Wrapper Classes

The wrapper classes in Java are used to convert primitive types (`int`, `char`, `float`, etc) into corresponding objects.

Each of the 8 primitive types has corresponding wrapper classes.

Primitive types	Wrapper Classes
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Convert Primitive Type to Wrapper Objects

We can also use the `valueOf()` method to convert primitive types into corresponding objects.

Wrapper Objects into Primitive Types
To convert objects into the primitive types, we can use the corresponding value methods (intValue(), doubleValue(), etc) present in each wrapper class.

Advantages of Wrapper Classes
- In Java, sometimes we might need to use objects instead of primitive data types. For example, while working with collections.

```
// error
ArrayList<int> list = new ArrayList<>();

// runs perfectly
ArrayList<Integer> list = new ArrayList<>();
```

Java Command-Line Arguments
The **command-line arguments** in Java allow us to pass arguments during the execution of the program. As the name suggests arguments are passed through the command line.

Java Scanner Class
The Scanner class of the java.util package is used to read input data from different sources like input streams, users, files, etc. Let's take an example.

Java Scanner Methods to Take Input
The `Scanner` class provides various methods that allow us to read inputs of different types.

Scanner Methods

Method	Description
nextInt()	reads an int value from the use
nextFloat()	reads a float value from the user
nextBoolean()	reads a boolean value from the user
nextLine()	reads a line of text from the user
next()	reads a word from the user
nextByte()	reads a byte value from the user
nextDouble()	reads a double value from the user
nextShort()	reads a short value from the user
nextLong()	reads a long value from the user

Java Scanner with BigInteger and BigDecimal

Java scanner can also be used to read the big integer and big decimal numbers.

- **nextBigInteger()** - reads the big integer value from the user
- **nextBigDecimal()** - reads the big decimal value from the user

Type Casting

The process of converting the value of one data type (int, float, double, etc.) to another data type is known as typecasting.
In Java, there are 13 types of type conversion. However, in this tutorial, we will only focus on the major 2 types.
1. Widening Type Casting
2. Narrowing Type Casting

Widening Type Casting

In **Widening Type Casting**, Java automatically converts one data type to another data type.

Narrowing Type Casting

In **Narrowing Type Casting**, we manually convert one data type into another using the parenthesis.

Java autoboxing and unboxing

Java Autoboxing - Primitive Type to Wrapper Object

In **autoboxing**, the Java compiler automatically converts primitive types into their corresponding wrapper class objects.

For example,
```
int a = 56;
// autoboxing
Integer aObj = a;
```

Java Unboxing - Wrapper Objects to Primitive Types

In **unboxing**, the Java compiler automatically converts wrapper class objects into their corresponding primitive types.

For example,

```
// autoboxing
Integer aObj = 56;

// unboxing
int a = aObj;
```

Java Lambda Expressions
The lambda expression was introduced first time in Java 8. Its main objective to increase the expressive power of the language.

What is Functional Interface?
If a Java interface contains one and only one abstract method then it is termed as functional interface. This only one method specifies the intended purpose of the interface.
For example, the Runnable interface from package java.lang; is a functional interface because it constitutes only one method i.e. run().

Introduction to lambda expressions
Lambda expression is, essentially, an anonymous or unnamed method. The lambda expression does not execute on its own. Instead, it is used to implement a method defined by a functional interface.
How to define lambda expression in Java?
Here is how we can define lambda expression in Java.

```
(parameter list) -> lambda body
```

The new operator (->) used is known as an arrow operator or a lambda operator. The syntax might not be clear at the moment. Let's explore some examples,
Suppose, we have a method like this:

```
double getPiValue() {
    return 3.1415;
}
```

We can write this method using lambda expression as:

```
() -> 3.1415
```

Types of Lambda Body

In Java, the lambda body is of two types.

1. A body with a single expression

```
() -> System.out.println("Lambdas are great”);
```

This type of lambda body is known as the expression body.

2. A body that consists of a block of code.

```
() -> {
    double pi = 3.1415;
    return pi;
};
```

Java Packages:

it is a way of categorizing the classes and interfaces

A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are:

Here:

→ **java** is a top level package

→ **util** is a sub package

→ and **Scanner** is a class which is present in the sub package **util**.

