IO

Обмен данными

- Часто программы обмениваются данными.
- Источниками и адресатами данных могут быть файлы, сетевые соединения, блоки памяти, клавиатура и т.д.
- Информация, размещенная в файлах, и информация, полученная из сетевого соединения, обрабатывается, по существу, одинаково.

Система ввода-вывода

- Система, отвечающая за обмен данными, называется системой ввода-вывода.
- В Java реализация системы представлена пакетом *java.io*.
- Обмен данными можно представить в виде:
  - двоичном;
  - символьном;
  - текстовом (в различных кодировках);
  - числовом.

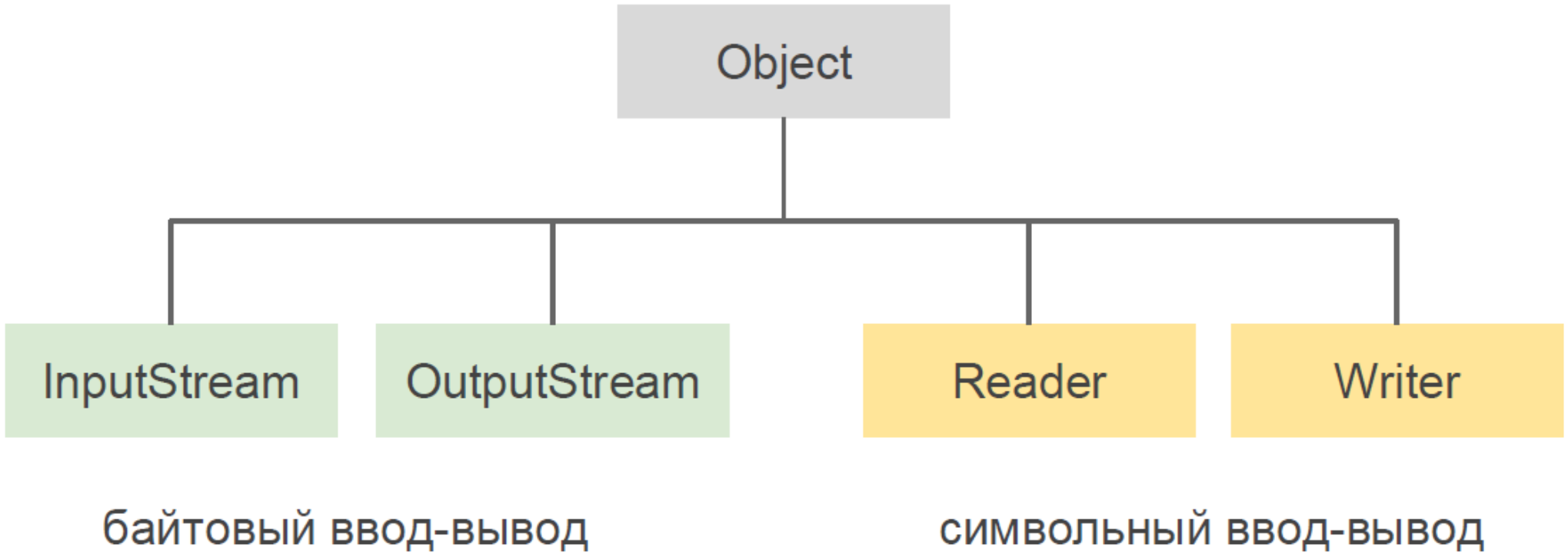
Потоки данных

- Для описания работы механизма ввода-вывода используется понятие потока данных (stream).
- Поток связан с некими источником и приемником данных, способных получать и предоставлять информацию.
- Объект, из которого можно считывать последовательность байтов, называется потоком ввода (*input stream*), а объект, в который эту последовательность данных можно записывать, называется потоком вывода (*output stream*).

Объекты потоков

- Каждая разновидность потока представлена объектом (более 60 типов).
- Разделены на две части: входящие и исходящие потоки.
- Базовые классы считывают и записывают информацию в виде набора байт.
- Если требуется запись данных других типов, то они сначала преобразуются в байтовый массив (сериализуются). При считывании происходит обратное преобразование.

