انجمن جاوا کاپ تقدیم می‌کند

دوره برنامه‌نویسی جاوا

# فایل و ورودی/خروجی در جاوا
# Java IO and Files

صادق علی اکبری

# حقوق مؤلف

- کلیه حقوق این اثر متعلق به انجمن جاواکاپ است
- بازنشر یا تدریس آن‌چه توسط جاواکاپ و به صورت عمومی منتشر شده است، با ذکر مرجع (جاواکاپ) بلامانع است
- اگر این اثر توسط جاواکاپ به صورت عمومی منتشر نشده است و به صورت اختصاصی در اختیار شما یا شرکت شما قرار گرفته، بازنشر آن مجاز نیست
- تغییر محتوای این اثر بدون اطلاع و تأیید انجمن جاواکاپ مجاز نیست


# سرفصل مطالب

- ورودی و خروجی برنامه‌ها (IO)
- برنامه‌نویسی فایل
- جریان‌های ورودی و خروجی (Stream)
- خواننده و نویسنده (Reader/Writer)
- مفهوم Serialization
- برنامه‌نویسی تحت شبکه (Socket Programming)
- امکانات نسخه‌های جدید جاوا برای کار با فایل‌ها

# درباره فایل‌ها

# فایل

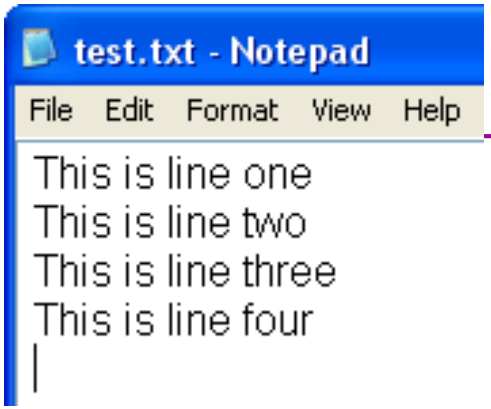
- یک فایل چیست؟
  - مجموعه‌ای از بایت‌ها که در حافظه جانبی ذخیره شده‌اند
- چرا ما به فایل نیاز داریم؟
  - ذخیره‌سازی ماندگار
- فرایند استفاده از یک فایل در یک برنامه چگونه است؟

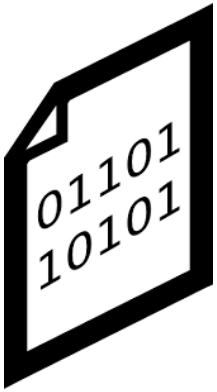
۱- باز کردن (open) ۲- خواندن/نوشتن (read/write) ۳- بستن (close)

(این موارد توسط سیستم‌عامل به برنامه‌ها ارائه می‌شوند)

- انواع فایل‌ها کدامند؟
  - باینری (binary)، متنی (text)

# انواع فایل




- **فایل متنی** (text files)
  - کوچکترین واحد سازنده: کاراکترها
  - مانند فایل‌های txt ، xml ، html و ...
- **فایل باینری** (binary files)
  - واحدهای سازنده: بایت‌ها
  - مانند فایل‌های exe ، zip ، pdf ، docx و ...
- نکته: دسته‌بندی فوق درباره بر نحوه "ذخیره‌سازی" فایل است، نه نمایش آن
  - مثلاً pdf و docx نمایش متنی دارند، ولی به صورت باینری ذخیره می‌شوند
- نکته: فایل‌های متنی هم از بایت‌ها تشکیل می‌شوند
  - هر کاراکتر از یک یا چند بایت تشکیل می‌شود

