

**Table of Contents**

[1. Purpose 3](#_Toc51830557)

[2. Some Useful Features 4](#_Toc51830558)

[2.1. String in switch statement 4](#_Toc51830559)

[2.2. Arrays.asList() method 5](#_Toc51830560)

[2.3. Arrays.copyOf() method 6](#_Toc51830561)

[2.4. forEach Method 7](#_Toc51830562)

[2.5. Streams 9](#_Toc51830563)

[2.6. Try-with-Resources 13](#_Toc51830564)

[2.7. Hashmap Methods 14](#_Toc51830565)

[2.8. Default and Static methods 17](#_Toc51830566)

[2.9. Optional<t> 18](#_Toc51830567)

# Purpose

The purpose of this document is to have an overview over some Java language new features in order to improve the application performance and to give developers significant flexibility when dealing with some of the key productivity improvements like Lambda expressions, the Streams API, Optional, …

These new features allow you application to run faster, to have fewer lines of code (through embracing a more functional style for the new APIs, focusing on what you want to achieve and not how to do it) and to have simplified code (through using [Lambda expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)).

Java, with syntax changes, new methods and types, has many benefits for your business, your existing application, and for developers looking to improve their productivity.

# Some Useful Features

## String in switch statement

One of the new features added in Java is the capability to switch on a String.

The switch statement when used with a String uses the equals() method to compare the given expression to each value in the case statement and is therefore case-sensitive and will throw a NullPointerException if the expression is null.

Switch case is more readable than if/else statement when having multiple conditions (more than five items), it's implemented using a lookup table or a hash list. This means that all items get the same access time, compared to a list of ifs where the last item takes much more time to reach as it has to evaluate every previous condition first.

|  |  |
| --- | --- |
| **Replace:**  if(str.equals("case1"))  {  *[...]*  }  else if(str.equals("case2"))  {  *[...]*  }  else if(str.equals("case3"))  {  *[...]*  }  else if(str.equals("case4"))  {  *[...]*  }  else  {  *[...]*  } | **By:**  switch (str)  {  case "case1":  *[...]*  break;  case "case2":  *[...]*  break;  case "case3":  *[...]*  break;  case "case4":  *[...]*  break;  default:  *[...]*  } |

## Arrays.asList() method

The Array.asList() method is used to return a fixed-size list backed by the specified array.

|  |  |
| --- | --- |
| **Replace:**  ArrayListlst = new ArrayList();  lst.add(2);  lst.add(3);  lst.add(4); | **By:**  List<Integer> lst1 = Arrays.asList(2,3,4); |

**Example:**

String[] rowId = request.getParameterValues("rowId");

**try**

{

ArrayList list = **new** ArrayList();

**for** (**int** i = 0; i < rowId.length; i++)

{

theVO = **new** TargetingStrategyVO();

theVO.setStrategyId(NumberUtil.*parseInteger*(rowId[i]));

list.add(theVO);

}

List list = **new** ArrayList();

list = Arrays.asList(rowId);

applySecurity(request);

setGridsPref(request);

UserContextVO userContext = UserContextVO.*getUserContext*(request);

targetingStrategyBO.deleteTargetingStrategy(list, userContext);

theForm.setNotification("Done);

}

## Arrays.copyOf() method

The Arrays.copyOf() method is used to copy a specified array, truncating or padding with false (if necessary) so the copy has the specified length.

|  |  |
| --- | --- |
| **Replace:**  int[] oldArray = new int[] {1, 2 ,3};  for(int i=0; i<oldArray.length; i++)  newArray[i] = oldArray[i]; | **By:**  int[] oldArray = new int[] {1, 2 ,3};  int[] newArray = Arrays.copyOf(oldArray, oldArray.length);  int[] newArray = Arrays.copyOf(oldArray, oldArray.length+2);  the result: 1, 2, 3, 0, 0 |

**Example:**

prevGroup = new Object[groupKeys.length];

for (int m = 0; m < curGroup.length; m++)

{

prevGroup[m] = curGroup[m];

}

prevGroup = Arrays.copyOf(curGroup, groupKeys.length);

## forEach Method

The forEach() method iterates over a collection such as (list, set or map) and perform a certain action on each element of it.

It has been introduced in Java with the introduction of Lambda expression.

**Collection**

|  |  |
| --- | --- |
| **Replace:**  while (it.hasNext())  {  vo = new NmsCommonVO();  vo = (NmsCommonVO) it.next();  request.setAttribute(vo.getColorName(),  vo.getColorRgbCode());  } | **By:**  MatchBgColor.forEach(vo->request.setAttribute(  vo.getColorName(),  vo.getColorRgbCode()));  **Or by:** (Multiple line of code)  MatchBgColor.forEach(vo->  {  request.setAttribute(vo.getColorName(),  vo.getColorRgbCode());    ……  }); |

**AtomicInteger**

It is an integer value that may be updated atomically.

An AtomicInteger is used in applications such as atomically incremented counters, and cannot be used as a replacement for an [Integer](https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html).

