Binaries All classes

are defined in the System.IO namespace.

File Handling Enums

filemode

FileMode enumeration - file *opening mode :* •
CreateNew (1) - create a new file (if the file already exists, then an exception is raised
IOException);
• Create (2) - create a new file; if the file already exists, its contents are cleared; • Open (3) -
open an existing file (if the file does not exist, then an exception is raised
FileNotFoundException);
• OpenOrCreate (4) - open an existing file; if the file does not exist, then it is created; • Truncate
(5) - clear the contents of *an existing* file, then open it (if the file is not
exists, a FileNotFoundException is thrown);
• Append (6) – open an existing file for writing and move to its end; if the file does not exist
exists, it is created.

File Access

Enumeration FileAccess - *a way to access* the file: •
Read (1) - read access; • Write (2) - write access; •
ReadWrite (3) - read and write access.

SeekOrigin

The SeekOrigin enumeration is the position from which the file pointer offset in the Seek method is counted:

• Begin (0) - the offset is relative to the beginning of the file (only non-negative
displacement);
• Current (1) - offset is determined relative to the current position of the file pointer; • End (2) -
the offset is relative to the end of the file (only non-positive
offsets).

Binary File Stream: FileStream class

The FileStream class *(file stream)* provides basic file handling capabilities (opening, determining and resizing a file, positioning a file pointer, reading/writing bytes and byte arrays, closing). Basic properties

string Name {get; }
The full name of the file associated with the file stream
this. long Length { get; } long Position { get; set; } The
Length property returns the size of the open file in *bytes,*
the Position property returns and allows you to change the current position of the file pointer (the properties are of type long, so they allow you to store the size and position of the file pointer for files of *9 million terabytes in size).*

Setting the Position property to a value greater than Length does not automatically resize the file. To increase the file size, it is necessary to write new elements to the specified position (in this case, the values of the bytes located between the old and new elements are assumed to be equal to zero). You can also use the SetLength method to increase the file size. Creation: constructor and methods of class File

FileStream(string name, FileMode mode[, FileAccess access]); Creates a FileStream object, associates this object with the file named name, and opens the given file in the mode specified by the mode parameter. If the access parameter is specified, then it specifies how the given file is to be accessed; otherwise, read and write access is established

Sharing a file from multiple file streams is only possible if all those streams are set *to read-only.* If a short file name is specified, then the file is searched for in *the current directory,* i.e. in the working directory of the application.

niya (work directory).

In addition to using the constructor, you can also create an object of type FileStream using the methods of the File class. static FileStream Create(string name); Creates a file named name (or clears the file if it already exists) and opens it for reading and for

writing.

static FileStream OpenRead(string name);
Opens *an existing* file named name for reading. static
FileStream OpenWrite(string name); Opens a file named
name for writing; if the file does not exist, then it is created. The main advantage of the
Create, OpenRead, and OpenWrite methods of the File class is the shorter
the form of their
call. Methods long
Seek(long offset, SeekOrigin origin); Changes the
current position of the file pointer for the file stream this and returns its new position; offset specifies the pointer offset, origin specifies the position in the file from which the offset is measured. Instead of calling this method, just change the Position property:

f.Seek(4, SeekOrigin.Begin) is equivalent to f.Position = 4
f.Seek(4, SeekOrigin.Current) - f.Position += 4 f.Seek(–4,
SeekOrigin.End) - f.Position = f .Length - 4
(assuming the file is at least 4 bytes in size).
void SetLength(long value);
Changes the file size (in bytes) assuming it is equal to value. The value parameter must be non-negative. You can either decrease the file size (removing the last bytes) or increase the file size (adding new null bytes to the end of the file).
int ReadByte();
Reads the value of a byte from the current file position, advances the file pointer to the next byte (i.e. increments the value of the Position property by 1), and returns the value of the byte read converted to an int.

