
`"The answer is " + 4`	`"The answer is 4"`
`"" + (1.0/3.0)`	`"0.33333333333333333"`
`1+"2"`	`"12"`
`"1"+2+3`	`"123"`
`1+2+"3"`	`"33"`

Comparisons

- The usual comparison operators **<**, **<=**, **>=**, and **>**, on numeric types
- Equality **==** and inequality **!=** on any type, including **double**

C# Expression	Value
`1<=2`	`true`
`1==2`	`false`
`true!=false`	`true`

Boolean Operators

- **`&&`** and **`||`**, short-circuiting
- **`!`**, like F#'s **`not`**
- **`a?b:c`**, like F#'s **`if a then b else c`**

C# Expression	Value
`1<=2 && 2<=3`	`true`
`1<2 || 1>2`	`true`
`1<2 ? 3 : 4`	`3`

Operators With Side Effects

- An operator has a *side effect* if it changes something in the program environment, like the value of a variable or array element
- In F#, and in C# so far, we have seen only *pure* operators—no side effects
- Now: C# operators with side effects

Assignment

- **`a=b`**: changes **`a's`** memory equal to **`b's`**
- Assignment is an important part of what makes a language *imperative*
- Assignment is dangerous when aliases allowed as in C#
- Aliases occur when multiple names reference a common memory location
- Aliases allow side-effects by changes to a memory location through more than one name

Rvalues and Lvalues

- Why does **`a=1`** make sense, but not **`1=a`**?
- Expressions on the right must have a value: **`a`**, **`1`**, **`a+1`**, **`f()`** (unless **`void`**), etc.
- Expressions on the left must have *memory* locations: **`a`** or **`d[2]`**, but not **`1`** or **`a+1`**
- These two attributes of an expression are sometimes called the *rvalue* and the *lvalue*

More Side Effects

- Compound assignments

Long C# Expression	Short C# Expression
`a=a+b`	`a+=b`
`a=a-b`	`a-=b`
`a=a*b`	`a*=b`

- Increment and decrement

Long C# Expression	Short C# Expression
`a=a+1`	`a++`
`a=a-1`	`a--`

Values And Side Effects

- Side-effecting expressions have both a value and a side effect
- Value of **`x=y`** is the value of **`y`**; side-effect is to change **`x`** to have that value

C# Expression	Value	Side Effect
`a+(x=b)+c`	the sum of **`a`**, **`b`** and **`c`**	changes value of **`x`**, making it equal to **`b`**
`(a=d)+(b=d)+(c=d)`	three times the value of **`d`**	changes values of **`a`**, **`b`** and **`c`**, making them all equal to **`d`**
`a=b=c`	the value of **`c`**	changes values of **`a`** and **`b`**, making them equal to **`c`**

Pre and Post

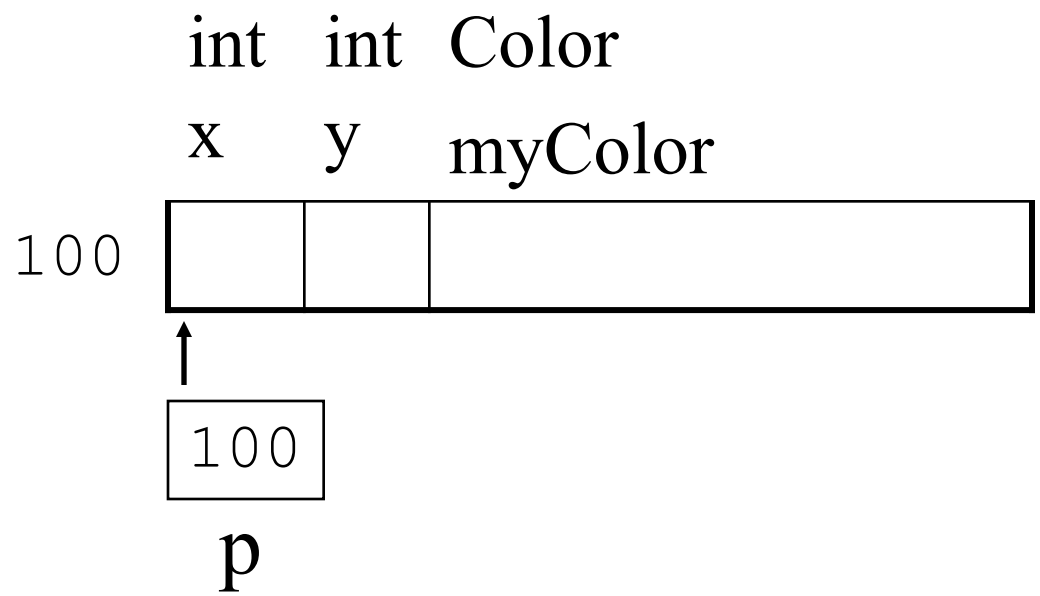
- Values from increment and decrement depend on placement