**Example:**

AtomicInteger d = **new** AtomicInteger(1);

lstBodyData.forEach(theVO -> generateSheet(d.getAndIncrement(), heads, ctx, beanHandler, sheet, theVO));

**Hashmap**

When using forEach() method over a Hashmap, two variables are defined for the **key** and the **value** (*[Hashmap]*.forEach((k, v)-> …… )

|  |
| --- |
| **Replace:**  map = kycCustLifeCycleMgmtBO.getSessionAttributes(criteria);    String sessionStatusCode = (String) map.get("sessionStatusCode");  String sessionStatusName = (String) map.get("sessionStatusName");  request.getSession().setAttribute("sessionStatusCode", sessionStatusCode);  request.getSession().setAttribute("sessionStatusName", sessionStatusName); |
| **By:** (if all the keys are to be set as attribute)  map.forEach((k, v)-> request.getSession().setAttribute(k, v)); |

## Streams

A stream is used to process collections of objects. It is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels, It doesn’t change the original data structure; it only provides the result as per the pipelined methods.

**Some Operations On Streams**

There are two categories of operations: Intermediate and Terminal.

Each intermediate operation is executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

**Intermediate Operations**

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List result = names.stream().filter(s->s.startsWith("V")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List result = names.stream().sorted().collect(Collectors.toList());

**Sorting by Property (using Comparator)**

List<User> sortedList = users.stream()

.sorted(Comparator.*comparing*(User::getAge))

.collect(Collectors.*toList*());

**Using Lambda**

List<User> sortedList = users.stream()

.sorted((u1,u2)->u1.getName().compareTo(u2.getName()))

.collect(Collectors.*toList*());

1. **distinct:** The *distinct* method creates a new stream of unique elements of the previous stream.

List result = names.stream().distinct().collect(Collectors.toList());

**Terminal Operations**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.

The reduce method takes a BinaryOperator as a parameter.

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i); // ans=0 initially and i is added to it

1. **count:** The count method returns stream's size.

long count = number.stream().distinct().count();

**Example:**

List<Integer> number = Arrays.asList(2,3,4,5);

List<Integer> square = number.stream().map(x -> x\*x).collect(Collectors.toList());

System.out.println(square); // [4, 9, 16, 25]

List<String> names = Arrays.asList("Reflection","Collection","Stream");

List<String> result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

System.out.println(result); // [Stream]

List<String> show = names.stream().sorted().collect(Collectors.toList());

System.out.println(show); // [Collection, Reflection, Stream]

List<Integer> numbers = Arrays.asList(2,3,4,5,2);

Set<Integer> squareSet = numbers.stream().map(x->x\*x).collect(Collectors.toSet());

System.out.println(squareSet); // [16, 4, 9, 25]

number.stream().map(x->x\*x).forEach(y->System.out.println(y)); // 4 // 9 // 16 // 25

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

System.out.println(even); // 6

//--------------------------------------------------------------------

Integer[] arr=new Integer[]{100,24,13,44,114,200,40,112};

List<Integer> list = Arrays.asList(arr);

    OptionalDouble average = list.stream().mapToInt(n->n\*n).filter(n->n>10000).average();

    if(average.isPresent())

        System.out.println(average.getAsDouble());

//--------------------------------------------------------------------

Employee e1=**new** Employee("John",21);

Employee e2=**new** Employee("Martin",19);

Employee e3=**new** Employee("Mary",31);

Employee e4=**new** Employee("Mary",18);

Employee e5=**new** Employee("John",26);

List<Employee> employeeList=**new** ArrayList<>();

employeeList.add(e1);

employeeList.add(e2);

employeeList.add(e3);

employeeList.add(e4);

employeeList.add(e5);

Map<String, List<Employee>> map = employeeList.stream().collect(Collectors.*groupingBy*(Employee::getName));

map.forEach((name,employeeListTemp)->System.out.println("Name: "+name+" ==>"+employeeListTemp));

// Name: John ==>[Employee Name: John age: 21, Employee Name: John age: 26]  
 // Name: Martin ==>[Employee Name: Martin age: 19]  
 // Name: Mary ==>[Employee Name: Mary age: 31, Employee Name: Mary age: 18]

//--------------------------------------------------------------------

**//Sorting by property**

List<User> users = Arrays.*asList*(

new User("C", 30), new User("D", 40),

new User("A", 10), new User("B", 20),

new User("E", 50));

// sort by age (Ascending)

List<User> sortedList = users.stream().sorted(Comparator.comparing(User::getAge))

.collect(Collectors.toList());

sortedList.forEach(System.out::println);

// sort by age (Descending)

List<User> sortedList = users.stream().sorted(Comparator.comparing(User::getAge)**.reversed()**)

.collect(Collectors.toList());

sortedList.forEach(System.out::println);

**//Sorting by property using Lambda (Ascending)**

List<User> sortedList = users.stream().sorted((u1,u2)->**u1**.getAge().compareTo(**u2**.getAge()))