If an attempt is made to read a byte beyond the end of the file, the method returns -1. If the file is open and readable, then executing this method will never raise an exception.
niya.
int Read(byte[] array, int start, int count); Reads count
or fewer bytes from the file stream this (starting at the byte pointed to by the file pointer), writes them sequentially to the elements of the byte array array, starting at the element at index start, and returns the number of bytes actually read. The return value will be equal to count if all required bytes have been successfully read. Otherwise, the return value will be less than the count parameter. After the method executes, the file pointer moves forward by the number of bytes actually read. void WriteByte(byte value); Writes one byte equal to value to the current file stream position this and advances the file pointer to the next byte.

void Write(byte[] array, int start, int count); Writes count
bytes from the byte array array, starting at the element at index start, to the file stream this, starting at the byte pointed to by the file pointer. After the method is executed, the file pointer is moved forward by count bytes. void Flush(); Writes the data contained in the *file buffer to the file,* and then clears the file buffer. Me

The method is automatically called when the file is closed by the Close method.

Close(); Closes

the file associated with file stream this and releases the unmanaged resources allocated nye to work with this file.

When an object associated with a file stream is destroyed (that is, removed from memory), the Close method is automatically called on it. Despite this possibility, *you should always close the file stream as soon as you are done with it by explicitly calling the* Close method.

Calling the Close method again is ignored. After closing the file, you can access the Name property.

Wrapper Streams: BinaryReader and BinaryWriter

The FileStream class discussed in the previous paragraph allows input/output of file data only in the form *of sets of bytes.* To be able to read or write more complex data structures, you need to use "add-ons" on top of a standard file stream: the BinaryReader class *(binary wrapper stream for reading)* or the BinaryWriter class *(binary wrapper stream for writing).* Constructors, General Properties, and Methods BinaryReader(Stream stream[, Encoding encoding]); BinaryWriter(Streamstream[, Encoding encoding]); Each constructor creates a corresponding binary wrapper stream that is associated with the underlying stream. When working with files, an object of the FileStream type is specified as the stream parameter. The stream does not need to be stored in a separate variable beforehand; it is permissible to create it on the fly, specifying as the first parameter a call to the constructor of the FileStream class or a method of the File class. The underlying stream can later be accessed using the BaseStream property of wrapper streams.

The encoding parameter for the BinaryReader class determines *the format for decoding* character data when it is read from a file, and for the BinaryWriter class it specifies *the format for encoding* character data when it is written to a file. If this parameter is not specified, the UTF-8 format is used.

Stream BaseStream { get; } A

read-only property that returns the underlying stream for the this wrapper thread. You should cast this property (of type Stream) to a FileStream type only if you need to access the Name property or the SetLength method of the FileStream class, since all other properties and methods are already defined in the Stream class, which is the ancestor of all stream classes. The base stream can be accessed using the BaseStream property only when the wrapper stream is *open .* void Close(); Closes the underlying BaseStream associated with the this wrapper stream and releases the unmanaged resources allocated to those streams. Re-execution of the Close method is ignored without raising an exception. It *is imperative* to call the Close method of the wrapper stream, since this method (unlike the FileStream class method of the same name) *is not automatically called when an object of type* BinaryReader *or* BinaryWriter is destroyed.

The wrapper thread's Close method *automatically* closes the underlying stream, so you do *not need to explicitly call the underlying thread's Close method after the wrapper thread has closed.* If two wrapper threads are connected to the same file, you cannot close one of them until the other is finished. Reading Data with a BinaryReader Object

In any of the following methods, reading data starts at the current file position (that is, at the position of the file pointer). After any read operation is performed, the file pointer moves forward by the number of bytes read.

When working with character data (of the char and string types), it should be taken into account that they are stored in the file in an encoded form, therefore, in order to read them correctly, you must specify the same *encoding format* when creating the BinaryReader wrapper stream that was used when writing this character data to the file .

For reading each elementary data type, the BinaryReader class provides a special method. bool ReadBoolean(); byte ReadByte();

int ReadInt32(); long
ReadInt64();
doubleReadDouble(); char
ReadChar(); string
ReadString(); Each of the
methods of this group reads one element of the required type from the stream and returns its value (with the exception of the ReadBoolean method, which reads one byte from the stream and returns false if the read byte is 0, and true otherwise).

