2: Basics

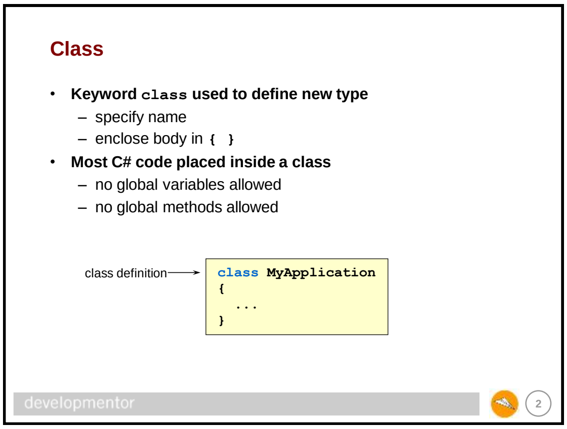
12/01/2003

Basics

(modified)

developmentor

A DEVELOPER SERVICES COMPANY

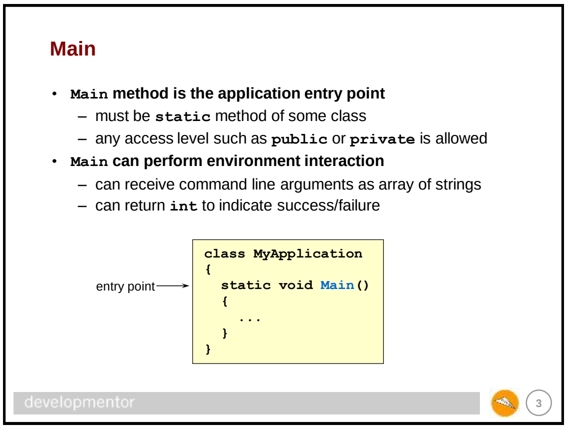


Class

- Keyword `class` used to define new type
  - specify name
  - enclose body in `{ }`
- Most C# code placed inside a class
  - no global variables allowed
  - no global methods allowed

class definition

```
class MyApplication
{
  ...
}
```

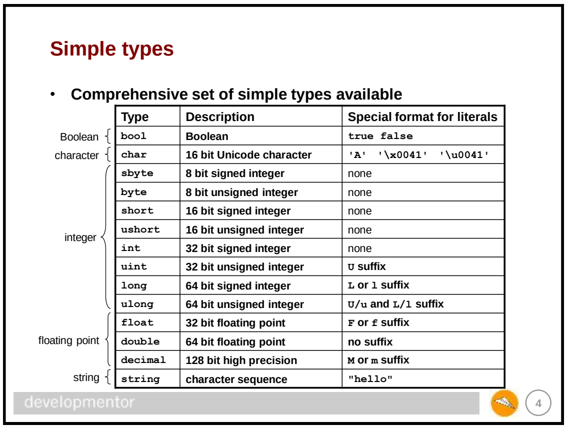


Main

- `Main` method is the application entry point
  - must be `static` method of some class
  - any access level such as `public` or `private` is allowed
- `Main` can perform environment interaction
  - can receive command line arguments as array of strings
  - can return `int` to indicate success/failure

entry point

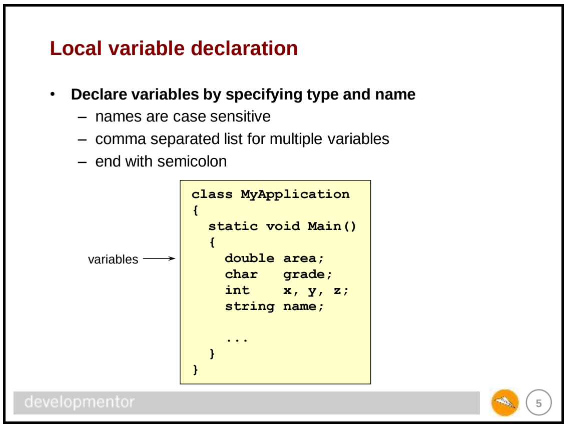
```
class MyApplication
{
  static void Main()
  {
    ...
  }
}
```