# کاراکترها در جاوا

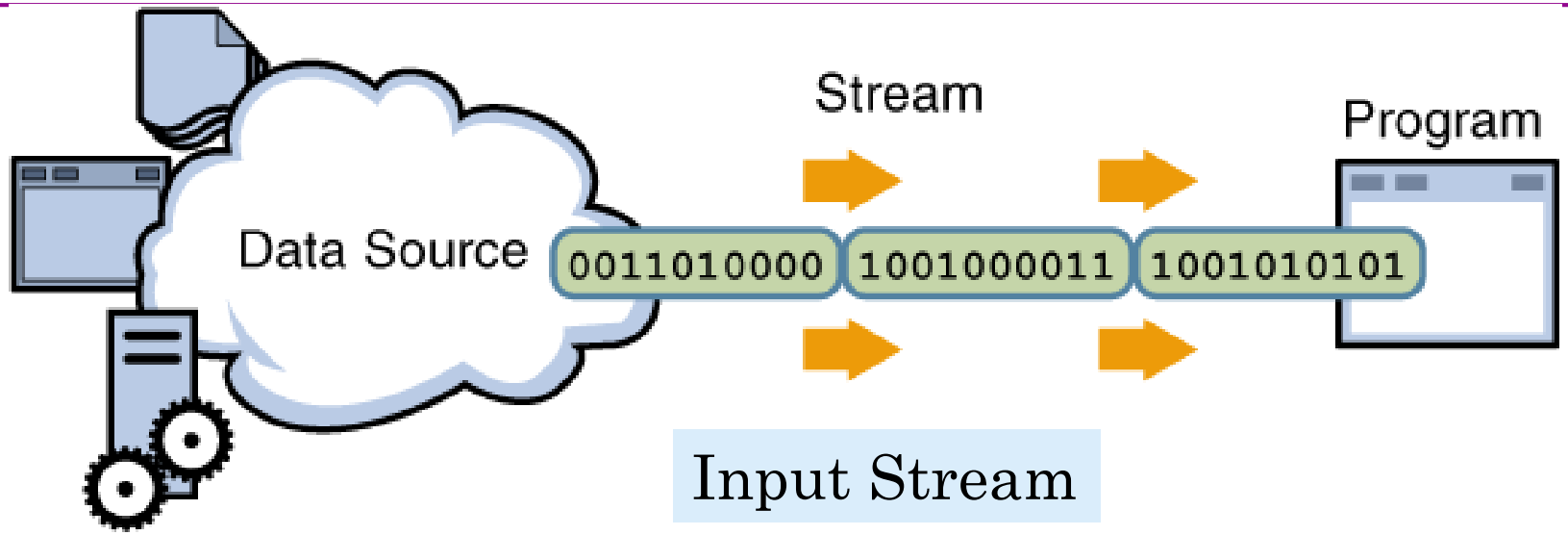
- جاوا از استاندارد Unicode برای کاراکترها پشتیبانی می‌کند
  - استانداردهای قدیمی‌تر مانند ASCII محدود بود
  - کاراکترهایی مثل حروف فارسی و ژاپنی در ASCII پشتیبانی نمی‌شود
- استاندارد Unicode یک مجموعه کاراکتر (character set یا charset) است
- هر کاراکتر، به صورت یک عدد در کامپیوتر ذخیره می‌شود
- نحوه تبدیل کاراکتر به عدد توسط روش‌های کدگذاری (encoding) تعیین می‌شود
  - روش‌های کدگذاری مختلفی برای یونیکد ارائه شده است، مانند UTF-8، UTF-16 و UTF-32
- جاوا از کدگذاری UTF-16 استفاده می‌کند (البته کدگذاری‌های دیگر هم پشتیبانی می‌شود)
  - هر کاراکتر در UTF-16 معمولاً در دو بایت ذخیره می‌شود
- در قدیم، روش‌های کدگذاری دیگری (غیریونیکد) رایج بودند، مثل Windows-1256

# جریان داده (Stream)

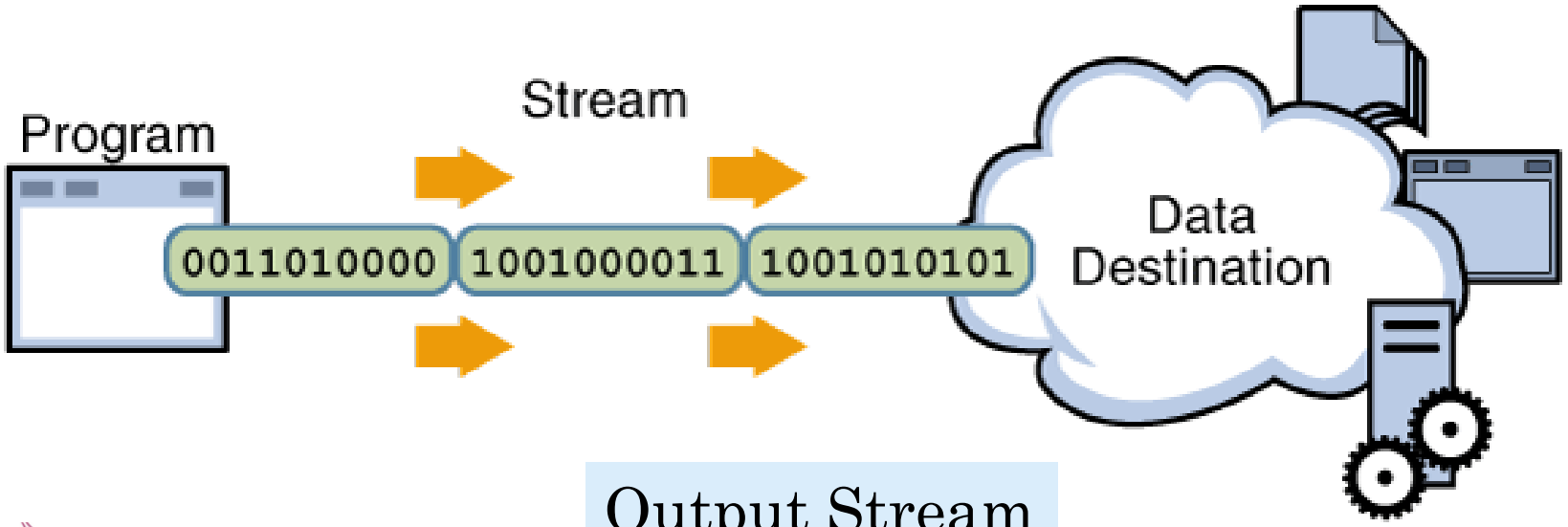
# جریان (Streams)

