**// inform compiler about Class definitions used in app**

# using System.Collection.Generic; using System.IO;

**Comments**

// Ignore from the // to the end of the line  
/\* Ignore everything until the closing \* /.  
 Even if it looks like code  
Console.WriteLine (“This will not print?”);  
 \*/

**Variables**

<type> name;

<type> can be a primitive

int, bool, long, char

or Class type

string, Integer, Money, MyClass;

name descriptive, use camelCase

Visible within the code block in which they are defined. Between the enclosing { and } braces.

**Methods**

Method signature

returnType methodName (0 or more parameters)

**returnType** class type orvoid – nothing returned or a primitive

**methodName** A method does work, it should describe the work. Start with a verb, use camelCase naming.

**Parameters** <type> name. A list of parameters passed into the method to accomplish its task.

**Method Code Block** – opens with { and continues to the matching } Between is all code for the method. Variables declared between the braces are only visible within the block

/\* Calculate Factorial using recursion \*/

long result = Factorial(10); // s/b 3\_628\_800

public static long Factorial ( long base ) {

if (base == 1)

return 1;

else

return base \* factorial(base-1);

//return base\*((base==1)? 1 : factorial(base-1));

}

// pick cnt number of random numbers

public byte[] pickLottery (int range, int cnt) {

byte[] picks = new byte[cnt];

for (int i=0; i < cnt; i++)

picks[i] = (byte)(Math.random()\*range) + 1;

return picks;

}

**Strings**

Class for dealing with text

string strName = “Edge” + “ Tech” + “ Academy”;

int len = strName.Length; // len = 17

if (strName.Contains(“Edge”)… // it does

int lastE = strName.LastIndexOf(‘e’); //=14

int firstE = strName.IndexOf(‘e’); //=3 lowercase e

string[] threeWords = strName.Split(” “);

string strTech = strName.Substring(5, 9); // “Tech”

// bigger is >= 0 b/c ETA > ET

int bigger = strName.CompareTo(“Edge Tech”);

bool ending = strName.EndsWith("h Academy");

string UPPER = strName.ToUpper(); //EDGE TECH…

int eIndex = strName.LastIndexOf("e"); // == 14

**Flow Control**

**if-then and if-then-else**

**if** (<**condition**>) // see below for conditional operators  
 **<code block>**

**else // optional**

**<code block>**

**condition** result must be Boolean (T/F).

**<code block>** if > 1 line of code statements must be inside curly braces { }

// Example of **if** statement:

**if** **( age > 16 && permit == true )**

Console.WriteLine(“You can Drive!”);

**else** **if** **( age > 16 )**

**Console.WriteLine(“You can take the test!”);**

**else**

**Console.WriteLine(“Wait ” + age – 16 + “ years”);**

**switch**

**switch** (<**condition**>) {

**case 1:**  
 **<statements>**

**else // optional**

**<code block>**

**condition** result must be Boolean (T/F).

**<code block>** if > 1 line of code statements must be inside curly braces { }

// Example of **switch** statement with enums:

**switch** **( weekDay )**

**case MONDAY :**

**case TUESDAY :**

**case WEDNESDAY :**

**case THURSDAY :**

**Console.WriteLine(“Yay! We have a new week!”);**

**break;**

**case FRIDAY :**

**Console.WriteLine(“Yay! We have a weekend!”);**

**break;**

**default :**

**Console.WriteLine(“Yay! It’s the Weekend”);**

**break;**

**}**

**Loops**

**for loop**

**for** (<**initialization**>; <**condition**>; <**increment**>) **{**  
**<code block>**

**}**

// Example for statements:

**for** **(i = 1; i < 10; i++)**   
**Console.WriteLine(i);**

**for** **(j = 10; j > 5; j--) {   
   total += j;   
 Console.WriteLine(j);  
}**

**Console.WriteLine(“Total: “ + total);**

// We have two additional lines of code for loops

**continue** Skip the rest of the loop and go to the TOP of the loop

**break** GET OUT of the loop. Go to the first line of code past the end of the loop

These also apply to the while and do-while loops

**while loop**

// no **initialization** or **increment**. Just test condition

**while** (<**condition**>){  
 **<code block>**

}

// Example of **while** statement:

linesRead = 0;

// read three numbers from the users  
**while** **( linesRead < 3 )** **{   
   string what = Console.ReadLine();  
   number = Convert.ToInt32(what);   
   Console.WriteLine("Number entered: " + number);**

**linesRead++;  
}**

**do-while loop**

// no **initialization** or **increment**. Just test condition at the END of the loop. Guaranteed one pass thru loop

**do** {  
**<code block>**

**} while** (<**condition**>);

Example of **do-while** statement:

// read three numbers from the users  
linesRead = 0;

