Object-oriented Programming in Java

Session: 2

java.lang Package

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

Introduction

- While writing programs in Java, it is often required to perform certain tasks on the data specified by the user.
- The data could be in any format such as strings, numbers, characters, and so on.
- To manipulate such data, special classes and methods are required.
- They are provided by a special package in Java called the `java.lang` package.

Overview of the java.lang Package

- The java.lang package provides classes that are fundamental for the creation of a Java program.
- This includes the root classes that form the class hierarchy, basic exceptions, types tied to the language definition, threading, math functions, security functions, and information on the underlying native system.
- The most important classes are as follows:
  - **Object**: Which is the root of the class hierarchy.
  - **Class**: Instances of this class represent classes at run time.
- The other important classes and interfaces in java.lang package are as follows:
  - Enum
  - Throwable
  - Error, Exception, and RuntimeException
  - Exception classes thrown for language-level and other common exceptions
  - Thread and String
  - StringBuffer and StringBuilder
  - Comparable and Iterable
  - Process, ClassLoader, Runtime, System, and SecurityManager
  - Math
  - Wrapper classes that encapsulate primitive types as objects

Working with Garbage Collection [1-3]

- 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.
- The 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
- The following metrics can be utilized to evaluate the performance of a garbage collector:
  - **Throughput**: It is the percentage of total time not spent in garbage collection, considering a longer time period.
  - **Garbage collection overhead**: It is the inverse of throughput. That is, the percentage of total time spent in garbage collection.
  - **Pause time**: It is the amount of time during which application execution is suspended while garbage collection is occurring.
  - **Frequency of collection**: It is a measure of how often collection occurs in relation to application execution.
  - **Footprint**: It is a measure of size, such as heap size.
  - **Promptness**: It is the time span between the time an object becomes garbage and the time when its memory becomes available.

Working with Garbage Collection [2-3]

- An important method for garbage collection is the `finalize()` method.
- The `finalize()` method is called by the garbage collector on an object when it is identified to have no more references pointing to it.
- A subclass overrides the `finalize()` method for disposing the system resources or to perform other cleanup.
- The following Code Snippet shows an example of automatic garbage collection:

Code Snippet

```
class TestGC{
 int num1;
 int num2;

public void setNum(int num1,int num2){
        this.num1=num1;
        this.num2=num2;
  }
  public void showNum(){
    System.out.println("Value of num1 is " + num1);
    System.out.println("Value of num2 is " + num2);
   }
```

Working with Garbage Collection [3-3]

```
 public static void main(String args[]){
             TestGC obj1 = new TestGC();
             TestGC obj2 = new TestGC();
             obj1.setNum(2,3);
             obj2.setNum(4,5);
             obj1.showNum();
             obj2.showNum();
             //TestGC obj3; // line 1
             //obj3=obj2; // line 2
             //objGC3.showNum(); // line 3
             //obj2=null; // line 4
            //obj3.showNum(); // line 5
            //obj3=null; // line 6
            //obj3.showNum(); // line 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.

Math Class [1-4]

- The `Math` class contains methods for performing basic mathematical/numeric operations such as square root, trigonometric functions, elementary exponential, logarithm, and so on.
- By default, many of the `Math` methods simply call the equivalent method of the `StrictMath` class for their implementation.
- The following lists some of the commonly used methods of the `Math` class:
  - `static double abs(double a)`
  - `static float abs(float a)`
  - `static int abs(int a)`
  - `static long abs(long a)`
  - `static double ceil(double a)`
  - `static double cos(double a)`
  - `static double exp(double a)`
  - `static double floor(double a)`
  - `static double log(double a)`
  - `static double max(double a, double b)`
  - `static float max(float a, float b)`
  - `static int max(int a, int b)`

Math Class [2-4]

The following Code Snippet shows the use of some of the methods of `Math` class:

Code Snippet

```
// creating a class to use Math class methods
class MathClass {

 int num1; // declaring variables
 int num2;
 // declaring constructors
 public MathClass(){}
 public MathClass(int num1, int num2){
  this.num1 = num1;
  this.num2 = num2;

 }
```

Math Class [3-4]

```
 // method to use max()
 public void doMax(){
  System.out.println("Maximum is: " + Math.max(num1,num2));

}
 // method to use min()
 public void doMin(){
  System.out.println("Minimum is: " + Math.min(num1,num2));
 }
 // method to use pow()
 public void doPow(){
   System.out.println("Result of power is: " +
Math.pow(num1,num2));
 }
 // method to use random()
 public void getRandom(){
  System.out.println("Random generated is: " + Math.random());