- جریان اطلاعات به داخل برنامه یا خارج از یک برنامه، مانند:
  - ذخیره و بازیابی از فایل
  - ارسال اطلاعات از طریق شبکه
  - تبادل داده با دستگاه‌های جانبی (مثلاً اسکنر)
  - و ...
- کتابخانه java.io کلاس‌های متنوعی برای کار با فایل‌ها و جریان‌ها دارد
- جریان ورودی به برنامه (آن‌چه برنامه می‌خواند) : Input Stream
- جریان خروجی از برنامه (آن‌چه برنامه تولید می‌کند) : Output Stream

# جریان ورودی و خروجی




Input Stream




Output Stream

# کلاس‌های Java IO

- ورودی و خروجی‌های متنی (جریان متنی)
  - جریانی از کاراکترها
  - امکانات جاوا برای این منظور: کلاس‌های Reader و Writer
  - مثال: خواندن/نوشتن یک فایل txt
    تبادل متن تحت شبکه (مثلاً برنامه چَت)
- ورودی و خروجی‌های باینری (جریان باینری)
  - جریانی از بایت‌ها
  - امکانات جاوا: کلاس‌های InputStream و OutputStream
  - مثال: برای خواندن و نوشتن یک فایل zip یا pdf

# بستن منبع (close)

- بسیاری از کلاس‌های مربوط به کار با فایل‌ها و جریان داده‌ها متد close() دارند
  - در انتهای کار با شیء، باید شیء مربوطه close شود
  - وگرنه، برنامه منابعی گرفته که آزاد نکرده است
- مثلاً باز شدن و بسته شدن فایل: از طریق سیستم‌عامل و سیستم‌فایل انجام می‌شود
  - فایل باز: یک منبع (resource) از طرف سیستم‌عامل که به برنامه تخصیص داده شده
  - این منبع، باید از طریق متد close() آزاد شود
  - اگر آن را نبندیم، یک منبع از سیستم عامل گرفته‌ایم که آزاد نشده است
  - فایل، یک منبع است که باید بعد از باز شدن و در انتهای کار با آن، آزاد شود. وگرنه:
    - تعداد فایل‌های قابل باز کردن محدود است،

    امکان باز کردن آن فایل در برنامه‌های دیگر کمتر می‌شود و ...

# بستن منبع با کمک متد close()

- مهمترین منبعی که برنامه‌ها می‌گیرند: حافظه
  - مثلاً با کمک عملگر new
  - آزادسازی حافظه به صورت خودکار توسط زباله‌روب انجام می‌شود
- اما برنامه‌ها منابع دیگری هم می‌گیرند
  - مانند فایل‌هایی که باز می‌کنند
  - یا سایر جریان‌ها (strream)
- برنامه‌نویس موظف است این منابع را آزاد کند (با کمک متد close)
  - آزادسازی این منابع به صورت خودکار انجام نمی‌شود

# فایل‌های متنی

# کلاس FileReader

- کلاس Reader یک کلاس انتزاعی (abstract class) است
  - کلاس FileReader یکی از زیرکلاس‌های Reader است
  - برای خواندن از "فایل متنی" به کار می‌رود
  - برای مطالعه Reader از کلاس FileReader مثال می‌زنیم:

```
FileReader inf = new FileReader("readme.txt");
int chCode;
while(-1 != (chCode=inf.read()) )
    System.out.println("Next:"+(char)chCode);
inf.close();
```

چرا Reader.read() یک int برمی‌گرداند، نه یک char ؟

پاسخ: برای تشخیص پایان فایل (با 1- مشخص می‌شود)

# کلاس FileReader

- ایجاد شیء: `FileReader fr = new FileReader(file_location);`
- با ایجاد شیء، فایل موردنظر برای خواندن متن باز می‌شود (open)
- خواندن یک کاراکتر: متد read()
- بستن فایل: متد close()
- اما معمولاً فایل متنی را کاراکتر-کاراکتر نمی‌خوانیم
- معمولاً از متد read() استفاده نمی‌شود
- متدهای دیگری برای خواندن حجم بیشتری از اطلاعات وجود دارد. مثال:

`int read(char[ ] cbuf)` (تعداد کاراکترهایی که خوانده‌شده را برمی‌گرداند)

- روشهای دیگری هم وجود دارد که بعداً خواهیم دید

# مثال برای Writer: کلاس FileWriter

- کلاس Writer یک کلاس انتزاعی (abstract class) است
  - کلاس FileWriter یکی از زیرکلاس‌های Writer است
  - برای نوشتن در "فایل متنی" به کار می‌رود. مثال:

```
FileWriter writer = new FileWriter("writeme.txt");
writer.write("This is a line. \n");
writer.write("This is the second line. \n");
writer.close();
```

  - در صورت وجود فایل موردنظر، محتوای آن پاک می‌شود
  - در غیر این صورت، فایل موردنظر ایجاد می‌شود
  - برای اضافه کردن به انتهای یک فایل موجود (append)، آن را این‌گونه ایجاد کنید:

```
FileWriter wr = new FileWriter("text.txt", true);
```

# خطای IOException

- هنگام کار با فایل‌ها و جریان‌ها، ممکن است خطای IOException پرتاب شود
- مثلاً:
  - خواندن از فایلی که وجود ندارد (FileNotFoundException)
  - نقض مجوز دسترسی به فایل
- مثال:

```
try {
   FileWriter writer= new FileWriter("f.txt");
   writer.write("a line. \n");
   writer.close();
} catch (IOException e) {
   e.printStackTrace();
}
```