Иерархия классов



Методы InputStream

Методы класса InputStream	Что метод делает
`int read(byte[] buff);`	— метод сразу читает блок байт в буфер (массив байт), пока буфер не заполнится или не закончатся байты там, откуда он их читает. Метод возвращает количество реально прочитанных байт (оно может быть меньше длины массива)
`int read();`	— метод читает один байт и возвращает его как результат. Результат расширяется до int, для красоты. Если все байты уже прочитаны, метод вернет «-1».
`int available();`	— метод возвращает количество непрочитанных (доступных) байт.
`void close();`	— метод «закрывает» поток – вызывается после окончания работы с потоком. Объект выполняет служебные операции, связанные с закрытием файла на диске и т.д. Из потока больше нельзя читать данные.

Методы OutputStream	Что метод делает
`void write(int c);`	— метод записывает один байт информации. Тип int сужается до byte, лишняя часть просто отбрасывается.
`void write(byte[] buff);`	— метод записывает блок байт.
`void write(byte[] buff, int from, int count);`	— метод записывает часть блока байт. Используется в случаях, когда есть вероятность, что блок данных был заполнен не целиком
`void flush();`	— если есть данные, которые хранятся где-то внутри и еще не записаны, то они записываются.
`void close();`	— метод «закрывает» поток – вызывается после окончания работы с потоком. Объект выполняет служебные операции, связанные с закрытием файла на диске и т.д.В поток больше нельзя писать данные, flush при этом вызывается автоматически.

Чтение и запись байтов

Чтение (класс *InputStream*):

abstract int read() // Возвращает -1, достигнув конца потока

Запись (класс *OutputStream*):

abstract int write(int b)

При операциях чтения-записи поток выполнения блокируется.

Метод available()

Чтобы избежать блокировки программы во время операций чтения-записи, следует проверять количество доступных байт в потоке:

```
   int bytesAvailable = System.in.available();
if(bytesAvailable >0)

    {
        byte[] data = new byte[byteAvailable];
        System.in.read(data);
    }
```

Освобождение ресурсов

Завершив чтение или запись в поток, его следует закрыть, вызвав метод *close*:

```
InputStream in = new FileInputStream("file.txt");
in.read();
// другие операции с потоком
in.close();
```

Если приложение открывает слишком много потоков, не закрывая их, системные ресурсы могут оказаться исчерпанными.

Освобождение ресурсов

Закрытие потока вывода очищает буфер, использованный этим потоком.

Если вы забыли закрыть файл, последний пакет байтов, записанный в буфере, может пропасть.

Буфер можно очистить и вручную, воспользовавшись методом *flush*:

```
InputStream in = new FileInputStream("file.txt");
// операции с потоком
in.flush();
```

Пример

```
//Создаем поток-чтения-байт-из-файла
FileInputStream inputStream = new FileInputStream("c:/data.txt");
// Создаем поток-записи-байт-в-файл
FileOutputStream outputStream = new FileOutputStream("c:/result.txt");

byte[] buffer = new byte[1000];
while (inputStream.available() > 0) //пока есть еще непрочитанные байты
{
    // прочитать очередной блок байт в переменную buffer и реальное количество в count
    int count = inputStream.read(buffer);
    outputStream.write(buffer, 0, count); //записать блок(часть блока) во второй поток
}

inputStream.close(); //закрываем оба потока. Они больше не нужны.
outputStream.close();
```

ZipInputStream

`ZipEntry` **`getNextEntry`**`()`	Возвращает объект, описывающий следующий ZipEntry (пропускает все байты текущего entry).
`void` **`closeEntry`**`()`	Закрывает чтение текущего ZipEntry (пропускает все байты текущего entry).
`int` **`available`**`()`	Возвращает 1, если есть доступные ZipEntry, иначе 0.
`int` **`read`**`(byte[] b, int off, int len)`	Читает байты из текущего ZipEntry.
`long` **`skip`**`(long n)`	Пропускает n байт при чтении из потока.
`void` **`close`**`()`	Закрывает поток.

ZipOutputStream

`void` **`setComment`**`(String comment)`	Устанавливает комментарий к архиву.
`void` **`setMethod`**`(int method)`	Указывает метод (тип) сжатия.
`void` **`setLevel`**`(int level)`	Указывает степень сжатия. Чем сильнее, тем медленнее.
`void` **`putNextEntry`**`(ZipEntry e)`	Добавляет новый ZipEntry.
`void` **`closeEntry`**`()`	Закрывает текущий ZipEntry.
`void` **`write`**`(byte[] b, int off, int len)`	Пишет набор байт в текущий ZipEntry.
`void` **`close`**`()`	Закрывает поток.