 }
```

Math Class [4-4]

```
 // method to use sqrt()
 public void doSquareRoot(){
  System.out.println("Square Root of " + num1 +" is: " +
Math.sqrt(num1));
 }


}


public class TestMath {
 public static void main(String[] args) {
  MathClass objMath = new MathClass(4,5);
  objMath.doMax();
  objMath.doMin();
  objMath.doPow();
  objMath.getRandom();
  objMath.doSquareRoot();


  }
 }
```

System Class [1-3]

- The `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, access to externally defined properties and environment variables, and a utility method for quickly copying a part of an array.
- The following lists some of the commonly used methods of the `System` class:
  - `static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`
  - `static long currentTimeMillis()`
  - `static void exit(int status)`
  - `static void gc()`
  - `static String getenv(String name)`
  - `static Properties getProperties()`
  - `static void loadLibrary(String libname)`
  - `static void setSecurityManager(SecurityManager s)`

System Class [2-3]

The following Code Snippet shows the use of some of the methods of `System` class:

Code Snippet

```
class SystemClass {
 int arr1[] = {1,3,2,4};
 int arr2[] = {6,7,8,0};


 public void getTime()
 {
     System.out.println("Current time in milliseconds is: " +
System. currentTimeMillis());
 }


 public void copyArray()
 {
  System.arraycopy(arr1, 0, arr2, 0, 3);
  System.out.println("Copied array is: " );
```

System Class [3-3]

```
  for(int i=0; i<4; i++)
  System.out.println(arr2[i]);
 }
 public void getPath(String variable)
 {
  System.out.println("Value of Path variable is: " +
System.getenv(variable));
  }
 }


 public class TestSystem {
  public static void main(String[] args) {
    SystemClass objSys = new SystemClass();
  objSys.getTime();
  objSys.copyArray();
  objSys.getPath("Path");
  }
}
```

Object Class [1-3]

- `Object` class is the root of the class hierarchy.
- Every class has `Object` as a superclass.
- All objects, including arrays, implement the methods of the `Object` class.
- The following lists some of the commonly used methods of the `Object` class:
  - `protected Object clone()`
  - `boolean equals(Object obj)`
  - `protected void finalize()`
  - `Class<? extends Object> getClass()`
  - `int hashCode()`
  - `void notify()`
  - `void notifyAll()`
  - `String toString()`
  - `void wait()`
  - `void wait(long timeout)`
  - `void wait(long timeout, int nanos)`

Object Class [2-3]

The following Code Snippet shows the use of some of the methods of `Object` class:

Code Snippet

```
class ObjectClass {
   Integer num;
   public ObjectClass(){}
   public ObjectClass(Integer num){
     this.num = num;
   }
   // method to use the toString() method
   public void getStringForm(){
      System.out.println("String form of num is: " +
num.toString());
  }
 }
 public class TestObject {
```

Object Class [3-3]

```
        // creating objects of ObjectClass class
        ObjectClass obj1 = new ObjectClass(1234);
        ObjectClass obj2 = new ObjectClass(1234);
        obj1.getStringForm();
        // checking for equality of objects
        if (obj1.equals(obj2))
          System.out.println("Objects are equal");
        else
          System.out.println("Objects are not equal");
         obj2=obj1; // assigning reference of obj1 to obj2
         // checking the equality of objects
        if (obj1.equals(obj2))
         System.out.println("Objects are equal");
        else
         System.out.println("Objects are not equal");
    }
}
```

Class Class [1-2]

- In an executing Java program, instances of the `Class` class represent classes and interfaces.
- An array belongs to a class that is reflected as a `Class` object that is shared by all arrays with the same element type and number of dimensions.
- The primitive Java data types such as `boolean`, `byte`, and `char` also represented as `Class` objects.
- `Class` objects are constructed automatically by the JVM, as the classes are loaded and by calling the `defineClass()` method in the class loader.
- The following lists some of the commonly used methods of the `Class` class:
  - `static Class forName(String className)`
  - `static Class forName(String name, boolean initialize, ClassLoader loader)`
  - `Class[]getClasses()`
  - `Field getField(String name)`
  - `Class[]getInterfaces()`

Class Class [2-2]

The following Code Snippet shows the use of some of the methods of `Class` class:

Code Snippet