Simple types

- Comprehensive set of simple types available

	Type	Description	Special format for literals
Boolean	`bool`	Boolean	`true false`
character	`char`	16 bit Unicode character	`'A' '\x0041' '\u0041'`
integer	`sbyte`	8 bit signed integer	none
	`byte`	8 bit unsigned integer	none
	`short`	16 bit signed integer	none
	`ushort`	16 bit unsigned integer	none
	`int`	32 bit signed integer	none
	`uint`	32 bit unsigned integer	`U` suffix
	`long`	64 bit signed integer	`L` or `l` suffix
	`ulong`	64 bit unsigned integer	`U/u` and `L/l` suffix
floating point	`float`	32 bit floating point	`F` or `f` suffix
	`double`	64 bit floating point	no suffix
	`decimal`	128 bit high precision	`M` or `m` suffix
string	`string`	character sequence	`"hello"`

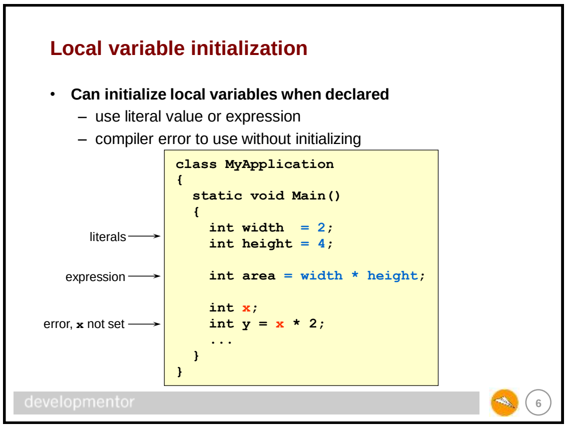


Local variable declaration

- Declare variables by specifying type and name
  - names are case sensitive
  - comma separated list for multiple variables
  - end with semicolon

variables

```
class MyApplication
{
  static void Main()
  {
    double area;
    char   grade;
    int    x, y, z;
    string name;

    ...
  }
}
```



Local variable initialization

- Can initialize local variables when declared
  - use literal value or expression
  - compiler error to use without initializing

literals

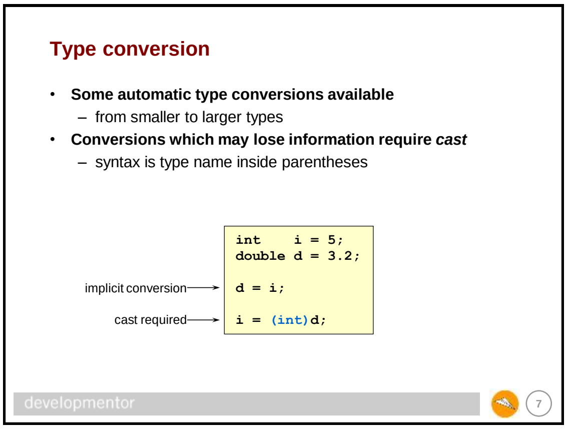
expression

error, `x` not set

```
class MyApplication
{
  static void Main()
  {
    int width  = 2;
    int height = 4;

    int area = width * height;

    int x;
    int y = x * 2;
    ...
  }
}
```

2: Basics

12/01/2003



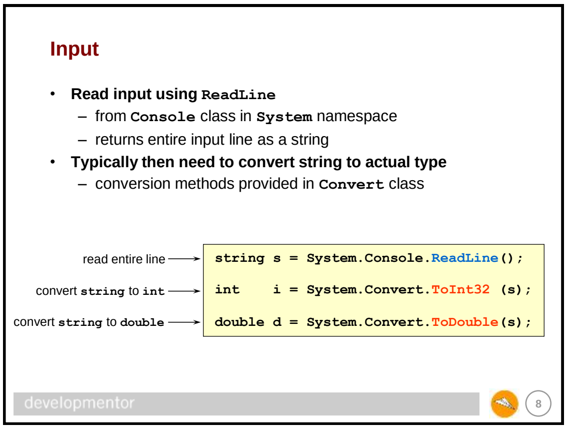
Type conversion

- Some automatic type conversions available
  - from smaller to larger types
- Conversions which may lose information require *cast*
  - syntax is type name inside parentheses