Attempting to read data past the end of the file throws an EndOfStreamException. The ReadString method first reads information about the length of the string *(in bytes) from the stream,* then reads the specified number of bytes and converts the bytes read into characters, given the encoding format used in the file. Line length information can take from 1 to 5 bytes. For example, if the characters of a string occupy no more than 127 bytes, then the length of the string is encoded in *one* byte, and the value of this byte is equal to the number of bytes (not characters!) of the text. Writing Data Using the BinaryWriter Object

To write data, the BinaryWriter class provides a single method that is overloaded for various types of recorded data.
In any of the above methods, data writing starts from the current file position (that is, from the position of the file pointer). After any data write operation is performed, the file pointer moves forward by the number of bytes written; this may increase the file size.

void Write(bool value); void
Write(numeric_type value); void Write(char
value); void Write(string value); Each of the
methods in this group writes the value of
the value parameter of the corresponding type to the stream. The exception is the value parameter of type bool, instead of which one byte with the value 0 (if the parameter is false) or 1 (if the parameter is true) is written to the file.

When writing a string to a file, information about the length of the string is first written (the length *of the encoded* string *in bytes is indicated),* and then the characters of the string themselves (characters are encoded according to the encoding format defined for the BinaryWriter stream).

Text files. Working with the File System All classes (except
Console) are defined in the System.IO namespace. *Text file streams: the StreamReader and StreamWriter classes There* are two classes for working with *text files :* a *text stream wrapper for reading* StreamReader and *text stream wrapper for writing* StreamWriter.

Creating and Closing Text Streams

Unlike binary wrapper streams (BinaryReader and BinaryWriter), for creating text streams kov and linking them to files, it is enough to specify the name name of the text file:
StreamReader(string name[, Encoding encoding]);
StreamWriter(string name[, bool append[, Encoding encoding]]); In the case of a
StreamReader to read, the specified file must exist. The file opens on read, and the file pointer is set to the beginning of the file.
In the case of a StreamWriter to write, the file may not be present; in this case it is automatically created. The file is opened for writing. If the append parameter is not specified or is false, then the contents of the existing file are cleared; if the append parameter is set to true, then the file is opened *for* appending (its contents are saved and the file pointer is set to the end-of-file marker). If the file is empty, then append can be anything.

If the encoding parameter is specified, then it specifies *the encoding format for the* file data; if this parameter is absent, UTF-8 format is used. To set an encoding format that matches the default ANSI encoding used by Windows, specify Encoding.Default as the encoding parameter.

Sharing a file from several text streams is possible only if all these currents are streams of type StreamReader.

void Close();

Closes the this text stream and releases the unmanaged resources associated with it. The repeated execution of the Close method is ignored. For text streams, you should *always* call the Close method when you're done with them. Reading data from a text stream

In any of the reading methods described below, reading data starts from the current position of the file. After any read operation is performed, the file pointer moves forward by the number of symbols.

int Read();

Reads a single character from the stream and returns its value converted to an int (i.e., the character's *code* in the Unicode table is returned). If an attempt is made to read a character beyond the end of the file, the method returns -1. string ReadLine(); Reads and returns the next line from the text stream this. The end of a line is considered to be the end of the file or the presence of one of two options for *end-of-line markers:* a character with code 10 ('\n') or a pair of characters with codes 13 and 10 ('\r', '\n'); the end-of-line marker is not included in the returned string. If this method is called after reaching the end of the file, then it returns null. string ReadToEnd(); Reads all remaining characters from the text stream this (starting at the current character) and returns them as a single string containing both "regular characters" and end-of-line markers (the end-of-file marker is not included in the returned string). If this method is called after reaching the end of the file, then it returns the empty string "".

When reading data from text files, the following class property is useful StreamReader.

bool EndOfStream { get; }

This property returns true if end of file is reached, false otherwise.

Write data to text stream

To write data, the StreamWriter class provides the Write and WriteLine methods, which are overloaded for various types of data being written. In any of the above writing methods, the data is appended to the end of the file. void Write(object value); void Write(bool value); void *Write(numeric_type* value); void Write(char value); void Write(string value); Each method writes the text representation of the value parameter to the stream (the specified parameter's ToString method is called to get the text representation). If the parameter is null, then nothing is written to the file. void Write(string fmt, params object[] args); This method provides *formatted output* using the fmt format string.

For any of the above Write methods, there is a “paired” WriteLine method with the same set of parameters. The WriteLine method writes the same data to the text stream as its corresponding Write method, and then appends *an end-of-line marker to the stream.* There is also a variant of the WriteLine method with no parameters: void WriteLine(); The end-of-line marker to use is taken from the NewLine property of the StreamWriter class: string NewLine { get; set; } By default, this property returns the string "\r\n".