C# Expression	Value	Side Effect
`a++`	the old value of **`a`**	adds one to **`a`**
`++a`	the new value of **`a`**	adds one to **`a`**
`a--`	the old value of **`a`**	subtracts one from **`a`**
`--a`	the new value of **`a`**	subtracts one from **`a`**

Instances

- A Point object is an instance of the Point class definition.
- Each field of the object requires a memory allocation.
- A simple view of the memory allocated for a Point object instantiation.

```
class Point {
        int x, y;
        Color myColor;
}
                :
Point p = new Point( );
```



Instance Method Calls

Require an object:

```
string s = new string();
string r = "Hello";
```

C# Expression	Value
`s.Length`	the length of the **`string s`**
`s.Equals(r)`	**`true`** if **`s`** and **`r`** are equal, **`false`** otherwise
`r.Equals(s)`	same
`r.ToUpper()`	A **`string`** object that is an uppercase version of the **`string r`**
`r.SubString(3,1)`	the **`char`** value in position 3 in the **`string r`** (that is, character 4)
`r.ToUpper().SubString(3,1)`	the **`char`** value in position 3 in the uppercase version of the **`string r`**

Class Method Calls

- *Class methods* define things the class itself knows how to do—not objects of the class
- The class just serves as a labeled namespace
- Similar to ordinary function calls in non-object-oriented languages, no object required

C# Expression	Value
`(1==2).ToString()`	`"False"`
`System.Math.Abs(-5);`	`5`
`(1.0/3.0).ToString()`	`"0.3333333333333333"`

Instance versus Class Methods

```
public class Point {
  private int x, y;
  private Color myColor = Color.blue;

  public int currentX() { return x; }
  public int currentY() { return y; }
  public void move(int newX, int newY)
  {
    x = newX;
    y = newY;
  }

  public static Color default() {
    return Color.blue;
  }
}
```

Instance method

Called with an instance.
Example:
p.move(4, 6);

Class method

Declared static, called with a class. Example:
Point.default();

Method Call Syntax

- Three forms:
  - Normal instance method call:

    <method-call> **::=** *<reference-expression>* **.** *<method-name>* **(** *<parameter-list>* **)**
  - Normal class method call

    <method-call> **::=** *<class-name>* **.** *<method-name>* **(** *<parameter-list>* **)**
  - Either kind, from within another method of the same class

    <method-call> **::=** *<method-name>* **(** *<parameter-list>* **)**

Object Creation Expressions

- To create a new object that is an instance of a given class

  <*creation-expression*> **::=** **new** <*class-name*> **(**<*parameter-list*>**)**

- Parameters are passed to a *constructor*—a special method of the class

C# Expression	Value
`char[] c = {'A','B','C'};`	a new **`string`** that contains
`string s = new string(c);`	the **`char`** values from array

No Object Destruction

- Objects are created with **`new`**
- Objects are never explicitly destroyed or deallocated
- Garbage collection reclaims storage automatically for any inaccessible object (chapter 14)
- Removes programmer memory management as source of errors

General Operator Info

- All left-associative, except for assignments
- 15 precedence levels
  - Some obvious: `*` higher than `+`
  - Others less so: `<` higher than `!=`
  - Use parentheses to make code readable
- Many coercions
  - `null` to any reference type
  - Any value to `string` for concatenation
  - One reference type to another sometimes (Chapter 15)

Numeric Coercions

- Numeric coercions (for our types):
  - **`char`** to **`int`** before any operator is applied (except string concatenation)
  - **`int`** to **`double`** for binary ops mixing them

C# expression	value
`'a'+'b'`	`195`
`1/3`	`0`
`1/3.0`	`0.3333333333333333`
`1/2+0.0`	`0.0`
`1/(2+0.0)`	`0.5`

Statements

- That's it for expressions
- Next, statements:
  - Expression statements
  - Compound statements
  - Declaration statements
  - The **`if`** statement
  - The **`while`** statement
  - The **`return`** statement
- Statements are executed for side effects: an important part of *imperative* languages

Expression Statements

<expression-statement> **::=** *<expression>* **;**

- Any expression followed by a semicolon
- Value of the expression, if any, is discarded
- C# does not allow the expression to be something without side effects, like **x==y**

C# Statement	Equivalent English Command
speed = 0;	Store a 0 in **speed**.
a++;	Increase the value of **a** by 1.
inTheRed = cost > balance;	If **cost** greater than **balance**, **inTheRed** to **true**, otherwise to **false**.

