Lecture 11
Java SE NIO and JDK 9 Modules/Features

presentation

Java Programming – Software App Development

Cristian Toma

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Agenda for Lecture 11

Buffers, Channels and Selectors

Java New Input Output

1. Java NIO

NIO Components:

Java NIO consist of the following CORE components:

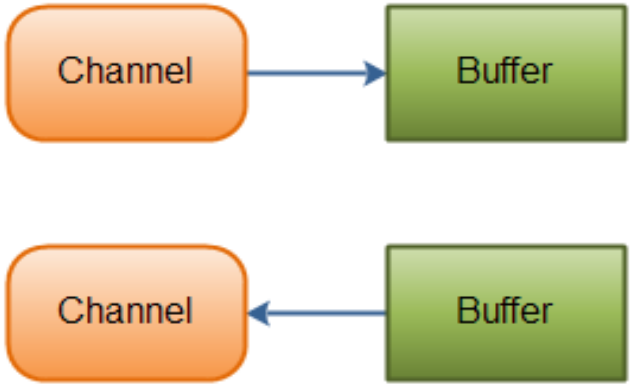
- Channels
- Buffers
- Selectors

Java NIO has more classes and components than these, but the Channel, Buffer and Selector forms the core of the API. The rest of the components, like Pipe and FileLock are merely utility classes to be used in conjunction with the three core components.

1. Java NIO

Channels and Buffers:

Typically, all IO in NIO starts with a Channel. A Channel is a bit like a stream. From the Channel data can be read into a Buffer. Data can also be written from a Buffer into a Channel.



Java NIO: Channels read data into Buffers, and Buffers write data into Channels

There are several Channel and Buffer types. Here is a list of the primary Channel implementations in Java NIO:

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

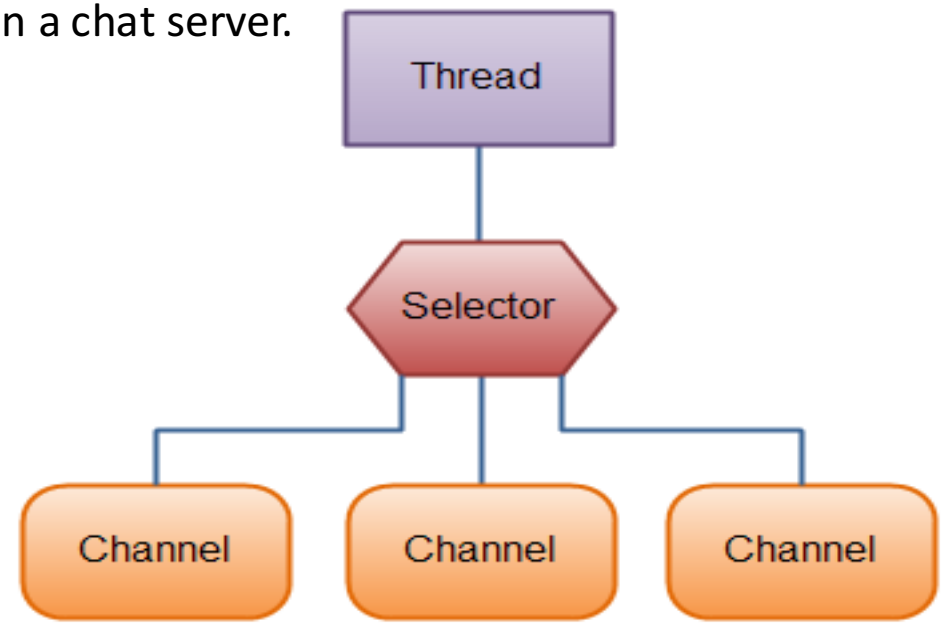
Here is a list of the core Buffer implementations in Java NIO:

- ByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

1. Java NIO

Selectors:

A Selector allows a single thread to handle multiple Channel's. This is handy if an application has many connections (Channels) open, but only has low traffic on each connection. For instance, in a chat server.



Java NIO: A Thread uses a Selector to handle 3 Channel's

To use a Selector the one registers the Channel's with it. Then the one calls it's select() method. This method will block until there is an event ready for one of the registered channels. Once the method returns, the thread can then process these events. Examples of events are incoming connection, data received etc.

1. Java NIO

Channels:

The most important Channel implementations in Java NIO:

- ***FileChannel***
- ***DatagramChannel***
- ***SocketChannel***
- ***ServerSocketChannel***

The ***FileChannel*** reads data from and to files.

The ***DatagramChannel*** can read and write data over the network via UDP.

The ***SocketChannel*** can read and write data over the network via TCP.

The ***ServerSocketChannel*** allows you to listen for incoming TCP connections, like a web server does. For each incoming connection a SocketChannel is created.

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
ByteBuffer buf = ByteBuffer.allocate(48); // 1. allocate the buffer
int bytesRead = inChannel.read(buf);       // 2. write into a buffer from the channel
while (bytesRead != -1) {
    System.out.println("Read " + bytesRead);
    buf.flip();  // 3. change the buffer from write mode to read mode
    while(buf.hasRemaining()) { System.out.print((char) buf.get()); } // 4. read data from the buffer
    //4. alt. read the data from buffer into the channel: int bytesWritten = inChannel.write(buf);
    buf.clear(); // 5. clear the buffer – in conjuction operation for flip
    bytesRead = inChannel.read(buf); // write into a buffer from the channel
}
aFile.close();
```

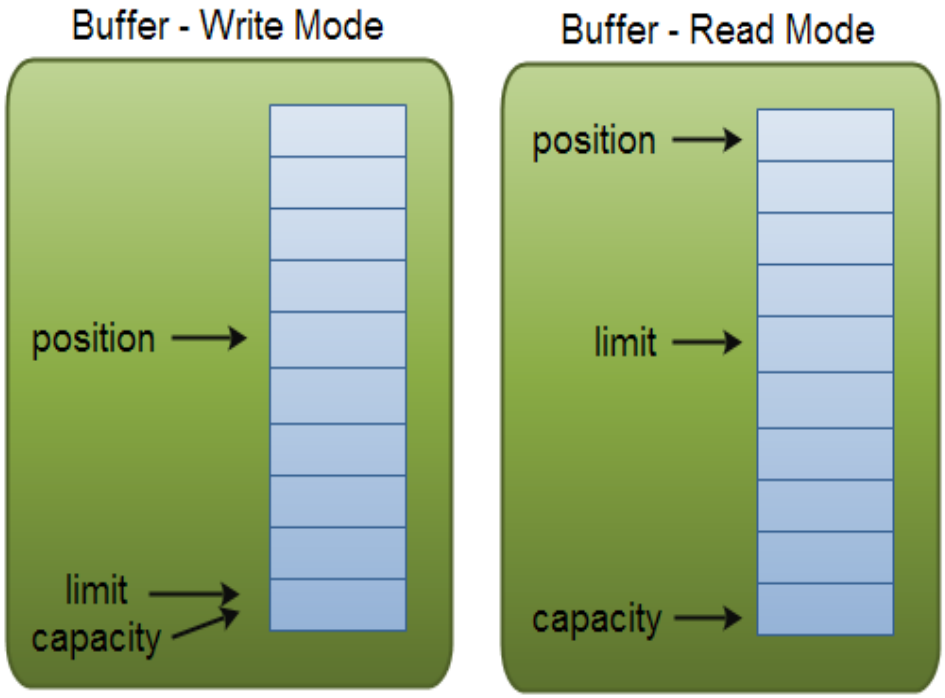
Buffer Capacity, Position and Limit:

A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

A Buffer has three properties you need to be familiar with, in order to understand how a Buffer works. These are:

- capacity
- position
- limit

The meaning of position and limit depends on whether the Buffer is in read or write mode. Capacity always means the same, no matter the buffer mode.



Buffer capacity, position and limit in write and read mode.

Buffer Capacity, Position and Limit:

Capacity

Being a memory block, a Buffer has a certain fixed size, also called its "capacity". You can only write capacity bytes, longs, chars etc. into the Buffer. Once the Buffer is full, you need to empty it (read the data, or clear it) before you can write more data into it.

Position

When you write data into the Buffer, you do so at a certain position. Initially the position is 0. When a byte, long etc. has been written into the Buffer the position is advanced to point to the next cell in the buffer to insert data into. Position can maximally become capacity - 1.

When you read data from a Buffer you also do so from a given position. When you flip a Buffer from writing mode to reading mode, the position is reset back to 0. As you read data from the Buffer you do so from position, and position is advanced to next position to read.

Limit

In write mode the limit of a Buffer is the limit of how much data you can write into the buffer. In write mode the limit is equal to the capacity of the Buffer.

When flipping the Buffer into read mode, limit means the limit of how much data you can read from the data. Therefore, when flipping a Buffer into read mode, limit is set to write position of the write mode. In other words, you can read as many bytes as were written (limit is set to the number of bytes written, which is marked by position).

flip() Buffer Capacity, Position and Limit:

The flip() method switches a Buffer from writing mode to reading mode. Calling flip() sets the position back to 0, and sets the limit to where position just was.

In other words, position now marks the reading position, and limit marks how many bytes, chars etc. were written into the buffer - the limit of how many bytes, chars etc. that can be read.

Flips this buffer. The limit is set to the current position and then the position is set to zero. If the mark is defined then it is discarded.

After a sequence of channel-read or put operations, invoke this method to prepare for a sequence of channel-write or relative get operations. For example:

```
buf.put(magic); // Prepend header
in.read(buf); // Read data into rest of buffer
buf.flip(); // Flip buffer
out.write(buf); // Write header + data to channel
```

This method is often used in conjunction with the ***compact*** method when transferring data from one place to another.

clear() and compact()

Once you are done reading data out of the Buffer you have to make the Buffer ready for writing again. You can do so either by calling clear() or by calling compact(). If you call clear() the position is set back to 0 and the limit to capacity. In other words, the Buffer is cleared. The data in the Buffer is not cleared. Only the markers telling where you can write data into the Buffer area. **clear()** - Clears this buffer. The position is set to zero, the limit is set to the capacity, and the mark is discarded. Invoke this method before using a sequence of channel-read or put operations to fill this buffer. For example:

