

# Обзор языка программирования C#

(продолжение)

Гутников С.Е.

# Ввод-вывод в C#

# ПОТОКИ И ВВОД-ВЫВОД

Современные операционные системы осуществляют управление устройствами ввода-вывода при помощи специализированных программ-драйверов, а затем обращаются к ним с помощью средств библиотеки ввода-вывода, предоставляющих максимально единообразную связь с различными источниками и приёмниками данных.

При использовании такой модели, вся входная и выходная информация может рассматриваться как последовательность (поток) байтов, обрабатываемых средствами библиотеки ввода-вывода.

# ПОТОКИ И ВВОД-ВЫВОД

При вводе программа читает байты из потока ввода, при выводе вставляет байты в поток вывода. Байты потоков могут интерпретироваться как текст или как бинарные данные. Система ввода-вывода связывает поток с файлом на диске компьютера или с физическим устройством, таким как, например, консоль.

# ПОТОКИ И ВВОД-ВЫВОД

Процесс работы с файлом из программы делится, обычно на три простых шага:

- создание нового / открытие существующего файла;
- запись данных в файл / чтение данных из файла;
- закрытие файла

# ПОТОКИ И ВВОД-ВЫВОД

Библиотека ввода-вывода C# основана на классах системы .NET Framework определенных в пространстве имен `System.IO` и позволяет решать следующие задачи:

- последовательный ввод-вывод текста  
(классы `StreamReader`, `StreamWriter`)

# ПОТОКИ И ВВОД-ВЫВОД

- последовательный или прямой ввод-вывод байтов и двоичного представления данных  
(классы `FileStream`, `BinaryReader`, `BinaryWriter`)
- создание, удаление и получение свойств файлов и каталогов  
(классы `File`, `FileInfo`, `Directory`, `DirectoryInfo`, `FileSystemWatcher`)

Рассмотрим решение каждой из этих задач ввода-вывода на примерах.

# Текстовый ввод-вывод.

## Чтение из файла

Последовательное чтение текста из файла осуществляется, используя класс `StreamReader`. Файл открывается при помощи конструктора `StreamReader`:

```
StreamReader sr = new  
    StreamReader("file.txt");
```

или метода `OpenText` класса `File`:

```
StreamReader sr =  
    File.OpenText("file.txt");
```



# Текстовый ввод-вывод. Чтение из файла

`StreamReader` по умолчанию принимает значения кодировки UTF-8, а не значения кодовой страницы ANSI для текущей системы. `StreamReader` пытается обнаружить кодировку при просмотре первых трех байтов потока. Он автоматически распознает текст в кодировке UTF-8, Юникод с прямым порядком следования байтов и Юникод с обратным порядком следования байтов, если файл начинается с соответствующих меток порядка следования байтов.

# Текстовый ввод-вывод. Чтение из файла

Значение необходимой кодировки можно указать при вызове конструктора `StreamReader`, например, для чтения текста на русском языке в кодировке Windows-1251:

```
StreamReader sr = new  
    StreamReader("file.txt",  
        System.Text.Encoding.GetEncoding(1251));
```

# Текстовый ввод-вывод.

## Чтение из файла

Метод `StreamReader.Close()` закрывает файл и освобождает все системные ресурсы, связанные с устройством чтения.

После вызова `StreamReader.Close()` все операции в устройстве чтения могут вызывать исключения.

Метод `StreamReader.Close()` неявно вызывается при использовании оператора `using`, например:

# Текстовый ввод-вывод. Чтение из файла

```
using (StreamReader sr =  
        new StreamReader("file.txt"))  
{  
    // читаем файл,  
    // Close() вызывается автоматически  
}  
// в этом месте файл уже закрыт
```

# Текстовый ввод-вывод. Чтение из файла

**Посимвольный ввод** осуществляется методом `StreamReader.Read()`, который выполняет чтение следующего символа из входного потока и перемещает положение потока на одну позицию вперед. `StreamReader.Read()` возвращает символ в виде значения целого типа или -1, если больше нет доступных символов. В следующем примере показано использование метода `Read`:

# Текстовый ввод-вывод. Чтение из файла

```
using System;  
using System.IO;  
  
class Read_example  
{  
    public static void Main()  
    {  
        string filename = "file.txt";  
  
        try {
```

# Текстовый ввод-вывод. Чтение из файла

```
using (StreamReader sr = new
        StreamReader( filename,
                        System.Text.Encoding.
                            GetEncoding(1251) )
    )
for (;;) // бесконечный цикл
{
    int ch = sr.Read(); // очередной символ
    if (ch == -1) break; // конец файла?
    // если да - выход из цикла
    Console.Write((char)ch);
    // выводим прочитанный символ
}
```

# Текстовый ввод-вывод. Чтение из файла

```
}  
catch (Exception e) // исключение при чтении  
{  
    Console.WriteLine(  
        "Ошибка при чтении файла {0}: {1}",  
        filename, e.Message);  
}}}
```



# Текстовый ввод-вывод.

## Чтение из файла

**Построчный ввод** осуществляется методом `StreamReader.ReadLine()`, который выполняет чтение строки символов и возвращает данные в виде строки. Если достигнут конец входного потока, `StreamReader.ReadLine()` возвращает `null`. Во входном потоке строка определяется как последовательность символов, за которыми следует символ перевода строки (`'\n'`), символ возврата каретки (`'\r'`) или за символом возврата каретки сразу следует символ перевода строки (`"\r\n"`). Результирующая строка не содержит завершающий знак возврата каретки или перевода строки.

В следующем примере показано использование метода `ReadLine`:

# Текстовый ввод-вывод. Чтение из файла

```
using System;  
using System.IO;  
  
class ReadLine_example  
{  
    public static void Main()  
    {  
        string filename = "file.txt";  
  
        try  
        {
```

# Текстовый ввод-вывод. Чтение из файла

```
using (StreamReader sr = new
    StreamReader(filename,
        System.Text.Encoding.
            GetEncoding(1251) )
    )
for (;;) // бесконечный цикл
{
    string str = sr.ReadLine();
    // очередная строка
    if (str == null) break; // конец файла?
    // если да - выход из цикла
    Console.WriteLine(str);
}
```

# Текстовый ввод-вывод. Чтение из файла

```
}  
catch (Exception e) // исключение при чтении  
{  
    Console.WriteLine(  
        "Ошибка при чтении файла {0}: {1}",  
        filename, e.Message);  
}  
}}
```

# Текстовый ввод-вывод. Чтение из файла

Текстовый файл можно загрузить в строку начиная с текущей позиции потока и до конца с помощью метода `StreamReader.ReadToEnd()`. Если текущее положение находится в конце потока, возвращается пустая строка (`""`).