implicit conversion

cast required

```
int    i = 5;
double d = 3.2;

d = i;

i = (int)d;
```



Input

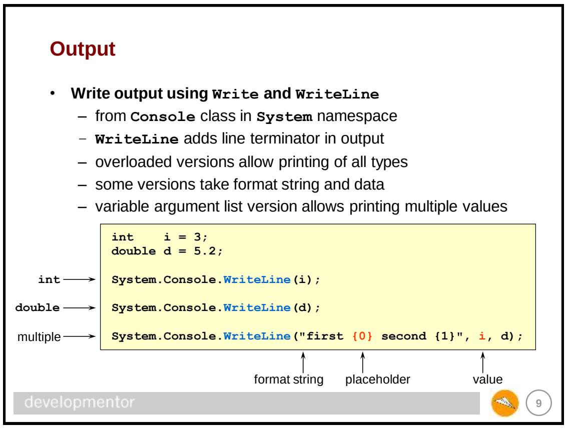
- Read input using `ReadLine`
  - from `Console` class in `System` namespace
  - returns entire input line as a string
- Typically then need to convert string to actual type
  - conversion methods provided in `Convert` class

read entire line

convert `string` to `int`

convert `string` to `double`

```
string s = System.Console.ReadLine();

int    i = System.Convert.ToInt32 (s);

double d = System.Convert.ToDouble(s);
```



Output

- Write output using `Write` and `WriteLine`
  - from `Console` class in `System` namespace
  - `WriteLine` adds line terminator in output
  - overloaded versions allow printing of all types
  - some versions take format string and data
  - variable argument list version allows printing multiple values

`int`

`double`

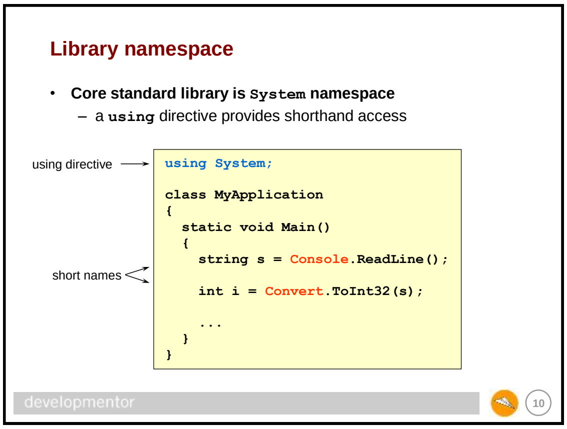
multiple

```
int    i = 3;
double d = 5.2;

System.Console.WriteLine(i);

System.Console.WriteLine(d);

System.Console.WriteLine("first {0} second {1}", i, d);
```

format string

placeholder

value



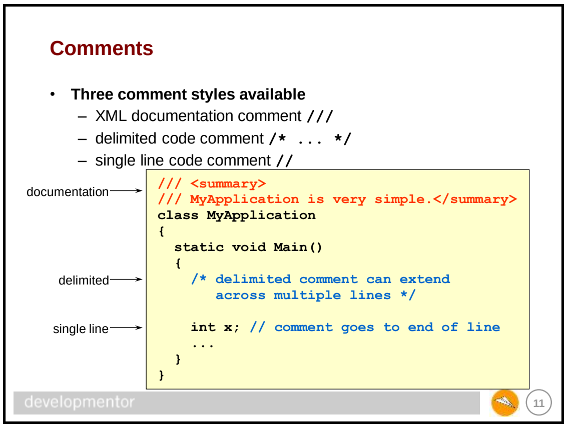
Library namespace

- Core standard library is `System` namespace
  - a `using` directive provides shorthand access

using directive

short names

```
using System;

class MyApplication
{
  static void Main()
  {
    string s = Console.ReadLine();

    int i = Convert.ToInt32(s);

    ...
  }
}
```



Comments

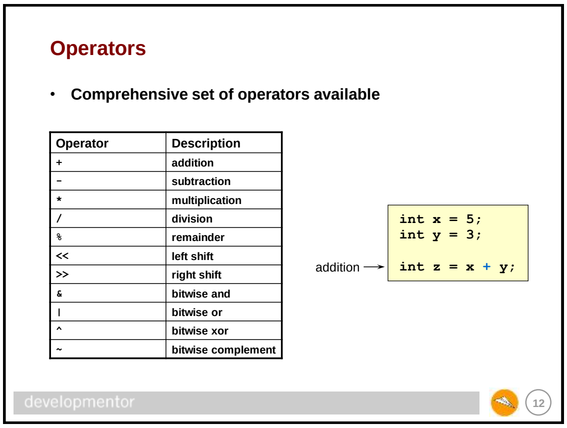
- Three comment styles available
  - XML documentation comment `///`
  - delimited code comment `/* ... */`
  - single line code comment `//`

documentation

delimited

single line

```
/// <summary>
/// MyApplication is very simple.</summary>
class MyApplication
{
  static void Main()
  {
    /* delimited comment can extend
       across multiple lines */

    int x; // comment goes to end of line
    ...
  }
}
```