```
buf.clear(); // Prepare buffer for reading
in.read(buf); // Read data
```

This method does not actually erase the data in the buffer, but it is named as if it did because it will most often be used in situations in which that might as well be the case.

Buffer Capacity, Position and Limit:

Position

When the one reads from a channel, the one puts the data that is read into an underlying array. The position variable keeps track of how much data the one has written. More precisely, it specifies into which array element the next byte will go. Thus, if the one has read three bytes from a channel into a buffer, that buffer's position will be set to 3, referring to the fourth element of the array.

Likewise, when you are writing to a channel, you get the data from a buffer. The position value keeps track of how much you have gotten from the buffer. More precisely, it specifies from which array element the next byte will come. Thus, if you've written 5 bytes to a channel from a buffer, that buffer's position will be set to 5, referring to the sixth element of the array.

Limit

The limit variable specifies how much data there is left to get (in the case of writing from a buffer into a channel), or how much room there is left to put data into (in the case of reading from a channel into a buffer).

The position is always less than, or equal to, the limit.

Capacity

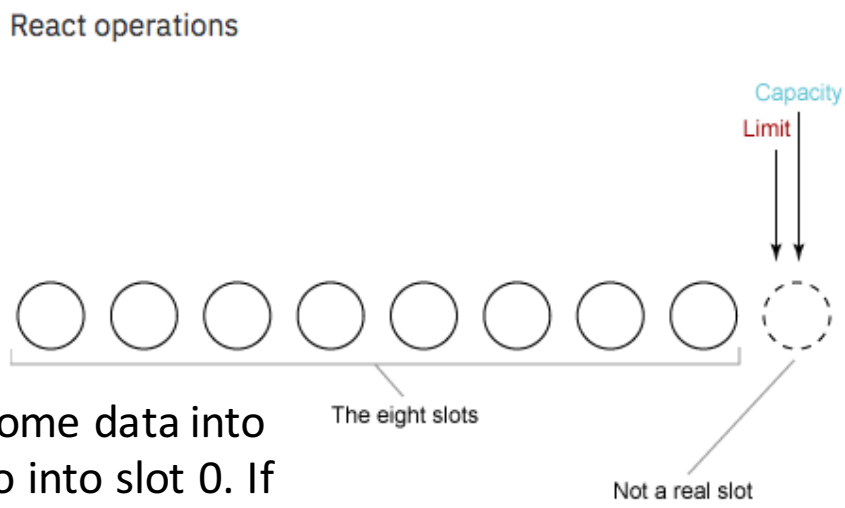
The capacity of a buffer specifies the maximum amount of data that can be stored therein. In effect, it specifies the size of the underlying array -- or, at least, the amount of the underlying array that we are permitted to use.

The limit can never be larger than the capacity.

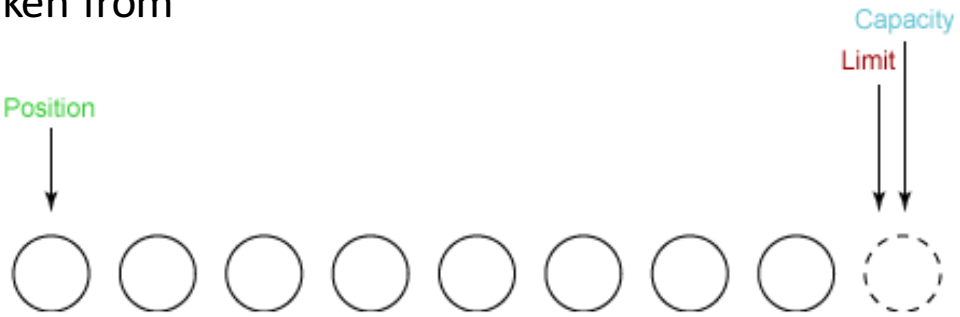
1. Java NIO

Buffer Use Case Example – ByteBufferSimpleProgMain.java:

Step 1. We'll start with a newly created buffer. For the sake of the example, let's assume that our buffer has a total capacity of eight bytes. The Buffer 's state is shown here. Recall that the limit can never be larger than the capacity, and in this case both values are set to 8. We show this by pointing them off the end of the array (which is where slot 8 would be if there were a slot 8):



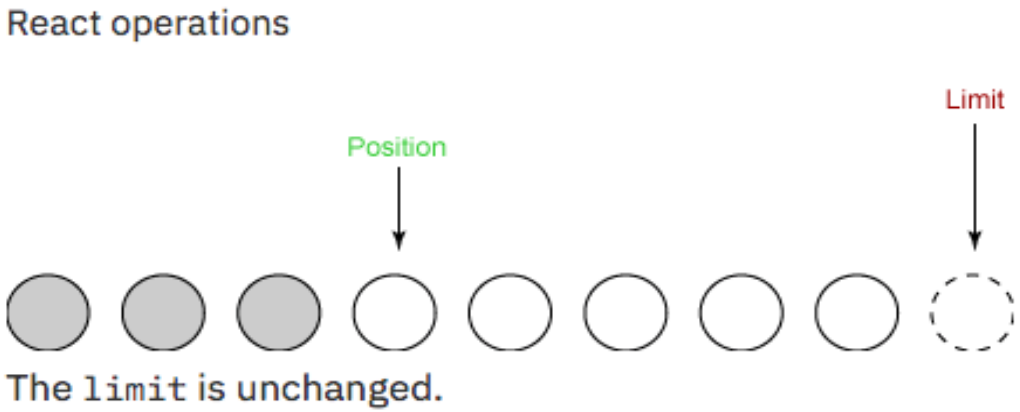
The position is set to 0. If we read some data into the buffer, the next byte read will go into slot 0. If we write from the buffer, the next byte taken from the buffer will be taken from slot 0.
The position setting is shown here:



Buffer Use Case Example:

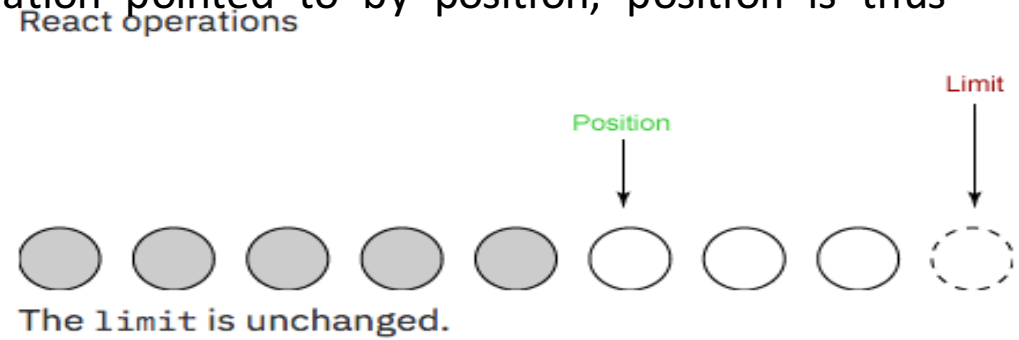
Step 2 - The first read – 3 bytes

Now we are ready to begin read/write operations on our newly created buffer. We start by reading some data from our input channel into the buffer. The first read gets three bytes. These are put into the array starting at the position, which was set to 0. After this read, the position is increased to 3, as shown here:



Step 3 - The second read – 2 bytes

For our second read, we read two more bytes from the input channel into our buffer. The two bytes are stored at the location pointed to by position; position is thus increased by two:



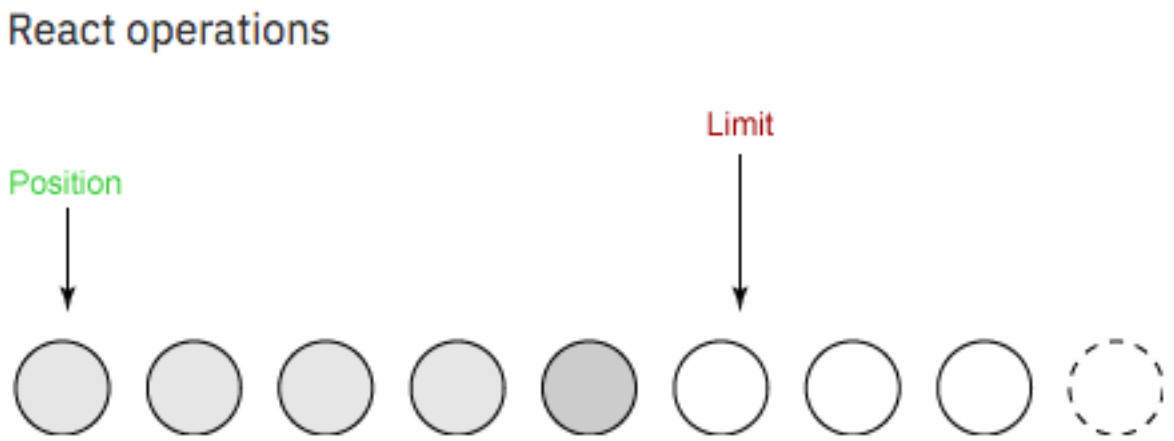
Buffer Use Case Example:

Step 4 - The flip

Now we are ready to write our data to an output channel. Before we can do this, we must call the flip() method. This method does two crucial things:

- It sets the limit to the current position.
- It sets the position to 0.

The figure on the section shows our buffer before the flip. Here is the buffer after the flip:



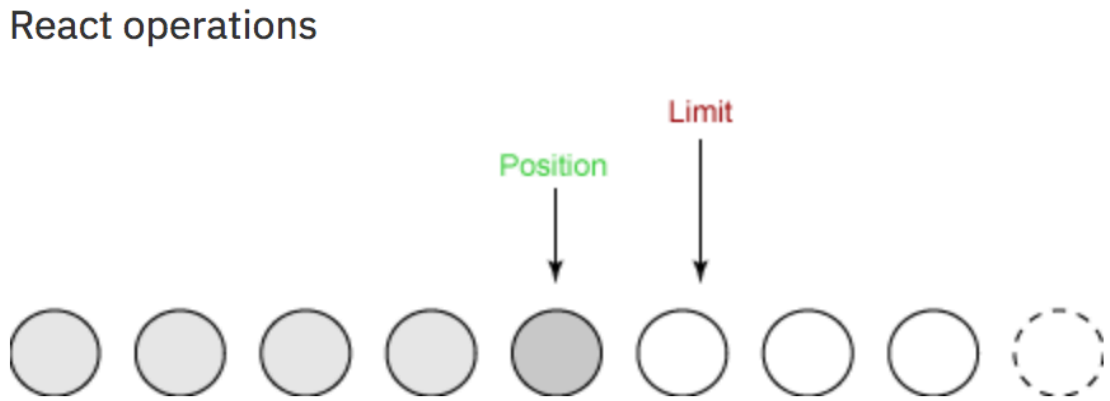
We are now ready to begin writing data to a channel from the buffer. The position has been set to 0, which means the next byte we get will be the first one. And the limit has been set to the old position, which means that it just includes all the bytes we read before, and no more.

1. Java NIO

Buffer Use Case Example:

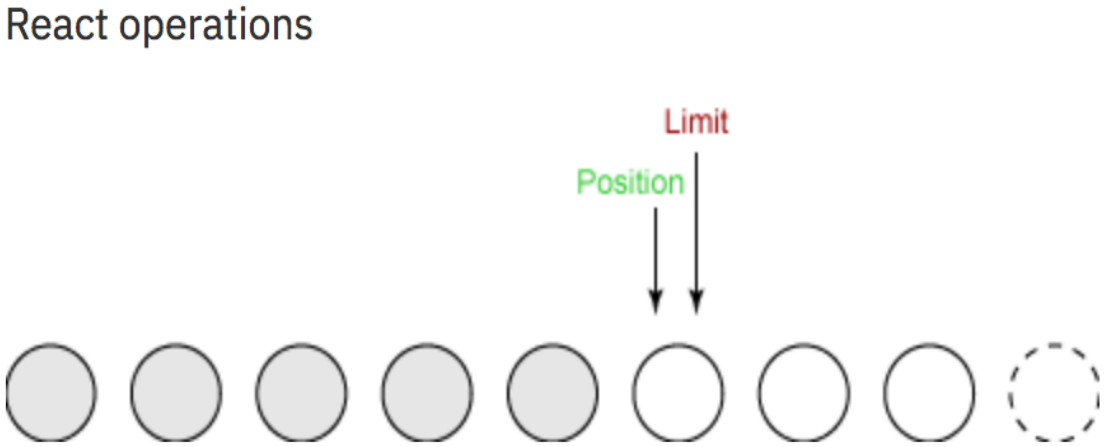
Step 5 - The first write – 4 bytes

In our first write, we take four bytes from the buffer and write them to our output channel. This advances the position to 4, and leaves the limit unchanged, as shown here:



Step 6 - The second write – 1 byte

We only have one byte left to write. The limit was set to 5 when we did our flip(), and the position cannot go past the limit. So the last write takes one byte from our buffer and writes it to the output channel. This advances the position to 5, and leaves the limit unchanged, as shown here:



Buffer Use Case Example – ByteBufferSimpleProgMain.java:

Step 7 - The clear

Our final step is to call the buffer's clear() method. This method resets the buffer in preparation for receiving more bytes. Clear does two crucial things:

- It sets the limit to match the capacity.
- It sets the position to 0.

This figure shows the state of the buffer after clear() has been called:

1. Java NIO

The buffer slicing – SliceBuffer.java

The slice() method creates a kind of *sub-buffer* from an existing buffer. That is, it creates a new buffer that shares its data with a portion of the original buffer.

This is best explained with an example. Let's start by creating a ByteBuffer of length 10:

```
ByteBuffer buffer = ByteBuffer.allocate( 10 );
```

We fill this buffer with data, putting the number *n* in slot *n*:

```
for (int i=0; i<buffer.capacity(); ++i) {
    buffer.put( (byte)i );
}
```

Now we'll *slice* the buffer to create a sub-buffer that covers slots 3 through 6. In a sense, the sub-buffer is like a *window* onto the original buffer.

You specify the start and end of the window by setting the position and limit values, and then call the Buffer's slice() method:

```
buffer.position( 3 );
buffer.limit( 7 );
ByteBuffer slice = buffer.slice();
```

slice is a sub-buffer of buffer. However, slice and buffer share the same underlying data array, as we'll see in the next section.

1. Java NIO

The buffer slicing – SliceBuffer.java

We've created a sub-buffer of our original buffer, and we know that the two buffers and the sub-buffers share the same underlying data array. Let's see what this means.

We run through the sub-buffer, and alter each element by multiplying it by 11. This changes, for example, a 5 into a 55.

```
for (int i=0; i<slice.capacity(); ++i) {
    byte b = slice.get( i );
    b *= 11;
    slice.put( i, b );
}
```

Finally, let's take a look at the contents of the original buffer:

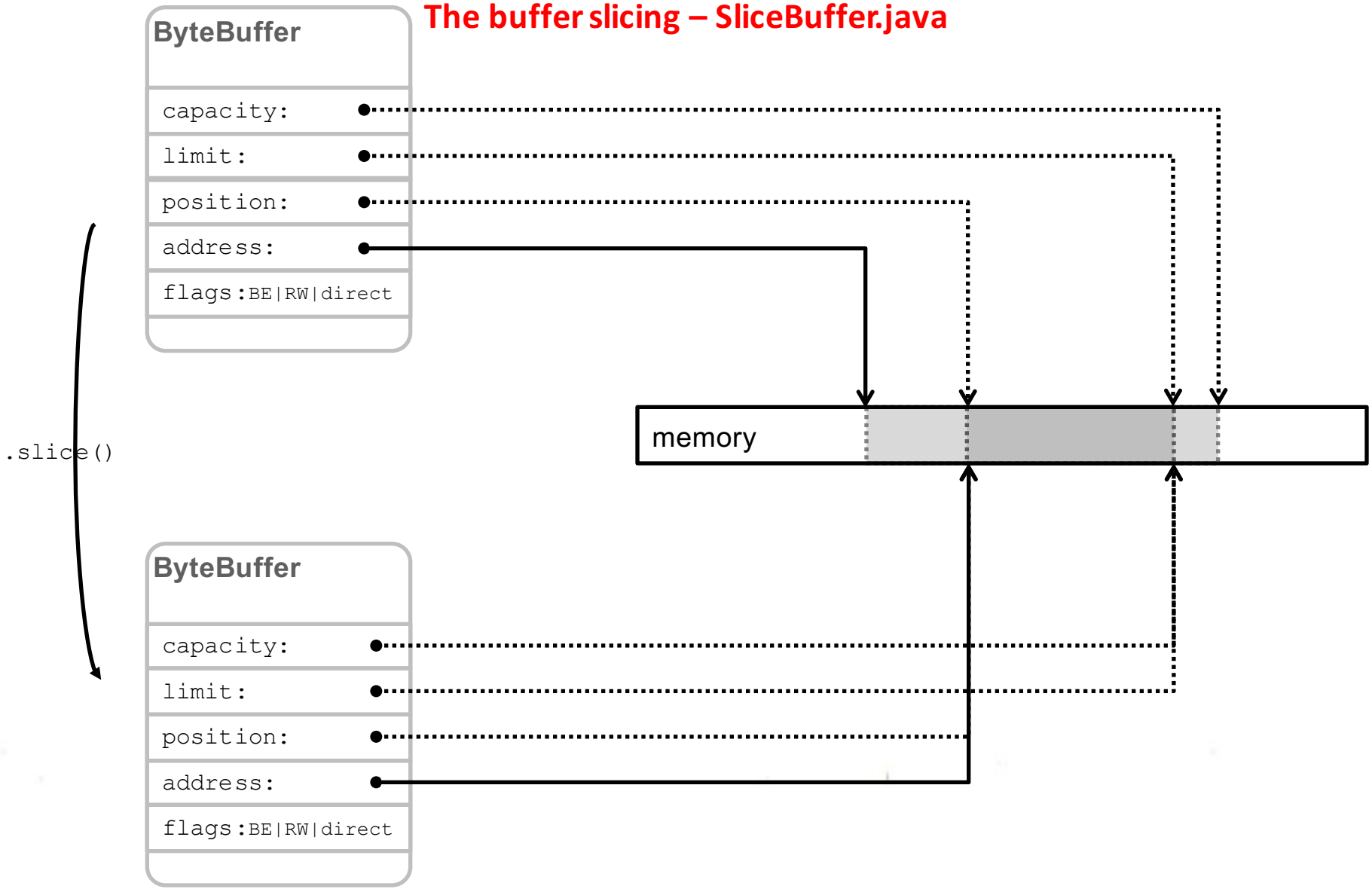
```
buffer.position( 0 );
buffer.limit( buffer.capacity() );

while (buffer.remaining()> 0) {
    System.out.print( buffer.get() + ", " );
}
```

The result shows that only the elements in the window of the sub-buffer were changed:

```
$java SliceBuffer
0, 1, 2, 33, 44, 55, 66, 7, 8, 9,
```

1. Java NIO


The buffer slicing – SliceBuffer.java
ByteBuffer
capacity:
limit:
position:
address:
flags:BE|RW|direct
.slice()
memory
ByteBuffer
capacity:
limit:
position:
address:
flags:BE|RW|direct

1. Java NIO

The buffer at work: An inner loop – CopyFile.java / FastCopyFile.java

The following inner loop summarizes the process of using a buffer to copy data from an input channel to an output channel.

```
while (true) {
    buffer.clear();
    int r = fcin.read( buffer );

