Power Programming with Java

Session: 2

Java Packages

Objectives

- Describe the `java.lang` package
- Explain the various classes of `java.lang` package
- Explain how to use and manipulate Strings
- Explain regular expressions, pattern, and matcher
- Explain String literal and Character classes
- Explain the use of quantifiers, capturing groups, and boundary matchers

java.lang Package

Enables developers to manipulate data belonging to various formats such as strings, numbers, characters, and so on.

Provides classes that are fundamental for creation of a Java program.

Includes root classes forming class hierarchy, basic exceptions, types tied to language definition, threading, math functions, security functions, and information on underlying native system.

Includes following important classes:

Object: The root of the class hierarchy

Class: Instances of this class represent classes at runtime

Is implicitly imported in every program. Hence, an explicit `import` statement is not required for using this package.

Working with Garbage Collection

Garbage collector is an automatic memory management program. Garbage collection helps to avoid the problem of dangling references.

Garbage collection also solves the problem of memory leak problem.

Following parameters must be studied while designing or selecting a garbage collection algorithm:

- Serial versus Parallel
- Concurrent versus Stop-the-world
- Compacting versus Non-compacting versus Copying

Following metrics can be utilized to evaluate the performance of a garbage collector:

- Throughput
- Garbage collection overhead
- Pause time
- Frequency of collection
- Footprint
- Promptness

Classes of the java-lang Package [1-7]

Wrapper Classes

A typical wrapper class contains a value of primitive data type and various methods for managing the data types.

Wrapper classes are used to manage primitive values as objects.

Each of these classes wraps a primitive data types within a class.

An object of type Integer, for example, contains a field whose type is `int`.

It represents that value in such a way that a reference to it, can be stored in a variable of reference type.

The wrapper classes also provide a number of methods for processing variables of specified data type to another type.

Classes of the java.lang Package [2-7]

Math Class

Contains methods for performing basic mathematical/numeric operations

By default, many of its methods call the equivalent method of the `StrictMath` class for their implementation.

Some of the commonly used methods of the `Math` class are as follows:

- `static double abs(double a)`
- `static float abs(float a)`
- `static int abs(int a)`

Classes of the java-lang Package [3-7]

System Class

Provides several useful class fields and methods. However, it cannot be instantiated.

It provides several facilities such as standard input, standard output, and error output streams, a means of loading files and libraries, and so on.

Commonly used methods of `System` class:

- `static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`
- `static long currentTimeMillis()`

Classes of the java.lang Package [4-7]

Object Class

Is the root of the class hierarchy.

Is the superclass for every class in a program.

All objects, including arrays, implement methods of this class.

Commonly used methods of `Object`:

```
protected Object clone()
```

```
boolean equals(Object obj)
```

```
protected void finalize()
```

Classes of the java.lang Package [5-7]

Class

- In a running program, instances of `Class` class represent classes and interfaces.
- An array belongs to a class reflected as a `Class` object shared by all arrays with same element type and number of dimensions.
- Primitive Java data types such as `boolean`, `byte`, and `char` are also represented as `Class` objects.
- `Class` objects are constructed automatically by JVM, as classes are loaded and by calling the `defineClass()` method in the class loader.

Commonly used methods of `Class`:

`static Class forName(String className)`	`static Class forName(String name, boolean initialize, ClassLoader loader)`	`Class[] getClasses()`	`Field getField(String name)`	`Class[]getInterfaces()`

ThreadGroup Class

- A thread group represents a set of threads. Besides this, a thread group can also include other thread groups.
- The thread groups forms a tree in which all the thread group except the initial thread group has a parent.
- Commonly used methods of `ThreadGroup` class:
  - `int activeCount()`
  - `int activeGroupCount()`
  - `void checkAccess()`
  - `void destroy()`

Runtime Class

- There is a single instance of class `Runtime` for every Java application allowing the application to interface with the environment in which it is running.
- The current runtime is obtained by invoking the `getRuntime()` method.
- An application cannot create its own instance of this class.
- Commonly used methods of the `Runtime` class are as follows:
  - `int availableProcessors()`
  - `Process exec(String command)`
  - `void exit(int status)`
  - `long freeMemory()`

Strings

Strings are widely used in Java programming.