# جریان باینری

# جریان (Stream)

- در جاوا کلاس‌هایی برای کار با جریان داده (Stream) داریم:
  - کلاس InputStream برای خواندن از جریان داده
  - کلاس OutputStream برای نوشتن در جریان داده
- هر شیء از این کلاس‌ها به یک جریان داده متصل می‌شود
- مثال‌هایی از یک جریان داده:
  - فایل، تبادل اطلاعات در بستر شبکه، تبادل اطلاعات با یک دستگاه جانبی (مثل اسکنر)
  - ورودی و خروجی استاندارد:

```
InputStream is = System.in;
OutputStream os = System.out;
```

# کلاس FileInputStream

- کلاس **FileInputStream** برای خواندن از فایل:

- **FileInputStream** extends **InputStream**

- مثال:

```
List<Byte> list = new ArrayList<>();
FileInputStream inf = null;
try{
    inf = new FileInputStream("file");
    int bCode;
    while(-1 != (bCode=inf.read()) )
      list.add((byte)bCode);
}finally{
    if(inf!=null)
      inf.close();
}
```

# کلاس FileOutputStream

- کلاس **FileOutputStream** برای نوشتن در فایل:

- **FileOutputStream** extends **OutputStream**

```
int[] numbers = {1234567890, 1234567891, 1234567892};
byte[] array = intToByteArray(numbers);//length=12
FileOutputStream out = null;
try{
   out = new FileOutputStream("file");
   out.write(array);
}finally{
   if(out!=null)
     out.close();
}
```

**سؤال**: اگر این اعداد را در یک فایل متنی ذخیره می‌کردیم، چه تفاوتی داشت؟

**پاسخ**: به جای ۱۲ بایت، حداقل ۳۰ بایت اشغال می‌شد

# کلاس File

# کلاس File

- کلاس java.io.File یک کلاس کمکی مفید برای کار با فایل‌ها و فولدرها است
- از این کلاس برای خواندن از فایل و نوشتن در فایل استفاده نمی‌شود
  - شبیه Reader/Writer یا InputStream/OutputStream نیست
- مثال:

```
File f = new File("1.txt");
long length = f.length();
boolean isdir = f.isDirectory();
long lastModified = f.lastModified();
```

## برخی از متدهای کلاس File

```
boolean canRead();
boolean canWrite();
boolean canExecute();
long lastModified();
boolean exists();
boolean isFile() ;
boolean isDirectory() ;
String getName();
String getAbsolutePath() ;
String getParent();
long length() ;//zero for folders
String[] list() ;
```

# نکته: مسیرها و نام فایل‌ها

- بیشتر سیستم‌های عامل ازکاراکتر / برای جدا کردن فولدرها استفاده می‌کنند. مثال:
- /home/ali/file.txt
- اما در سیستم‌عامل ویندوز ازکاراکتر \ برای جدا کردن فولدرها استفاده می‌شود. مثال:
- c:\textfiles\newfile.txt
- از طرف دیگر، کاراکتر \ در جاوا یک کاراکتر خاص (escape character) است
- در تعیین نام فایل یا فولدر در ویندوز دقت کنید
- مثلاً این آدرس: new File ("c:\textfiles\newfile.txt");
  یعنی: c:**{tab}**extfiles**{newline}**ewfile.txt
- راه‌حل: "c:\\textfiles\\newfile.txt" و یا "c:/textfiles/newfile.txt"

# چند کلاس کمکی و مفید

# کلاس RandomAccessFile

- این کلاس نه reader/writer است، و نه inputstream/outputstream
- با کمک این کلاس می‌توانید فایل را به‌صورت باینری یا متنی استفاده کنید
- برای خواندن یا نوشتن
- شیئی از این کلاس دارای یک عدد به عنوان **اشاره‌گر فایل** (file pointer) است
  - محلی از فایل که از آن می‌خوانیم یا در آن می‌نویسیم
- می‌توانید با استفاده از متد seek(long) این اشاره‌گر را جابجا کنید
- متدهای مختلفی برای خواندن یا نوشتن دارد

# کلاس RandomAccessFile

```
RandomAccessFile raf =
                    new RandomAccessFile("1.txt", "rw");
//reads a single byte:
byte ch = raf.readByte();
//reads a 32-bit integer (binary read)
int i = raf.readInt();
//reads text
String line = raf.readLine();
//5 bytes from the beginning of the file
raf.seek(5);
//write text
raf.writeBytes("This will complete the Demo");
//wriet 8-bytes (binary)
raf.writeDouble(1.2);

raf.close();
```

# کلاس Scanner

- کلاسی کمکی است که برای دریافت و پردازش متنی مناسب است
- کلاس Scanner در بسته java.io نیست (در java.util است)
- قبلاً از این کلاس برای دریافت اطلاعات از کاربر استفاده می‌کردیم
- مثال:

```
Scanner s = new Scanner(System.in);
String line = s.nextLine();
int i = s.nextInt();
```

- اکنون می‌دانیم که System.in یک جریان ورودی (InputStream) است
- کلاس Scanner می‌تواند با انواع InputStream ها و Reader ها کار کند

```
s = new Scanner("1.txt");
s = new Scanner(new File("1.txt"));
s = new Scanner(new FileInputStream("1.txt"));
s = new Scanner(new FileReader("1.txt"));
```