Compound Statements

<compound-statement> **::=** **{** *<statement-list>* **}**
< statement-list> **::=** *<statement>* *<statement-list>* | *<empty>*

- Do statements in order
- Also serves as a block for scoping

C# Statement	Equivalent English Command

`{`
`  a = 0;`
`  b = 1;`
`}`
	Store a zero in **a**, then store a 1 in **b**.

`{`
`  a++;`
`  b++;`
`  c++;`
`}`
	Increment **a**, then increment **b**, then increment **c**.

`{ }`	Do nothing.

Declaration Statements

<declaration-statement> **::=** *<declaration>* **;**
<declaration> **::=** *<type>* *<variable-name>*
| *<type>* *<variable-name>* **=** *<expression>*

- Block-scoped definition of a variable

`boolean done = false;`	Define a new variable named **done** of type **boolean**, and initialize it to **false**.
`Point p;`	Define a new variable named **p** of type **Point**. (Do not initialize it.)

`{`
`  int temp = a;`
`  a = b;`
`  b = temp;`
`}`
	Swap the values of the integer variables **a** and **b**.

The `if` Statement

<*if-statement*> **::=** **if (**<*expression*>**)** <*statement*>
| **if (**<*expression*>**)** <*statement*> **else** <*statement*>

- Dangling else resolved in the usual way

C# Statement	Equivalent Command in English
`if (i > 0) i--;`	Decrement **i**, but only if it is greater than zero.
`if (a < b) b -= a;` `else a -= b;`	Subtract the smaller of **a** or **b** from the larger.

`if (reset) {`
`  a = b = 0;`
`  reset = false;`
`}`
	If **reset** is **true**, zero out **a** and **b** and then set **reset** to **false**.

The **`while`** Statement

<*while-statement*> **`::=`** **`while (`**<*expression*>**`)`** <*statement*>

- Evaluate expression; if false do nothing
- Otherwise execute statement, then repeat
- Iteration is another hallmark of imperative languages
- (Note that this iteration would not make sense without side effects, since the value of the expression must change)
- C# also has **`do`** and **`for`** loops

C# Statement	Equivalent Command in English
`while (a<100) a+=5;`	As long as **a** is less than 100, keep adding 5 to **a**.

`while (a!=b)`
`  if (a < b) b -= a;`
`  else a -= b;`
	Subtract the smaller of **a** or **b** from the larger, over and over until they are equal. (This is Euclid's algorithm for finding the GCD of two positive integers.)

`while (time>0) {`
`  simulate();`
`  time--;`
`}`
	As long as **time** is greater than zero, call the **simulate** method of the current class and then decrement **time**.

`while (true) work();`	Call the **work** method of the current class over and over, forever.

The **`return`** Statement

<return-statement> **`::=`** **`return`** *<expression>*`;`
| **`return;`**

- Methods that return a value must execute a return statement of the first form
- Methods that do not return a value (methods with return type **`void`**) may execute a return statement of the second form

Outline

- 13.2 Thinking about objects
- 13.3 Simple expressions and statements
- 13.4 Class definitions
- 13.5 About references and pointers
- 13.6 Getting started with a C# language system

