How to search the last position of a substring

This example shows how to determine the last position of a substring inside a string with the help of strOrig.lastIndexOf(Stringname) method.

```
public class SearchlastString {
   public static void main(String[] args) {
      String strOrig = "Hello world ,Hello Reader";
      int lastIndex = strOrig.lastIndexOf("Hello");

      if(lastIndex == - 1){
         System.out.println("Hello not found");
      } else {
         System.out.println("Last occurrence of Hello is at index "+
lastIndex);
      }
   }
}
```

```
Output:
Last occurrence of Hello is at index 13
```

```
public class HelloWorld{
   public static void main(String []args) {
      String t1 = "HelloWorld";
      int index = t1.lastIndexOf("1");
      System.out.println(index);
   }
}
```

```
public class StringReverseExample{
    public static void main(String[] args) {
        String string = "abcdef";
        String reverse = new StringBuffer(string).reverse().toString();
        System.out.println("\nString before reverse: "+string);
        System.out.println("String after reverse: "+reverse);
    }
}
```

Output:

```
String before reverse:abcdef
String after reverse:fedcba
```

```
import java.io.*;
import java.util.*;

public class HelloWorld {
    public static void main(String[] args) {
        String input = "abcdef";
        char[] try1 = input.toCharArray();
        for (int i = try1.length-1;i >= 0; i--)
          System.out.print(try1[i]);
    }
}
```

```java
//program to search a string

public class SearchStringEmp{
   public static void main(String[] args) {
      String strOrig = "Hello readers";
      int intIndex = strOrig.indexOf("Hello");

      if(intIndex == - 1) {
         System.out.println("Hello not found");
      } else {
         System.out.println("Found Hello at index " +
intIndex);
      }
   }
}

public class HelloWorld {
   public static void main(String[] args) {
      String text = "The cat is on the table";
      System.out.print(text.contains("the"));
   }
}
```

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

6) StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

Important methods of StringBuilder class

1) StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this string.

2) StringBuilder insert() method

The StringBuilder insert() method inserts the given string with this string at the given position.

3) StringBuilder replace() method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

4) StringBuilder delete() method

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

5) StringBuilder reverse() method

The reverse() method of StringBuilder class reverses the current string.

6) StringBuilder capacity() method

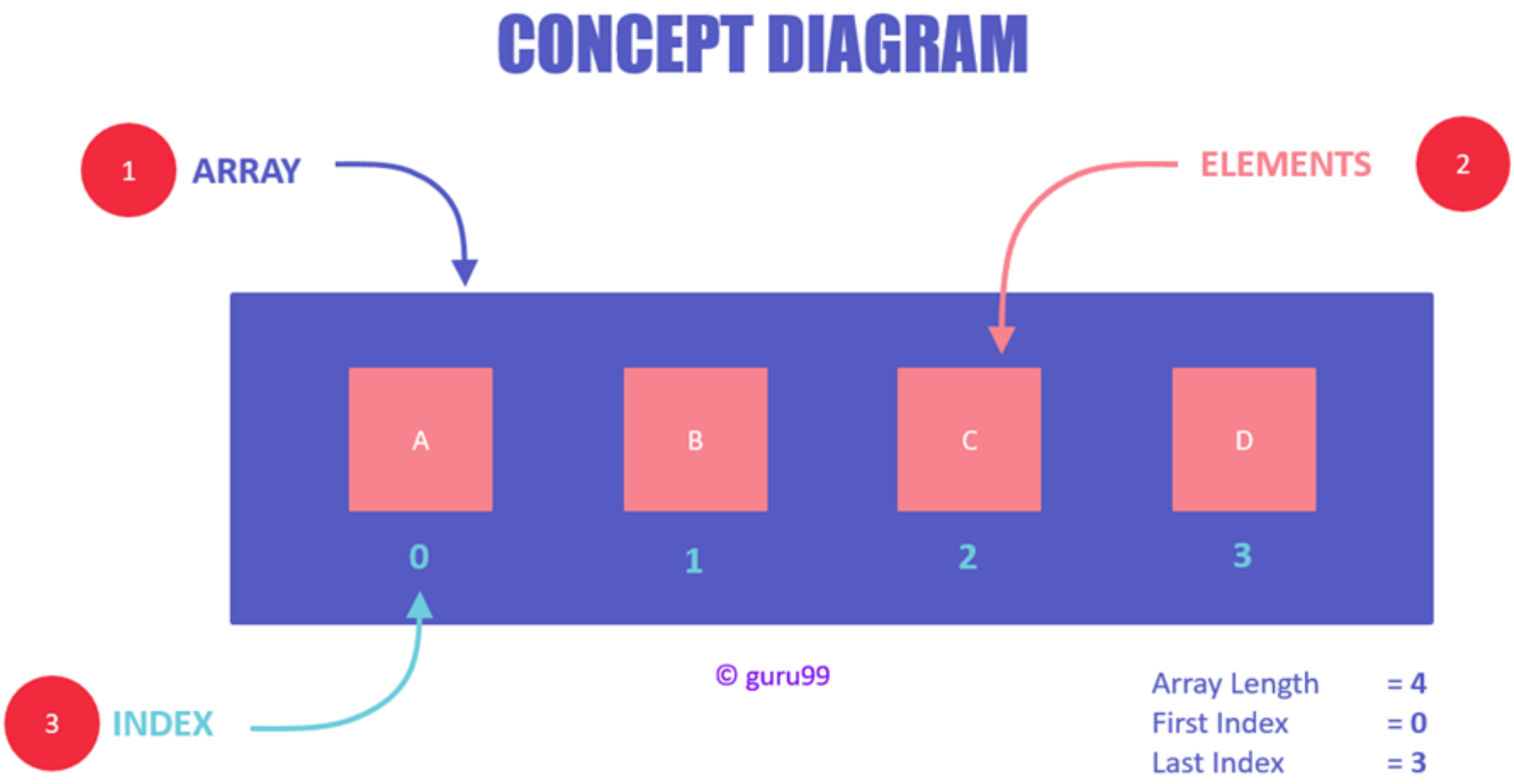
The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