    if (r==-1) {
      break;
    }

    buffer.flip();
    fcout.write( buffer );
}
```

The read() and write() calls are greatly simplified because the buffer takes care of many of the details. The clear() and flip() methods are used to switch the buffer between reading and writing.

1. Java NIO

Main Differences between Java NIO and IO

The table below summarizes the main differences between Java NIO and IO. I will get into more detail about each difference in the sections following the table.

Java IO	Java NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
-	Selectors

Stream Oriented vs. Buffer Oriented

The first big difference between Java NIO and IO is that IO is stream oriented, where NIO is buffer oriented. So, what does that mean?

Java IO being stream oriented means that you read one or more bytes at a time, from a stream. What you do with the read bytes is up to you. They are not cached anywhere. Furthermore, you cannot move forth and back in the data in a stream. If you need to move forth and back in the data read from a stream, you will need to cache it in a buffer first.

Java NIO's buffer oriented approach is slightly different. Data is read into a buffer from which it is later processed. You can move forth and back in the buffer as you need to. This gives you a bit more flexibility during processing. However, you also need to check if the buffer contains all the data you need in order to fully process it. And, you need to make sure that when reading more data into the buffer, you do not overwrite data in the buffer you have not yet processed.

1. Java NIO

Main Differences between Java NIO and IO

Blocking vs. Non-blocking IO (and Completion Handlers)

Java IO's various streams are blocking. That means, that when a thread invokes a read() or write(), that thread is blocked until there is some data to read, or the data is fully written. The thread can do nothing else in the meantime.

Java NIO's non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available, or nothing at all, if no data is currently available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else.

The same is true for non-blocking writing. A thread can request that some data be written to a channel, but not wait for it to be fully written. The thread can then go on and do something else in the mean time.

What threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output.

Selectors

Java NIO's selectors allow a single thread to monitor multiple channels of input. You can register multiple channels with a selector, then use a single thread to "select" the channels that have input available for processing, or select the channels that are ready for writing. This selector mechanism makes it easy for a single thread to manage multiple channels.

1. Java NIO

Main Differences between Java NIO and IO

Java NIO vs. IO - the API Calls

Of course the API calls when using NIO look different than when using IO. This is no surprise. Rather than just read the data byte for byte from e.g. an InputStream, the data must first be read into a buffer, and then be processed from there.

The Processing of Data – IO Implementation

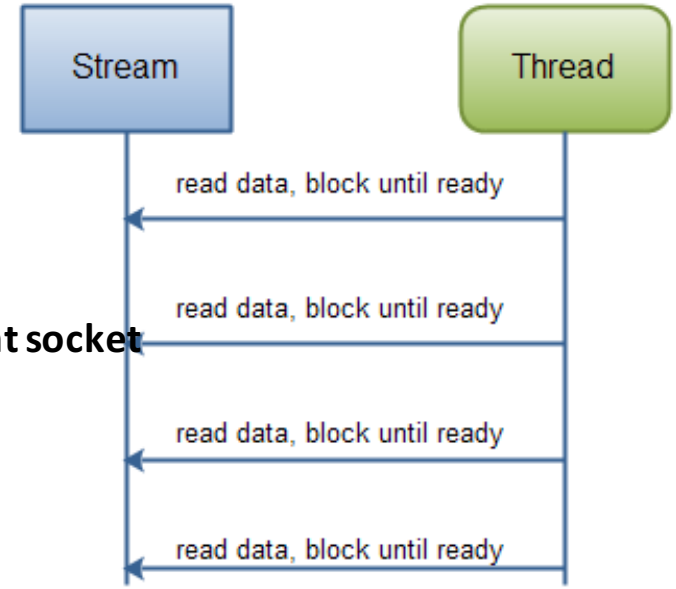
The processing of the data is also affected when using a pure NIO design, vs. an IO design.

In an IO design you read the data byte for byte from an InputStream or a Reader. Imagine you were processing a stream of line based textual data. For instance:

Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890

This stream of text lines could be processed like this:

```
InputStream input = ... ; // get the InputStream from the client socket
BufferedReader reader = new BufferedReader(new
InputStreamReader(input));
String nameLine = reader.readLine();
String ageLine = reader.readLine();
String emailLine = reader.readLine();
String phoneLine = reader.readLine();
```



Java IO: Reading data from a blocking stream.

1. Java NIO

Main Differences between Java NIO and IO

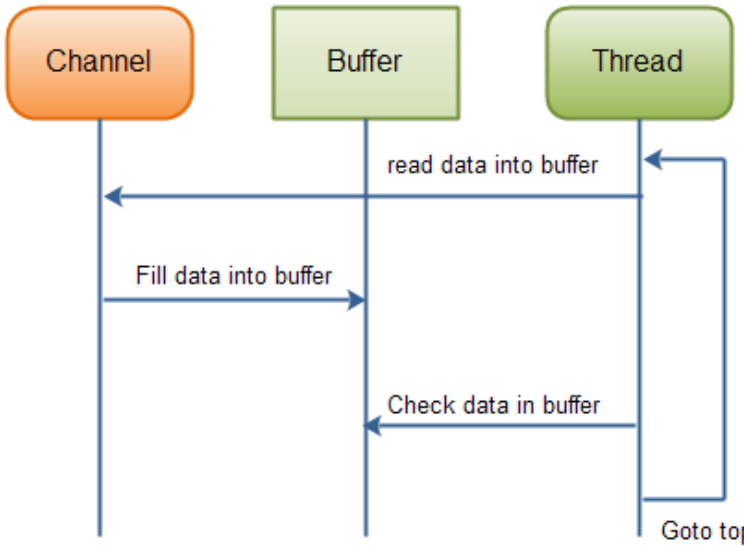
The Processing of Data - NIO Implementation

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

Notice the second line which reads bytes from the channel into the ByteBuffer. When that method call returns you don't know if all the data you need is inside the buffer. All you know is that the buffer contains some bytes. This makes processing somewhat harder.
Imagine if, after the first read(buffer) call, that all what was read into the buffer was half a line. For instance, "Name: An". Can you process that data? Not really. You need to wait until at leas a full line of data has been into the buffer, before it makes sense to process any of the data at all.

So how do you know if the buffer contains enough data for it to make sense to be processed? Well, you don't. The only way to find out, is to look at the data in the buffer. The result is, that you may have to inspect the data in the buffer several times before you know if all the data is in there. This is both inefficient, and can become messy in terms of program design. For instance:

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(! bufferFull(bytesRead) ) {
   bytesRead = inChannel.read(buffer);
}
```

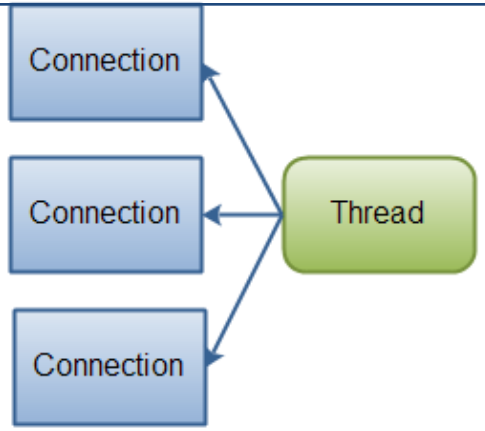


Java NIO: Reading data from a channel until all needed data is in buffer.

1. Java NIO

The Processing of Data – Multithreading in NIO vs. IO

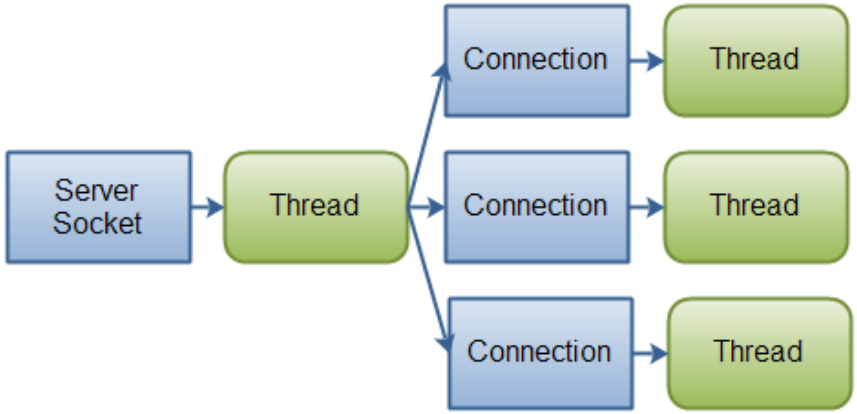
NIO – New Input / Output allows you to manage multiple channels (network connections or files) using only a single (or few) threads, but the cost is that parsing the data might be somewhat more complicated than when reading data from a blocking stream. If you need to manage thousands of open connections simultaneously, which each only send a little data, for instance a chat server, implementing the server in NIO is probably an advantage. Similarly, if you need to keep a lot of open connections to other computers, e.g. in a P2P network, using a single thread to manage all of your outbound connections might be an advantage:



Java NIO: A single thread managing multiple connections.

Main Differences between Java NIO and IO

Classic IO - fewer connections with very high bandwidth, sending a lot of data at a time, perhaps a classic IO server implementation might be the best fit. This diagram illustrates a classic IO server design:



Java IO: A classic IO server design - one connection handled by one thread.

Section Conclusion

Fact: **Java Dev may take advantage of NIO**

In few **samples** it is simple to understand:

- Buffers
- Channels
- Selectors
- Completion Handlers

Java Regular Expressions for parsing data

Java Regular Expressions - RegEx

2. Java RegEx

Java provides the **java.util.regex** package for pattern matching with regular expressions. Java regular expressions are very similar to the Perl programming language and very easy to learn.

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

The java.util.regex package primarily consists of the following three classes –

- **Pattern Class** – A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static **compile()** methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
- **Matcher Class** – A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the **matcher()** method on a Pattern object.
- **PatternSyntaxException** – A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

2. Java RegEx

Capturing Groups:

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression (dog) creates a single group containing the letters "d", "o", and "g".

Capturing groups are numbered by counting their opening parentheses from the left to the right. In the expression **((A)(B(C)))**, for example, there are four (4) such groups −

- ((A)(B(C)))
- (A)
- (B(C))
- (C)

To find out how many groups are present in the expression, call the **groupCount** method on a matcher object. The groupCount method returns an **int** showing the number of capturing groups present in the matcher's pattern.

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by **groupCount**.

2. Java RegEx

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexMatches {
  public static void main( String args[] ) {
    // String to be scanned to find the pattern.
    String line = "This order was placed for QT3000! OK?";
    String pattern = "(.*)(\\d+)(.*)";
    // Create a Pattern object
    Pattern r = Pattern.compile(pattern);
    // Now create matcher object.
    Matcher m = r.matcher(line);

    if (m.find( )) {
      System.out.println("Found value: " + m.group(0) );
      System.out.println("Found value: " + m.group(1) );
      System.out.println("Found value: " + m.group(2) );
    } else {
      System.out.println("NO MATCH");
    }
  }
}
```

Capturing Groups - Example:

Following example illustrates how to find a digit string from the given alphanumeric string:

This will produce the following result –

Output