Strings are nothing but a sequence of characters.

In the Java programming language, strings are objects.

The Java platform provides the `String` class to create and manipulate strings.

Whenever a string literal is encountered in a code, the compiler creates a `String` object with its value.

String Class

- The `String` class represents character strings.
- All string literals in Java programs, such as 'xyz', are implemented as instances of the `String` class.

Syntax

```
public final class String extends Object
implements Serializable, Comparable<String>, CharSequence
```

- Strings are constant, that is, their values cannot be changed once created.
- However, string buffers support mutable strings. Since, `String` objects are immutable, they can be shared.
- Similar to other objects, a `String` object can be created by using the `new` keyword and a constructor.
- The `String` class has 13 overloaded constructors that allow specifying the initial value of the string using different sources.

String Methods

String class provides several methods for manipulating strings.

- `char charAt(int index)`
- `int compareTo(String anotherString)`
- `String concat(String str)`
- `Boolean contains(CharSequence s)`
- `boolean endsWith(String suffix)`
- `boolean equals(Object anObject)`
- `Boolean equalsIgnoreCase(String anotherString)`
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
- `int indexOf(int ch)`
- `boolean isEmpty()`
- `int lastIndexOf(int ch)`
- `int length()`
- `boolean matches(String regex)`
- `String replace(char oldChar, char newChar)`
- `String[]split(String regex)`
- `String substring(int beginIndex)`
- `char[]toCharArray()`
- `String toLowerCase()`
- `String toString()`
- `String toUpperCase()`
- `String trim()`

StringBuilder and StringBuffer Classes [1-2]

`StringBuilder` objects are same as String objects, except that they are mutable.

Internally, runtime treats these objects similar to variable-length arrays containing a sequence of characters.

The `StringBuilder` class also has a `length()` method that returns length of the character sequence in the builder.

The main operations on a `StringBuilder` class that the `String` class does not possess, are `append()` and `insert()` methods.

The `StringBuilder` class provides `void setLength(int newLength)` and `void ensureCapacity(int minCapacity)` methods related to length and capacity which are not available with the `String` class.

StringBuilder and StringBuffer Classes [2-2]

StringBuffer:

- Creates a thread-safe, mutable sequence of characters.
- `StringBuilder` class should be preferred over `StringBuffer`, as it is faster since it performs no synchronization.
- Is declared as follows:

```
public final class StringBuffer extends Object
   implements Serializable, CharSequence
```

- All operations that can be performed on `StringBuilder` class are also applicable to `StringBuffer` class.

String resolveConstantDesc

- Is a public bytecode parsing method introduced in Java 12.
- Is declared in `ConstantDesc` interface implemented within the `String` class.

Code Snippet shows an example for the `resolveConstantDesc()` method.

Code Snippet

```
public class StringResolveConstantDescDemo {
   public static void main(String[] args) {
      String strA = "Don't just learn the tricks of the trade,
         learn the trade.";
      String strB = strA.resolveConstantDesc(null);
      System.out.println("Outcome of the code: Comparing the
      objects");
      System.out.println(strB.equals(strA));
      System.out.println(strB == strA);
   }
}
```

indent (int n)

Was introduced in Java SE 12 in `java.lang.String` class

Is used for adding or removing whitespaces from start of a line to fine-tune indentation for each string line

Syntax for `indent (int n)`:

- `public String indent (int n)`

transform (Function f)[1-2]

Is used to apply a function to an input string. A single string argument should be accepted by this function returning an object.

Accepts a `String` as input and transforms it into a new `String` using a Function. Addition of this utility method was done to `String` class in Java 12.

A String input data pill is required in String transform functions.

When the consumer reads the Pill, this item shuts down for a separate distributed consumption process.

Syntax:

```
public <R> R transform(Function<? super String, ? extends R> f) {
return f.apply(this); }
```

where, `Function` is a functional interface that allows one argument and creates an object of type `R`.

transform (Function f)[2-2]

Code Snippet

```
public class String transform {
    public static void main(String[] args) {
        String str = "Life's too short";
        var result = str.transform(input -> input.concat(" to eat bad
            food")).transform(String::toUpperCase);
        System.out.println(String(result));
    }
}
```

Code Snippet transforms the string input to single corrector output.