# کلاس Formatter

- این کلاس نیز در بسته java.util است
- برای تولید خروجی متنی استفاده می‌شود

```
Formatter f = new Formatter(new FileWriter("1.txt"));
f = new Formatter(new FileOutputStream("1.txt"));
f = new Formatter(new File("1.txt"));
f = new Formatter("1.txt");
f = new Formatter(System.out);
```

- دارای متد format : مشابه printf در C خروجی قالب‌بندی‌شده تولید می‌کند
- مثال:

```
f.format("age=%d,name=%s,grade=%.2f", 20, "Ali", 18.453);
```

```
age=20,name=Ali,grade=18.45
```

# کلاس‌های Closeable

- بسیاری از کلاس‌های مربوط به خواندن/نوشتن از فایل‌ها و جریان‌ها Closeable هستند
  - واسط **Closeable** را پیاده‌سازی می‌کنند که دارای متد close است
- InputStream, OutputStream, Reader, Writer, Scanner, Formatter, Socket, ServerSocket, …
  - در انتهای کار با این اشیاء، باید آن‌ها را close کنیم
- از نسخه ۷ (JDK 1.7)، واسط Closeable زیرواسط AutoCloseable شده‌است
- کلاس‌های AutoCloseable امکان try-with-resources دارند
  - با این امکان، منابع به صورت خودکار در انتهای بلوک try بسته می‌شوند

# امکان try-with-resources

```
FileReader fr = new FileReader("1.txt");
try {
   int read = fr.read();
   ...
}finally {
    if (fr != null) fr.close();
}
```

قبل از جاوا ۷

```
try (FileReader fr = new FileReader("1.txt")){
    int read = fr.read();
    ...
}
```

بعد از جاوا ۷

# کوییز

# سؤال‌های کوتاه

- برای تشخیص اندازه یک فایل از FileReader استفاده کنیم یا File ؟
  - *پاسخ: File*
- کدام کلاس هم برای خواندن و هم برای نوشتن در فایل قابل استفاده است؟
- File, RandomAccessFile, Formatter, InputStream
  - *پاسخ: RandomAccessFile*
- اگر محتوای فایلی که با کمک FileWriter ایجاد شده را با کمک یک FileInputStream بخوانیم، با خطا (exception) مواجه می‌شویم؟
  - *پاسخ: خیر*
  - *هر محتوای متنی را به صورت یک جریان باینری هم می‌توانیم بخوانیم، و برعکس*
  - *اما معمولاً این کار بی‌فایده است*

# تمرین عملی

# تمرین

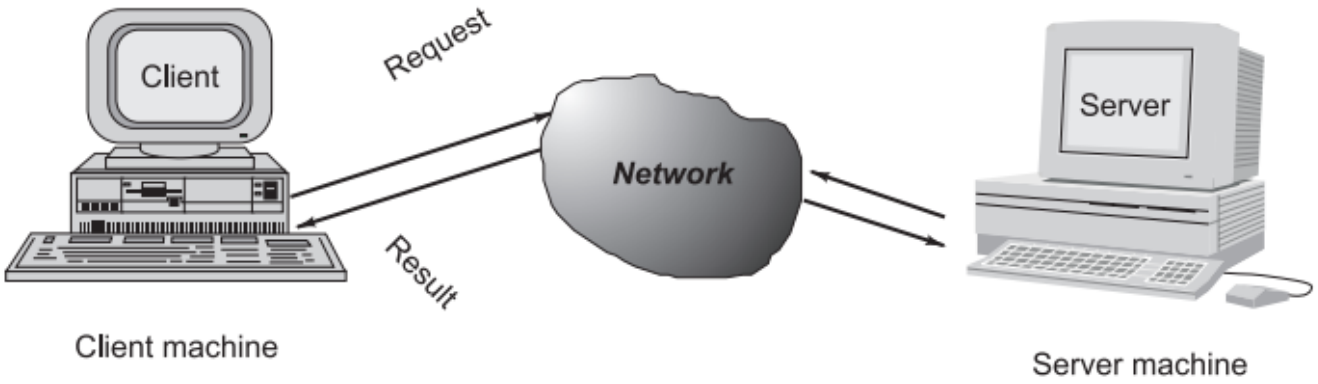
- متدی بنویسید که از یک فایل متنی یک کپی ایجاد کند ولی همه خطوطی که با BAD شروع می‌شوند حذف کند
- کار با کلاس File

# برنامه‌نویسی تحت شبکه

# برنامه‌نویسی شبکه

- روشی که دو برنامه را برای تبادل اطلاعات به هم متصل می‌کند
  - این دو برنامه می‌توانند در دو کامپیوتر مختلف باشند که با شبکه به هم متصلند
- برنامه‌نویسی سوکت (socket programming): رایج‌ترین روش برنامه‌نویسی شبکه
  - برنامه کلاینت (client socket) به برنامه سرور (server socket) متصل می‌شود
  - برای اتصال، باید آدرس (نام یا IP) و شماره پورت (port) برنامه سرور مشخص شود
- معماری مشتری-خدمت‌گذار (client-server)

کاربردها:
بازی تحت شبکه
چَت
انتقال فایل
...