```
Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT300
Found value: 0
```

2. Java Regular Expression – RegEx

Regular Expression Syntax:

The table listing down the regular expression meta-character syntax available in Java

Sub-expression	Matches
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character except newline. Using **m** option allows it to match the newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
\A	Beginning of the entire string.
\z	End of the entire string.
\Z	End of the entire string except allowable final line terminator.

2. Java Regular Expression – RegEx

Regular Expression Syntax:

Sub-expression	Matches
re*	Matches 0 or more occurrences of the preceding expression.
re+	Matches 1 or more of the previous thing.
re?	Matches 0 or 1 occurrence of the preceding expression.
re{ n}	Matches exactly n number of occurrences of the preceding expression.
re{ n,}	Matches n or more occurrences of the preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of the preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers the matched text.
(?: re)	Groups regular expressions without remembering the matched text.
(?> re)	Matches the independent pattern without backtracking.

2. Java Regular Expression – RegEx

Regular Expression Syntax:

Sub-expression	Matches
\w	Matches the word characters.
\W	Matches the nonword characters.
\s	Matches the whitespace. Equivalent to [\t\n\r\f].
\S	Matches the nonwhitespace.
\d	Matches the digits. Equivalent to [0-9].
\D	Matches the nondigits.
\A	Matches the beginning of the string.
\Z	Matches the end of the string. If a newline exists, it matches just before newline.
\z	Matches the end of the string.
\G	Matches the point where the last match finished.

2. Java Regular Expression – RegEx

Regular Expression Syntax:

Sub-expression	Matches
\n	Back-reference to capture group number "n".
\b	Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets.
\B	Matches the nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\Q	Escape (quote) all characters up to \E.
\E	Ends quoting begun with \Q.

2. Java Regular Expression – RegEx

Methods of the Matcher Class – Index Methods:

Index methods provide useful index values that show precisely where the match was found in the input string

No.	Method & Description
1	**public int start()** Returns the start index of the previous match.
2	**public int start(int group)** Returns the start index of the subsequence captured by the given group during the previous match operation.
3	**public int end()** Returns the offset after the last character matched.
4	**public int end(int group)** Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

2. Java Regular Expression – RegEx

Methods of the Matcher Class – Index Methods:

The **start** and **end** Methods Example:

Following is the example that counts the number of times the word "cat" appears in the input string.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {
 private static final String REGEX = "\\bcat\\b";
 private static final String INPUT = "cat cat cat cattie cat";

 public static void main( String args[] ) {
   Pattern p = Pattern.compile(REGEX);
   Matcher m = p.matcher(INPUT); // get a matcher object
   int count = 0;

   while(m.find()) {
      count++;
      System.out.println("Match number "+count);
      System.out.println("start(): "+m.start());
      System.out.println("end(): "+m.end());
   }
 }
}
```

Output

```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22
```

2. Java Regular Expression – RegEx

Methods of the Matcher Class – Study Methods:

Study methods review the input string and return a Boolean indicating whether or not the pattern is found

No.	Method & Description
1	**public boolean lookingAt()** Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
2	**public boolean find()** Attempts to find the next subsequence of the input sequence that matches the pattern.
3	**public boolean find(int start)** Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
4	**public boolean matches()** Attempts to match the entire region against the pattern.

2. Java Regular Expression – RegEx

Methods of the Matcher Class – Study Methods:

The **matches** and **lookingAt** Methods Example:

The **matches** and **lookingAt** methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while **lookingAt** does not. Both methods always start at the beginning of the input string. Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexMatches {
   private static final String REGEX = "foo";
   private static final String INPUT = "fooooooooooooooooo";
   private static Pattern pattern;
   private static Matcher matcher;

   public static void main( String args[] ) {
      pattern = Pattern.compile(REGEX);
      matcher = pattern.matcher(INPUT);
      System.out.println("Current REGEX is: "+REGEX);
      System.out.println("Current INPUT is: "+INPUT);
      System.out.println("lookingAt(): "+matcher.lookingAt());
      System.out.println("matches(): "+matcher.matches());
   }
}
```

Output

```
Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt(): true
matches(): false
```

2. Java Regular Expression – RegEx

Methods of the Matcher Class – Replacement Methods:

Replacement methods are useful methods for replacing text in an input string

No.	Method & Description
1	**public Matcher appendReplacement(StringBuffer sb, String replacement)** Implements a non-terminal append-and-replace step.
2	**public StringBuffer appendTail(StringBuffer sb)** Implements a terminal append-and-replace step.
3	**public String replaceAll(String replacement)** Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
4	**public String replaceFirst(String replacement)** Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
5	**public static String quoteReplacement(String s)** Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement **s**in the appendReplacement method of the Matcher class.

2. Java Regular Expression – RegEx

Methods of the Matcher Class – Replacement Methods:

The **replaceFirst** and **replaceAll** Methods Example:

The replaceFirst and replaceAll methods replace the text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences. Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexMatches {
   private static String REGEX = "dog";
   private static String INPUT = "The dog says meow. " + "All dogs say meow.";
   private static String REPLACE = "cat";

   public static void main(String[] args) {
      Pattern p = Pattern.compile(REGEX); // get a matcher object
      Matcher m = p.matcher(INPUT);
      INPUT = m.replaceAll(REPLACE);
      System.out.println(INPUT);
   }
}
```

Output

```
The cat says meow. All cats say meow.
```

2. Java Regular Expression – RegEx

Methods of the Matcher Class – Replacement Methods:

The **appendReplacement** and **appendTail** Methods Example:

The Matcher class also provides **appendReplacement** and **appendTail** methods for text replacement.

Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexMatches {
   private static String REGEX = "a*b";
   private static String INPUT = "aabfooaabfooabfoob";
   private static String REPLACE = "-";

   public static void main(String[] args) {
      Pattern p = Pattern.compile(REGEX); // get a matcher object
      Matcher m = p.matcher(INPUT);
      StringBuffer sb = new StringBuffer();
      while(m.find()) { m.appendReplacement (sb, REPLACE); }
      m.appendTail(sb);
      System.out.println(sb.toString());
   }
}
```

Output