7) StringBuilder ensureCapacity() method

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34

Arrays:

An Array is a collection of variables of the same type.



Creating an array:

```
public class TestArray{

public static void main(String[] args){
double[] myList ={1.9,2.9,3.4,3.5};
// Print all the array elements
for(int i =0; i < myList.length; i++)
{
System.out.println(myList[i]);
} } }
```

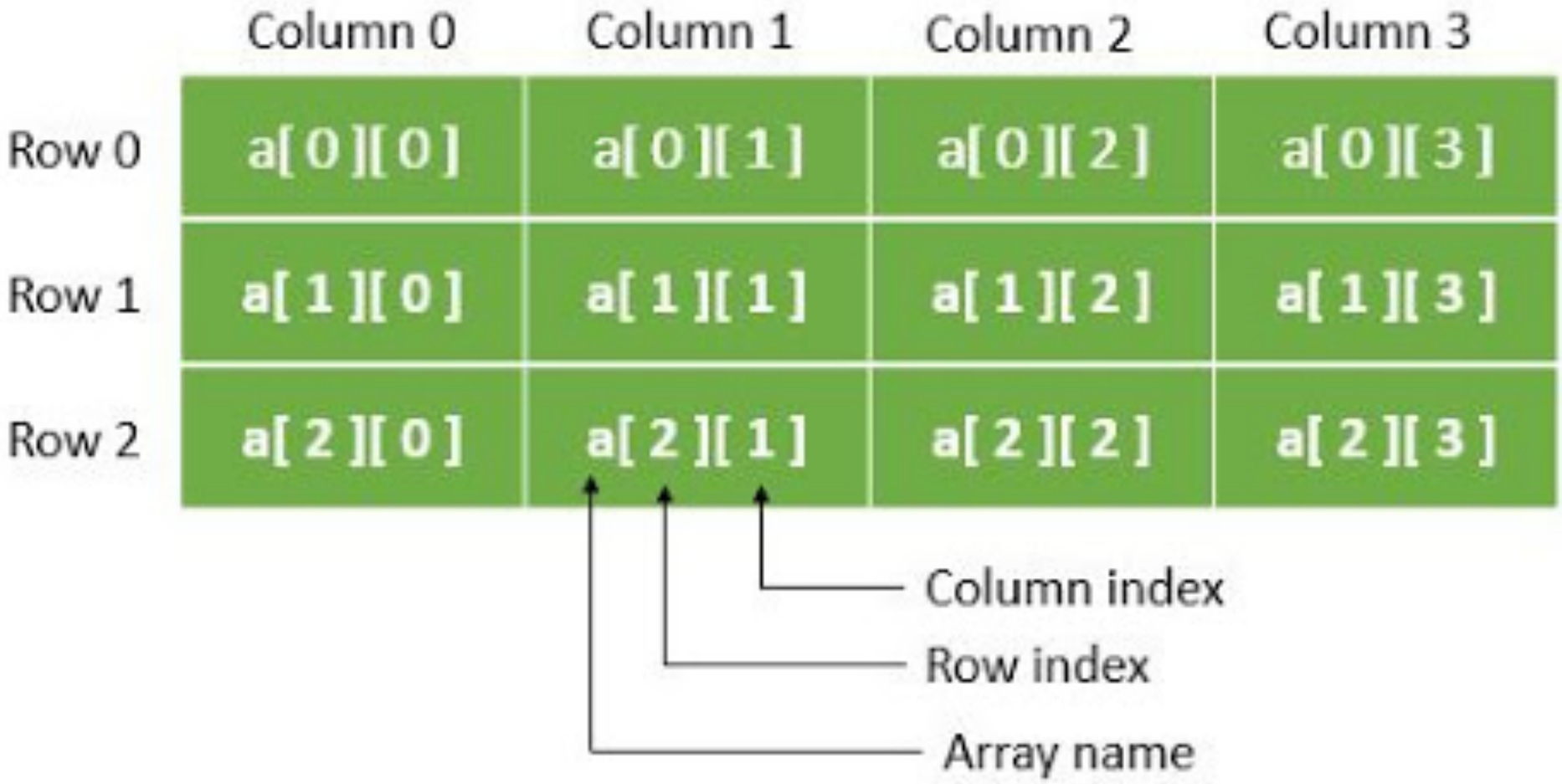
The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
      for (double element: myList) {
         System.out.println(element);
      } } }
```


Column 0
Column 1
Column 2
Column 3
Row 0
Row 1
Row 2
a[0][0]
a[0][1]
a[0][2]
a[0][3]
a[1][0]
a[1][1]
a[1][2]
a[1][3]
a[2][0]
a[2][1]
a[2][2]
a[2][3]
Column index
Row index
Array name

Passing Arrays to Methods

```
public static void printArray(int[] array) {
   for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + " ");
   }
}
```

Returning an Array from a Method

```
public static int[] reverse(int[] list) {
   int[] result = new int[list.length];

   for (int i = 0, j = result.length - 1; i < list.length; i++, j--)
{
      result[j] = list[i];
   }
   return result;
}
```

An array in Java is a type of object that can contain a number of variables. ... You can have arrays of any of the Java primitives or reference variables. The important point to remember is that when created, primitive arrays will have default values assigned, but object references will all be null.