# کلاس Socket

- کلاس‌های جاوا برای برنامه‌نویسی شبکه در بسته java.net قرار دارند
- کلاس java.net.Socket ارتباط بین دو برنامه را ممکن می‌کند
- یک جریان داده (stream) بین دو برنامه ایجاد می‌کند
- مثال: `socket = new Socket("google.com", 80);`
- یک برنامه برای ارسال داده به برنامه دیگر در «خروجی سوکت» می‌نویسد
- `OutputStream out = socket.getOutputStream();`
- برای دریافت داده از برنامه دیگر از «ورودی سوکت» می‌خواند
- `InputStream inp = socket.getInputStream();`
- مشابه نوشتن و خواندن فایل

# مثال: نوشتن در سوکت

```
Socket socket = new Socket("192.168.10.21", 8888);
OutputStream outputStream = socket.getOutputStream();
Formatter formatter = new Formatter(outputStream);
formatter.format("Salam!\n");
formatter.flush();
formatter.format("Chetori?\n");
formatter.flush();
formatter.format("exit");
formatter.flush();
socket.close();
System.out.println("finished");
```

# خواندن از سوکت

```
InputStream inputStream = socket.getInputStream();
Scanner scanner = new Scanner(inputStream);


while(true){
  String next = scanner.next();
  if(next.contains("exit"))
      break;
  System.out.println("Server : " + next);
  System.out.flush();
}
socket.close();
```

# کلاس ServerSocket

- برای اتصال به برنامه دیگر، آن برنامه باید «منتظر» دریافت تماس باشد
  - به این کار، «گوش به زنگ بودن» (listen) می‌گوییم
- کلاس ServerSocket انتظار برای دریافت ارتباط را فراهم می‌کند
  - برای سرور لازم است: مثلاً سرورهای گوگل و یاهو منتظر تماس از طرف کاربران هستند
- هر شیء از نوع ServerSocket روی یک پورت کار می‌کند
  - با فراخوانی متد accept گوش‌به‌زنگ می‌شود
  - هر گاه یک برنامه به آن وصل شود، متد accept پایان می‌یابد
  - با پایان متد accept یک شیء از نوع Socket برای برقراری ارتباط ایجاد می‌شود

```
ServerSocket serverSocket = new ServerSocket(8888);
Socket socket = serverSocket.accept();
```

# مثال برنامه سِرور

```
try(ServerSocket server = new ServerSocket(8765);
Socket socket = server.accept();
Scanner in = new Scanner(socket.getInputStream());
Formatter out = new Formatter(socket.getOutputStream());){
    String next;
    do{
        next = in.next();
        String translate = translate(next);
        out.format(translate+"\n");
        out.flush();
    }while(!next.equals("exit"));
}
```

# مثال: برنامه کلاینت

```
try(Socket socket = new Socket("localhost", 8765);
Scanner socketIn = new Scanner(socket.getInputStream());
Formatter socketOut =
              new Formatter(socket.getOutputStream());
Scanner systemIn = new Scanner(System.in);){
   String next;
   do{
      next = systemIn.next();
      socketOut.format(next+"\n");
      socketOut.flush();
      String received = socketIn.next();
      System.out.println("received: "+received);
   }while(!next.equals("exit"));
}
```

# چند نکته

- برنامه‌های سرور و کلاینت می‌توانند بر روی یک کامپیوتر اجرا شوند
  - بدون اتصال به یک شبکه
  - آدرس کامپیوتر جاری: localhost یا 127.0.0.1
- تبادل اطلاعات با کمک سوکت می‌تواند به صورت باینری باشد

```
byte[] bytes = new byte[]{1,5,127,7};
socket.getOutputStream().write(bytes);
```

- معمولاً برنامه سرور، یک برنامه چندنخی (Multi-Thread) است
  - تا بتواند همزمان به چند کاربر خدمت‌رسانی کند

(در یک جلسه مستقل درباره برنامه‌نویسی چندنخی صحبت می‌کنیم)

# مفهوم Serialization

# مفهوم Serialization

- در جاوا می‌توانیم یک شیء را به یک جریان داده ارسال کنیم
  - مثلاً در یک فایل ذخیره کنیم، یا از طریق شبکه به کامپیوتر دیگری بفرستیم
  - به این فرایند، Serialization می‌گویند
- سپس می‌توانیم از یک جریان ورودی، شیء را بازیابی کنیم
  - مثلاً با کمک فایلی که شیء را ذخیره کرده، آن شیء را دوباره ایجاد کنیم
- به این فرایند، Deserialization می‌گویند (برعکس Serialization)
- در جاوا امکاناتی برای این کارها وجود دارد
- این امکان، برای هر شیئی که واسط **Serializable** را پیاده‌سازی کند، ممکن است
- واسط java.io.Serializable هیچ متدی ندارد:

```
public interface Serializable {}
```

# اشیاء Serializable

- بسیاری از کلاس‌ها در جاوا **Serializable** هستند
  - مثل String ، Integer ، ArrayList و ...
  - اشیاء این کلاس‌ها قابل تبدیل به «جریانی از بایت‌ها» هستند
  - و قابل بازسازی از «جریانی از بایت‌ها» هستند
- عملیات Serialization یعنی تبدیل شیء به یک جریان باینری
- در این عملیات، همه ویژگی‌های درون شیء (یعنی فیلدها) ذخیره می‌شوند
- البته به جز فیلدهایی که با کلیدواژه **transient** مشخص شده باشند
- معنای برچسب transient برای فیلدهای کلاس:
  - هنگام عملیات Serialization فیلدهای transient ذخیره نمی‌شوند