```
-foo-foo-foo-
```

Section Conclusions

Java Regular Expressions – RegEx Programming has the following key concepts:

- Pattern and Matcher classes
- Capturing Groups
- Meta-Character Syntax
- Matcher class methods:
  - Index
  - Study
  - Replacement

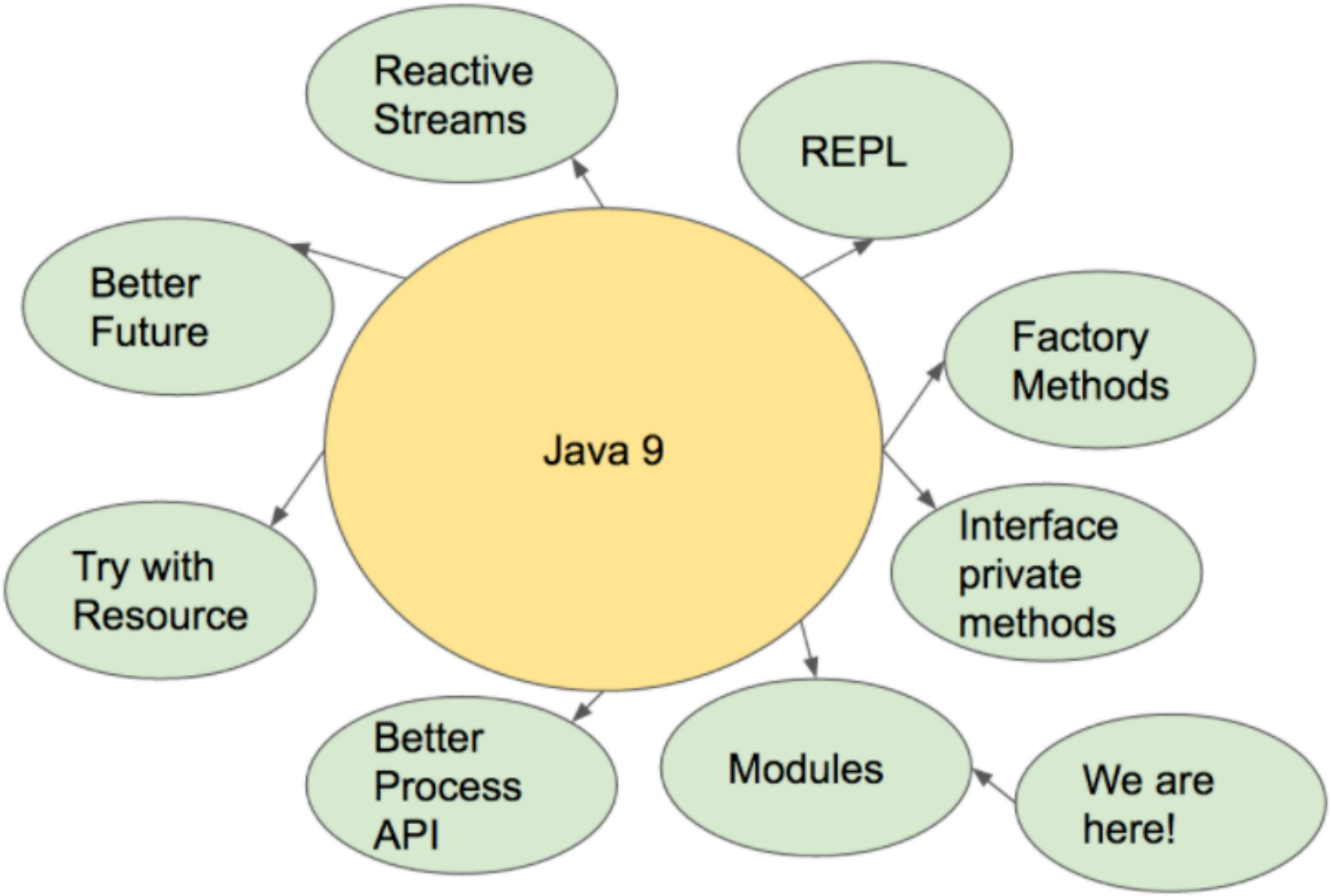
Java RegEx Programming
for easy sharing

JDK 9 Modules & HTTP2 plus New Features

JDK9 Modules & New Features

3. JDK 9 New Features

Java 9 New Features (incl. HTTP2 classes):



3. JDK 9 New Features

Java 9 Modules:

Java SE 9: Jigsaw Project

Jigsaw project is going to introduce completely new concept of Java SE 9: **Java Module System**. It is very big and prestigious project from Oracle Corp in Java SE 9 release. Initially they have started this project as part of Java SE 7 Release. However with huge changes, it's postponed to Java SE 8 then again postponed. Now it has been released with Java SE 9 in 2017.

Main Goals of Jigsaw Project:

- The Modular JDK - As we know, Current JDK system is too big. So they have decided to divide JDK itself into small modules to get a number of benefits (We will discuss them soon in the coming sections).
- Modular Source Code - Current source code jar files are too big, especially rt.jar is too big right. So they are going to divide Java Source code into smaller modules.
- Modular Run-Time Images - The main goal of this Feature is "Restructure the JDK and JRE run-time images to accommodate modules".
- Encapsulate Most Internal APIs - The main goal of this feature is "Make most of the JDK's internal APIs inaccessible by default but leave a few critical, widely-used internal APIs accessible".
- Java Platform Module System - The main goal of this Feature is "Allowing the user to create their modules to develop their applications".
- jlink: The Java Linker - The main goal of this jlink Tool is "Allowing the user to create executable to their applications".

3. JDK 9 New Features

Java 9 Modules:

Problems of Current Java System?

In this section, we will discuss about "Why we need Java SE 9 Module System" that means the problems of Current Java System.

Java SE 8 or earlier systems have following problems in developing or delivering Java Based applications.

- As JDK is too big, it is bit tough to scale down to small devices. Java SE 8 has introduced 3 types of compact profiles to solve this problem: compact1, compact2 and compact3. But it does not solve this problem.
- JAR files like rt.jar etc are too big to use in small devices and applications.
- As JDK is too big, our applications or devices are not able to support better Performance.
- There is no Strong Encapsulation in the current Java System because "public" access modifier is too open. Everyone can access it.
- As JDK, JRE is too big, it is hard to Test and Maintain applications.
- As public is too open, They are not to avoid the accessing of some Internal Non-Critical APIs like sun.*, *.internal.* etc.
- As User can access Internal APIs too, Security is also big issue.
- Application is too big.
- Its a bit tough to support Less Coupling between components.

3. JDK 9 New Features

Java 9 Modules:

Advantages of Java SE 9 Module System

Java SE 9 Module System is going to provide the following benefits:

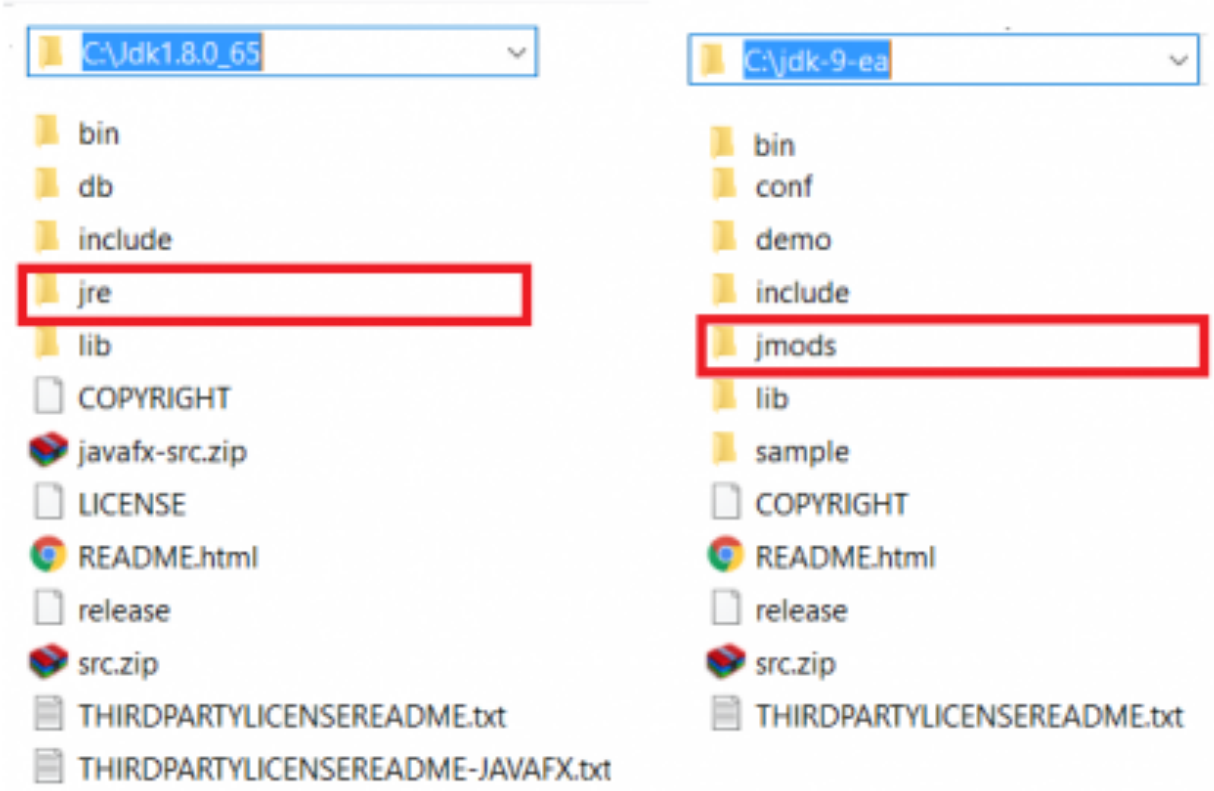
- As Java SE 9 is going to divide JDK, JRE, JARs etc, into smaller modules, we can use whatever modules we want. So it is very easy to scale down the Java Application to Small devices.
- Ease of Testing and Maintainability.
- Supports better Performance.
- As public is not just public, it supports very Strong Encapsulation. (Don't worry its a big concept. we will explore it with some useful examples soon).
- We cannot access Internal Non-Critical APIs anymore.
- Modules can hide unwanted and internal details very safely, we can get better Security.
- Application is too small because we can use only what ever modules we want.
- Its easy to support Less Coupling between components.
- Its easy to support Single Responsibility Principle (SRP).

3. JDK 9 New Features

Java 9 Modules:

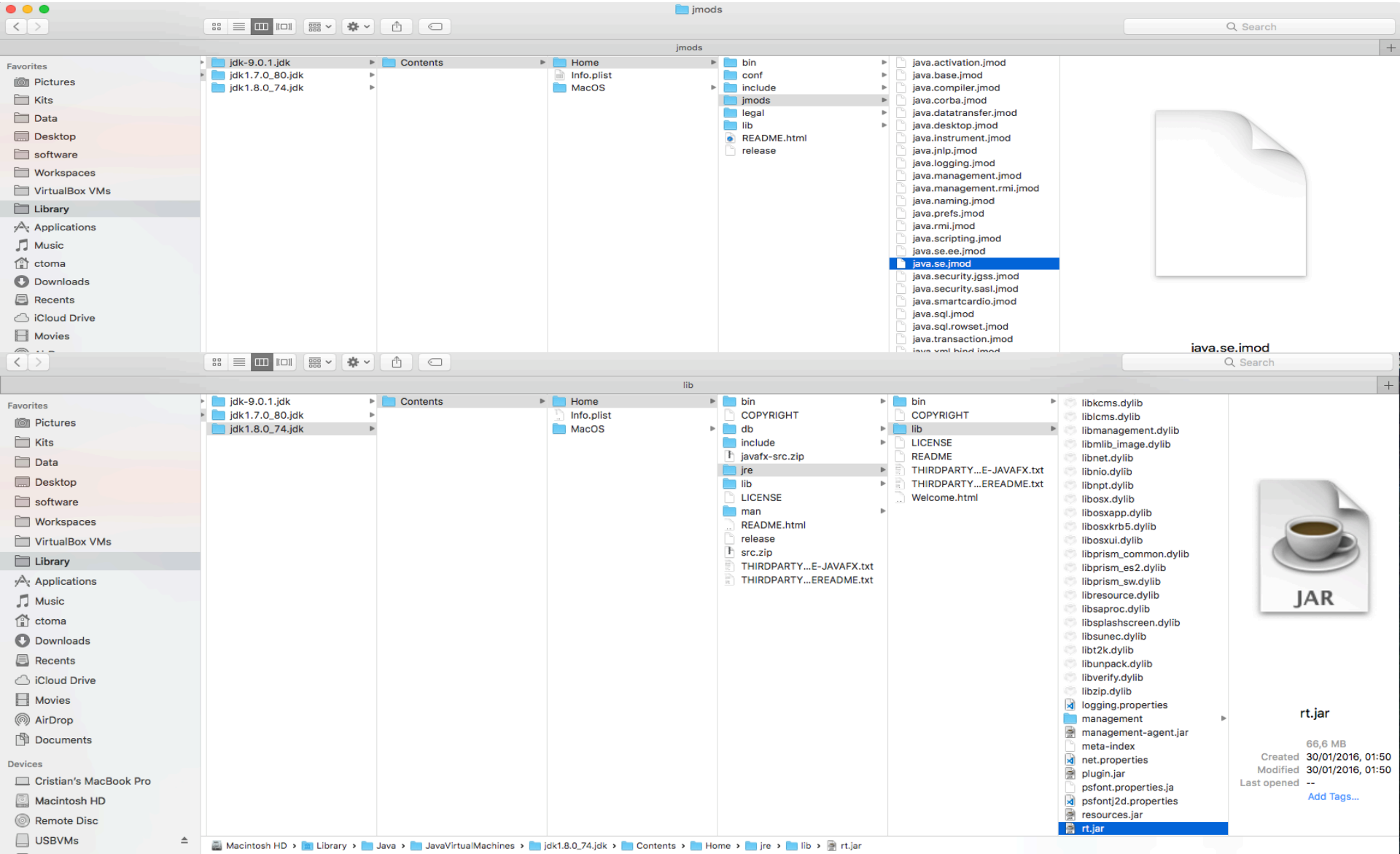
Compare JDK 8 and JDK 9

We know what a JDK software contains. After installing JDK 8 software, we can see couple of directories like bin, jre, lib etc in Java Home folder. However Oracle Corp / OpenJDK has changed this folder structure a bit differently as shown below.



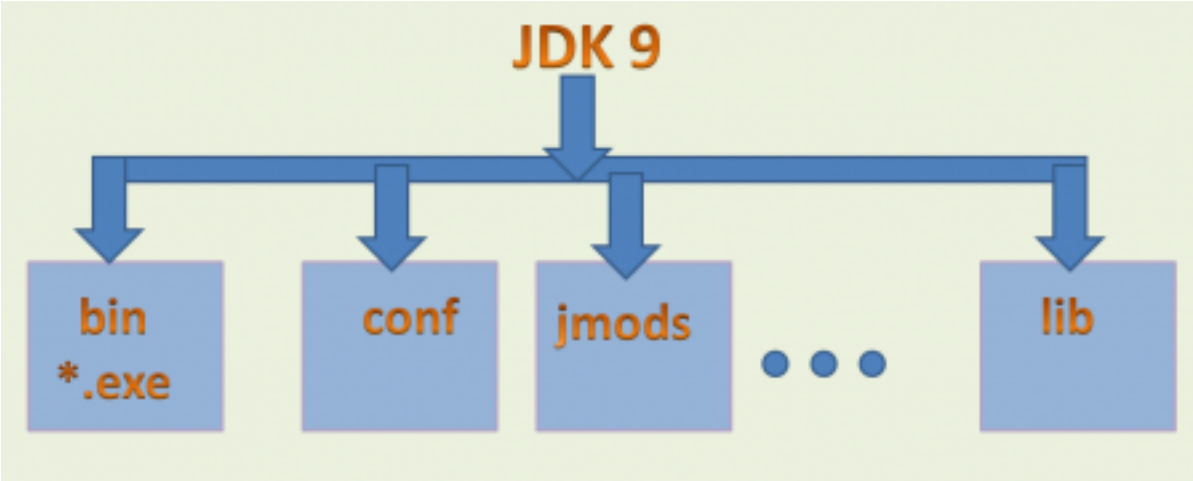
3. JDK 9 New Features

Java 9 Modules - Compare JDK 8 and JDK 9 Directory Structure:



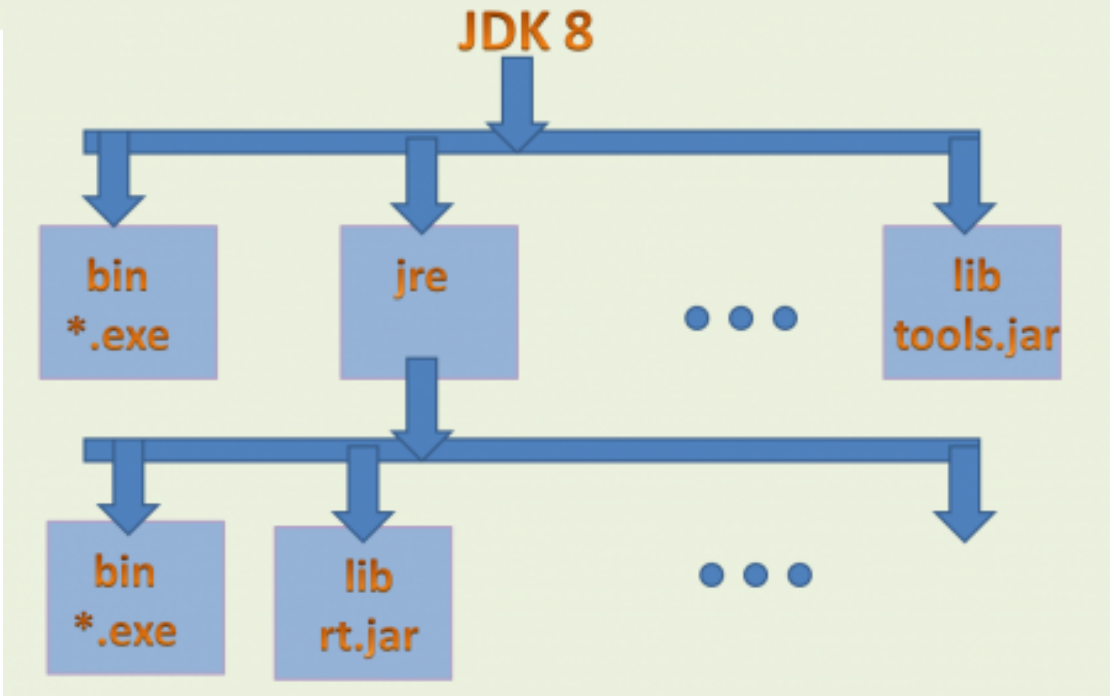
3. JDK 9 New Features

Java 9 Modules – SDK Distro:



Here JDK 9 does NOT contain JRE.

In JDK 9, JRE is separated into a separate distribution folder. JDK 9 software contains a new folder "jmods". As of today, "jmods" contains 97 modules. The "jmods" folder is available at ${JAVA_HOME}/jmods. These are known as JDK Modules.



3. JDK 9 New Features