Declare an array

```
String[] aArray = new String[5];
String[] bArray = {"a","b","c", "d", "e"};
String[] cArray = new String[]{“a","b","c","d","e"};
```

1. Print an array in Java

```
int[] intArray = { 1, 2, 3, 4, 5 };
String intArrayString = Arrays.toString(intArray);

// print directly will print reference value
System.out.println(intArray);
// [I@7150bd4d

System.out.println(intArrayString);
// [1, 2, 3, 4, 5]
```

Create an ArrayList from an array

```
String[] stringArray = { "a", "b", "c", "d", "e" };
ArrayList<String> arrayList = new
ArrayList<String>(Arrays.asList(stringArray));
System.out.println(arrayList);
// [a, b, c, d, e]
```

Check if an array contains a certain value

```
String[] stringArray = { "a", "b", "c", "d", "e" };
boolean b =
Arrays.asList(stringArray).contains("a");
System.out.println(b);
// true
```

Concatenate two arrays

```
int[] intArray = { 1, 2, 3, 4, 5 };
int[] intArray2 = { 6, 7, 8, 9, 10 };
// Apache Commons Lang library
int[] combinedIntArray = ArrayUtils.addAll(intArray,
intArray2);
```

Declare an array inline

```
method(new String[]{"a", "b", "c", "d", “e"});
```

Joins the elements of the provided array into a single String