Class Definitions

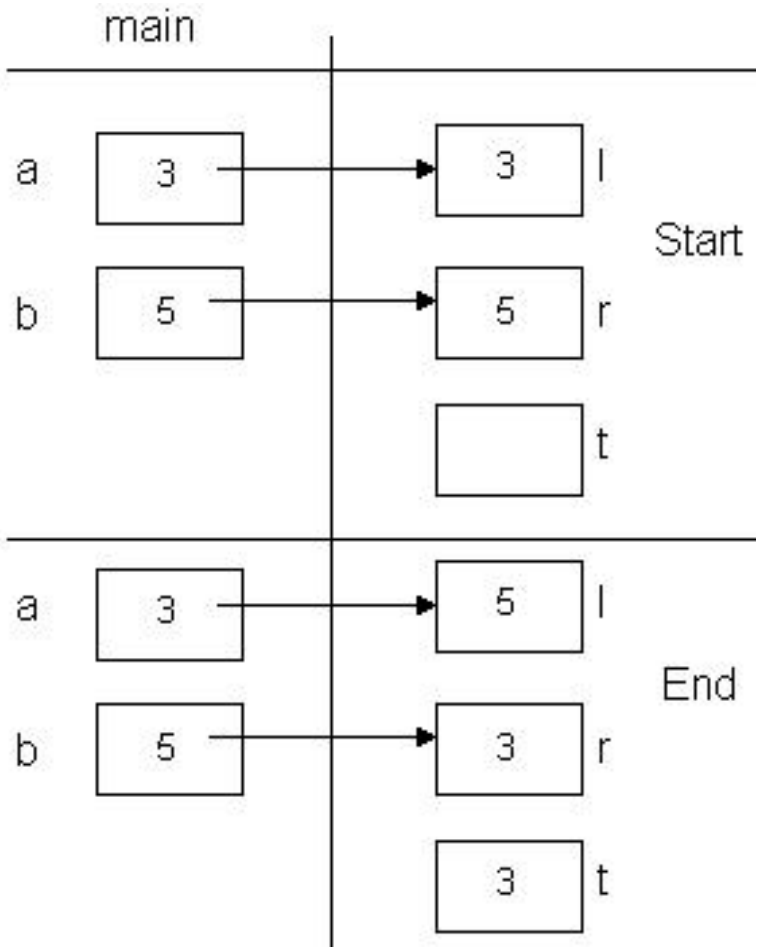
- We have enough expressions and statements
- Now we will use them to make a definition of a class
- Example: **`Int`**, a simple class wrapper of **`int`** to demonstrate parameter passing
- Example: **`ConsCell`**, a class for building linked lists of integers as F#'s **`int list`** type

Parameter Passing

- In C#, parameters can be passed either by value or by reference.
- Passing parameters by reference allows function members (methods, properties, indexers, operators, and constructors) to change the value of the parameters and have that change persist. To pass a parameter by reference, use the **ref** or **out** keyword.
- Passing Value-Type Parameters - A value-type variable contains its data directly as opposed to a reference-type variable, which contains a reference to its data. Therefore, passing a value-type variable to a method means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no affect on the original data stored in the variable. For a called method to change the value of the parameter, pass by reference, using the **ref** or **out** keyword.

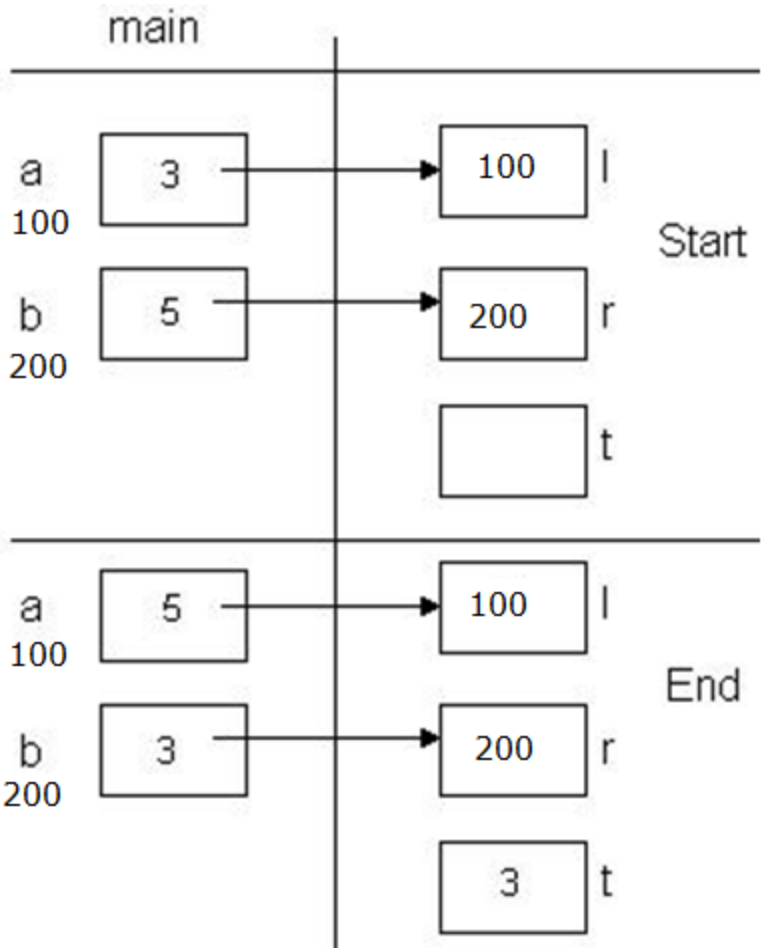
Value - Swap int

```
using System;
public class SwapProg {
  static void swap(int l, int r)
  {
                int t;
                t = l;
                l = r;
                r = t;
  }
  public static void Main(string [ ]s){
    int a = 3, b = 5;
    swap(a, b);
    Console.Write(a+" ”+b);
  }
}                          Output?
```