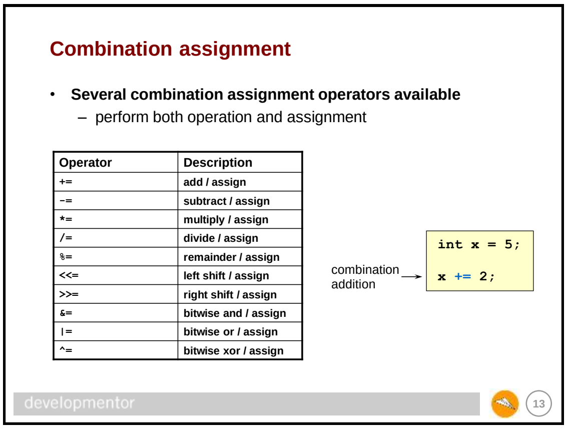
Operators

- Comprehensive set of operators available

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	remainder
<<	left shift
>>	right shift
&	bitwise and
|	bitwise or
^	bitwise xor
~	bitwise complement

addition

```
int x = 5;
int y = 3;

int z = x + y;
```



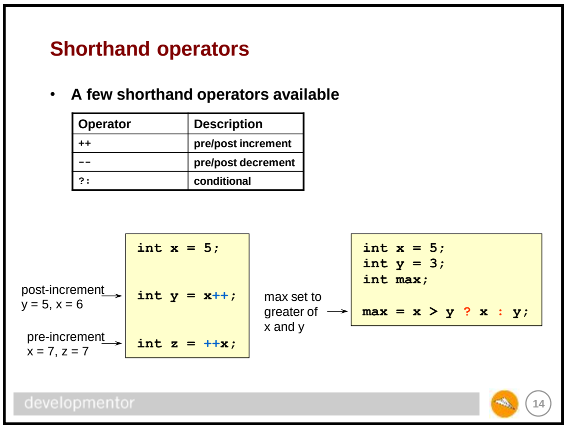
Combination assignment

- Several combination assignment operators available
  - perform both operation and assignment

Operator	Description
+=	add / assign
-=	subtract / assign
*=	multiply / assign
/=	divide / assign
%=	remainder / assign
<<=	left shift / assign
>>=	right shift / assign
&=	bitwise and / assign
|=	bitwise or / assign
^=	bitwise xor / assign

combination addition →

```
int x = 5;

x += 2;
```



Shorthand operators

- A few shorthand operators available

Operator	Description
++	pre/post increment
--	pre/post decrement
?:	conditional

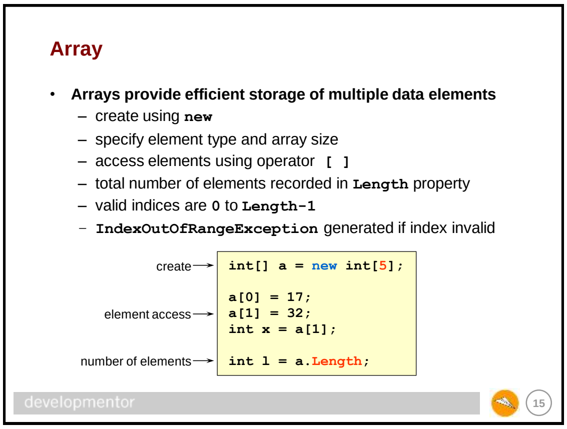
post-increment → y = 5, x = 6

pre-increment → x = 7, z = 7

```
int x = 5;

int y = x++;

int z = ++x;
```

max set to greater of x and y →

```
int x = 5;
int y = 3;
int max;

max = x > y ? x : y;
```