```
class ClassClass extends MathClass{
    public ClassClass(){}
}
public class TestClass {
 public static void main(String[] args) {
     ClassClass obj = new ClassClass();
     System.out.println("Class is: " + obj.getClass());
   }
}
```

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.
- The following lists some of the commonly used methods of the `ThreadGroup` class:
  - `int activeCount()`
  - `int activeGroupCount()`
  - `void checkAccess()`
  - `void destroy()`
  - `int enumerate(Thread[] list)`
  - `int enumerate(ThreadGroup[] list)`
  - `int getMaxPriority()`
  - `String getName()`
  - `ThreadGroup getParent()`
  - `void interrupt()`
  - `boolean isDaemon()`

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.
- The following lists some of the commonly used methods of the `Runtime` class:
  - `int availableProcessors()`
  - `Process exec(String command)`
  - `void exit(int status)`
  - `long freeMemory()`
  - `void gc()`
  - `static Runtime getRuntime()`
  - `void halt(int status)`
  - `void load(String filename)`

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.
- The syntax of `String` class is as follows:

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

- 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 [1-4]

- `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 length and content of the sequence can be modified at any point through certain method calls.
- It is advisable to use `String` unless `StringBuilder` offer an advantage in terms of simpler code or better performance.
- The `StringBuilder` class also has a `length()` method that returns the length of the character sequence in the builder.
- The following lists the constructors of `StringBuilder` class:
  - `StringBuilder()`
  - `StringBuilder(CharSequence cs)`
  - `StringBuilder(int initCapacity)`
  - `StringBuilder(String s)`

StringBuilder and StringBuffer Classes [2-4]

The following Code Snippet explains the use of `StringBuilder`:

Code Snippet

```
class StringBuild {
 // creating string builder
 StringBuilder sb = new StringBuilder(); // line 1


 public void addString(String str){
    // appending string to string builder
    sb.append(str); // line 2
    System.out.println("Final string is: " +
     sb.toString());
}
}
public class TestStringBuild {
```

StringBuilder and StringBuffer Classes [3-4]

```
public static void main(String[] args) {
   StringBuild sb = new StringBuild();
   sb.addString("Java is an ");
   sb.addString("object-oriented ");
   sb.addString("programming ");
   sb.addString("language.");
 }
}
```

- The `StringBuilder` class provides some methods related to length and capacity which are not available with the `String` class. They are:
  - `void setLength(int newLength)`
  - `void ensureCapacity(int minCapacity)`
- The main operations on a `StringBuilder` class that the `String` class does not possess, are the `append()` and `insert()` methods.

StringBuilder and StringBuffer Classes [4-4]

StringBuffer:

- The `StringBuffer` creates a thread-safe, mutable sequence of characters.
- Since JDK 5, this class has been supplemented with an equivalent class designed for use by a single thread, `StringBuilder`.
- The `StringBuilder` class should be preferred over `StringBuffer`, as it supports all of the same operations but it is faster since it performs no synchronization.
- The `StringBuffer` class declaration is 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.

Parsing of Text Using StringTokenizer Class [1-3]

- There are different ways of parsing text. The usual tools are as follows:
  - `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
- The `StringTokenizer` class belongs to the `java.util` package and is used to break a string into tokens. The class is declared as follows:

```
public class StringTokenizer
extends Object
implements Enumeration
```

- The following lists the constructors of `StringTokenizer` class:
  - `StringTokenizer(String str)`
  - `StringTokenizer(String str, String delim)`
  - `StringTokenizer(String str, String delim, boolean returnDelims)`
- An instance of `StringTokenizer` class internally maintains a current position within the string to be tokenized.

Parsing of Text Using StringTokenizer Class [2-3]

- The following lists some of the methods of `StringTokenizer` class:
  - `int countTokens()`
  - `boolean hasMoreElements()`
  - `boolean hasMoreTokens()`
  - `Object nextElement()`
  - `String nextToken()`
  - `String nextToken(String delim)`
- The following Code Snippet shows the use of `StringTokenizer`:

Code Snippet

```
import java.util.StringTokenizer;
class StringToken {
        public void tokenizeString(String str, String
delim){
              StringTokenizer st = new StringTokenizer(str,
delim);
                while (st.hasMoreTokens()) {
```

Parsing of Text Using StringTokenizer Class [3-3]