ZipEntry

Хранит служебную информацию

`String` **`getName`**`()`	Внутреннее имя файла.
`long` **`getTime`**`(), ` **`setTime`**`(long)`	Время последней модификации entry.
`long` **`getCrc`**`(), ` **`setCrc`**`(long)`	Контрольная сумма.
`long` **`getSize`**`(), ` **`setSize`**`(long)`	Размер до компрессии.
`int` **`getMethod`**`(), ` **`setMethod`**`(int)`	Метод сжатия.
`long` **`get/setCompressedSize`**`()`	Размер после архивации.
`boolean` **`isDirectory`**`()`	Является ли entry директорией.

Пример

```
// создаем архив
        FileOutputStream zipFile = new FileOutputStream("c:/archive.zip");
        ZipOutputStream zip = new ZipOutputStream(zipFile);

//кладем в него ZipEntry – «архивный объект»
        zip.putNextEntry(new ZipEntry("document.txt"));

//копируем файл «document-for-archive.txt» в архив под именем «document.txt»
        File file = new File("c:/document-for-archive.txt");
        Files.copy(file.toPath(), zip);

// закрываем архив
        zip.close();
```

Decorator

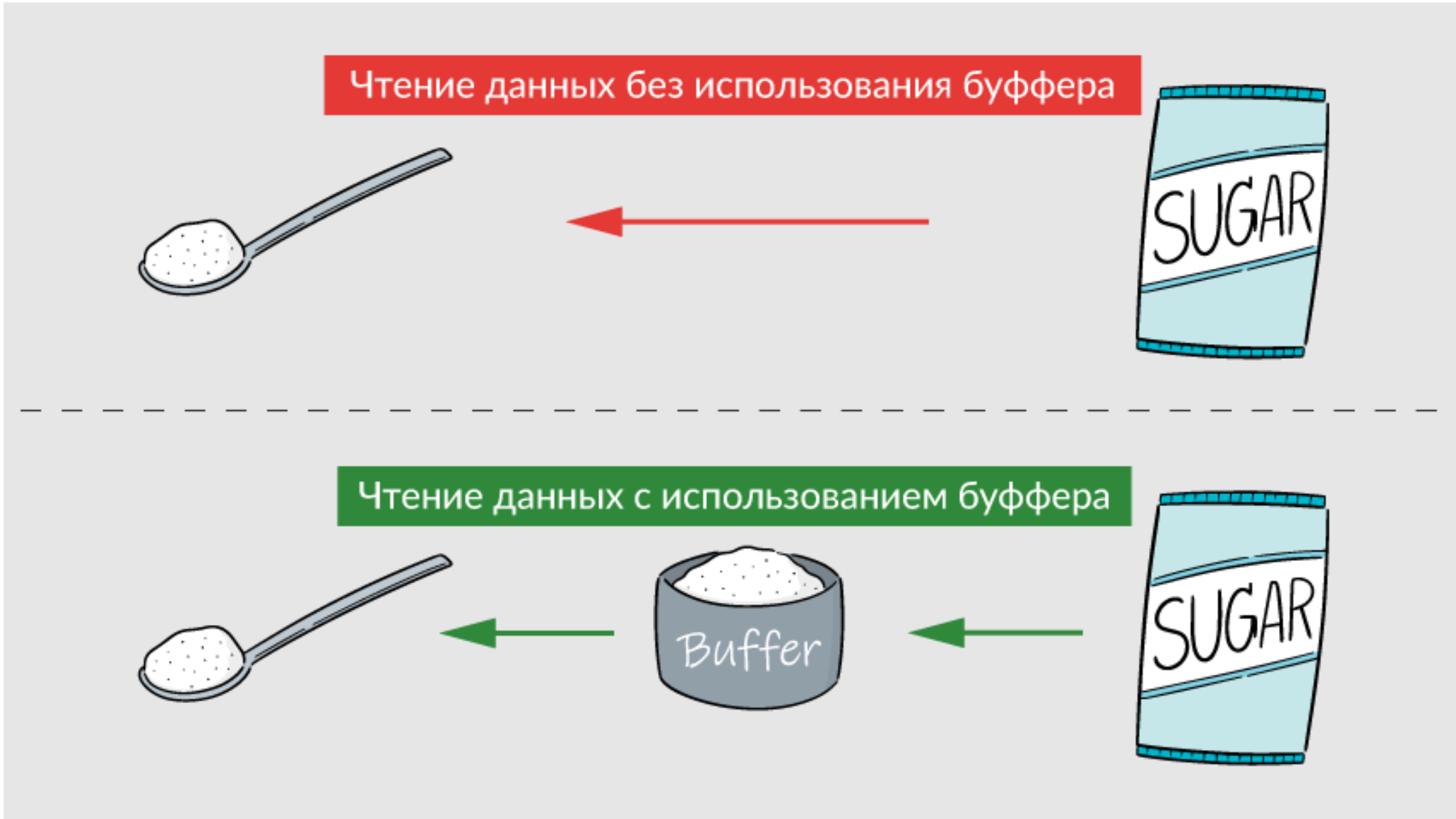
Если мы хотим «обернуть» вызовы методов какого-то объекта своим кодом, то нам нужно:

1) Создать свой класс-обертку и унаследоваться от класса/интерфейса, для которого делаем обертку.

2) Передать оборачиваемый объект в конструктор нашего класса.

3) Переопределить все методы в нашем новом классе, и вызвать в них методы оборачиваемого объекта.

4) Внести свои изменения «по вкусу»: менять результаты вызовов, параметры или делать что-то еще.


Чтение данных без использования буффера
SUGAR
Чтение данных с использованием буффера
Buffer
SUGAR

Reader

Методы класса Reader	Что метод делает
`int read(char[] cbuf);`	— метод сразу читает много символов в буфер (массив символов), пока буфер не заполнится или не закончатся символы там, откуда он их читает. Метод возвращает количество реально прочитанных символов (оно может быть меньше длины массива)
`int read();`	— метод читает один символ и возвращает его как результат. Результат расширяется до int, для красоты. Если доступных символов нет, метод вернет «-1».
`boolean ready();`	— метод возвращает true если есть еще непрочитанные символы для методов read
`void close();`	— метод «закрывает» поток, вызывается после окончания работы с потоком. Объект выполняет служебные операции, связанные с закрытием файла на диске и т.д. Из потока больше нельзя читать данные.

Writer

Метод	Что метод делает
`void write(int c);`	— метод записывает один символ. Тип int сужается до char, лишняя часть просто отбрасывается.
`void write(char[] cbuff);`	— метод записывает массив символов.
`void write(String s);`	— метод записывает строку. Она просто преобразовывается в массив символов и вызывается второй метод.
`void flush();`	— если есть данные, которые хранятся где-то внутри и еще не записаны, то они записываются.
`void close();`	— метод «закрывает» поток – вызывается после окончания работы с потоком. Объект выполняет служебные операции, связанные с закрытием файла на диске и т.д. В поток больше нельзя писать данные, flush при этом вызывается автоматически.

Пример

```
public static void main(String[] args) throws Exception
{
    FileReader reader = new FileReader("c:/data.txt");
    FileWriter writer = new FileWriter("c:/result.txt");

    while (reader.ready()) //пока есть непрочитанные символы в потоке ввода
    {
        int data = reader.read(); //читаем один символ (char будет расширен до int)
        writer.write(data); //пишем один символ (int будет обрезан/сужен до char)
    }

    //закрываем потоки после использования
    reader.close();
    writer.close();
}
```

Фильтры потоков

В Java предусмотрен механизм для разделения двух видов ответственности.

Некоторые потоки (например, *FileInputStream*) могут извлекать из файла байты и другие элементы. Другие потоки (такие как *DataInputStream* и *PrintWriter*) могут объединять байты, создавая более полезные типы данных.

Программист должен объединять эти две разновидности потоков, создавая так называемые профильтрованные потоки.

Фильтры потоков

Для создания фильтра нужно передать существующий поток конструктору другого потока.

Например, для считывания чисел из файла необходимо сначала создать объект класса *FileInputStream*, а затем передать его конструктору класса *DataInputStream*:

```
FileInputStream in = new FileInputStream("file.txt");
DataInputStream din = new DataInputStream(in);
double s = din.readDouble();
```

Таким образом интерфейс потока приобретает новые функциональные возможности

Фильтры потоков

Объединяя дочерние классы потоков, можно создать свой собственный поток.