Output:

```
[L, I, F, E, ', S, , T, O, O, , S, H, O, R, T, , T, O, , E, A, T,
, B, A, D, , F, O, O, D]
```

Optional describeConstable()

Is a method from the class `String`.

It is not necessary for a Constable to (or may choose not to) explain all its instances in the form of a `ConstantDesc`.

Returns an empty Optional to show that a nominal descriptor could not be generated for an instance.

Is also a Byte level method obtained from interface Constable.

It returns an optional instance of object which includes nominal descriptor for this instance.

Parsing of Text Using StringTokenizer Class

Ways of parsing text:

- `String.split()` method
- `StringTokenizer` and `StreamTokenizer` classes
- `Scanner` class
- `Pattern` and `Matcher` classes, which implement regular expressions
- For the most complex parsing tasks, tools such as JavaCC can be used

`StringTokenizer` class belongs to `java.util` package and is used to break a string into tokens.

An instance of `StringTokenizer` class internally maintains a current position within the string to be tokenized.

`StringTokenizer` is a legacy class that has been retained for compatibility reasons and its use is discouraged in new code.

Regular Expressions

- Regular expressions are used to describe a set of strings based on the common characteristics shared by individual strings in the set.
- They are used to edit, search, or manipulate text and data.
- To create regular expressions, one must learn a particular syntax beyond normal syntax of Java.
- Regular expressions differ in complexity, but once the basics of their creation are understood, it is easy to decipher or create any regular expression.
- For creating regular expressions, there are many different options available such as Perl, grep, Python, Tcl, Python, awk, and PHP.
- In Java, one can use `java.util.regex` API to create regular expressions.
- The syntax for regular expression in the `java.util.regex` API is very similar to that of Perl.

Regular Expression API

Three classes in the `java.util.regex` package required for creation of regular expression:

`Pattern`	`Matcher`	`PatternSyntaxExpression`
A `Pattern` object is a compiled form of a regular expression.	A `Matcher` object is used to interpret pattern and perform match operations against an input string.	`PatternSyntaxExpression` object is unchecked exception used to indicate a syntax error in a regular expression pattern.

Pattern Class

- Any regular expression that is a string must first be compiled into an instance of `Pattern` class.
- Resulting `Pattern` object can then be used to create a `Matcher` object.
- Once the `Matcher` object is obtained, it can then match arbitrary character sequences against the regular expression.
- All different state involved in performing a match resides in the matcher, so several matchers can share the same pattern.

Syntax

```
public final class Pattern
extends Object
implements Serializable
```

- `matches()` method of `Matcher` is used when a regular expression appears just once.

Matcher Class [1-2]

- A `Matcher` object is created from a pattern by invoking `matches()` method on the `Pattern` object.
- A `Matcher` object is the engine that performs the match operations on a character sequence by interpreting a Pattern.

Syntax

```
public final class Matcher extends Object implements MatchResult
```

- After creation, a `Matcher` object can be used to perform three different types of match operations:

The `matches()` method is used to match the entire input sequence against the pattern.

The `lookingAt()` method is used to match the input sequence, from the beginning, against the pattern.

The `find()` method is used to scan the input sequence looking for the next subsequence that matches the pattern.

Matcher Class [2-2]

`Matcher` class has index methods that provide useful index values to indicate exactly where match was found in input string.

Explicit state of a `matcher` includes:

- Start and end indices of the most recent successful match.
- Start and end indices of the input subsequence captured by each capturing group in the pattern.
- Total count of such subsequences.

The implicit state of a `matcher` includes:

- input character sequence.
- append position, which is initially zero.

The `reset()` method helps the matcher to be explicitly reset.

If a new input sequence is desired, the `reset(CharSequence)` method can be invoked.

The reset operation on a matcher discards its explicit state information and sets the append position to zero.