Standard text stream for I/O: Console

Any .NET console application has a special text stream available for I/O: the *console window.* This window is managed by the Console class, defined in the System.

In .NET windowed applications, a console window is not created by default, but attempts to enter output using the Console class do not result in exceptions being thrown because the Console class is associated with "empty" text streams in this situation. To use the console window in windowed applications, you must specify "Console Application" as the application type. static TextReader In { get; } static TextWriter Out { get; } static TextWriter Error { get; } Each of these properties provides access to the corresponding text stream associated with the console window: In is the standard input stream, Out is the standard output stream, Error is the error message stream.

Standard console streams use the default encoding format for console windows in the current version of Windows; in particular, the Russian version of Windows uses code page 866 "Cyrillic (DOS)". static void SetIn(TextReader newIn); static void SetOut(TextWriter newOut); static void SetError(TextWriter newError); The methods are designed to redirect standard console I/O streams; as

their new values can be specified, for example, text files. static int Read(); static string ReadLine(); These methods are designed to enter characters and strings from the standard input stream In. They work similarly to the corresponding methods of the StreamReader class, but note that text typed in the console window is only passed to the program (and therefore processed by the input methods) when the [Enter] key is pressed. You should also keep in mind that pressing the [Enter] key writes two characters with codes 13 and 10 to the input stream, which can be read by the Read method as normal

symbols.

static void Write(object value); static void Write(bool value); static void *Write(numeric_type* value); static void Write(char value); static void Write(string value); static void Write(string fmt, params object[] args); These methods are designed to output data of various types to the standard output stream Out. In addition, the Console class has WriteLine methods “paired” to Write with the same set of parameters and a WriteLine method without parameters.

Helper classes for working with files, directories, and drives DriveType enumeration

This enumeration defines the *types of logical disks:* • Unknown (0) — unknown type of logical disk; • NoRootDirectory (1) — logical drive that does not have a root directory; • Removable (2) — device for removable media; • Fixed (3) — local hard drive; • Network (4) — network drive; • CDRom (5) - a device for reading CDs; • Ram (6) is a virtual disk created in RAM.

Machine Translated by Google

class

The Path class is for manipulating *filenames.* All of its methods are class methods. The main part of the methods is designed to extract the required element from the name of a file or directory. All of these methods have one name parameter of type string and return an object of type string:

• GetPathRoot — name of the root directory (of the form "C:\", "C:" or "\"); • GetDirectoryName - the path to the file, including the drive name, but not containing the trailing "\" character (if name ends with the "\" character, then the string name is returned without the trailing "\" character); • GetFileName - the name of the file along with the extension. If name ends with a delimiter character drive or directory, an empty string is returned;

• GetFileNameWithoutExtension — file name *without extension;*

• GetExtension - file extension, including the preceding dot; • GetFullPath - the full name of the file (if name contains a relative name, then the name of the current directory is added to it along with the name of the drive; if name begins with the character "\", then the name of the current drive is added to it).

static bool HasExtension(string name);

Returns true if the file name name contains a non-empty extension, false otherwise. static string ChangeExtension(string name, string ext); Returns name with extension changed to ext. The ext option can either keep or not contain the initial character ".". Part of the methods of the Path class is related to the creation *of temporary files.* Let's specify one of them: static string GetRandomFileName(); Returns a random string that can be used as the name of a file or directory. The string consists of numbers and lowercase Latin letters and includes the name itself of 8 characters and an extension of 3 characters.

The File and FileInfo Classes

The File and FileInfo classes are designed to treat files as elements of a file system, without access to their contents. The File class contains only class methods; in this case, the first parameter of any method is the name name of the file being processed. The FileInfo class contains properties and instance methods, to access which you need to create an object of this class, specifying the name of the file being processed in its constructor.