Reference - Swap int

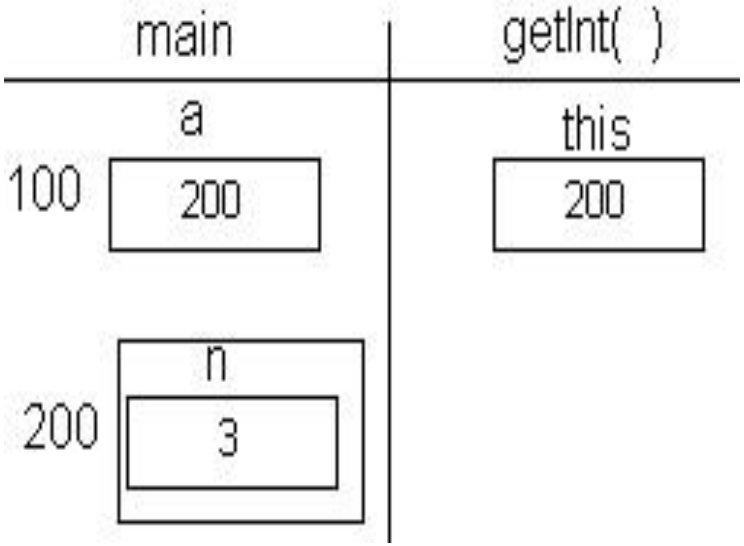
```
using System;
public class SwapProg {
  static void swap(ref int l, ref int r)
  {
                int t;
                t = l;
                l = r;
                r = t;
  }
  public static void Main(string [ ]s){
    int a = 3, b = 5;
    swap(ref a, ref b);
    Console.Write(a+" ”+b);
  }
}                          Output?
```



Instance method invocation

- C# invokes methods by reference to an object instance as the first parameter.
- Example: *a.getInt( )* passes object referenced by *a* to the *Int* method *getInt()*.
- *this* implicit name of formal parameter of method object in an instance method.

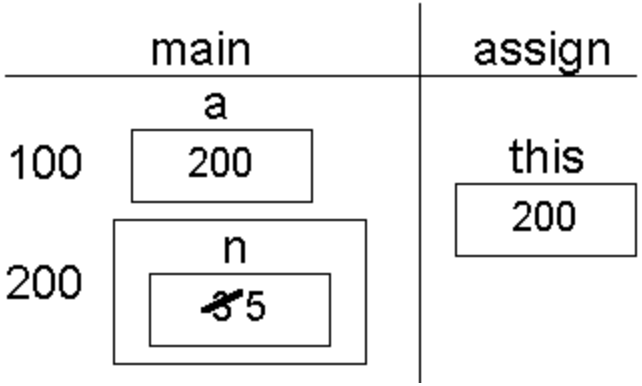
```
using System;

public class IntProg {
  public static void Main(string [ ] s) {
      Int a = new Int(3);
      Console.WriteLine( a.getInt( ) );
  }
}
class Int {
  private int n;
  public Int(int n) { this.n = n; }
  public int getInt( ) { return this.n; }
}
```



C# instance method side-effects

- Only instance methods can access an object's *private* attributes. Below *a.assign(5)* passes *a* object reference to parameter *this* of the *Int* method *assign()* giving access to the private attribute *n*.