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

The import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

The Directory Structure of Packages:

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Built-in Package

Built-in packages are existing java packages that come along with the JDK. For example, `java.lang`, `java.util`, `java.io`, etc.

```
import java.util.Date; // imports only Date class
import java.io.*;      // imports everything inside java.io package
```

User-defined Package

Java also allows you to create packages as per your need. These packages are called user-defined packages.

```
package packageName;
```

```
└─── com
  └─── test
    └─── Test.java
```

```
import package.name.ClassName;   // To import a certain class only
import package.name.*   // To import the whole package
```

Regular Expressions:

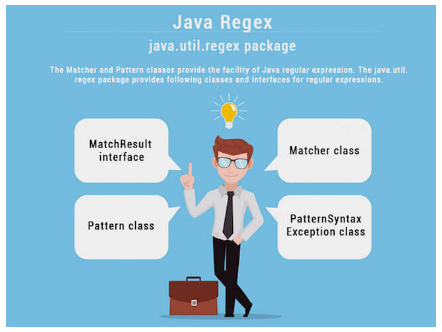
A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings,

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


Java Regex
java.util.regex package
The Matcher and Pattern classes provide the facility of Java regular expression. The java.util. regex package provides following classes and interfaces for regular expressions.
MatchResult interface
Matcher class
Pattern class
PatternSyntax Exception class

Java Matcher group() Method

String group()
Returns the matched sequence captured by the previous match as the string.

String group(int group)
Returns the matched sequence captured by the given group during the previous match operation as the string.

String group(String name)
Returns the matched sequence captured by the given named group during the previous match operation or null if the match fails.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexGroupExample1 {
   public static void main(String[] args) {
      // TODO Auto-generated method stub

      Pattern p=Pattern.compile("a(bb)");
      Matcher m=p.matcher("aabbabbabbaaa");
      while(m.find())

   System.out.println("Start :"+m.start()
+", End : "+m.end()+", Group "+m.group());
      }
 }
```

Output:

```
Start :1, End : 4, Group abb
Start :4, End : 7, Group abb
Start :7, End : 10,Group abb
```

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

public class Main {
  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("javaTraining", 
Pattern.CASE_INSENSITIVE);
    Matcher matcher = pattern.matcher(“Welcome JavaTraining");
    boolean matchFound = matcher.find();
    if(matchFound) {
      System.out.println("Match found");
    } else {
      System.out.println("Match not found");
    }
  }
}
```

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

import java.util.regex.Pattern;

public class RegularExample {

    public static void main(String args[]) {

    String line = "This order was placed for QT3000! OK?";

    Pattern pattern = Pattern.compile("(.*?)(\\d+)(.*)");

    Matcher matcher = pattern.matcher(line);

    while (matcher.find()) {

        System.out.println("group 1: " + matcher.group(1));

        System.out.println("group 2: " + matcher.group(2));

        System.out.println("group 3: " + matcher.group(3));

}

    }

}
```

```
Found value:This order was places for QT
Found value:3000
Found value:)OK
```

Regular Expression Syntax:

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

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

\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\n	Back-reference to capture group number "n"
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches 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

```
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());
}
}
}
```

The start method returns the start index of the subsequence captured by the given group during the previous match operation, and end returns the index of the last character matched, plus one

Methods:

Modifiers: The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.

Return Type: A method may return a value. The returnValueType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the returnValueType is the keyword **void**.

Method Name: This is the actual name of the method. The method name and the parameter list together constitute the method signature.

Parameters: A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

Method Body: The method body contains a collection of statements that define what the method does.

```
/** Return the max between two numbers */ public static int
max(int num1,int num2){ int result;
if(num1 > num2) result = num1; else
  result = num2;

return result;
}
```

```
public class TestVoidMethod{

public static void main(String[] args){ printGrade(78.5);
}

public static void printGrade(double score){ if(score >=90.0)
{
System.out.println('A');
}
elseif(score >=80.0){ System.out.println('B');
}
elseif(score >=70.0){ System.out.println('C');
}
elseif(score >=60.0){ System.out.println('D');
}
else{ System.out.println('F');
}

int i =5;
int j =2;
int k = max(i, j);
System.out.println("The maximum between "+ i + " and "+ j +"
is "+ k);
}

/** Return the max between two numbers */
public static int max(int num1,int num2){
int result;
if(num1 > num2) result = num1; else
  result = num2;

return result;
}
}
```

Static Class, Block, Methods and Variables

Static variable in Java is variable which belongs to the class and initialized only once at the start of the execution. It is a variable which belongs to the class and not to object(instance). Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.

- A single copy to be shared by all instances of the class
- A static variable can be accessed directly by the class name and doesn’t need any object

What is Static Method in Java?
Static method in Java is a method which belongs to the class and not to the object. A static method can access only static data. It is a method which belongs to the class and not to the object(instance). A static method can access only static data. It cannot access non-static data (instance variables).

- A static method can call only other static methods and can not call a non-static method from it.
- A static method can be accessed directly by the class name and doesn’t need any object
- A static method cannot refer to "this" or "super" keywords in anyway

-

What is Static Block in Java?
The static block is a block of statement inside a Java class that will be executed when a class is first loaded into the JVM. A static block helps to initialize the static data members, just like constructors help to initialize instance members.

Java Static Variables

A static variable is common to all the instances (or objects) of the class because it is a class level variable.

- Static variables are also known as Class Variables.
- Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.