String Literal

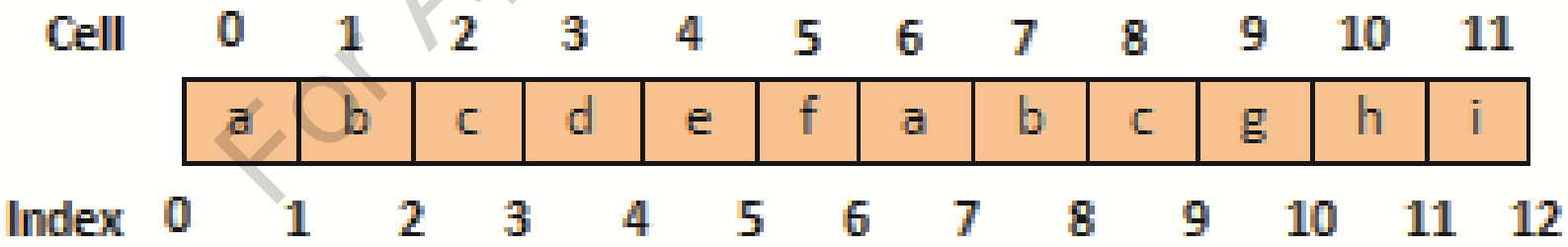
The most basic form of pattern matching supported by the `java.util.regex` API is the match of a string literal.

The match will succeed because the regular expression is found in the string.

Note that in the match, the start index is counted from 0.

By convention, ranges are inclusive of the beginning index and exclusive of the end index.

Each character in the string resides in its own cell, with the index positions pointing between each cell as shown in the following figure:



Metacharacters

This API also supports many special characters.

The match still succeeds, even though the dot '.' is not present in the input string.

This is because the dot is a metacharacter, that is, a character with special meaning as interpreted by the matcher.

For the matcher, the metacharacter '.' stands for 'any character'.

The metacharacters supported by the API are: <([{\^-=$!|]})?*+.>

Character Classes

- The word ‘class’ in ‘character class’ phrase does not mean a `.class` file.
- With respect to regular expressions, a character class is a set of characters enclosed within square brackets.
- It indicates characters that will successfully match a single character from a given input string.
- Following table summarizes supported regular expression constructs in ‘Character Classes’:

Construct	Type	Description
[abc]	Simple class	a, b, or c
[^abc]	Negation	Any character except a, b, or c
[a-zA-Z]	Range	a through z, or A through Z (inclusive
[a-d[m-p]]	Union	a through d, or m through p: [a-dm-p]
[a-z&&[def]]	Intersection	d, e, or f
[a-z&&[^bc]]	Subtraction	a through z, except for b and c: [ad-z]
[a-z&&[^m-p]]	Subtraction	a through z, and not m through p: [a-lq-z]

Simple Classes

This is the most basic form of a character class.

It is created by specifying a set of characters side-by-side within square brackets.

For example, the regular expression [fmc]at will match the words ‘fat’, ‘mat’, or ‘cat’.

This is because the class defines a character class accepting either ‘f’, ‘m’, or ‘c’ as the first character.

Negation

- Negation is used to match all characters except those listed in the brackets.
- The '^' metacharacter is inserted at the beginning of the character class to implement Negation.
- Following figure shows the use of Negation:

```
C:\WINDOWS\system32\cmd.exe - java RegexTest

E:\>java RegexTest

Enter expression: [^fmc]at
Enter string to search: fat
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [^fmc]at
Enter string to search: mat
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [^fmc]at
Enter string to search: cat
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [^fmc]at
Enter string to search: rat
Found the text "rat" starting at index 0 and ending at index 3.
Press x to exit or y to continue
Enter your choice: _
```

Ranges

At times, it may be required to define a character class that includes a range of values, such as the letters 'a to f' or numbers '1 to 5'.

A range can be specified by simply inserting the '-' metacharacter between the first and last character to be matched.

For example, `[a-h]` or `[1-5]` can be used for a range.

One can also place different ranges next to each other within the class in order to further expand the match possibilities.

For example, `[a-zA-Z]` will match any letter of the alphabet from a to z (lowercase) or A to Z (uppercase).