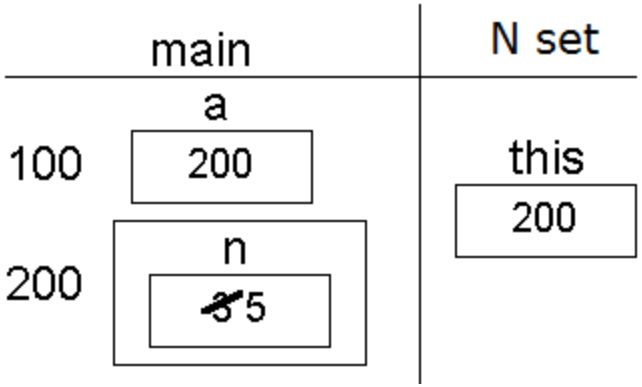
```
using System;
public class IntProg {
  public static void Main(string [ ] s) {
      Int a = new Int(3);
      a.assign(5);
      Console.WriteLine( a.getInt( ) );
  }
}
class Int {
  private int n;
  public Int(int j) { this.n = n; }
  public int getInt( ) { return this.n; }
  public void assign(int n) { this.n = n; }
}
```



C# Properties get and set

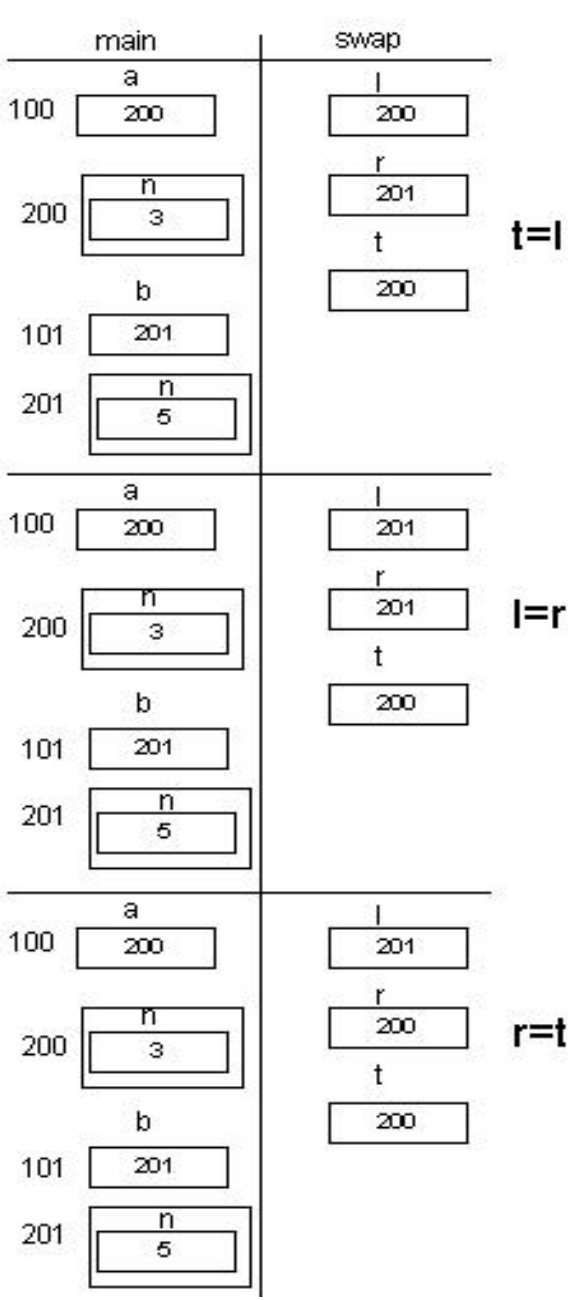
- Special *get* and *set* methods access an object's *private* attributes. Below *a.N = 5* passes *a* object reference to parameter *this* of the *Int* method *set* giving access to the private attribute *n*.

```
public class IntProg {
  public static void Main(string [ ] s) {
      Int a = new Int(3);
      a.N = 5;
      System.Console.WriteLine( a.N );
  }
}
class Int {
    private int n;
    public Int(int n) { this.n = n; }
    public int N {
        get { return this.n; }
        set { this.n = value; }
    }
}
```



Swap Int Objects

```
public class SwapProg {
   static void swap(Int l, Int r)
   {
       Int t;
       t = l;
       l = r;
       r = t;
   }
   public static void Main(string []s)   {
        Int a = new Int(),   b = new Int();
        a.N = 3;
        b.N = 5;
        swap( a, b);
        System.Console.Write(a.N+" "+b.N);
    }                          Output?
}
```