В следующем примере показано использование метода `ReadToEnd`:

# Текстовый ввод-вывод. Чтение из файла

```
using System;  
using System.IO;  
  
class ReadToEnd_example  
{  
    public static void Main()  
    {  
        string filename = "file.txt";  
  
        try  
        {
```

# Текстовый ввод-вывод. Чтение из файла

```
using (StreamReader sr = new
    StreamReader(filename,
        System.Text.Encoding.
            GetEncoding(1251) )
    )
{
    string str = sr.ReadToEnd();
    // читаем весь файл в строку
    Console.WriteLine(str);
    // выводим на консоль
}
```

# Текстовый ввод-вывод. Чтение из файла

```
}  
catch (Exception e) // исключение  
{  
    Console.WriteLine(  
        "Ошибка при чтении файла {0}: {1}",  
        filename, e.Message);  
}  
}}
```



# Текстовый ввод-вывод.

## Запись в файл

Последовательная запись в текстовый файл осуществляется при помощи класса `StreamWriter`.

`StreamWriter` разработан для вывода символов в определенной кодировке. Если не указано иначе, `StreamWriter` по умолчанию использует UTF-8 без метки порядка следования байтов.

Чтобы создать `StreamWriter`, используя кодировку UTF-8 и метку порядка следования байтов, следует использовать конструктор, задающий кодировку, например

# Текстовый ввод-вывод. Запись в файл

```
StreamWriter sw = new StreamWriter(  
    "file.txt", // имя файла  
    false, // если файл существует-  
            перезаписать  
    System.Text.Encoding.Default  
); // по умолчанию использует UTF-8
```

# Текстовый ввод-вывод.

## Чтение из файла

Если файл не существует, конструктор создает новый файл. Если файл существует, он может быть либо перезаписан, либо в него могут быть добавлены данные.

В приведенном выше примере, второй параметр определяет, требуется ли добавить в файл данные.

Если файл существует и значение параметра равно `false`, файл перезаписывается.

Если файл существует и значение параметра равно `true`, в файл добавляются данные. В противном случае создается новый файл.

# Текстовый ввод-вывод.

## Запись в файл

Метод `StreamWriter.Close()` закрывает файл и освобождает все системные ресурсы, связанные с устройством записи.

После вызова `StreamWriter.Close()` все операции в устройстве записи могут вызывать исключения.

Метод `StreamWriter.Close()` неявно вызывается при использовании оператора `using`, например:

# Текстовый ввод-вывод. Запись в файл

```
using (StreamWriter sw = new
        StreamWriter("file.txt"))
{
    // здесь читаем файл,
    // Close() вызывается автоматически
}
// в этом месте файл уже закрыт
```

# Текстовый ввод-вывод. Запись в файл

Для вывода всех основных типов данных используются методы `Write` и `WriteLine`.

Для форматирования вывода неявно вызывается метод `String.Format`. При форматировании выходных строк в качестве первого параметра методу `Write/WriteLine` всегда передается строковый литерал, в котором могут содержаться ссылки на далее переданные параметры в виде `{0}`, `{1}`, `{2}` и т.д., где цифра в фигурных скобках означает порядковый номер параметра, начиная с нуля.

# Текстовый ввод-вывод.

## Запись в файл

При форматировании, каждая ссылка на параметр вместе с фигурными скобками, заменяется на текстовое представление соответствующего объекта полученное обращением к его методу `ToString()`.

Ссылки на параметры могут располагаться в любом порядке их номеров.

Можно использовать более сложное форматирование для числовых данных, включая в каждую ссылку на параметр различные **символы форматирования**, некоторые из которых перечислены ниже:

# Текстовый ввод-вывод. Запись в файл

**C** или **c** -для форматирования денежных значений

**D** или **d** -для форматирования десятичных чисел, можно также задавать минимальное количество цифр для представления значения.

**E** или **e** -для экспоненциального представления

**F** или **f** -для представления числовых данных в формате с фиксированной точкой, можно также задавать минимальное количество цифр для представления значения



# Текстовый ввод-вывод. Запись в файл

**N** или **n** -для числового форматирования (с запятыми)

**X** или **x** -для представления числовых данных в шестнадцатеричном формате.

# Текстовый ввод-вывод.

## Запись в файл

Признак конца строки записывается в текстовый поток всякий раз при вызове одного из методов `WriteLine`. Признак конца строки по умолчанию - это символ возврата каретки, за которым следует символ перевода строки ("`\r\n`"). Текущее значение признака конца строки хранится в строковом свойстве `NewLine`. Можно присвоить собственное значение свойству `NewLine`, если нужен признак конца строки отличный от принятого по умолчанию.

В следующем примере показано использование методов `Write`, `WriteLine` и свойства `NewLine`:

# Текстовый ввод-вывод. Запись в файл

```
using System;  
using System.IO;  
  
class Write_example  
{  
    public static void Main()  
    {  
        string filename = "filew.txt";  
  
        try  
        {
```

# Текстовый ввод-вывод. Запись в файл

```
using (StreamWriter sw = new
        StreamWriter(filename, false,
            System.Text.Encoding.Default))
{ // Назначаем признак в стиле UNIX:
  sw.NewLine = "\n";
  // вывод текста
  sw.WriteLine("Строка текста");
  // вывод целого числа в шестнадцатичном
  int intg = 1234;
  sw.WriteLine("0x{0:X}", intg);
```

# Текстовый ввод-вывод.

## Запись в файл

```
// вывод действительного числа
double flt = 1234.098765;
sw.WriteLine(flt); // формат по умолчанию
sw.WriteLine("{0:f3}", flt);
// формат с тремя цифрами после запятой
}}
catch (Exception e)
{
    Console.WriteLine(
        "Ошибка при записи в файл {0}: {1}",
        filename, e.Message);
}}}
```

# Текстовый ввод-вывод. Запись в файл

Содержимое filew.txt после выполнения программы:

Строка текста

0x4D2

1234,098765

1234,099

# Текстовый ввод-вывод. Стандартные потоки

При запуске консольного приложения операционная система предоставляет ему три открытых стандартных потока: *ввода*, *вывода* и *диагностики* (для вывода сообщений об ошибках и т.п.).