- Following figure shows the use of Range and Negation:

```
E:\>java RegexTest

Enter expression: [p-t]
Enter string to search: s
Found the text "s" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [p-t]
Enter string to search: q
Found the text "q" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: rno[5-9]
Enter string to search: rno7
Found the text "rno7" starting at index 0 and ending at index 4.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [x-z]
Enter string to search: a
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: rno[5-9]
Enter string to search: rno2
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: rno[^5-9]
Enter string to search: rno2
Found the text "rno2" starting at index 0 and ending at index 4.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [1-5]
Enter string to search: 5
Found the text "5" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: x

E:\>
```

Unions

- Unions can be used to create a single character class comprising two or more separate character classes.
- This can be done by simply nesting one class within the other.
- For example, the union `[a-d[f-h]]` creates a single character class that matches the characters `a`, `b`, `c`, `d`, `f`, `g`, and `h`.
- Following figure shows the use of Unions:

```
E:\>java RegexTest

Enter expression: [a-d[f-h]]
Enter string to search: c
Found the text "c" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [a-d[f-h]]
Enter string to search: g
Found the text "g" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [a-d[f-h]]
Enter string to search: e
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [a-d[f-h]]
Enter string to search: i
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Intersections

- Intersection is used to create a single character class that matches only the characters which are common to all of its nested classes.
- This is done by using the &&, such as in `[0-6&&[234]]`.
- This creates a single character class that will match only the numbers common to both character classes, that is, 2, 3, and 4.
- Following figure shows the use of Intersections:

```
E:\>java RegexTest

Enter expression: [0-6&&[234]]
Enter string to search: 3
Found the text "3" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[234]]
Enter string to search: 2
Found the text "2" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[234]]
Enter string to search: 4
Found the text "4" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[234]]
Enter string to search: 5
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Subtraction

- Subtraction can be used to negate one or more nested character classes, such as `[0-6&&[^234]]`. In this case, the character class will match everything from 0 to 6, except the numbers 2, 3, and 4.

- Following figure shows the use of Subtraction:

```
E:\>java RegexTest

Enter expression: [0-6&&[^234]]
Enter string to search: 2
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[^234]]
Enter string to search: 3
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[^234]]
Enter string to search: 4
No match found.
Press x to exit or y to continue
Enter your choice: y

Enter expression: [0-6&&[^234]]
Enter string to search: 5
Found the text "5" starting at index 0 and ending at index 1.
Press x to exit or y to continue
Enter your choice: x

E:\>
```

Pre-defined Character Classes

- Table lists the pre-defined character classes.

Construct	Description
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Quantifiers

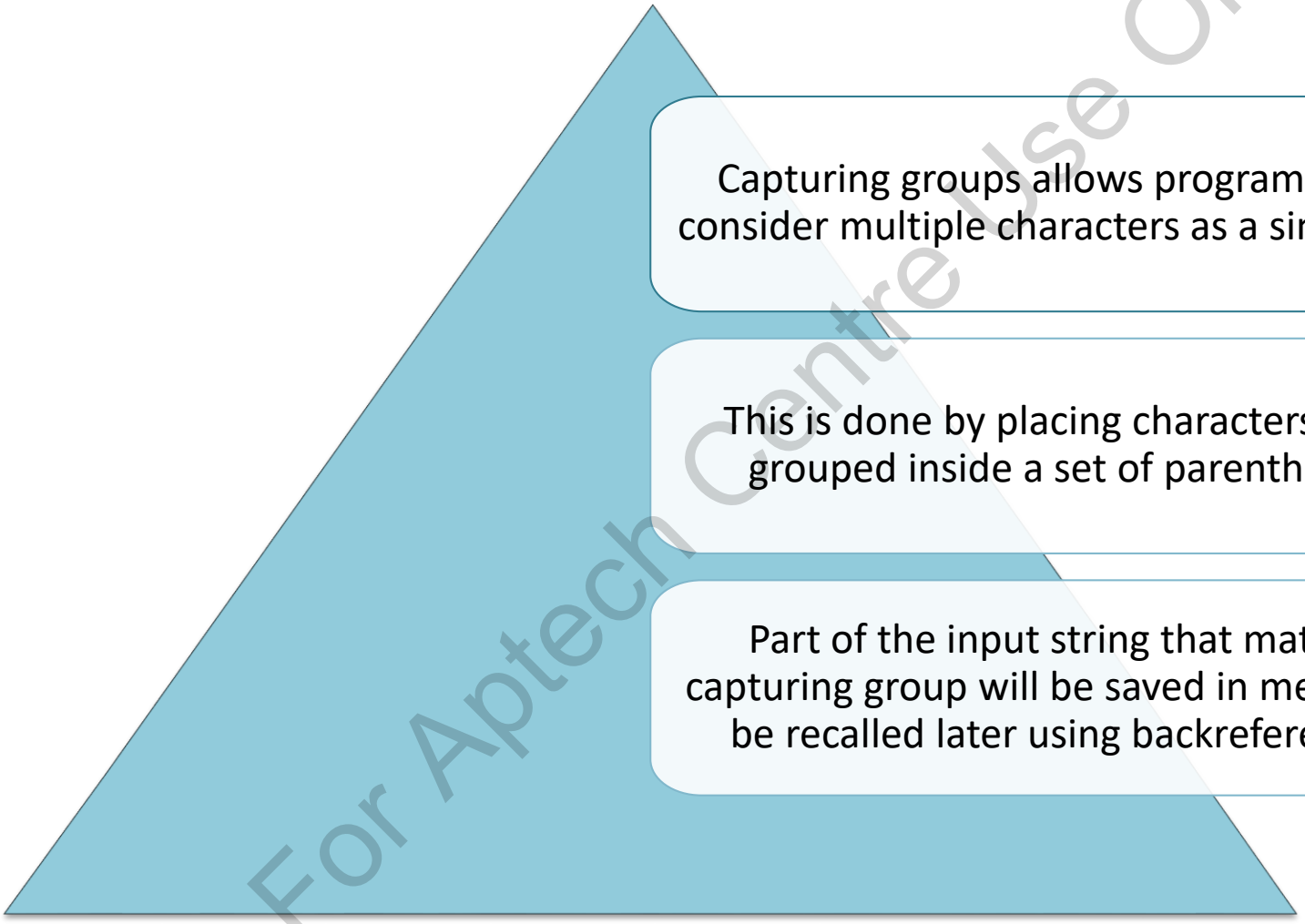
- Quantifiers can be used to specify number of occurrences to match against.
- Following table shows the greedy, reluctant, and possessive quantifiers:

Greedy	Reluctant	Possessive	Description
X?	X??	X?+	once or not at all
X*	X*?	X*+	zero or more times
X+	X+?	X++	one or more times
X{n}	X{n}?	X{n}+	exactly n times
X{n,}	X{n,}?	X{n,}+	at least n times
X{n,m}	X{n,m}?	X{n,m}+	at least n but not more than m times

Differences among the Quantifiers

Greedy	Reluctant	Possessive
Greedy quantifiers are termed 'greedy' because they force matcher to read entire input string before to attempting first match.	Reluctant quantifiers take the opposite approach.	Possessive quantifiers always eat entire input string, trying once and only once for a match.
If in first attempt to match the entire input string, fails, then, matcher backs off input string by one character and tries again.	They start at the beginning of the input string and then, reluctantly read one character at a time looking for a match.	Unlike greedy quantifiers, they never back off, even if doing so would allow the overall match to succeed.
It repeats the process until a match is found or there are no more characters left to back off from.	Last thing they try is to match the entire input string.	
Depending on quantifier used in the expression, last thing it will attempt is to try to match against 1 or 0 characters.		

Capturing Groups



Capturing groups allows programmer to consider multiple characters as a single unit.

This is done by placing characters to be grouped inside a set of parentheses.

Part of the input string that matches capturing group will be saved in memory to be recalled later using backreferences.

Numbering [1-2]

Capturing groups are numbered by counting their opening parentheses from left to right.

`groupCount()` method can be invoked on matcher object to find out how many groups are present in expression.

This method will return an `int` value indicating number of capturing groups present in the matcher's pattern.

There is another special group, group 0, which always represents entire expression.

However, this group is not counted in the total returned by `groupCount()`.

Groups beginning with the character '**?**' are pure, non-capturing groups as they do not capture text and also do not count towards the group total.