```
                    System.out.println(st.nextToken());
                }
            }
}
public class TestProject {
    public static void main(String[] args) {
        StringToken objST = new StringToken();
        objST.tokenizeString("Java,is,a,programming,langu
age", ",");
    }
}
```

- `StringTokenizer` is a legacy class that has been retained for compatibility reasons.
- However, its use is discouraged in new code. It is advisable to use the `split()` method of `String` class or the `java.util.regex` package for tokenization rather than using `StringTokenizer`.

Regular Expression

- 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 that goes beyond the normal syntax of the 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 [1-2]

There are primarily three classes in the `java.util.regex` package that are required for creation of regular expression. They are as follows:

- `Pattern`
- `Matcher`
- `PatternSyntaxExpression`

Pattern:

- A `Pattern` object is a compiled form of a regular expression.
- There are no public constructors available in the `Pattern` class.
- To create a pattern, it is required to first invoke one of its `public static compile()` methods.
- These methods will then return an instance of `Pattern` class.
- The first argument of these methods is a regular expression.

Regular Expression API [2-2]

`Matcher`:

- A `Matcher` object is used to interpret the pattern and perform match operations against an input string.
- Similar to the `Pattern` class, the `Matcher` class also provides no public constructors.
- To obtain a `Matcher` object, it is required to invoke the `matches()` method on a `Pattern` object.

`PatternSyntaxExpression`:

A `PatternSyntaxException` object is an unchecked exception used to indicate a syntax error in a regular expression pattern.

Pattern Class

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

Syntax

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

- The `matches()` method of the `Matcher` class is defined for use when a regular expression is used just once.

Matcher Class [1-6]

- A `Matcher` object is created from a pattern by invoking the `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.
- The syntax of the `Matcher` class is as follows:

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 [2-6]

- `Matcher` class consists of index methods that provide useful index values that can be used to indicate exactly where the match was found in the input string.
- These are as follows:
  - `public int start()`
  - `public int start(int group)`
  - `public int end()`
  - `public int end(int group)`
- The following lists some of the important methods of the `Matcher` class:
  - `Matcher appendReplacement(StringBuffer sb, String replacement)`
  - `StringBuffer appendTail(StringBuffer sb)`
  - `boolean find()`
  - `boolean find(int start)`
  - `String group()`
  - `String group(int group)`
  - `String group(String name)`
  - `int groupCount()`

Matcher Class [3-6]

- The explicit state of a `matcher` includes:
  - The start and end indices of the most recent successful match.
  - The start and end indices of the input subsequence captured by each capturing group in the pattern.
  - Te total count of such subsequences.
- The implicit state of a `matcher` includes:
  - input character sequence.
  - append position, which is initially zero. It is updated by the `appendReplacement()` method.
- 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.
- Instances of the `Matcher` class are not safe for use by multiple concurrent threads.

Matcher Class [4-6]

The following Code Snippet explains the use of `Pattern` and `Matcher` for creating and evaluating regular expressions:

Code Snippet

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class RegexTest{
    public static void main(String[] args){
     String flag;
     while (true) {
         Pattern pattern1 =
         Pattern.compile(System.console().readLine("%nEnter
expression: "));
         Matcher matcher1 =
         pattern1.matcher(System.console().readLine("Enter
string to search: "));
         boolean found = false;
```

Matcher Class [5-6]

```
        while (matcher1.find()) {
          System.console().format("Found the text" + "
\"%s\" starting at " +
"index %d and ending at index %d.%n", matcher1.group(),
matcher1.start(), matcher1.end());
found = true;
}
if(!found){
 System.console().format("No match found.%n");
}
// code to exit the application
 System.console().format("Press x to exit or y to
continue");
 flag=System.console().readLine("%nEnter your choice: ");
if(flag.equals("x"))
```

Matcher Class [6-6]

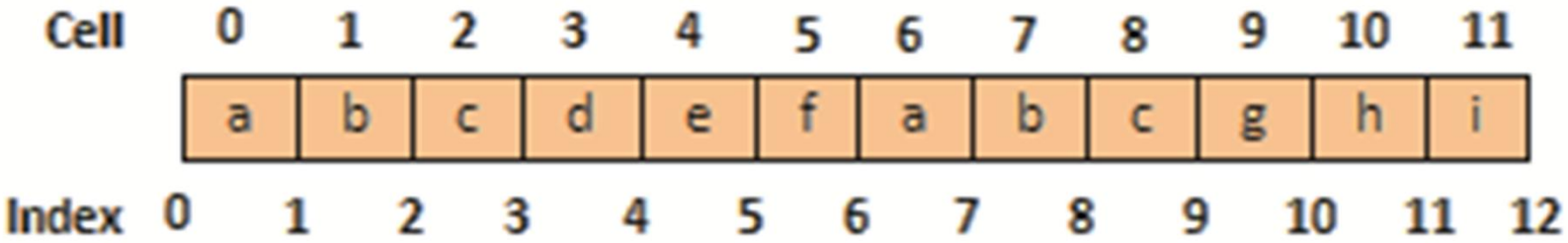
```
 System.exit(0);
else
  continue;
  }
 }
}
```

In the code:

- A `while` loop has been created inside the `RegexTest` class.
- Within the loop, a `Pattern` object is created and initialized with the regular expression specified at runtime using the `System.console().readLine()` method.
- Similarly, the `Matcher` object has been created and initialized with the input string specified at runtime.
- Next, another `while` loop has been created to iterate till the `find()` method returns true.