В программе C# эти стандартные потоки доступны в пространстве имён `System` с помощью соответствующих свойств класса `Console`: `Console.In`, `Console.Out` и `Console.Error`. `Console.In` представляет собой объект класса `StreamReader`, а `Console.Out` и `Console.Error` - объекты класса `StreamWriter`.

# Текстовый ввод-вывод.

## Стандартные потоки

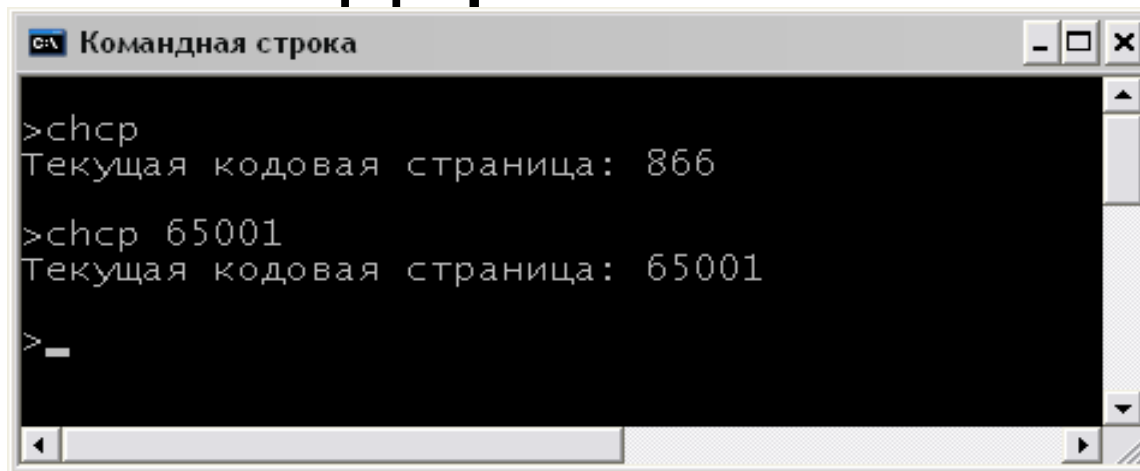
Стандартные потоки принимают по умолчанию кодировку установленную в Windows для окна консоли.

Например, в русской версии Windows XP, по умолчанию устанавливается кодовая страница 866.

Для распечатки текущей кодировки или её изменения можно пользоваться командой `chcp`. В следующем примере сначала печатается значение текущей кодировки окна консоли, затем устанавливается кодировка UTF-8 (кодовая страница 65001).



# Текстовый ввод-вывод. Стандартные потоки



```
Командная строка

>chcp
Текущая кодовая страница: 866

>chcp 65001
Текущая кодовая страница: 65001

>_
```

В программе C# кодировку стандартных потоков можно изменить, присвоив значения свойствам класса `Console` – `InputEncoding` и `OutputEncoding`. `Console.InputEncoding` устанавливает кодировку стандартного входного потока, а `Console.OutputEncoding` – стандартного выходного потока и потока диагностики.

# Текстовый ввод-вывод. Стандартные потоки

Работу со стандартными потоками рассмотрим на примере программы `fnd.cs`.

Программа считывает данные из стандартного входного потока, и находит во входных строках те из них, которые содержат искомую подстроку. Искомая подстрока вводится в качестве параметра программы. Найденные строки выводятся в стандартный выходной поток.

Программа проверяет ошибки в задании параметра, сообщения об ошибках выводит в стандартный поток диагностики.

Для иллюстрации, программа также устанавливает значения кодировки для стандартных потоков в UTF-8.

# Текстовый ввод-вывод. Стандартные потоки

```
using System;

class fnd {
static int Main(string[] args)
{ // Устанавливаем кодировку ст.потоков в UTF-8
    Console.InputEncoding =
        System.Text.Encoding.UTF8;
    Console.OutputEncoding =
        System.Text.Encoding.UTF8;
    // Проверяем что один аргумент
    if (args.Length != 1 ) {
```

# Текстовый ввод-вывод. Стандартные потоки

```
// Выводим сообщение
Console.Error.WriteLine(
    "Ошибка: " +
    "Неверное количество аргументов
программы " );
return 1;
// выходим, возвращаем код завершения:
// 1 означает ошибка
}

string s; // Читаем строку в переменную s
```

# Текстовый ввод-вывод. Стандартные потоки

```
// Проверяем закончились ли входные данные
while ((s= Console.In.ReadLine()) != null)
{
    if (s.IndexOf(args[0]) >= 0)
        // Поиск подстроки args[0]
        Console.Out.WriteLine(s);
        // Выводим s если нашли
}
return 0;
// выходим, возвращаем код завершения:
// 0 показывает отсутствие ошибок
}}
```

# Текстовый ввод-вывод. Стандартные потоки

По умолчанию все стандартные потоки являются символьными и связаны с клавиатурой и консолью, но их можно переадресовывать в файлы или другие устройства ввода-вывода, для этого при запуске программы, в командной строке используются операторы перенаправления.

В следующей таблице перечислены операторы перенаправления стандартных потоков, к которым можно обращаться по номерам: 0-ввод, 1-вывод, 2-диагностика.