Array

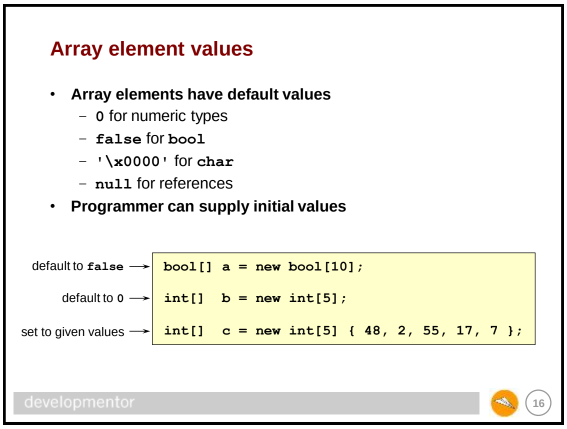
- Arrays provide efficient storage of multiple data elements
  - create using `new`
  - specify element type and array size
  - access elements using operator `[ ]`
  - total number of elements recorded in `Length` property
  - valid indices are `0` to `Length-1`
  - `IndexOutOfRangeException` generated if index invalid

create →

element access →

number of elements →

```
int[] a = new int[5];

a[0] = 17;
a[1] = 32;
int x = a[1];

int l = a.Length;
```



Array element values

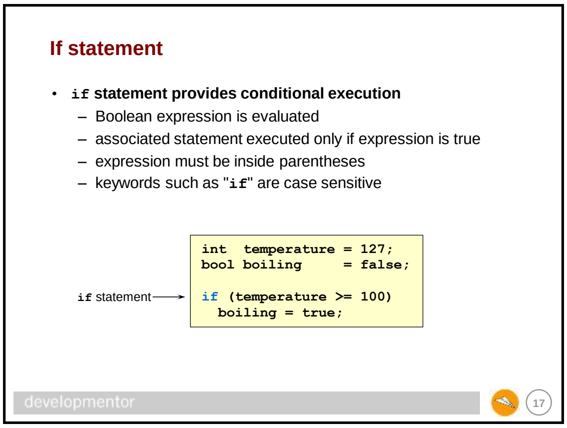
- Array elements have default values
  - `0` for numeric types
  - `false` for `bool`
  - `'\x0000'` for `char`
  - `null` for references
- Programmer can supply initial values

default to `false` →

default to `0` →

set to given values →

```
bool[] a = new bool[10];

int[]  b = new int[5];

int[]  c = new int[5] { 48, 2, 55, 17, 7 };
```

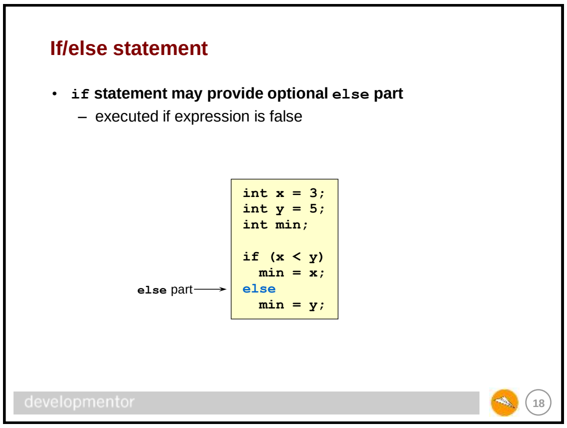


If statement

- `if` statement provides conditional execution
  - Boolean expression is evaluated
  - associated statement executed only if expression is true
  - expression must be inside parentheses
  - keywords such as "`if`" are case sensitive

`if` statement →

```
int  temperature = 127;
bool boiling     = false;

if (temperature >= 100)
  boiling = true;
```

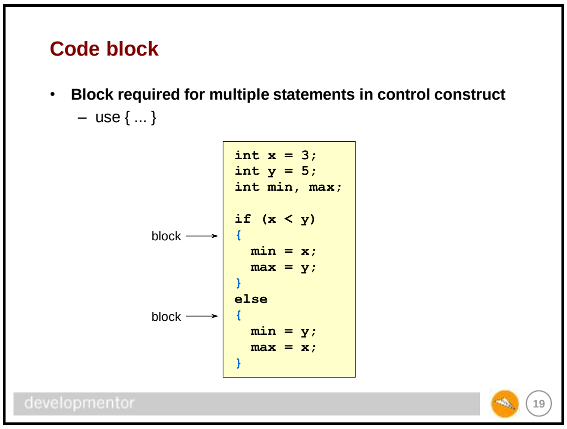


If/else statement

- `if` statement may provide optional `else` part
  - executed if expression is false

`else` part →

```
int x = 3;
int y = 5;
int min;

if (x < y)
  min = x;
else
  min = y;
```