Java 9 Modules:

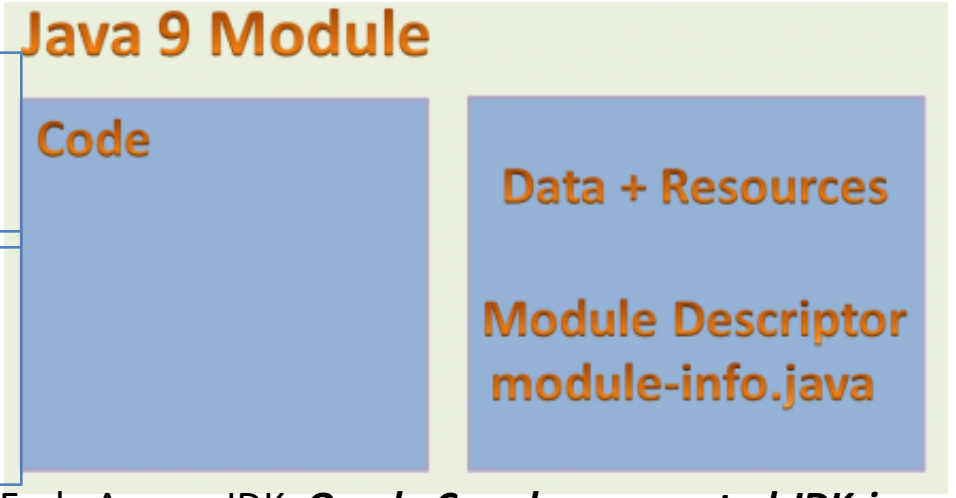
What is Java 9 Module?

A Module is a self-describing collection of Code, Data and some Resources. It is a set of related Packages, Types (classes, abstract classes, interfaces etc) with Code & Data and Resources.

Each Module contains only a set of related code and data to support Single Responsibility (Functionality) Principle (SRP).

The main goal of Java 9 Module System is to support:
Modular Programming in Java.

In Simple Terms,
Module = Code + Data
Module Descriptor ("module-info.java") is a one of the Resources in Java 9 Module.



As of now, Java 9 Module System has 97 modules in Early Access JDK. ***Oracle Corp has separated JDK jars and Java SE Specifications into two set of Modules.***

- ***All JDK Modules starts with "jdk.*"***
- ***All Java SE Specifications Modules starts with "java.*"***

Java 9 Module System has a "java.base" Module. It's known as Base Module. It's an Independent module and does NOT dependent on any other modules. By default, all other Modules dependent on this module.

That's why "java.base" Module is also known as the root of the Java 9 Modules.

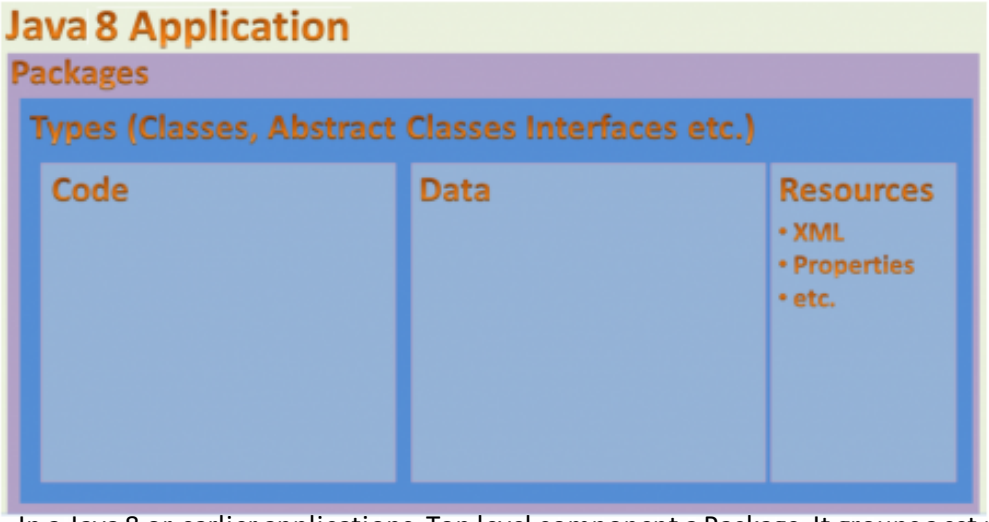
It's default module for all JDK Modules and User-Defined Modules.

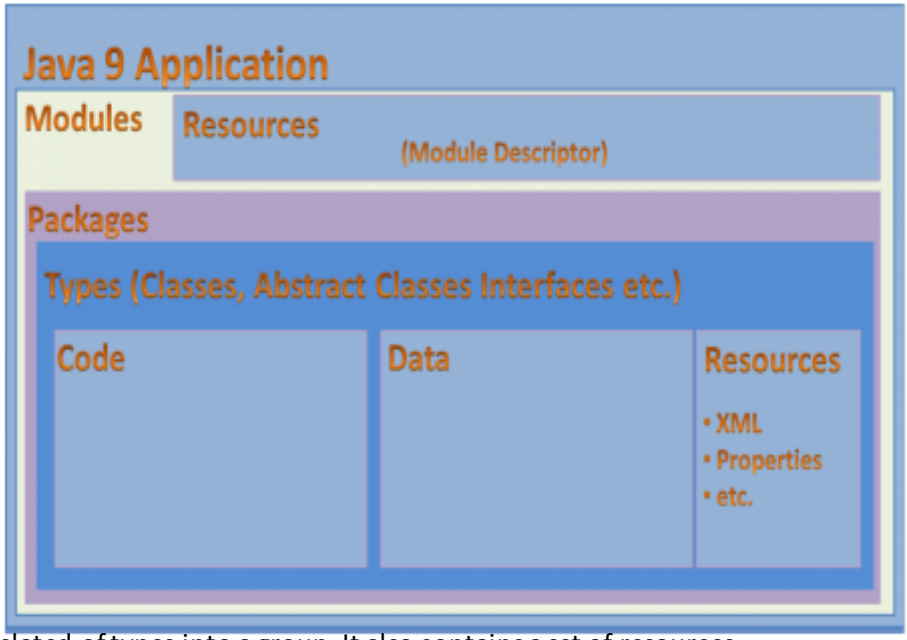
3. JDK 9 New Features

Java 9 Modules:

Compare Java 8 and Java 9 Applications

We have already developed many Java applications using Java 5, 6,7, or 8. We know how a Java 8 earlier applications looks like and what it contains.





In a Java 8 or earlier applications, Top level component a Package. It groups a set related of types into a group. It also contains a set of resources.

Java 9 Applications does not have much difference with this. It just introduced a new component called "Module", which is used to group a set of related Packages into a group. And one more new component that **Module Descriptor ("module-info.java").** That's it.

Like Java 8 applications have Packages as a Top level components, Java 9 applications have Module as Top Level components.

NOTE: -Each Java 9 Module have one and only one Module and one Module Descriptor. Unlike Java 8 Packages, We cannot create multiple modules into a single Module. In brief we can say a Java 9 Module contains the following main components:

- One Module + Module Name
- Module Descriptor
- Set of Packages
- Set of Types and Resources

Here Resources may be **module-info.java (Module Descriptor)** or any other properties or XML.

3. JDK 9 New Features

Java 9 Modules:

The two main goals of Java 9 Module System:

- Reliable Configuration
- Strong Encapsulation

In Java 9 Module System, a Module is a First class citizen.
In Simple Terms, Java has these many First class citizens:

- In OOP (Object Oriented Programming) = Object
- In FP (Functional Programming) = Function or Lambda
- In Module System (Modular Programming) = Module

Java 9 Module Basics

We should remember the following important points about Java 9 Module:

- Each module has a unique Name.
- Each module has some description in a source file.
- A Module description is expressed in a source file called “module-info.java”.
- As “module-info.java” file describes a Module, it is also known as “Module Descriptor”.
- A Module Descriptor is a Java file. It is not an XML, Text file or Properties file.
- By convention, we should use same name “module-info.java” for Module Descriptor.
- By convention, Module Descriptor file is placed in the top level directory of a Module.
- Each Module can have any number of Packages and Types.
- As of now, year 2017, JDK 9 have 97 modules.
- We can create our own modules.
- One Module can dependent on any number of modules.
- Each Module should have one and only one Module Descriptor (“module-info.java”).

Using Java SE 9, we develop Modules that means we can do Modular Programming in Java.

3. JDK 9 New Features

Java 9 Modules:

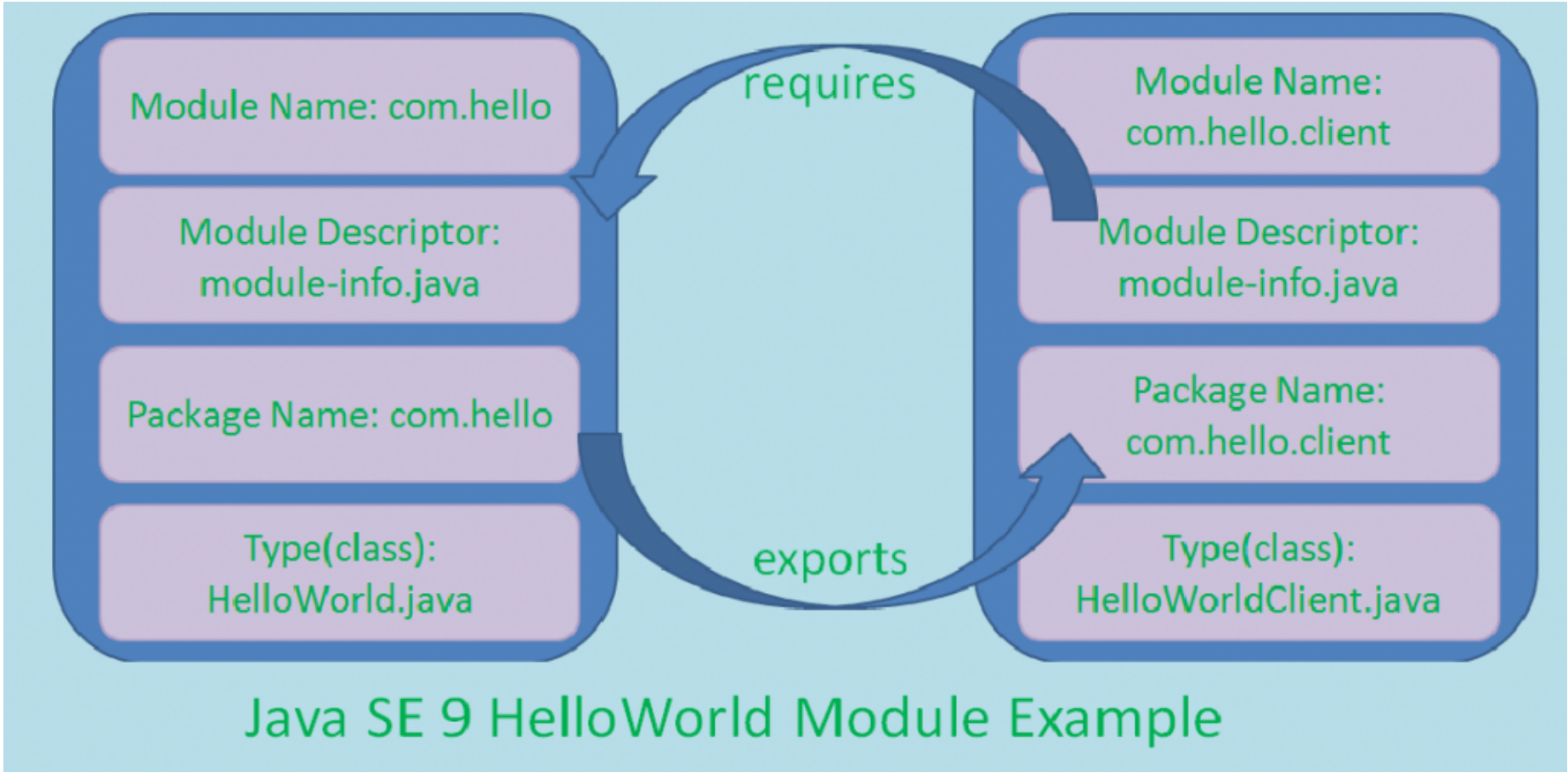
Steps to Develop a Java 9 Module

Follow these steps one by one to develop and test the “HelloWorld” Module.

- Create Module name folder, for example “com.hello”.
- Create Module Packages, for example “com.hello”.
- Create our Java component, for example “HelloWorld.java”.
- Create Module Descriptor, for example “module-info.java”.
- Define Module Description in Module Descriptor, for example “exports com.hello;” of “module-info.java” in “com.hello” module.
- Create Module Jars if required.
- Test the modules.
- These steps are common for almost all modules development.

3. JDK 9 New Features

Java 9 Modules:



3. JDK 9 New Features

Java 9 New Features:

1. Java 9 REPL (JShell)

Java has introduced a new tool called “jshell”. It stands for Java Shell and also known as REPL (Read Evaluate Print Loop). It is used to execute and test any Java Constructs like class, interface, enum, object, statements etc. very easily.