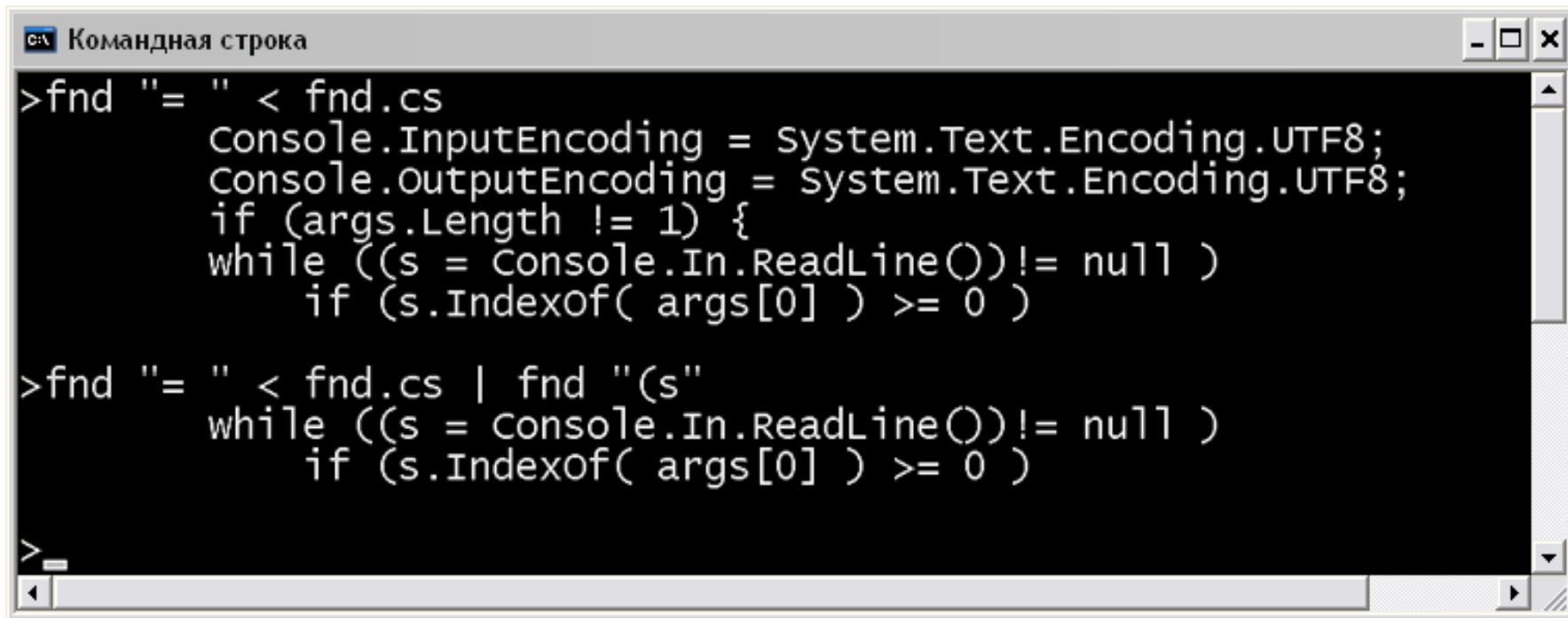
# Текстовый ввод-вывод. Стандартные потоки

Оператор перенаправления	Описание
>	Записывает данные выходного потока в файл или на устройство
<	Читает поток входных данных команды из файла, а не с клавиатуры.
>>	Добавляет данные выходного потока в конец файла, не удаляя при этом существующей информации из файла.
>&	Считывает данные на выходе одного потока как входные данные для другого потока.
<&	Считывает входные данные одного потока как выходные данные другого потока.
	Считывает выходной поток программы и записывает их на вход другой программы (эта процедура известна под названием “канал”)

# Текстовый ввод-вывод.

## Стандартные потоки

Для иллюстрации операторов перенаправления стандартных потоков, запустим представленную ранее программу `fnd` с параметром `"= "` и переопределим её входной поток на исходный текст `fnd.cs`. Затем, вывод предыдущей команды направим на вход программы `fnd` с параметром `"(s"`.



```
Командная строка

>fnd "= " < fnd.cs
    Console.InputEncoding = System.Text.Encoding.UTF8;
    Console.OutputEncoding = System.Text.Encoding.UTF8;
    if (args.Length != 1) {
    while ((s = Console.In.ReadLine()) != null )
        if (s.IndexOf( args[0] ) >= 0 )

>fnd "= " < fnd.cs | fnd "(s"
    while ((s = Console.In.ReadLine()) != null )
        if (s.IndexOf( args[0] ) >= 0 )

>
```



# Текстовый ввод-вывод.

## Стандартные потоки

В завершение обсуждения стандартных потоков, заметим, что класс `Console`, позволяет не ссылаться явно на входной и выходной поток, для методов `Read`, `Write`, `ReadLine` и `WriteLine`. Например, обращение к методу `Console.WriteLine` эквивалентно обращению `Console.Out.WriteLine` (аналогично для метода `Console.Write`), а обращение к методу `Console.ReadLine` эквивалентно `Console.In.ReadLine`. (аналогично для метода `Console.Read`).

Класс `Console` предоставляет ещё одну полезную возможность – переопределение в программе стандартного входного и/или выходного потока. Эта возможность реализуется методами `Console.SetIn` и `Console.SetOut`.

# Текстовый ввод-вывод.

## Стандартные потоки

Предположим, что мы разрабатываем программу, которая обрабатывает текст из стандартного входного потока и выводит результат в стандартный выходной поток.

Предположим также, что мы хотим дать пользователю дополнительную возможность, при необходимости, задавать в командной строке имена входного и выходного файлов. Фрагмент кода ниже демонстрирует, как это можно решить с помощью методов `Console.SetIn` и `Console.SetOut`:

# Текстовый ввод-вывод. Стандартные потоки

```
using System;
using System.IO;
class SomeUtility
{
    static int DoWork() {
        // Здесь выполняем обработку текста и
        Т.П.
        // ...
    }
    static int Main(string[] args) {
        try {
            // Заданы ли имена в командной строке
            if (args.Length > 0) {
```

# Текстовый ввод-вывод. Стандартные потоки

```
// Задан входной файл
using ( StreamReader istr = new
        StreamReader(args[0])) {
    Console.SetIn(istr);
    // переопределили ст.входной
    файл

    // Задан выходной файл
    if (args.Length > 1) {
```

# Текстовый ввод-вывод. Стандартные потоки

```
using (StreamWriter ostr = new
    StreamWriter(args[1])) {
    Console.SetOut(ostr);
    //переоопределили ст.выход
    return DoWork();
    //работа: переопределены
    //два ст.потока
}
} else return DoWork();
//работа: переопределен входной поток
}
```

# Текстовый ввод-вывод. Стандартные потоки

```
    } else return DoWork();  
    //работа: без переопределения  
    ст.потоков  
}  
catch ( Exception e) {  
    /* Обработка ошибок */  
}  
return 1;  
//возвращаем код сигнализирующий ошибку  
}}
```