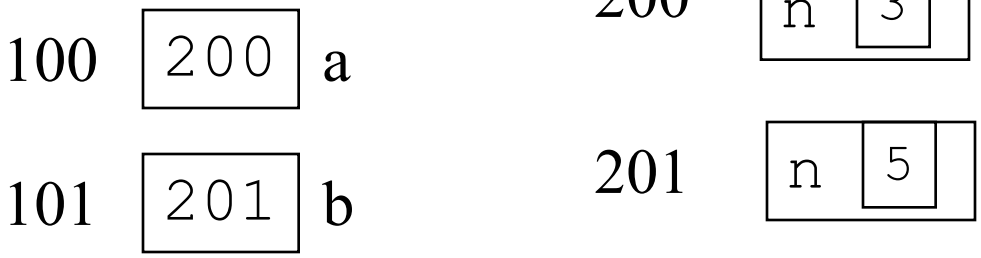


Assignment on Objects

- Swapping Int objects fails because parameters are passed by *value*
- Only *this* can access the object directly by its reference.
- *get/set* methods have access to *this.n* attribute.

```
class Int {
    private int n;
    public Int(int j) { this.n = n; }
    public int N {
        get { return this.n; }
        set { this.n = value; }
    }
}
```

```
Int a = new Int(3),
    b = new Int(5);
a.N = b.N;
```



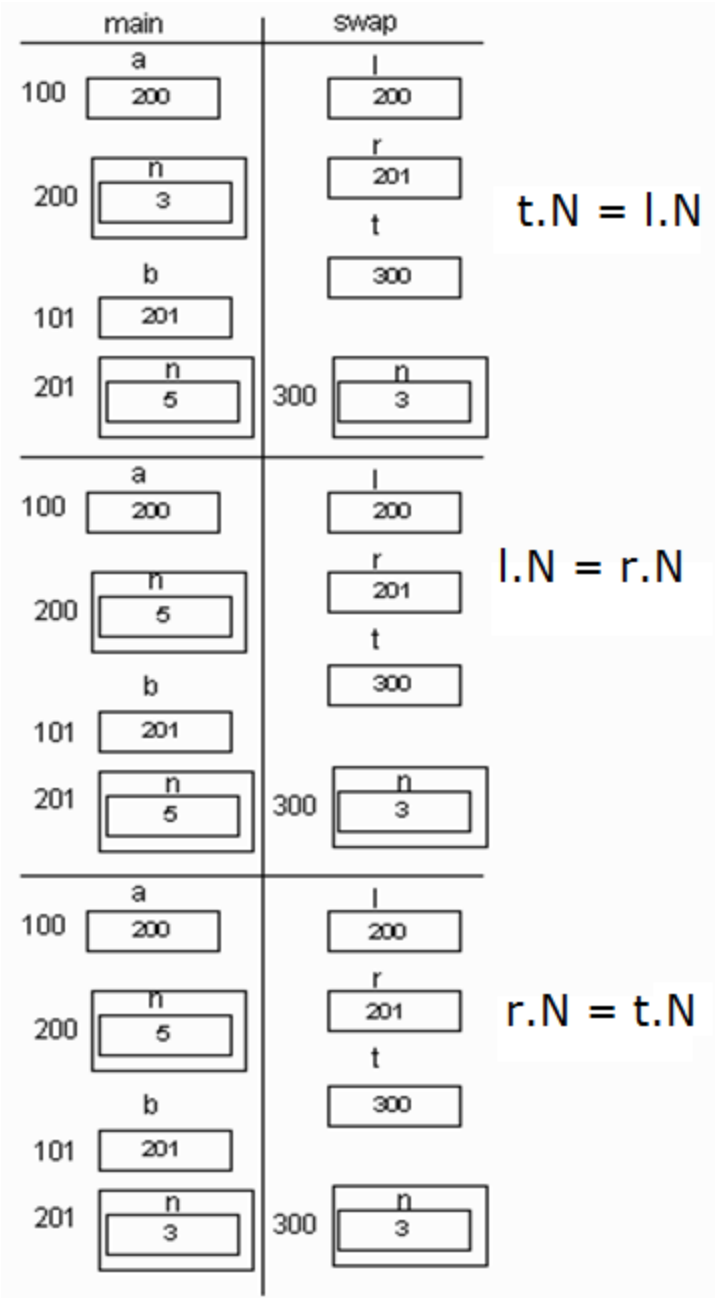
Exercise 1: Give result of:

```
a.N = b.N;
```

Swap Int Objects

```
using System;
public class SwapProg {
    public static void Main(string [] s)
    {
        Int a = new Int(),  b = new Int();
        a.N = 3;
        b.N = 5;
        swap(a, b);
        Console.Write(a.N+" "+b.N);
    }
    static void swap(Int l, Int r) {
         Int t = new Int();
         t.N = l.N;
         l.N = r.N
         r.N = t.N;
     }
}
```

Output?