Since all methods of the File class have their counterparts (properties or methods) in the FileInfo class, below only the methods of the File class are described.

static void Copy(string name, string newName[, bool overwrite]); Creates a copy of the file name with the name newName. The file name and path specified in newName must exist; newName must not be the name of an existing directory; if overwrite is not specified or is false, then newName cannot be the name of an existing file. If the overwrite parameter is true and the file newName exists, then its contents are replaced with the contents of the file name (if the file newName is writable, an exception is raised). static void Move(string name, string newName); Renames the file name, replacing its name with newName. name cannot be a directory name; the file name and the path specified in newName must exist; newName must not be the name of an existing directory or file. The name newName can be a directory located on another drive. The new filename may be the same as the old one; in this case, the method does nothing. static void Delete(string name); Deletes the file named name. If the file does not exist, then the method does nothing. If a non-existent drive and/or directory is specified in the file name, or if the file exists but cannot be deleted (for example, if it is currently being used by another application), and also if the name of an existing directory is specified as name, then an exception is raised.

static bool Exists(string name);

Returns true if a

file named name exists, false otherwise. In particular, the method returns false if the name parameter is an empty string or the name of *a directory* (even if it exists). This method never throws an exception.

The FileInfo class also contains a number of read-only instance properties that have no counterpart in the File class. Let's list these properties (all of them, except for Length, can also be used in the case when the this object of the FileInfo type is associated not with a file, but with a directory):

• Directory - an object of type DirectoryInfo containing information about the directory that contains file;
• DirectoryName - a string containing the full path to the file (this path does not have to exist; for the terminating character "\" is indicated only in the case of the root directory);
• Extension — a string containing the file extension (a non-empty extension is padded with a dot on the left); • Name — a string containing the file name (with extension) without the preceding path; • FullName - a string containing the full name of the file; • Length is a long number equal to the length of the file in bytes. The Directory and DirectoryInfo Classes

The Directory and DirectoryInfo classes are designed to work with *directories.* The Directory class, like the File class discussed earlier, contains only class methods; in this case, the first parameter of most methods is the name name of the directory being processed. The DirectoryInfo class, like the FileInfo class, contains properties and instance methods that must be accessed by creating an object.
of this class by specifying the directory name in its constructor.

Some methods of the Directory class do not have a match in the DirectoryInfo class. static string GetCurrentDirectory(); static void SetCurrentDirectory(string newName); The GetCurrentDirectory method allows you to determine, and the SetCurrentDirectory method allows you to change the current
directory, i.e. *the application's working directory.*

All other methods of the Directory class have analogues in the DirectoryInfo class (these analogues are not described here). static string[] GetFiles(string name[, string mask[, SearchOption option]]); static string[] GetDirectories(string name[, string mask[,SearchOption option]]); static string[] GetFileSystemEntries(string name[, string mask]); These methods return an array of strings with full names of files (GetFiles method), subdirectories (GetDirectories method), or both files and subdirectories (GetFileSystemEntries method) from the name directory. Returned subdirectory names do not end with "\".

If the mask parameter is specified, then only the names of those files/directories that match the specified *mask are returned;* if the mask parameter is not specified, then its value is assumed to be "*", which matches *any* file/directory names. In addition to the "*" character, which stands for any number of any characters, you can specify ordinary characters in the mask, as well as the "?" character, which stands for exactly one arbitrary character.

If the option parameter of the SearchOption enumerated type is specified, then, depending on its value, the search for files/directories can be performed not (in the specified SearchOption.TopDirectoryOnly directory), but also in all its subdirectories of any nesting level (option

SearchOption.AllDirectories). If the option parameter is absent, then the search is carried out only in the specified
nom directory.

static DirectoryInfo CreateDirectory(string name); Creates the sequence of subdirectories specified in the name string and returns a DirectoryInfo object associated with the created directory. If the specified directory already exists, then the method returns that directory without performing any further action. If the name parameter is the name *of an existing file,* then an exception is thrown. static void Move(string name, string newName); Renames the file or directory name, replacing its name with newName. The file/directory name, as well as the path specified in the newName parameter, must exist; newName must not be the name of an existing that in this method (unlike the Move method of the File class), the new name cannot be the same as the old one, and the renaming must be done within the same disk.

static void Delete(string name[, bool recursive]); If the

recursive parameter is not specified or is equal to false, then the method ensures the removal of *an empty* directory named name (if the directory is not empty, then an exception is raised). If the recursive parameter is true, then the directory being deleted may contain files and subdirectories that are also deleted. If the directory with the specified name does not exist or is read-only, an exception is thrown. An exception is also thrown if the specified directory is the working directory of a running application.

static bool Exists(string name); Returns

true if the directory named name exists, false otherwise. In particular, the method returns false if the name parameter is an empty string or the name *of a file* (even if it exists). This method never throws an exception.
DriveInfo class