Например, по умолчанию не предусмотрена буферизация потоков - каждая очередная операция чтения обращается к операционной системе с просьбой выдать очередной байт.

Пример буферизации ввода данных из файла:

```
DataInputStream din = new DataInputStream(
        new BufferedInputStream(
                new FileInputStream("file.txt")));
```

Потоки данных

Часто возникает необходимость записать результат вычислений или прочитать его снова. В потоках предусмотрены методы для повторного считывания данных всех основных типов языка.

Для того чтобы записать число, символ, булевское значение

или строку, используется один из перечисленных ниже методов

интерфейса *DataOutput*:

writeChars	*writeChar*
writeByte	*writeFloat*
writeInt	*writeDouble*
writeShort	*writeBoolean*
writeLong	*writeUTF*

Потоки с произвольным доступом

Класс *RandomAccessFile* обеспечивает чтение и запись данных в любом месте файла. Он одновременно реализует интерфейсы *DataInput* и *DataOutput*.

Файлы, записанные на дисках, обладают произвольным доступом, а потоки данных из сетевых соединений - нет.

RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");

Файл с произвольным доступом имеет указатель файла, который всегда указывает на положение новой записи.

Метод *seek* устанавливает этот указатель на произвольный байт внутри файла.

Пример

```
        //r- read, файл открыт только для чтения
        RandomAccessFile raf = new RandomAccessFile("input.txt", "r");

//перемещаем «курсор» на 100-й символ.
        raf.seek(100);

//читаем строку, начиная с текущего положения курсора и до конца строки
        String text = raf.readLine();

//перемещаем «курсор» на 0-й символ.
        raf.seek(0);
        String text3 = raf.readLine();

//закрываем файл
        raf.close();
```

Запись текста в поток

Для вывода текста следует использовать класс PrintWriter.

Объекты этого класса могут выводить на печать строки и числа в текстовом формате.

PrintWriter out = new PrintWriter(new FileWriter("file.txt"));

Записать данные в объект класса PrintWriter можно с помощью методов *print* и *println*.

Чтение текста из потока

BufferedReader - класс, предназначенный для обработки входных данных в текстовом формате. Объекты этого класса нужно комбинировать с источником данных:

BufferedReader in = new BufferedReader(new FileReader("file.txt"));

Если дальнейший ввод данных невозможен, метод *readLine* возвращает **null**.

```
String line;
while((line = in.readLine()) != null) {
// Обработка строки
}
```

Сохранение объектов

Для того чтобы сохранить данные, сначала необходимо открыть объект класса *ObjectOutputStream*:

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("file.txt"));

Сохранить объект можно с помощью метода *writeObject* класса *ObjectOutputStream*:

Manager boss = new Manager(“Carl”, 80000);

out.writeObject(boss);

Чтение объектов

Чтобы прочитать объект из файла в память, сначала нужно получить объект класса *ObjectInputStream*:

ObjectInputStream out = new ObjectInputStream(new FileInputStream("file.txt"));

Затем нужно извлечь объекты в том же порядке, в котором они были записаны, используя метод *readObject*:

Employee el = (Employee) in.readObject();

При считывании объектов из файла следует внимательно следить за их количеством, порядком и типами.

Serializable

Каждый класс, который записывает и считывает объекты из объектного потока должен реализовывать интерфейс *Serializable*:

class Employee implements Serializable (...)

В интерфейсе Serializable нет методов (маркерный интерфейс), поэтому сам класс изменять не требуется.

Работа с файлами

Возможности, позволяющие работать с файловой системой, инкапсулированы в классах *File, Path, Files*.

Потоковые классы сосредоточивают внимание на содержании файлов, а *File, Path, Files* — на особенностях хранения файлов на диске. Может представлять собой либо файл, либо каталог.

File

`boolean isDirectory()`	Является ли «объект файла» директорией
`boolean isFile()`	Является ли объект файлом
`long length()`	Возвращает размер/длину файла в байтах.
`boolean createNewFile()`	Создает файл. Если такой файл уже был, возвращает false.
`boolean mkdir()`	Создает директорию. Название mkdir происходит от «make directory».
`boolean mkdirs()`	Создает директорию и все поддиректории.
`boolean delete()`	Удаляет файл объекта на диске. Если объект – директория, то только, если в ней нет файлов.
`void deleteOnExit()`	Добавляет файл в специальный список файлов, которые будут автоматически удалены при закрытии программы.