# Ввод-вывод байтов

Класс `FileStream` предоставляет поток байтов с последовательным или прямым доступом. Для создания байтового потока связанного с файлом, создаётся объект класса `FileStream`. В этом классе определено несколько перегруженных конструкторов. Вот самые распространенные среди них:

```
public FileStream( string path,  
    FileMode mode,      FileAccess access,  
    FileShare share, int bufferSize );
```

# Ввод-вывод байтов

```
public FileStream( string path,  
    FileMode mode, FileAccess access,  
    FileShare share );
```

```
public FileStream( string path,  
    FileMode mode, FileAccess access );
```

```
public FileStream( string path,  
    FileMode mode );
```

Параметры перечисленных выше конструкторов и их назначение перечислены ниже:



# Ввод-вывод байтов

`string path`

Абсолютный или относительный путь к файлу

`FileMode mode`

Способ открытия или создания файла, может принимать значения:

`FileMode.CreateNew` – Указывает, что операционная система должна создавать новый файл. Если файл уже существует, создается исключение `IOException`.

# Ввод-вывод байтов

`FileMode.Create` – Указывает, что операционная система должна создавать новый файл. Если файл уже существует, он будет переписан.

`FileMode.Open` – Указывает, что операционная система должна открыть существующий файл. Если данный файл не существует, создается исключение `FileNotFoundException`.

# Ввод-вывод байтов

`FileMode.OpenOrCreate` – Указывает, что операционная система должна открыть файл, если он существует, в противном случае должен быть создан новый файл.

`FileMode.Truncate` – Указывает, что операционная система должна открыть существующий файл. После открытия должно быть выполнено усечение файла таким образом, чтобы его размер стал равен нулю.

`FileMode.Append` – Открывает файл, если он существует, и переводит указатель в конец файла, или же создает новый файл.

# Ввод-вывод байтов

`FileAccess` access

Определяет, каким образом объект может получить доступ к файлу, может принимать значения:

`FileAccess.Read` – данные можно прочитать из файла

`FileAccess.Write` – данные можно записать в файл

`FileAccess.ReadWrite` – данные можно записать в файл и прочитать из файла

# Ввод-вывод байтов

`FileShare` share

Определяет, каким образом файл будет совместно использоваться процессами, может принимать значения:

`FileShare.None` – отклоняет совместное использование текущего файла

`FileShare.Read` – разрешает последующее открытие файла для чтения

`FileShare.Write` – разрешает последующее открытие файла для записи

# Ввод-вывод байтов

`FileShare.ReadWrite` – разрешает последующее открытие файла для чтения или записи

`FileShare.Delete` – разрешает последующее удаление файла

`int bufferSize`

Определяет размер буфера, для значений от 0 до 8 фактический размер буфера устанавливается равным 8 байт.

# Ввод-вывод байтов

Метод `FileStream.Close()` закрывает файл и освобождает все системные ресурсы, связанные с файлом или устройством. После вызова `FileStream.Close()` все операции с объектом могут вызывать исключения. Метод `FileStream.Close()` неявно вызывается при использовании оператора `using`, например:

```
using (FileStream fs = new FileStream(
    "file.bin", FileMode.Open)) {
    // здесь читаем,
    // Close() вызывается автоматически
}
// в этом месте файл уже закрыт
```

# Ввод-вывод байтов

**Побайтный ввод-вывод** осуществляется с помощью методов `FileStream.ReadByte()` и `FileStream.WriteByte()`.

Метод `FileStream.ReadByte()` выполняет чтение следующего символа из потока и перемещает положение потока на одну позицию вперед. `FileStream.ReadByte()` возвращает байт в виде значения целого типа или -1, если достигнут конец файла.

В следующем примере выполняется побайтное копирование файла, имена файла источника и приёмника вводятся как аргументы программы:



# Ввод-вывод байтов

```
using System;
using System.IO;
class FCopy_example
{
    static int Main(string[] args)
    {
        // Проверяем что два аргумента
        if (args.Length != 2) {
            // Выводим сообщение
            Console.Error.WriteLine(
                "Ошибка: " +
                "Неверное количество
                аргументов");
            return 1; // Выходим по ошибке
        }
    }
}
```

# Ввод-вывод байтов

```
try {  
    // открываем входной файл для  
    // последовательного чтения  
        using (FileStream fsrc = new  
                FileStream( args[0],  
                            FileMode.Open,  
                            FileAccess.Read )  
            )  
  
    // открываем выходной файл,  
    // если существует - перезаписать
```

# Ввод-вывод байтов

```
using (FileStream ftarg = new
    FileStream( args[1],
        FileMode.Create,
        FileAccess.Write))
{
    int b; // читаем до конца файла
    while ( (b = fsrc.ReadByte()) != -1 )
        ftarg.WriteByte( (byte) b );
    // выводим байт в выходной файл
    return 0; //нет ошибок
}
```

# Ввод-вывод байтов

```
} catch (Exception e) {  
    // исключения при копировании файла  
    Console.WriteLine(  
        "Ошибка копирования " +  
        "файла {0} в {1}: {2}",  
        args[0], args[1],  
        e.Message);  
}  
return 1; //код завершения по ошибке  
}  
}
```

# Ввод-вывод байтов

**Блочный ввод-вывод** осуществляется с помощью методов `FileStream.Read()` и `FileStream.Write()`.

Метод

```
int FileStream.Read(byte[] array,  
                    int offset, int count)
```

выполняет чтение блока байт размером `count` в буфер `array` с позиции `offset` и возвращает количество байтов считанных в буфер.

# Ввод-вывод байтов

Метод

```
void FileStream.Write(byte[] array,  
                      int offset, int count)
```

записывает блок байтов в указанный поток с использованием данных из буфера.