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.
- This affects the way a pattern is matched.
- 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'.
- This is why the match succeeds in the example.
- The metacharacters supported by the API are: `<([{\^-=$!|]})?*+.>`
- One can force metacharacters to be treated as an ordinary character in one of the following ways:
  - By preceding the metacharacters with a backslash.
  - By enclosing it within \Q (starts the quote) and \E (ends the quote). The \Q and \E can be placed at any location within the expression. However, the \Q must comes first.

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 the characters that will successfully match a single character from a given input string.
- The following table summarizes the 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.
- The 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).

- The 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`.
- The 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.
- The 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.
- The 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 the number of occurrences to match against.
- At first glance it may appear that the quantifiers X?, X??, and X?+ do exactly the same thing, since they all promise to match **X**, once or not at all.
- However, there are subtle differences so far as implementation is concerned between each of these quantifiers.
- The 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
The greedy quantifiers are termed ‘greedy’ because they force the matcher to read the entire input string before to attempting the first match.	The reluctant quantifiers take the opposite approach.	The possessive quantifiers always eat the entire input string, trying once and only once for a match.
If in the first attempt to match the entire input string, fails, then the matcher backs off the 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 the 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.	The last thing they try is to match the entire input string.	
Depending on the quantifier used in the expression, the last thing it will attempt is to try to match against 1 or 0 characters.		

Capturing Groups

- Capturing groups allows the programmer to consider multiple characters as a single unit.
- This is done by placing the characters to be grouped inside a set of parentheses.
- For example, the regular expression **(bat)** creates a single group.
- The group contains the letters '**b**', '**a**', and '**t**'.
- The part of the input string that matches the 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.
- For example, in the expression **((X)(Y(Z)))**, there are four such groups namely, **((X)(Y(Z)))**, **(X)**, **(Y(Z))**, and **(Z)**.
- The `groupCount()` method can be invoked on the matcher object to find out how many groups are present in the expression.
- This method will return an `int` value indicating the number of capturing groups present in the matcher's pattern.
- There is another special group, group 0, which always represents the 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]

The 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 the input string that matches the 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`.
- The 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.
- Also, one can use the additional useful methods of the `Pattern` class.

Creating a Pattern with Flags

- The `Pattern` class provides an alternate `compile()` method that accepts a set of flags.
- These flags affect the way the pattern is matched.
- The flags parameter is a bit mask including any of the following public static fields:
  - `Pattern.CANON _ EQ`
  - `Pattern.CASE _ INSENSITIVE`
  - `Pattern.COMMENTS`
  - `Pattern.DOTALL`
  - `Pattern.LITERAL`
  - `Pattern.MULTILINE`
  - `Pattern.UNICODE _ CASE`
  - `Pattern.UNIX _ LINES`

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.
- The following example uses the 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 [1-3]

- 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);
```

The split(String) Method [2-3]

```
  for(String s : days) {
   System.out.println(s);
  }
 }
}
```

- In the code, the `split()` method is used to extract the words 'Sun Mon Tue Wed Thu Fri Sat' from the string '**Mon:Tue:Wed:Thu:Fri:Sat**'.
- The `split()` method can also be used to get the text that falls on either side of any regular expression.
- The following Code Snippet explains the example to split a string on digits:

Code Snippet

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class SplitTest{
 private static final String REGEX = "\\d";
```

The split(String) Method [3-3]

```
 private static final String DAYS =
"Sun1Mon2Tue3Wed4Thu5Fri6Sat";
 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 [1-2]

- The 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 the stop-the-world garbage collection approach, during garbage collection, application execution is completely suspended.
- The 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 the root of the class hierarchy. Every class has Object as a superclass.
- All objects, including arrays, implement the methods of the 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.

Summary [2-2]

- 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.
- The greedy quantifiers are termed ‘greedy’ because they force the matcher to read the entire input string before to attempting the first match.