```
// containing the provided list of elements
// Apache common lang
String j = StringUtils.join(new String[] { "a", "b", "c" }, ",
");
System.out.println(j);
// a, b, c
```

Convert an ArrayList to an array

```
String[] stringArray = { "a", "b", "c", "d", "e" };
ArrayList<String> arrayList = new
ArrayList<String>(Arrays.asList(stringArray));
String[] stringArr = new String[arrayList.size()];
arrayList.toArray(stringArr);
for (String s : stringArr)
   System.out.println(s);
```

Convert an array to a set

```
Set<String> set = new HashSet<String>(Arrays.asList(stringArray));
System.out.println(set);
//[d, e, b, c, a]
```

Reverse an array

```
int[] intArray = { 1, 2, 3, 4, 5 };
ArrayUtils.reverse(intArray);
System.out.println(Arrays.toString(intArray));
//[5, 4, 3, 2, 1]
```

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:

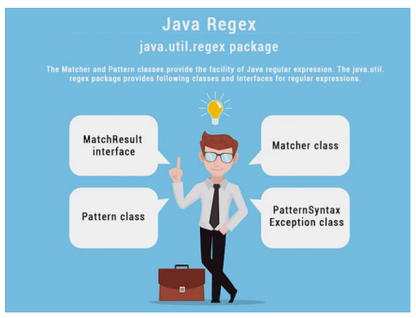
A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings,

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 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
```

```
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.

Object & Classes

Java is an Object-Oriented Language. It supports

- **Polymorphism**
- **Inheritance**
- **Encapsulation**
- **Abstraction**
- **Classes**
- **Objects**

- **Object - Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.**

```
public class Puppy{
public Puppy(){
}

public Puppy(String name){
// This constructor has one parameter, name.
}
}
```

- **Class - A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.**

```
public class Dog{
String breed;
int age;
String color;

void barking(){
}

void hungry(){
}

void sleeping(){
}
}
```

A class can contain any of the following variable types.

- **Local variables: Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.**
- **Instance variables: Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.**
- **Class variables: Class variables are variables declared within a class, outside any method, with the static keyword.**

Constructors:

- **Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.**
- **Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor**

Creating an object:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' keyword is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.
-

```
public class Puppy{

public Puppy(String name){
// This constructor has one parameter,
name. System.out.println("Passed Name
is :"+ name );
}
 public static void main(String[]args){
// Following statement would create an
object myPuppy
Puppy myPuppy =new Puppy("tommy");
}
}
```

Accessing Instance variables and Methods:

Instance variables and methods are accessed via created objects.

```
public class Puppy{

int puppyAge;

public Puppy(String name){
// This constructor has one parameter, name.
System.out.println("Passed Name is :"+ name );
}
public void setAge(int age ){
puppyAge = age;
}

public int getAge(){
System.out.println("Puppy's age is :"+ puppyAge );
return puppyAge;
}
 public static void main(String[]args){
/* Object creation */
Puppy myPuppy =newPuppy("tommy");

/* Call class method to set puppy's age */
myPuppy.setAge(2);

/* Call another class method to get puppy's age */
  myPuppy.getAge();

/* You can access instance variable as follows as well */
System.out.println("Variable Value :"+ myPuppy.puppyAge );
}
}
```

Points to Remember:

1. Every class has a constructor whether it's a normal class or a abstract class.
2. Constructors are not methods and they don't have any return type.
3. Constructor name should match with class name .
4. Constructor can use any access specifier, they can be declared as private also. Private constructors are possible in java but there scope is within the class only.
5. **Like constructors method can also have name same as class name, but still they have return type, though which we can identify them that they are methods not constructors.**
6. If you don't implement any constructor within the class, compiler will do it for.
7. **this() and super() should be the first statement in the constructor code.** If you don't mention them, compiler does it for you accordingly.
8. Constructor overloading is possible but overriding is not possible. Which means we can have overloaded constructor in our class but we can't override a constructor.
9. Constructors can not be inherited.
10. If Super class doesn't have a no-arg(default) constructor then compiler would not insert a default constructor in child class as it does in normal scenario.
11. Interfaces **do not have constructors**.
12. Abstract class can have constructor and it gets invoked when a class, which implements interface, is instantiated. (i.e. object creation of concrete class).
13. A constructor can also invoke another constructor of the same class – By using this(). If you want to invoke a parameterized constructor then do it like this: **this(parameter list)**.

Difference between Constructor and Method

1. The purpose of constructor is to initialize the object of a class while the purpose of a method is to perform a task by executing java code.
2. Constructors cannot be abstract, final, static and synchronised while methods can be.
3. Constructors do not have return types while methods do.

Singleton Classes:

The Singleton's purpose is to control object creation, limiting the number of objects to one only. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources such as database connections or sockets. For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time