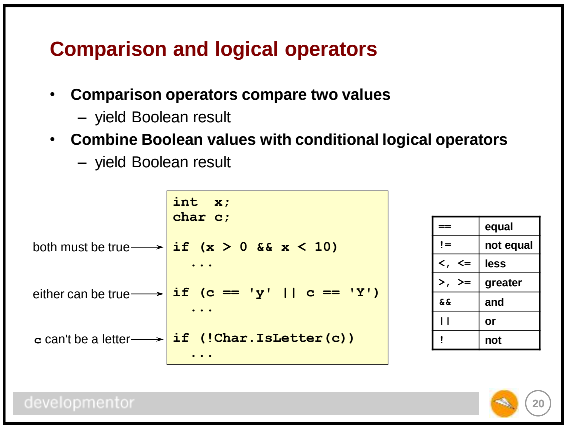
Code block

- Block required for multiple statements in control construct
  - use { ... }

```
int x = 3;
int y = 5;
int min, max;

if (x < y)
{
  min = x;
  max = y;
}
else
{
  min = y;
  max = x;
}
```

block →

block →



Comparison and logical operators

- Comparison operators compare two values
  - yield Boolean result
- Combine Boolean values with conditional logical operators
  - yield Boolean result

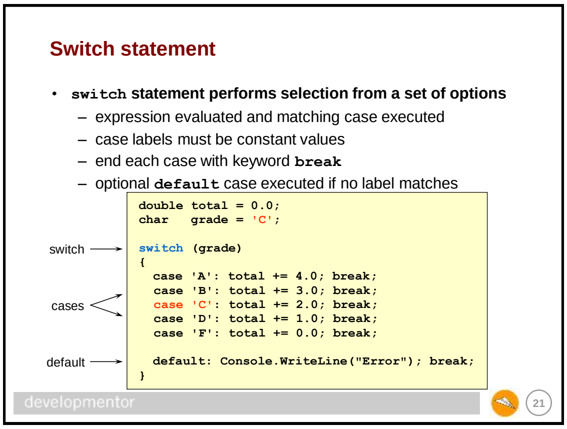
```
int  x;
char c;

if (x > 0 && x < 10)
  ...

if (c == 'y' || c == 'Y')
  ...

if (!Char.IsLetter(c))
  ...
```

both must be true →

either can be true →

c can't be a letter →

==	equal
!=	not equal
<, <=	less
>, >=	greater
&&	and
||	or
!	not



Switch statement

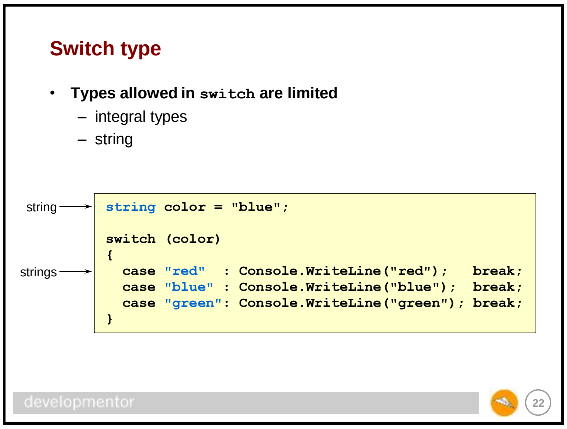
- `switch` statement performs selection from a set of options
  - expression evaluated and matching case executed
  - case labels must be constant values
  - end each case with keyword **break**
  - optional **default** case executed if no label matches

```
double total = 0.0;
char   grade = 'C';

switch (grade)
{
  case 'A': total += 4.0; break;
  case 'B': total += 3.0; break;
  case 'C': total += 2.0; break;
  case 'D': total += 1.0; break;
  case 'F': total += 0.0; break;

  default: Console.WriteLine("Error"); break;
}
```

switch →

cases

default →



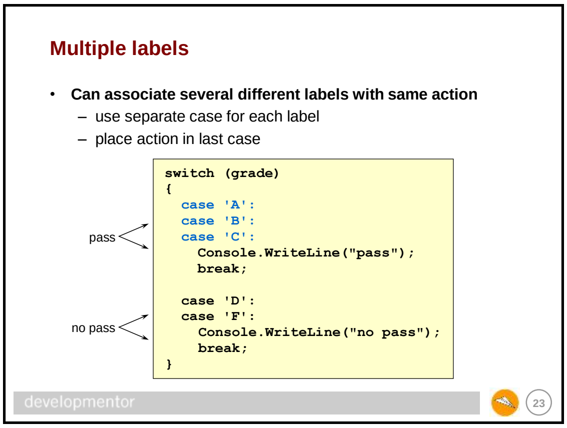
Switch type

- Types allowed in `switch` are limited
  - integral types
  - string

```
string color = "blue";

switch (color)
{
  case "red"  : Console.WriteLine("red");   break;
  case "blue" : Console.WriteLine("blue");  break;
  case "green": Console.WriteLine("green"); break;
}
```