`File createTempFile(`
`String prefix,`
`String suffix,`
`File directory)`
	Создает «временный файл» — файл с случайно сгенерированным уникальным именем – что-типа «dasd4d53sd».
	Дополнительные параметры – префикс к имени, суффикс (окончание). Если директория не указана, то файл создается в специальной директории ОС для временных файлов

`boolean exists()`	Возвращает true, если файл с таким именем существует на диске компьютера.
`String getAbsolutePath()`	Возвращает полный путь файла со всеми поддиректориями.
`String getCanonicalPath()`	Возвращает канонический путь файла. Например, преобразовывает путь «c:/dir/dir2/../a.txt» к пути «c:/dir/a.txt»
`String[] list()`	Возвращает массив имен файлов, которые содержатся в директории, которой является текущий объект-файл.
`File[] listFiles()`	Возвращает массив файлов, которые содержатся в директории, которой является текущий объект-файл.
`long getTotalSpace()`	Возвращает размер диска (количество байт) на котором расположен файл.
`long getFreeSpace()`	Возвращает количество свободного места (количество байт) на диске, на котором расположен файл.
`boolean renameTo(File)`	Переименовывает файл – содержимое файла фактически получает новое имя. Т.е. можно переименовать файл «c:/dir/a.txt» в «d:/out/text/b.doc».
`String getName()`	Возвращает только имя файла, без пути.
`String getParent()`	Возвращает только путь (директорию) к текущему файлу, без самого имени.

Пример

```
//какой-то текущий файл
        File originalFile = new File("c:/path/dir2/a.txt");

//объект-директория
        File folder = originalFile.getParentFile();

//печать списка файлов на экран
        for (File file : folder.listFiles())
        {
            System.out.println(file.getName());
        }
```

Path & Files - new File

Взяли за основу класс **File**, добавили в него немного нового, переименовывали методы и разделили класс на 2. **Path** – это, фактически новый аналог класса **File**, а Files – это утилитный класс (по аналогии с классами Arrays & Collections), в него вынести все статические методы класса File. Так «правильнее» с точки зрения ООП.

Path

`boolean isAbsolute()`	Возвращает true, если путь – абсолютный.
`Path getRoot()`	Возвращает корень текущего пути – директорию самого верхнего уровня.
`Path getFileName()`	Возвращает имя файла из текущего пути.
`Path getParent()`	Возвращает директорию из текущего пути.
`boolean startsWith(Path other)`	Проверяет, что текущий путь начинается с переданного пути.
`boolean endsWith(Path other)`	Проверяет, что текущий путь заканчивается на переданный путь.
`Path normalize()`	Нормализует текущий путь. Например, приводит путь «c:/dir/dir2/../a.txt» к пути «c:/dir/a.txt»
`Path relativize(Path other)`	Вычисляет относительный путь двух путей – «разницу путей»
`Path resolve(String other)`	Восстанавливает абсолютный путь по текущему и относительному.
`URI toUri()`	Возвращает URI текущего пути/файла.
`Path toAbsolutePath()`	Приводит путь к абсолютному, если был относительный.

Files

`Path createFile(…)`	Создает файл на диске.
`Path createDirectory(…)`	Создает директорию.
`Path createDirectories(…)`	Создает директорию и поддиректории.
`Path createTempFile(…)`	Создает «временный файл»
`Path createTempDirectory(…)`	Создает «временную директорию»
`void delete(Path path)`	Удаляет файл/директорию.
`Path copy(Path source, Path target,…)`	Копирует файл.
`Path move(Path source, Path target,…)`	Перемещает файл.
`boolean isSameFile(Path, Path)`	Сравнивает два файла.
`boolean isDirectory(Path)`	Путь — это директория?
`boolean isRegularFile(Path)`	Путь – это файл?
`long size(Path)`	Возвращает размер файла.
`boolean exists(Path)`	Объект с таким именем существует?
`boolean notExists(Path)`	Объект с таким именем не существует?
`long copy(InputStream, OutputStream)`	Копирует байты из InputStream в OutputStream.
`long copy(Path, OutputStream)`	Копирует все байты из Path в OutputStream.