Exercise 2.0

```
class Int {
    private int n;
    public int N {
        get { return this.n; }
        set { this.n = value; }
    }
}
```

```
Int a=new Int(), b=a;
```

1. Diagram above code showing memory references.

2. Diagram above code followed by:

```
b.N = 5;
b = null;
```

Exercise 2.1

```
int[] a = new int[2];
      a[0]=1;    a[1]=2;
int [] b = new int[2];
       b[0]=3;    b[1]=4;
swap(a, b);
Console.Write(a[0]+" "+a[1]+" "+b[0 ]+" "+ b[1]);
```

```
static void swap(int[] x,int[] y) {
      int t;
      t = x[0];
      x[0] = y[0];
      y[0] = t;
}
```

```
static void swap(int[] x,int[] y) {
      int [] t;
      t = x;
      x = y;
      y = t;
}
```

1. Diagram each swap using pass-by-value.

2. The output in each case?

Outline

- 13.2 Thinking about objects
- 13.3 Simple expressions and statements
- 13.4 Class definitions
- 13.5 About references and pointers
- 13.6 Getting started with a C# language system

What Is A Reference?

- A reference is a value that uniquely identifies a particular object (e.g. memory address).

```
public void swap(Int a, Int b)
```

- What is passed to **`swap`** is not an object—it is a reference to an object
- What gets stored in **`a`** is not a copy of an object—it is a reference to an object, and no copy of the object is made

Pointers

- If you have been using a language like C or C++, there is an easy way to think about references: a reference is a pointer
- That is, a reference is the address of the object in memory
- C# language systems can implement references this way

But I Thought…

- Is C# C++ *without* pointers?
- True from a certain point of view
- C and C++ expose the address nature of pointers (e.g. in pointer arithmetic)
- Programmer manipulates addresses directly, dangerous
- C# programs can't tell how references are implemented: they are just values that uniquely identify a particular object

C++ Comparison

- A C++ variable can hold an object or a pointer to an object. There are two selectors:
  - **`a->x`** selects method or field **`x`** when **`a`** is a pointer to an object
  - **`a.x`** selects **`x`** when **`a`** is an object
- A C# variable cannot hold an object, only a reference to an object. Only one selector:
  - **`a.x`** selects **`x`** when **`a`** is a reference to an object

C++/C# Reference Comparison

C++	Equivalent C#
`Int* p;` **`p = new Int ();`** **`p->set(5);`** **`p = q;`**	**`Int p;`** **`p = new Int (null);`** **`p.N = 5;`** **`p = q;`**
`Int p(); //` Implicit	No equivalent. Must invoke Int constructor explicitly using **`new.`**

class Int {
private: int n;
public:
int get( ) { return this.n; }
void set(int value){
this. n = value; }
}
}
	class Int {
	private int n;
	public int N {
	get { return this.n; }
	set { this.n = value; }
	}
	}

C# Reference Side-effect Example

- Object variables hold references to an object
- When two or more variables reference the same object, the object is aliased
- The following example illustrates how natural side-effects are in imperative languages
- And the dangerous assignment operation
- F# and functional languages in general discourage the use of assignment

Side-effect Example

```
public class Point {
  private int x, y;
  private Color myColor = Color.blue;

  public void move(int newX, int newY) {
    this.x = newX;
    this.y = newY;
  }
}

public class PointUse {
  public static void main( string arg[])
  {
    Point a = new Point();
    a.move( 100, 250);
    Point b = a;
    b.move( 0, 0);
  }
```

- What are `x` and `y` of `Point a` at the end?

Outline

- 13.2 Thinking about objects
- 13.3 Simple expressions and statements
- 13.4 Class definitions
- 13.5 About references and pointers
- 13.6 Getting started with a C# language system

Text Output

- A predefined object: **`System.Console`**
- Two methods: **`Write(x)`** to print **`x`**, and **`WriteLine(x)`** to print **`x`** and start a new line
- Overloaded for all parameter types

```
System.Console.Write("Hello there");
System.Console.Write(1.2);
```

The **`main`** Method

- A class can have a **`main`** method like this:

```
public static void Main(string[] args) {
   …
}
```

- This will be used as the starting point when the class is run as an application
- Keyword **`static`** makes this a class method; use sparingly!

C# at IUS

Visual Studio is installed in many labs around campus.

C# at Home

- C# programs can be written and executed using Visual Studio and other development environments like Xamarin.
- Freely available to download by IU students at: IUWare.iu.edu