В следующем примере выполняется копирование файла блоками по 1024 байтов; имена файла источника и приёмника вводятся как аргументы программы:

# Ввод-вывод байтов

```
using System;
using System.IO;
class FCopyb_example {
    static int Main(string[] args)
    { // Проверяем, что два аргумента
        if (args.Length != 2) {
            // Выводим сообщение
            Console.Error.WriteLine(
                "Ошибка: " +
                "Неверное количество
аргументов");
            return 1; // ошибка
        }
    }
}
```

# Ввод-вывод байтов

```
try {  
    // открываем входной файл  
    // для последовательного чтения  
  
    using (FileStream fsrc = new  
        FileStream(args[0],  
            FileMode.Open,  
            FileAccess.Read )  
    )
```



# Ввод-вывод байтов

```
// открываем выходной файл
using (FileStream ftarg = new
    FileStream( args[1],
        FileMode.Create,
        FileAccess.Write)) {
    byte[] Buf = new byte[1024];
    int nread;
    while(( nread = fsrc.Read(
        Buf, 0, 1024 )) > 0 )
        ftarg.Write(Buf, 0, nread);
    return 0;
}
```

# Ввод-вывод байтов

```
}  
catch (Exception e) {  
    // исключение при копировании  
    Console.WriteLine(  
        "Ошибка копирования файла "+  
        "{0} в {1}: {2}",  
        args[0], args[1],  
        e.Message);  
}  
return 1; //завершение с ошибкой  
}  
}
```

# Ввод-вывод байтов

В приведенных выше примерах, продемонстрирован последовательный доступ к файлам.

Прямой доступ к файлу осуществляется при помощи метода:

```
long Seek(long offset, SeekOrigin origin);
```

Функция возвращает новую позицию в потоке, начиная с начала файла;

# Ввод-вывод байтов

первый параметр задаёт новую позицию в потоке, второй параметр указывает точку отсчета для первого параметра и может принимать значения

```
SeekOrigin.Begin, SeekOrigin.Current,  
SeekOrigin.End.
```

При установке позиции за пределами файла, размер файла увеличивается добавлением необходимого числа нулевых байтов.

# Ввод-вывод байтов

Метод `FileStream.Seek()` можно использовать для получения текущей позиции в файле и текущего размера файла, как в следующем фрагмента кода:

```
// открываем существующий файл
```

```
using (FileStream fsrc = new  
    FileStream("file.bin",  
        FileMode.Open,  
        FileAccess.ReadWrite))  
{
```

# Ввод-вывод байтов

```
// сохраняем текущую позицию
long currentPosition = fsrc.Seek(
    0L, SeekOrigin.Current);

...

// получаем размер файла
long currentSize =
    fsrc.Seek(0L, SeekOrigin.End);

...

// восстанавливаем позицию
fsrc.Seek(currentPosition,
    SeekOrigin.Begin);

...

}
```

# Ввод-вывод двоичного представления данных

Двоичные файлы хранят данные в том же виде, в котором они располагаются в оперативной памяти.

Двоичный файл открываются на основе базового потока, в качестве которого чаще всего используется поток `FileStream`.

Для записи в поток используется перегруженный метод `Write`, для чтения из двоичного потока предназначена группа методов `ReadXXXX` (например, `ReadBoolean`, `ReadByte`, `ReadInt32` и т.д.).

# Ввод-вывод двоичного представления данных

В следующем примере создаются несколько объектов-описаний товаров в продуктовом магазине; эти описания товаров сохраняются в файл в двоичном представлении, затем они считываются из файла и печатаются на консоли.



# Ввод-вывод двоичного представления данных

```
using System;
using System.IO;

class Bynary_example
{
    static string filename = "file.dat";
    // Тип данных для описания
    // товара в магазине
    class ProductData
    {
```

# Ввод-вывод двоичного представления данных

```
public const int MaxName = 256;
// Макс. длина наименования товара
private string Name;
// наименование товара
private long Code;      // код товара
private double Price;
// цена за единицу
// инициализация объекта:
public ProductData(string AName,
                   long ACode, double APrice)
{
```

# Ввод-вывод двоичного представления данных

```
        if (AName.Length >
            ProductData.MaxName)
            throw new
                ArgumentException();
        Name = AName;
        Code = ACode;
        Price = APrice;
    }
    // инициализация объекта из файла
    public ProductData(
        BinaryReader file) {
        Read(file);
    }
}
```

# Ввод-вывод двоичного представления данных

```
}  
// чтение из файла  
public void Read(BinaryReader file)  
{  
    Name = file.ReadString();  
    Code = file.ReadInt64();  
    Price = file.ReadDouble();  
}
```

# Ввод-вывод двоичного представления данных

```
// сохранение в файл
```

```
public void Write(BinaryWriter file)
{
    file.Write(Name);
    file.Write(Code);
    file.Write(Price);
}
```

# Ввод-вывод двоичного представления данных

```
// Распечатка в строку текста
public override string ToString()
{
    return Name + " " +
        Code.ToString() + " " +
        Price.ToString();
}
```

# Ввод-вывод двоичного представления данных

```
// Записываем описания товаров в файл
static void WriteData() {
    try {
        using (BinaryWriter file = new
            BinaryWriter( new
                FileStream( filename,
                    FileMode.Create ))) {
            // Создаём несколько описаний
            // товаров и сохраняем в файл
            ProductData p = new ProductData(
                "Чай чёрный DILMAN 50 гр.",
                9312631122268, 9370);
        }
    }
}
```

# Ввод-вывод двоичного представления данных

```
p.Write(file);  
p = new ProductData(  
    "Шоколад чёрный 20 гр.",  
    4810410024338, 1700);  
p.Write(file);  
p = new ProductData(  
    "Зефир ванильный 250 гр.",  
    4810411002236, 5890);  
p.Write(file);  
}  
}
```



# Ввод-вывод двоичного представления данных

```
catch (Exception e)
// исключение во время записи в файл
{
    Console.WriteLine(
        "Ошибка при записи {0}: {1}",
        filename, e.Message);
}
}
// Читаем описания товаров из файла
// и печатаем на консоль
static void ReadData() {
    try {
```

# Ввод-вывод двоичного представления данных

```
using (BinaryReader file = new
    BinaryReader(new FileStream(
        filename, FileMode.Open))) {
    ProductData p = new
        ProductData(file);
    Console.WriteLine(p);
    p = new ProductData(file);
    Console.WriteLine(p);
    p = new ProductData(file);
    Console.WriteLine(p);
}
}
```

# Ввод-вывод двоичного представления данных

```
catch (Exception e)
//исключение при чтении из файла
{
    Console.WriteLine(
        "Ошибка чтения {0}: {1}",
        filename, e.Message);
}
}
static void Main(string[] args) {
    WriteData();
    ReadData();
} }
```

# Ввод-вывод двоичного представления данных

Для сохранения объектов в файле, C# предоставляет специальное средство под названием **сериализация**. При **сериализации** объект преобразуется в массив байтов, который затем сохраняется в файл. Для **сериализации** объекта его необходимо пометить атрибутом `[Serializable]`.