# مثال

```
class User implements Serializable {
   private String username;
   private transient String password;
   ...
}
```

```
class Student implements Serializable {
   private String name;
   private double[] grades;
   private transient double average = 17.27;
}
```

## مثال

```
FileOutputStream f1 = new FileOutputStream("c:/1.txt");
ObjectOutputStream out = new ObjectOutputStream(f1);
Student st = new Student("Ali", new double[]{17.0, 18.0});
System.out.println(st.name);
System.out.println(st.average);
out.writeObject(st);
out.close();
```

```
Ali
17.5
```

```
FileInputStream f2 = new FileInputStream("c:/1.txt");
ObjectInputStream in = new ObjectInputStream(f2);
Student s2 = (Student) in.readObject();
System.out.println(s2.name);
System.out.println(s2.average);
in.close();
```

```
Ali
0.0
```

```
class Student implements Serializable {
    String name;  double[] grades;
    transient double average;
    ...
}
```

# برخی کلاس‌های مهم

# برخی کلاس‌های مهم java.io

- **DataInputStream و DataOutputStream**
  - برای نوشتن و خواندن باینری
  - دارای امکاناتی برای خواندن مقادیر اولیه:

readBoolean, readChar, readDouble, readInt, readFloat, readLong,...

- **BufferedReader و BufferedWriter**
  **BufferedInputStream و BufferedOutputStream**
  - ورودی و خروجی بافر شده. بافر (buffer): تکنیکی برای افزایش کارایی
  - هر عملیات «نوشتن»، لزوماً بلافاصله اجرا نمی‌شود (شاید بافر شود، متد flush)
  - هر «خواندن» شاید به خوانده شدن گسترده منجر شود (ایجاد بافر برای خواندن‌های بعدی)
  - مثلاً خواندن با کمک BufferedReader سریعتر از Scanner است

# برخی کلاس‌های مهم java.io (ادامه)

- ByteArrayInputStream و ByteArrayOutputStream
  - برای خواندن و نوشتن باینری در یک آرایه از بایت‌ها
- StringReader و StringWriter
  - برای خواندن و نوشتن متنی در یک رشته
- PrintStream
  - تولید خروجی متنی در یک OutputStream
  - دارای متدهای متنوع print و println (مثلاً System.out)

# الگوی طراحی Decorator در کلاس‌های java.io

- اشیائی که برای خواندن و نوشتن در جریان‌های داده استفاده می‌شوند می‌توانند در ترکیب و تعامل با هم استفاده شوند
- سازنده (constructor) بسیاری از کلاس‌های io ، امکان دریافت منبعی دیگر را دارند
- مثال:

```
FileOutputStream file = new FileOutputStream("c:/f.txt");
BufferedOutputStream buffer = new BufferedOutputStream(file);
PrintStream print = new PrintStream(buffer);

print.println("salam");
```

- طراحی کلاس‌های io در جاوا از الگوی Decorator استفاده می‌کند
  - یک الگوی طراحی (Design Pattern)

# مثال

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
byte[] bytes;

dos.writeInt(2147483647);
bytes = baos.toByteArray();
System.out.println(bytes.length);   4

baos.reset();

dos.writeDouble(1);
bytes = baos.toByteArray();
System.out.println(bytes.length);   8
```

# کوییز

```java
class User implements Serializable{
    public String username;
    public transient String password;
    public User(String username, String password) {
      this.username = username;
      this.password = password;
    }
}
```

کوییز: خروجی قطعه برنامه زیر چیست؟

root null

```java
byte[] bytes;
try(
ByteArrayOutputStream bos= new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bos);){
    out.writeObject(new User("root", "1234"));
    bytes = bos.toByteArray();
}
try(
ByteArrayInputStream bis=new ByteArrayInputStream(bytes);
ObjectInputStream in = new ObjectInputStream(bis)){
    User des = (User) in.readObject();
    System.out.println(des.username+" "+ des.password);
}
```

# امکانات جدید جاوا در زمینه IO

# امکانات NIO و NIO.2

- بسته java.io از قدیم در جاوا بوده است
- از نسخه ۱.۴ بسته java.nio شامل امکانات جدید (new io) اضافه شد (۲۰۰۲)
- از نسخه ۱.۷ بسته java.nio.file اضافه شد (به آن NIO.2 گفته می‌شود)
- کلاس‌ها و واسط‌های جدیدی در java.nio.file ارائه شدند. مانند:
- Path, Paths, Files
- این امکانات تلاش می‌کنند محدودیت‌های کلاس java.io.File را برطرف کنند
  - قابلیت‌هایی مورد نیاز است که در کلاس File وجود ندارد (مانند کپی فایل‌ها)

# کلاس Paths و واسط Path

- هر دو در بسته‌ی java.nio.file هستند (NIO.2)
- کلاس Paths یک کلاس کمکی است که فقط شامل متد get است
  - متد get یک آدرس می‌گیرد و شیئی از نوع Path برمی‌گرداند