string →

strings →



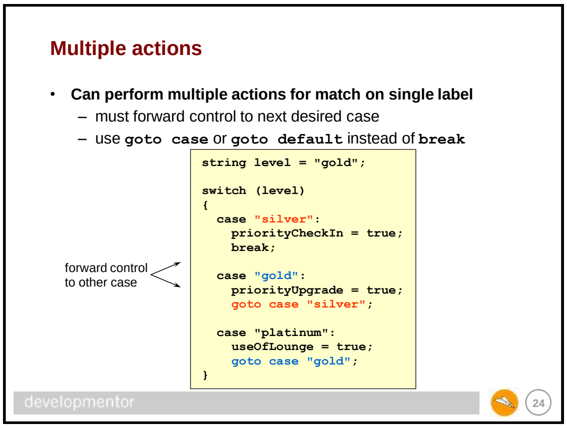
Multiple labels

- Can associate several different labels with same action
  - use separate case for each label
  - place action in last case

```
switch (grade)
{
  case 'A':
  case 'B':
  case 'C':
    Console.WriteLine("pass");
    break;

  case 'D':
  case 'F':
    Console.WriteLine("no pass");
    break;
}
```

pass

no pass

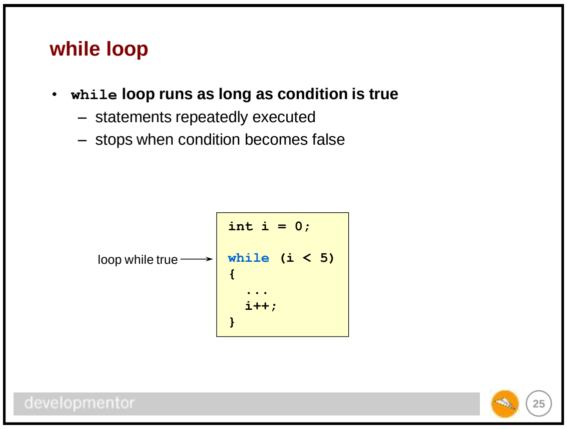


Multiple actions

- Can perform multiple actions for match on single label
  - must forward control to next desired case
  - use `goto case` or `goto default` instead of `break`

```
string level = "gold";

switch (level)
{
  case "silver":
    priorityCheckIn = true;
    break;

  case "gold":
    priorityUpgrade = true;
    goto case "silver";

  case "platinum":
    useOfLounge = true;
    goto case "gold";
}
```

forward control
to other case

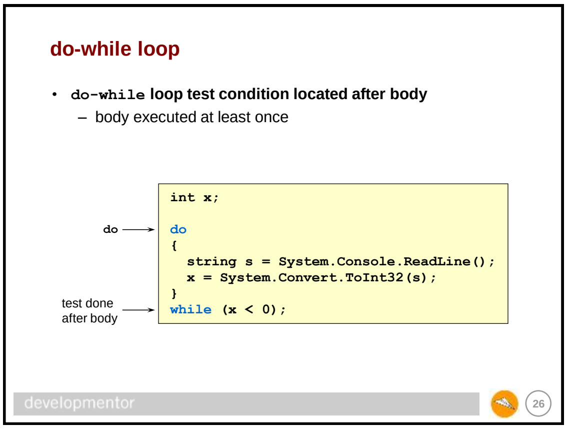


while loop

- `while` loop runs as long as condition is true
  - statements repeatedly executed
  - stops when condition becomes false