Те поля, которые сохранять не требуется, помечаются атрибутом `[NonSerialized]`.

Для сохранения объектов в двоичном формате используется класс `BinaryFormatter`.

Рассмотрим решение задачи из предыдущего примера с помощью **сериализации**.

# Ввод-вывод двоичного представления данных

```
using System;
using System.IO;
using System.Runtime.Serialization.
    Formatters.Binary;

class Serialize_example
{
    static string filename = "files.dat";

    // Тип данных для описания товара
    [Serializable]
```

# Ввод-вывод двоичного представления данных

```
class ProductData
{
    public const int MaxName = 256;
        // Длина наименования товара
    private string Name;
        // наименование товара
    private long Code;
        // код товара
    private double Price;
        // цена за единицу
}
```

# Ввод-вывод двоичного представления данных

```
// инициализация объекта
public ProductData(string AName,
                    long ACode, double APrice)
{
    if (AName.Length >
        ProductData.MaxName)
        throw new
            ArgumentException();
    Name = AName;
    Code = ACode;
    Price = APrice;
}
```

# Ввод-вывод двоичного представления данных

```
public ProductData()  
{  
}  
// Распечатка в строку текста  
public override string ToString()  
{  
    return Name + " " +  
        Code.ToString() + " " +  
        Price.ToString();  
}  
}
```



# Ввод-вывод двоичного представления данных

```
// Записываем описания товаров в файл
static void WriteData() { try {
    using (FileStream file = new
        FileStream( filename,
            FileMode.Create)) {
        BinaryFormatter bf = new
            BinaryFormatter();
        // Создаём несколько описаний
        // товаров и сохраняем в файл
        ProductData p= new ProductData(
            "Чай чёрный DILMAN 50 гр.",
            9312631122268, 9370);
```

# Ввод-вывод двоичного представления данных

```
        bf.Serialize(file, p);  
        p = new ProductData(  
            "Шоколад чёрный 20 гр.",  
            4810410024338, 1700);  
        bf.Serialize(file, p);  
        p = new ProductData(  
            "Зефир ванильный 250 гр.",  
            4810411002236, 5890);  
        bf.Serialize(file, p);  
    }  
}
```

# Ввод-вывод двоичного представления данных

```
catch (Exception e) {  
    // исключение во время записи в файл  
    Console.WriteLine(  
        "Ошибка записи {0}: {1}",  
        filename, e.Message);  
}  
  
// Читаем описания товаров из файла  
// и печатаем на консоль  
static void ReadData() {  
    try {
```

# Ввод-вывод двоичного представления данных

```
using (FileStream file = new
    FileStream(filename,
        FileMode.Open)) {
    BinaryFormatter bf = new
        BinaryFormatter();
    ProductData p = (ProductData)
        bf.Deserialize(file);
    Console.WriteLine(p);
    p = (ProductData)
        bf.Deserialize(file);
    Console.WriteLine(p);
}
```

# Ввод-вывод двоичного представления данных

```
p = (ProductData)
    bf.Deserialize(file);
Console.WriteLine(p);
}

}
catch (Exception e)
//исключение при чтении из файла
{
    Console.WriteLine(
        "Ошибка чтения {0}: {1}",
        filename, e.Message);
}
```

# Ввод-вывод двоичного представления данных

```
}
```

```
}
```

```
static void Main(string[] args)
```

```
{
```

```
    WriteData();
```

```
    ReadData();
```

```
}
```

```
}
```

# Асинхронный ввод-вывод

Во всех рассмотренных ранее примерах мы использовали так называемый **синхронный** механизм ввода-вывода. Это означает, что при каждом обращении к функциям чтения из файла или записи в файл, выполнение программы приостанавливается, пока не завершится выполнение оператора чтения или записи.

**Асинхронный ввод-вывод** – это механизм, предоставляемый операционной системой Windows. Асинхронный ввод-вывод позволяет программе осуществлять обмен данными с каким-либо устройством ввода-вывода и одновременно с этим выполнять другую полезную работу.

# Асинхронный ввод-вывод

Компьютер выполняет многие действия с данными в памяти и даже работу с дисплеем намного быстрее, чем чтение-запись данных в файлы, в том числе через сеть.

Применение асинхронного ввода-вывода позволяет оптимизировать использование ресурсов и создавать более эффективные приложения.



# Асинхронный ввод-вывод

Механизм асинхронного ввода-вывода реализован в классе байтового потока `FileStream`, при этом для того, чтобы открыть файл в асинхронном режиме, используется специальная версия конструктора:

```
public FileStream( string path,  
                  FileMode mode, FileAccess access,  
                  FileShare share, int bufferSize,  
                  bool useAsync);
```

# Асинхронный ввод-вывод

Первые пять параметров этого конструктора имеют тот же смысл, что и при организации синхронного ввода-вывода (см. подробное описание выше в разделе «Ввод-вывод байтов»). Шестой параметр `useAsync` должен принимать значение `true`.

Асинхронная операция ввода запускается при помощи метода `BeginRead`, в этот метод передаются параметры буфера и ссылка на метод, который вызывается после завершения операции ввода.

# Асинхронный ввод-вывод

После завершения операции ввода необходимо вызвать метод `EndRead`.

Аналогично выполняется асинхронная операция вывода, при этом используются методы `BeginWrite` и `EndWrite`.

В следующем примере демонстрируется использование операций асинхронного ввода-вывода.