The DriveInfo class is designed to obtain information about the computer's *logical drives .* To

obtain an object of type DriveInfo, you can call the class constructor, passing it the drive letter of the desired logical drive, or any valid file or directory name that contains the drive name, as a string parameter. There is also a GetDrives class method (with no parameters) that returns an array of DriveInfo objects associated with *all* logical drives found on the computer. All information about the logical drive associated with an object of type

DriveInfo is available through its properties. The first group includes properties (read-only) that never result in an exception being thrown:

• Name — string containing the name of the root directory of the disk in the format *"<letter>:\",* for example,
"C:\"; • RootDirectory - an object of type DirectoryInfo associated with the root directory of the disk; •
DriveType - an enumeration of the DriveType type that defines *the type* of disk; • IsReady - A Boolean
property that determines whether the specified drive is available. The second group is formed by properties

that make sense only for available disks; if the disk is not available, accessing them results in an exception being thrown. All of these properties, except for the VolumeLabel property,
read-only:

• DriveFormat — a string describing the drive format, for example, "FAT", "FAT32", "NTFS" (hard drives),
"CDFS" (CDs and DVDs), etc.; • TotalSize — disk size in bytes (integer of long type); • TotalFreeSize —
size of free disk space in bytes (integer of long type); • AvailableFreeSize — size of free disk space in bytes,
which is available for the current user (integer of long type);

• VolumeLabel - string with the name of the disk label.

Reading and writing data using methods of the File class

The File class includes a number of methods that allow you to organize reading or writing file data that does not require initial actions to open the file and final actions to close it (these actions are performed automatically).

static byte[] ReadAllBytes(string path); static
string[] ReadAllLines(string path[, Encoding encoding]); static string
ReadAllText(string path[, Encoding encoding]); These methods read the

contents of a file named path and write it to RAM. To read binary data, the ReadAllBytes method is used, which writes the file contents to an array of bytes. To read text data, the ReadAllLines and ReadAllText methods are intended, the first of which writes the file contents to an array of strings, and the second to a single line containing, in addition to text, end-of-line markers. When reading from a text file, you can additionally specify its encoding using the encoding parameter. static void WriteAllBytes(string path, byte[] bytes);

static void WriteAllLines(string path, string[] contents[, Encoding encoding]); static void WriteAllText(string path, string contents[, Encoding encoding]); static void AppendAllText(string path, string contents[, Encoding encoding]); These methods are for writing data to a file named path. If a file with the specified name does not exist, it is created. The WriteAllBytes method writes a set of bytes to the file, while the other methods write contents string data, represented either as an array of strings or as a single string including end-of-line markers. The AppendAllText method, unlike other methods, does not delete the previous contents of the file (new data is appended to the end of the file). All methods associated with writing text data can contain an encoding parameter that specifies the encoding to use when writing.

The use of these methods requires placing all file data in RAM, which, as a rule, turns out to be less efficient than algorithms that perform element-by-element file processing. To reduce the overhead associated with allocating memory, the .NET Framework 4.0 introduced the following variants of methods that process rowsets into the specified set of methods:

static IEnumerable<string> ReadLines(string path[, Encoding encoding]); static void WriteAllLines(string path, IEnumerable<string> contents[, Encoding encoding]); static void AppendAllLines(string path, IEnumerable<string> contents[, Encoding encoding]); In these methods, strings can be placed in any collection that implements the interface IEnumerable<string> (for example, in dynamic List<string> arrays), and, most importantly, when using sequences associated with LINQ technology, it is possible to perform file read/write operations “element by element”. For example, in the case of using the ReadLines method, elements are read from a file not at the moment of executing this method (as when using the ReadAllLines method), but later, when processing each element of the resulting sequence in the foreach loop, which makes it possible to obtain a more efficient *line-by-line* version of processing text files. The new variants of the WriteAllLines and AppendAllLines methods are also more efficient if the string sequence contents they specify is generated during LINQ queries, because in this situation the string elements of the sequence are written to the file *as they are generated,* and there is no need to allocate space in RAM for *simultaneous* storage of all

elements.

These advantages of methods using LINQ sequences are provided by a special property of LINQ queries - their *delayed,* or "lazy" nature.