**do {   
   string what = Console.ReadLine();  
   number = Convert.ToInt32(what);   
   Console.WriteLine("Number entered: " + number);   
 linesRead++;  
} while** **( linesRead < 3 )**

**Arrays**

int[] grades = new int[6];  // new array of 6 ints

Int[] other = { 1, 1, 2, 3, 5, 8 }; // array of 6 ints

grades[0] = 100; // first element   
grades[grades.Length-1] = 42; // last element

// an array of 200 Strings

String[] names = new String[200];

// an array of 200 Actor objects

Actor[] actors = new Actor[200];

**// Example of printing the contents of an array**

 for (i = 0; i < grades.Length; i++)   
    Console.WriteLine(

“Student “ + i + “ grade = “ + grades[i]);

**Objects**

Objects share two characteristics

**State** The condition or state of the object. How many, how long, what color, description, flavor. This is the state of the object. The defining attributes.

**Behavior** What action can be performed on the object. Get status, set availability, calculate taxes, reduce temperature. They generally are named starting with an action verb. These are the class methods.

Access to the state attributes is usually hidden from the outside world. The object has rules for how it can be modified. The Behavior is encapsulated in methods that have the rules for the object. The car can accelerate but the accelerate method determines how fast it can accelerate. The color of the sky can be chaned but only within certain bounds.

The Java class file defines the object. This is blue print of the object. In the code a programmer will do new Movie(“Casablanca”); to use the blue print to create an object that can now be manipulated by the class methods. This is called instantiating an object. Creating an instance of this object in the application memory.

Constructors – Special methods that allocate space for the object (done for you) and initialize the class attributes (you write).

Inheritance – Do not recreate the wheel. Many objects already exist and in many cases you want to do what they already do, just a little bit more. Your object will extend an existing object and inherit all the parent object did and then you add in the new more valuable methods. A parent or base class will have the generic methods and attributes. A Vehicle class will have wheels, color, engineSize trunkSpace attribute and startEngine, paintCar, brake methods. Your SUV class will extend Vehicle and add in gpsLocation, SiriusXMRadio or laneDepartureWaring attributes with enableAutoDriveMode, changeRadioByVoice or whereAmI methods.

Interface - Your class may adopt the attributes of an Interface. An Interface is a contract that says any object that implements an interface must define these methods. Anyone implementing those methods can do so as they wish but the signatures of the methods must follow the interface definition. If your car implements the WarpDrive interface it must include code to setMaxWarp, calculateTimeDilationFactor divertPowerToForwardNacelles and of course engage methods. This permits a third party to integrate with your SUV through these methods without worrying about your design decisions.

**Try-catch-finally**

You may believe you are the greatest programmer ever. But you are not. You will write bad code and worse you will interface with users and they will make mistakes.The try-catch-finally pattern lets you create code well aware of possible mistakes. Users will enter bad file names, input files will not be formatted correctly. You can test for each and every scenario and leave your code looking to noisy you can’t read it or you can bracket the areas of your code that are likely to have errors and send those errors to code that deals with the bad paths while most of your code deals with the positive paths.

// Example of error catching when access a file

try { // In the try block we trust everything is OK

stream = new FileStream(“output.txt”, FileMode.Open, FileAccess.Read);

out = new StreamReader(stream);

for (int i = 0; i < SIZE; i++) {

/\* get throws **exception** if list is too small. \*/

out.println("Value at:" +i+ "=" + list.get (i));

}

// we have just open a file and written to it.

// the whole time we never checked for any errors

// if there were errors they would be caught below

} catch (**System.IO.FileNotFoundException:** e) {

// if the list array was smaller than SIZE an exception would be thrown

// here we catch the exception and provide an error message

// rather than just having the application die.

// this error cannot be fixed by the user and will require a programming

// change but at least the programmer will have somewhere to start.

System.err.println ("IndexOutOfBoundsException: " + e.getMessage ());

} catch (**IOException** e) {

// if the OutFile.txt is not writeable an exception will be thrown

// here we catch the exception and provide an error message.

// This error the user may be able to resolve and can run the app again.

System.err.println ("Caught IOException: " + e.getMessage ());

} finally {

// we will go here no matter what happens. We can clean up the code

// and exit cleanly with all resources freed up.

if (out != null) {

Console.WriteLine ("Closing stream");

out.close ();

} else {

Console.WriteLine ("stream not open");