loop while true

```
int i = 0;

while (i < 5)
{
  ...
  i++;
}
```



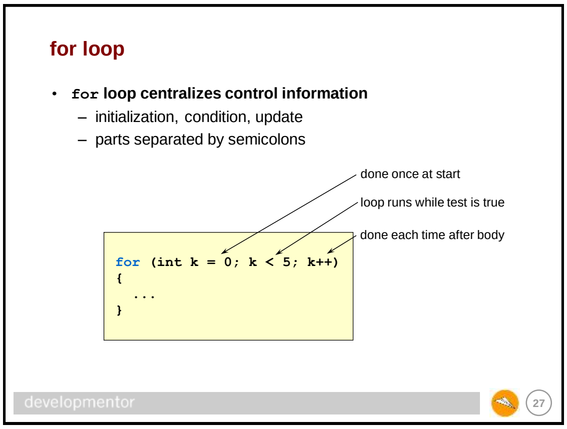
do-while loop

- `do-while` loop test condition located after body
  - body executed at least once

do

test done after body

```
int x;

do
{
  string s = System.Console.ReadLine();
  x = System.Convert.ToInt32(s);
}
while (x < 0);
```



for loop

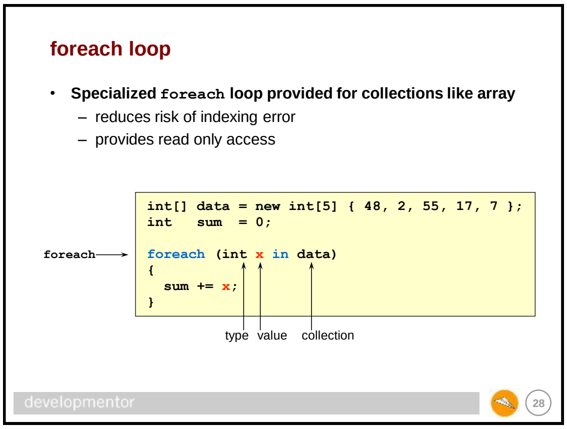
- `for` loop centralizes control information
  - initialization, condition, update
  - parts separated by semicolons

done once at start

loop runs while test is true

done each time after body

```
for (int k = 0; k < 5; k++)
{
  ...
}
```



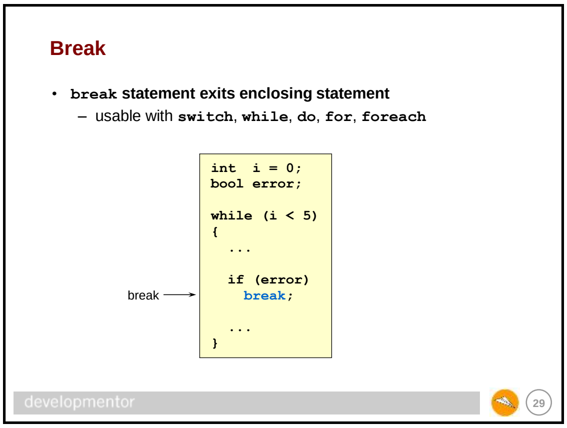
foreach loop

- Specialized `foreach` loop provided for collections like array
  - reduces risk of indexing error
  - provides read only access

foreach

```
int[] data = new int[5] { 48, 2, 55, 17, 7 };
int   sum  = 0;

foreach (int x in data)
{
  sum += x;
}
```

type value collection

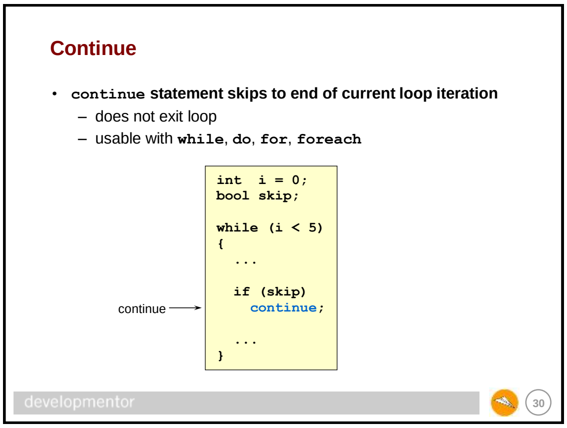


Break

- `break` statement exits enclosing statement
  - usable with `switch`, `while`, `do`, `for`, `foreach`

break

```
int  i = 0;
bool error;

while (i < 5)
{
  ...

  if (error)
    break;

  ...
}
```



Continue

- `continue` statement skips to end of current loop iteration
  - does not exit loop
  - usable with `while`, `do`, `for`, `foreach`

continue

```
int  i = 0;
bool skip;

while (i < 5)
{
  ...

  if (skip)
    continue;

  ...
}
```