```
// File Name: Singleton.java
public class Singleton{

private static Singleton singleton =new Singleton();

/* A private Constructor prevents any other
  * class from instantiating.
  */
private Singleton(){}

/* Static 'instance' method */
public static Singleton getInstance(){
return singleton;
}
/* Other methods protected by singleton-ness */
protected static void demoMethod(){
System.out.println("demoMethod for singleton");
}
}

// File Name: SingletonDemo.java
public class SingletonDemo{
public static void main(String[] args){
Singleton tmp =Singleton.getInstance();

   tmp.demoMethod();
}
}
```

Inheritance

The most commonly used keyword would be extends and implements. These words would determine whether one object IS-A type of another.

In Object Oriented terms the following are true:

- **Animal is the superclass of Mammal class.**
- **Animal is the superclass of Reptile class.**
- **Mammal IS-A Animal**
- **Reptile IS-A Animal**

```
public class Dog extends Mammal{
public static void main(String args[]){
Animal a =new Animal();
Mammal m =new Mammal();
Dog d =new Dog();

System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal);
}
}
```

IS-A (Inheritance) :

IS-A relationship denotes “one object is type of another”. IS-A relation denotes Inheritance methodology.

Has-A (Association) :

In Object orientation design, We can say “class one is in Has-A relationship with class B if class A holds reference of Class B”.

```
class Laptop {
   // Code for Laptop class goes here.
}
interface Formatable {
   // Members of Interface.
}
class Dell extends Laptop implements Formatable {
   // More code related to Dell goes here.
   // Dell class will inherit all accessible members of Laptop class.
   // Dell IS-A Laptop.
   // And Dells class also implements all method of Formatable interface, since
   // Dell is not an abstract class.
   // so Dell IS-A Formatable.
}
public class TestRelationship {
   public static void main(String[] args) {
      Dell oDell = new Dell();
      if (oDell instanceof Laptop) {
         System.out.println("Dell IS-A Laptop.");
      }
      if (oDell instanceof Formatable) {
         System.out.println("Dell IS-A Formatable.");
```

```
class HardDisk {
    public void writeData(String data) {
        System.out.println("Data is being written : " + data);
    }
}

class UseDell {
    // segate is referece of HardDisk class in UseDell class.
    // So, UseDell Has-A HardDisk
    HardDisk seagate = new HardDisk();
    public void save (String data) {
        seagate.writeData(data);
    }
}
```

Overridding:

The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement.

```
class Animal{

public void move(){
System.out.println("Animals can move");
}
}
class Dog extendsAnimal{
public void move(){
System.out.println("Dogs can walk and run");
}
public void bark(){
System.out.println("Dogs can bark");
}
}

public class TestDog{

public static void main(String args[]){
Animal a =new Animal();// Animal reference and object
Animal b =new Dog();// Animal reference but Dog object

  a.move();// runs the method in Animal class
  b.move();//Runs the method in Dog class
  b.bark();
}
}
```

super Keyword:

When invoking a superclass version of an overridden method the super keyword is used.

```
class Animal{

public void move(){
System.out.println("Animals can move");
}
}
class Dog extends Animal{
public void move(){
super.move();// invokes the super class method
System.out.println("Dogs can walk and run");
}
}

public class TestDog{

public static void main(String args[]){

Animal b =new Dog();// Animal reference but Dog
 object
b.move();//Runs the method in Dog class
}
}
```

this is a **keyword in Java.** It can be used inside the method or constructor of a class. It(this) works as a reference to the current object, whose method or constructor is being invoked.

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){
System.out.println(rollno+" "+name+" “+fee);
}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:
0 null 0.0
0 null 0.0

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

```
//this keyword to invoke a method
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

```
//this keyword to invoke current constructor
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```