```
$ jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
jshell> int a = 10
a ==> 10

jshell> System.out.println("a value = " + a )
a value = 10
```

3. JDK 9 New Features

Java 9 New Features:

2. Factory Methods for Immutable List, Set, Map and Map.Entry

Java has introduced some convenient factory methods to create Immutable List, Set, Map and Map.Entry objects. These utility methods are used to create empty or non-empty Collection objects.

In Java SE 8 and earlier versions, We can use Collections class utility methods like unmodifiableXXX to create Immutable Collection objects. For instance, if we want to create an Immutable List, then we can use Collections.unmodifiableList method.

However these Collections.unmodifiableXXX methods are very tedious and verbose approach. To overcome those shortcomings, Oracle corp has added couple of utility methods to List, Set and Map interfaces.

List and Set interfaces have "of()" methods to create an empty or no-empty Immutable List or Set objects as shown below:

Empty List Example

```
List immutableList = List.of();Non-Empty List Example
List immutableList = List.of("one","two","three");
```

3. JDK 9 New Features

Java 9 New Features:

3. Private methods in Interfaces

In Java 8, we can provide method implementation in Interfaces using Default and Static methods. However we cannot create private methods in Interfaces.
To avoid redundant code and more re-usability, Oracle Corp is going to introduce private methods in Java SE 9 Interfaces. From Java SE 9 on-wards, we can write private and private static methods too in an interface using 'private' keyword.
These private methods are like other class private methods only, there is no difference between them.

```
public interface Card
{
 private Long createCardID(){ // Method implementation goes here. }
 private static void displayCardDetails(){ // Method implementation goes here. }
}
```

3. JDK 9 New Features

Java 9 New Features:

4. Java 9 Module System

One of the big changes or java 9 feature is the Module System. Oracle Corp is going to introduce the following features as part of Jigsaw Project.

- Modular JDK
- Modular Java Source Code
- Modular Run-time Images
- Encapsulate Java Internal APIs
- Java Platform Module System

Before Java SE 9 versions, we are using Monolithic Jars to develop Java-Based applications. This architecture has lot of limitations and drawbacks. To avoid all these shortcomings, Java SE 9 is coming with Module System.

3. JDK 9 New Features

5. Process API Improvements

Java SE 9 is coming with some improvements in Process API. They have added couple new classes and methods to ease the controlling and managing of OS processes. Two new interfaces in Process API:

1. java.lang.ProcessHandle
2. java.lang.ProcessHandle.Info

Process API example

```
ProcessHandle currentProcess = ProcessHandle.current();
System.out.println("Current Process Id: = " + currentProcess.getPid());
```

6. Try With Resources Improvement

We know, Java SE 7 has introduced a new exception handling construct: Try-With-Resources to manage resources automatically. The main goal of this new statement is "Automatic Better Resource Management". Java SE 9 is going to provide some improvements to this statement to avoid some more verbosity and improve some Readability.

Java SE 7 example

```
void testARM_Before_Java9() throws IOException {
 BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));
 try (BufferedReader reader2 = reader1) { System.out.println(reader2.readLine()); }
}
```

Java SE 9 example

```
void testARM_Java9() throws IOException {
 BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));
 try (reader1) { System.out.println(reader1.readLine()); }
}
```

3. JDK 9 New Features

Java 9 New Features:

7. CompletableFuture API Improvements
8. Reactive Streams
9. Diamond Operator for Anonymous Inner Class
10. Optional Class Improvements
11. Functional Programming - Stream API Improvements
12. Enhanced @Deprecated annotation
13. HTTP 2 Client
14. Multi-Resolution Image API
15. Miscellaneous Java 9 Features

- GC (Garbage Collector) Improvements
- Stack-Walking API
- Filter Incoming Serialization Data
- Deprecate the Applet API
- Indify String Concatenation
- Enhanced Method Handles
- Java Platform Logging API and Service
- Compact Strings
- Parser API for Nashorn
- Javadoc Search
- HTML5 Javadoc

Questions & Answers!

But wait…

There's More!

Thanks!

Java SE
End of Lecture 11 – Summary of Java SE NIO,
RegEx & JDK 9 and 10