Numbering [2-2]

Following Code Snippet is an example of using `groupCount()`:

Code Snippet

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class RegexTest1{
    public static void main(String[] args){
        Pattern pattern1 =
            Pattern.compile("((X)(Y(Z)))");
        Matcher matcher1 =
            pattern1.matcher("((X)(Y(Z)))");
        System.console().format("Group count is:
            %d",matcher1.groupCount());
    }
}
```

Backreferences

- The portion of input string matching capturing group(s) is saved in memory for later recall with the help of backreference.
- A backreference is specified in the regular expression as a backslash (\) followed by a digit indicating the number of the group to be recalled.
- For example, the expression `(\d\d)` defines one capturing group matching two digits in a row, which can be recalled later in the expression by using the backreference `\1`.
- Following figure shows an example for using backreferences:

```
E:\>java RegexTest

Enter expression: (\d\d)\1
Enter string to search: 2323
Found the text "2323" starting at index 0 and ending at index 4.
Press x to exit or y to continue
Enter your choice: y

Enter expression: (\d\d)\1
Enter string to search: 2312
No match found.
Press x to exit or y to continue
Enter your choice: x

E:\>_
```

Boundary Matchers

- Table lists the boundary matchers.

Boundary Matchers	Description
^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Additional Methods of the `Pattern` Class

- Until now, the `RegexTest` class has been used to create `Pattern` objects in their most basic form.
- One can also use advanced techniques such as creating patterns with flags and using embedded flag expressions.

Creating a Pattern with Flags:

- `Pattern` class provides an alternate `compile()` method that accepts a set of flags.
- These flags affect the way the pattern is matched.

Embedded Flag Expressions

- Embedded flag expressions can also be used to enable various flags. They are an alternative to the two-argument version of `compile()` method.
- They are specified in the regular expression itself.
- Following example uses original `RegexTest.java` class with the embedded flag expression (?i) to enable case-insensitive matching:

```
Enter your regex: (?i)bat
Enter input string to search: BATbatBaTbaT
I found the text "BAT" starting at index 0 and ending at index
3.
I found the text "bat" starting at index 3 and ending at index
6.
I found the text "BaT" starting at index 6 and ending at index
9.
I found the text "baT" starting at index 9 and ending at index
12.
```

The matches(String CharSequence) Method

The `Pattern` class defines the `matches()` method that allows the programmer to quickly check if a pattern is present in a given input string.

Similar, to all public static methods, the `matches()` method is invoked by its class name, that is, `Pattern.matches("\\d","1");`.

In this case, the method will return true, because the digit `'1'` matches the regular expression `'\d'`.

The split(String) Method

- The `split()` method of `Pattern` class is used for obtaining the text that lies on either side of the pattern being matched.
- Consider the `SplitTest.java` class in the following Code Snippet:

Code Snippet

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class SplitTest{
private static final String REGEX = ":";
 private static final String DAYS = "Sun:Mon:Tue:Wed:Thu:Fri:Sat";
 public static void main(String[] args) {
  Pattern objP1 = Pattern.compile(REGEX);
  String[] days = objP1.split(DAYS);
  for(String s : days) {
   System.out.println(s);
  }
 }
}
```

Other Useful Methods

`public static String quote(String s):`

- This method returns a literal pattern `String` for the specified String argument.
- This `String` produced by this method can be used to create a pattern that would match the argument, `s` as if it were a literal pattern.
- Metacharacters or escape sequences in the input string will hold no special meaning.

`public String toString():`

Returns the `String` representation of this pattern.

Summary

- `java.lang` package provides classes that are fundamental for the creation of a Java program.
- Garbage collection solves the problem of memory leak because it automatically frees all memory that is no longer referenced.
- In stop-the-world garbage collection approach, during garbage collection, application execution is completely suspended.
- `finalize()` method is called by the garbage collector on an object when it is identified to have no more references pointing to it.
- `Object` class is root of the class hierarchy. Every class has `Object` as a superclass.
- All objects, including arrays, implement methods of `Object` class. `StringBuilder` objects are same as String objects, except that they are mutable.
- Internally, the runtime treats these objects similar to variable-length arrays containing a sequence of characters.
- The `StringTokenizer` class belongs to the `java.util package` and is used to break a string into tokens.
- Any regular expression that is specified as a string must first be compiled into an instance of the Pattern class.
- A Matcher object is the engine that performs the match operations on a character sequence by interpreting a Pattern.
- Intersection is used to create a single character class that matches only the characters which are common to all of its nested classes.
- Greedy quantifiers are termed ‘greedy’ because they force the matcher to read entire input string before to attempting the first match. Reluctant quantifiers take the opposite approach. Possessive quantifiers always eat entire input string, trying once and only once for a match.