# Асинхронный ввод-вывод

```
using System;
using System.IO;
using System.Threading;

public class Async_exam {
    const int BufSize = 1024;
        // размер буфера
    static byte[] byteData =
        new byte[BufSize]; // буфер
    static FileStream fs;
    static IAsyncResult aResult;
        // результат асинхронной операции
```

# Асинхронный ввод-вывод

```
public static void Main(string[] args) {  
    try {  
        // Открываем файл для асинхронного  
        чтения  
        fs = new FileStream(  
            "example.txt",  
            FileMode.Open, FileAccess.Read,  
            FileShare.Read, BufferSize, true);  
  
        // Запуск асинхронного  
        чтения (см. ниже)  
        AsyncRead();  
    }  
}
```

# Асинхронный ввод-вывод

```
// ... Здесь можно выполнять к-либо  
действия  
// одновременно с асинхронным чтением  
  
// Приостанавливаем выполнение программы  
// пока не завершилось асинхронное чтение  
    while (!aResult.IsCompleted)  
    {  
        Thread.Sleep(100);  
    }  
    fs.Close(); // Закрываем файл
```

# Асинхронный ввод-вывод

```
// Открываем для ас.записи
fs = new FileStream(
    "write.txt", FileMode.Create,
    FileAccess.Write,
    FileShare.None, BufferSize,
    true);

// Запуск асинхронной
записи (см. ниже)
    AsyncWrite();

// ... Здесь можно выполнять к-либо действия
// одновременно с асинхронной записью
```

# Асинхронный ВВОД-ВЫВОД

```
// Приостанавливаем выполнение программы
// пока не завершилась асинхронная
запись

    while (!aResult.IsCompleted)
    {
        Thread.Sleep(100);
    }
    fs.Close(); // Закрываем файл
}
catch (IOException err)
// ошибки при чтении/записи
{
```



# Асинхронный ВВОД-ВЫВОД

```
        Console.WriteLine(err.Message);
    }
}

// Обработчик завершения асинхронного чтения
public static void HandleRead(
    IAsyncResult ar) {
    int nbytes = fs.EndRead(ar);
    // Сколько байт прочитано?
    Console.WriteLine(
        "{0} bytes read", nbytes);
}
```

# Асинхронный ВВОД-ВЫВОД

```
// Запуск операции асинхронного чтения
public static void AsyncRead() {
    // Создаём ссылку на обработчик
    AsyncCallback cb = new
        AsyncCallback(HandleRead);
    aResult = fs.BeginRead(byteData,
        0, byteData.Length, cb, null);
}
// Обработчик завершения ас.записи
public static void HandleWrite(
    IAsyncResult ar) {
```

# Асинхронный ВВОД-ВЫВОД

```
fs.EndWrite(ar);  
Console.WriteLine("{0} bytes write",  
    byteData.Length);  
}  
// Запуск операции асинхронной записи  
public static void AsyncWrite() {  
    // Создаём ссылку на обработчик  
    // завершения  
    AsyncCallback cb = new  
        AsyncCallback(HandleWrite);  
    aResult = fs.BeginWrite(byteData,  
        0, byteData.Length, cb, null);  
} }
```

# Делегат

**Делегат** — это тип, который безопасно инкапсулирует метод, т. е. его действие схоже с указателем функции в С и С++. В отличие от указателей функций в С делегаты объектно-ориентированы, строго типизированы и безопасны. Тип делегата задается его именем.

Объявление типа делегата аналогично прототипу метода. Оно имеет возвращаемое значение и некоторое число параметров какого-либо типа:

```
public delegate void  
    TestDelegate (string message) ;  
public delegate int  
    TestDelegate1 (MyType m, long num) ;
```

# Делегат

Экземпляр делегата может инкапсулировать статический метод или метод экземпляра.

Делегаты являются основой событий

Хотя делегат может использовать параметр `out`, с делегатами событий его использование не рекомендуется, так как при этом нельзя определить, какой делегат будет вызван (и, соответственно, сколько раз и в какой последовательности будет изменяться значение параметра `out`). По тем же причинам не рекомендовано использовать в событиях делегаты возвращающие значение.

# Делегат

**Анонимная функция** – это оператор или выражение "inline", которое можно использовать каждый раз, когда ожидается тип делегата. Ее можно использовать для инициализации именованного делегата или подставить вместо типа именованного делегата в качестве параметра метода.

**Лямбда-выражение** — это анонимная функция, которая содержит выражения и операторы и может использоваться для создания делегатов или типов дерева выражений.

Рассмотрим пример:

# Делегат

```
using System;
using System.IO;

namespace ExampleDelegate
{
    // Простой делегат:
    public delegate int IntOp(int x, int y);
    // Обобщённый делегат:
    //      (сравните с C++ template)
    public delegate T Top<T>(T x, T y);
```

# Делегат

```
// простые арифметические операции:
public class Simple
{ // масштаб:
    public int X = 1;
    public int Y = 1;
    // конструктор
    public Simple(int Ax, int Ay) {
        X = Ax; Y = Ay;
    }
    // сумма с масштабированием
    public int Add(int x, int y) {
        return (X * x + Y * y);
    }
}
```



# Делегат

```
// статические умножение и деление
public static int Mul(int x, int y) {
    return (x * y);
}
public static int Div(int x, int y) {
    return (x / y);
}
}

class Program
{
```

# Делегат

```
static void Main(string[] args)
{
    Console.WriteLine(
        "Delegate example:\n");

    // объявляем переменные-делегаты
    // создаём и инициализируем объекты
    Simple ps = new Simple(2, 2);
    IntOp a = new IntOp(ps.Add);
```

# Делегат

```
// анонимный метод
IntOp s = delegate(
    int ax, int ay)
{
    return (ax - ay);
};

// ещё один с лямбда выражением:
IntOp s1 = (int ax, int ay) =>
{
    return (ax - ay);
};
```

# Делегат

```
// ещё короче:
```

```
IntOp s2 = (int ax, int ay) =>  
    (ax - ay);
```

```
TOp<int> m = Simple.Mul;
```

```
TOp<int> d = new  
    TOp<int>(Simple.Div);
```

```
int x = 100, y = 50;
```

```
// вызовы через переменные-делегаты:
```

```
Console.WriteLine(  
    "a({0}, {1}) = {2}",  
    x, y, a(x, y));
```

# Делегат

```
Console.WriteLine("s({0}, {1}) = {2}",  
    x, y, s(x, y));  
Console.WriteLine("s1({0}, {1}) = {2}",  
    x, y, s1(x, y));  
Console.WriteLine("s2({0}, {1}) = {2}",  
    x, y, s2(x, y));  
Console.WriteLine("m({0}, {1}) = {2}",  
    x, y, m(x, y));  
Console.WriteLine("d({0}, {1}) = {2}",  
    x, y, d(x, y));  
}}}
```

# Делегат

Результат работы программы:

Delegate example:

$a(100, 50) = 300$

$s(100, 50) = 50$

$s1(100, 50) = 50$

$s2(100, 50) = 50$

$m(100, 50) = 5000$

$d(100, 50) = 2$