```
Path path = Paths.get("c:/f.txt");
```

- هر شیء از نوع واسط Path ، اطلاعاتی درباره فایل یا فولدر موردنظر دارد
  - برخی از امکاناتی که در کلاس java.io.File دیدیم، در واسط Path وجود دارد
  - در نسخه‌های جدید جاوا بهتر است حتی‌الامکان از Path به جای File استفاده کنیم
  - امکان تبدیل File به Path و برعکس هم وجود دارد
    - متد toFile در واسط Path و متد toPath در کلاس File وجود دارد

```
Path parent = path.getParent();
File f = path.toFile();
```

# کلاس java.nio.file.Files

- کلاسی کمکی دارای متدهای متنوع استاتیک متنوع و مفید
  - برای دریافت ویژگی‌های فایل، کپی فایل، جابجایی، حذف و ...
- برخی از امکانات کلاس قدیمی java.io.File را دارد
  - بهتر است حتی‌الامکان از امکانات Files به جای File استفاده کنیم
- امکانات این کلاس با کمک Path پیاده شده‌اند
- امکانات و متدهای جدیدی نیز دارد
  - برای فایل‌های پیوندی نمادین (symbolic link)
  - ویژگی‌های فایل‌ها
  - و ...

## مثال

```
Path src = Paths.get("/home/ali/src.txt");
if(!Files.exists(src)) return;
Files.createDirectory(Paths.get("/folder/newfolder"));
Files.createSymbolicLink(Paths.get("/home/l.txt"), src);
byte[] bytes = Files.readAllBytes(src);
List<String> lines = Files.readAllLines(src);
boolean writable = Files.isWritable(src);
long size = Files.size(src);
Path copy = Paths.get("c:/folder/copy.txt");
Files.write(copy, bytes);
Files.write(copy, lines, StandardOpenOption.APPEND);
Files.copy(src, Paths.get("c:/folder/dest.txt"));
Files.delete(src);
```

# تمرین عملی

# تمرین عملی

- کار با امکانات NIO.2


aliakbary@asta.ir

# جمع‌بندی

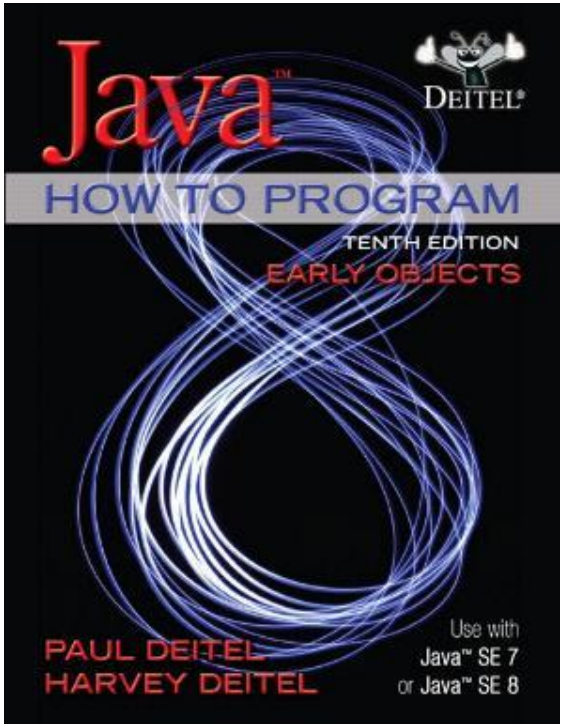
# جمع‌بندی

- برنامه‌نویسی فایل
- مفهوم جریان (stream)
- جریان باینری و جریان متنی
- کلاس‌های جاوا برای برنامه‌نویسی فایل و جریان داده
- File, Reader, Writer, InputStream, OutputStream
- Files, Path, Paths
- مفهوم Serialization
- برنامه‌های شبکه‌ای و Socket Programming

# مطالعه کنید

- فصل ۱۵ کتاب دایتل (Deitel & Deitel) Java How to Program






- تمرین‌های همین فصل از کتاب دایتل

# تمرین

- برنامه‌ای بنویسید که محتوای یک فایل متنی را uppercase کند
- یک برنامه تحت شبکه بنویسید که شیئی از نوع Student را از یک برنامه به برنامه دیگر بفرستد
  - در برنامه فرستنده اطلاعات را از کاربر بگیرید و در گیرنده برای کاربر چاپ کنید
  - راهنمایی: برای ارسال شیء از Serialization استفاده کنید
  - برخی از ویژگی‌های Student (مثل معدل کل) نباید ارسال شوند
    - این ویژگی‌ها را transient کنید
- یک برنامه چَت (chat) متنی بنویسید
  - دو کاربر از دو کامپیوتر مختلف بتوانند با هم چَت کنند

# جستجو کنید و بخوانید

- استاندارد Unicode و روش‌های UTF8 و UTF16 و UTF32
- سایر کلاس‌های I/O جاوا
  - سایر امکانات io و NIO و NIO.2
- الگوی طراحی Decorator
  - کاربرد این الگو در کلاس‌های java.io
  - الگوهای طراحی (Design Pattern)
- مفاهیم شبکه و برنامه‌نویسی شبکه
- کاربردهای serialization چیست؟
  - مثال: RMI

# پایان