# Summary of Operators

|  |  |
| --- | --- |
| Simple Assignment Operator | |
| = | Simple assignment operator |
| Arithmetic Operators | |
| + | Additive operator (also for Strings) |
| - | Subtraction operator |
| \* | Multiplication operator |
| / | Division operator |
| % | Remainder operator |
| Unary Operators | |
| + | Unary plus operator |
| - | Unary minus operator; negates an expression |
| ++ | Increment operator; increments by 1 |
| -- | Decrement operator; decrements by 1 |
| ! | Logical complement operator;  inverts the value of a boolean |
| Equality and Relational Operators | |
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| >= | Greater than or equal to |
| < | Less than |
| <= | Less than or equal to |
| Conditional Operators | |
| && | Conditional-AND |
| || | Conditional-OR |
| ?: | Ternary (shorthand for if-then-else statement) |
| Type Comparison Operator | |
| instanceof | Compares an object to a specified type |
| Bitwise and Bit Shift Operators | |
| ~ | Unary bitwise complement |
| << | Signed left shift |
| >> | Signed right shift |
| >>> | Unsigned right shift |
| & | Bitwise AND |
| ^ | Bitwise exclusive OR |
| | | Bitwise inclusive OR |

# Operator Precedence

|  |  |
| --- | --- |
| Operators | Precedence |
| Postfix | expr++ expr-- |
| unary | ++expr –-expr  +expr –expr ~ ! |
| multiplicative | \* / % |
| Additive | + - |
| Shift | << >> >>> |
| relational | < > <= >= instanceof |
| Equality | == != |
| bitwise AND | & |
| bitwise XOR | ^ |
| bitwise OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &=  ^= |= <<= >>= >>>= |

# Data Types

|  |  |
| --- | --- |
| Type | Definition |
| bool | Two possible values:  true and false. |
| char | A single 16-bit character. Minimum value of 0 and a maximum value of '\uffff' (or 65,535 inclusive). |
| (s)byte | An 8-bit (un)signed integer. Min value: -128 and Max value: 127. |
| (u)short | A 16-bit (un)signed integer. Min value: -32,768 and Max value: 32,767. |
| (u)int | A 32-bit (un)signed integer. Min value: -231 and Max value: 231-1. |
| (u)long | A 64-bit (un)signed integer. ±9 Q (18 Q) |
| double | A double-precision 64-bit floating point. Default for decimal values. |
| float | A single-precision 32-bit floating point. |

# Naming Conventions

|  |  |  |
| --- | --- | --- |
| Type | Rules for Naming | Examples |
| Packages | All lowercase, should be top-level domain names, (com, edu, gov, mil, net, org). other components vary by company conventions. | com.sun.eng  edu.cmu.cs |
| Classes | **Nouns**, in mixed case first letter of all words capitalized. Simple and descriptive. Use whole words-avoid acronyms and abbreviations. | Raster;  ImageSprite; |
| Interfaces | Capitalized like class names. | WarpCapable;  Storing; |
| Methods | Verbs, mixed ProperCase, all words capitalized. | Run ();  RunFast ();  GetBackground (); |
| Variables | Mixed case with a lowercase first letter. Do not start with \_ or $ characters. Indicates intent. Avoid One-character names except for temporary variables. | i;  myWidth;  lastName; |
| Constants | Variables declared class constants should be all uppercase with words separated by underscores ("\_"). | WIDTH = 4;  PI = 3.1415; |

# Escape Sequences

|  |  |
| --- | --- |
| Escape Sequence | Description |
| \t | Tab |
| \b | Backspace |
| \n | Newline |
| \r | Carriage return |
| \f | Form feed |
| \' | Single quote |
| \" | Double quote |
| \\Z | Backslash |

# Reg-Ex Patterns

|  |  |
| --- | --- |
| Pattern | Will match |
| Characters | |
| x | character x |
| \\ | Backslash |
| \t | *tab*('\u0009') |
| \n | newline (<LF>) ('\u000A') |
| \r | carriage-return ('\u000D') |
| Character classes | |
| [abc] | a, b, or c (simple class) |
| [^abc] | Any character except a, b, or c (negation) |
| [a-zA-Z] | a through z or A through Z (range) |
| [a-d[m-p]] | a through d, or m through p (union) |
| [a-z&& [def]] | d, e, or f (intersection) |
| [a-z&& [^bc]] | a through z, except for b and c: [ad-z] (subtraction) |
| Predefined character classes | |
| . (period) | Any char (including line terminators) |
| \d | digit: [0-9] |
| \s | whitespace: [\t\n\x0B\f\r] |
| \w | *W*ord character: [a-zA-Z\_0-9] |
| Boundary matchers | |
| ^ | The beginning of a line |
| $ | The end of a line |
| \b | A word boundary |
| Greedy quantifiers | |
| X? | X, once or not at all |
| X\* | X, zero or more times |
| X+ | X, one or more times |
| X{n} | X, exactly n times |
| X{n,} | X, at least n times |
| X{n,m} | X, >= n but < m times |
| Logical operators | |
| XY | X followed by Y |
| X|Y | Either X or Y |
| (X) | X, as a capturing group |
| /n | Whatever the nth capturing group matched |