.collect(Collectors.*toList*());

sortedList.forEach(System.out::println);

//Sorting by property using Lambda (Descending)

List<User> sortedList = users.stream().sorted((u1,u2)->**u2**.getAge().compareTo(**u1**.getAge()))

.collect(Collectors.*toList*());

sortedList.forEach(System.out::println);

**Stream helps to substitute *for*, *for-each*, and *while* loops**

|  |  |
| --- | --- |
| **This code**  for (String string : list)  {  if (string.contains("a"))  {  return true;  }  } | **can be changed just with one line of code**  boolean isExist = list.stream().anyMatch(element -> element.contains("a")); |

**Multi-Threading with Streams**

With both the new forEach method and the Stream API, you can create a stream of elements in a collection and then pipeline the stream to a forEach method for iteration.

**Example:**

public static void iterateThroughListStream(List<String> list) {

list.stream().forEach(System.out::println);

}

List<String> countryList = Arrays.asList("Argentina", "Brasil", "China", "United States");

iterateThroughListStream(countryList); //Argentina

//Brasil

//China

//United States

For parallel streams, the only difference is that you need to call the parallelStream() method instead of stream() on the list. Then iterate through the stream of elements using forEach, like this.

**Example:**

public static void iterateThroughListParallelStream(List<String> list) {

list.parallelStream().forEach(System.out::println);

}

List<String> countryList = Arrays.asList("Argentina", "Brasil", "China", "United States");

iterateThroughListParallelStream(countryList); //China

//United States

//Brasil

//Argentina

As you can notice, the order in which the list elements are processed is not the order in which the elements are stored in the list. However, when dealing with larger sets of data, parallel streams bring in considerable performance gain to your program.

## Try-with-Resources

Try-with-resources allows to declare resources to be used in a *try* block with the assurance that the resources will be closed after the execution of the block.

|  |  |
| --- | --- |
| **Replace:**  PrintWriter writer = newPrintWriter(newFile("test.txt"));  try  {  writer.println(str);  }  finally  {  if (writer! = null) writer.close();  } | **By:**  try(PrintWriter writer =  newPrintWriter(newFile("test.txt")))  {      writer.println(str);  } |
| Scanner scanner = null;  PrintWriter writer = null;  try  {  writer = newPrintWriter(newFile("testWrite.txt"));  scanner = newScanner(newFile("testRead.txt"));     while(scanner.hasNext())  {          writer.println(scanner.nextLine());     }  }  catch(FileNotFoundException e)  {      e.printStackTrace();  }  finally  {  if (writer! = null) writer.close();  if (scanner != null) scanner.close();  } | try(Scanner scanner =  newScanner(newFile("testRead.txt"));      PrintWriter writer =  newPrintWriter(newFile("testWrite.txt")))  {      while(scanner.hasNext())  {       writer.print(scanner.nextLine());      }  }  catch(FileNotFoundException e)  {      e.printStackTrace();  } |

## Hashmap Methods

**getOrDefault() Method**

This method overrides the get() method, it is used to return the value of the key where exists or the default value (instead of null) when the key doesn’t exist.

|  |  |
| --- | --- |
| **Replace:**  Map<String, Integer> products = new HashMap<>();  products.put("1234", 1000);  products.put("4567", 2000);  Integer price = products.get("7890");  if(price == null)  {  price = 500;  } | **By:**  Map<String, Integer> products = new HashMap<>();  products.put("1234", 1000);  products.put("4567", 2000);  Integer price = products.getOrDefault("7890", "500"); |

**ComputeIfAbsent(Key, Function) Method**

This method computes the value for a given key using the given mapping function, if key is not already associated with a value (or is mapped to null) then add the computed value in Hashmap else do nothing.

**Example 1:**

HashMap<String, Integer> map = new HashMap<>();

      map.put("key1", 10000);

      map.put("key2", 55000);

      System.out.println("HashMap: " + map.toString()); // HashMap: {key1=10000, key2=55000}

      map.computeIfAbsent("key3", k -> 2000 + 33000);

      System.out.println("New HashMap: " + map); // New HashMap: {key1=10000, key2=55000, key3=35000}

map.computeIfAbsent("key2", k -> 60000 + 1000);

System.out.println("New HashMap: " + map); // New HashMap: {key1=10000, key2=55000, key3=35000}

**Example 2:**

public OperatingSystem createOS()

{

OperatingSystem os = new OperatingSystem()

os.setId(50);

os.setName(“Valoores”);

return os;

}

OperatingSystem os1 = new OperatingSystem()

os1.setId(10);

os1.setName(“Microsoft”);

os1.setDesc(“active”);

HashMap<String, OperatingSystem> map = new HashMap<>();

      map.put("os1", os1);

System.out.println("HashMap: " + map); // HashMap: {os1=OperatingSystem(id=10,name=Microsoft,Desc=active)}

map.computeIfAbsent(“os2”, (v) -> createOS());

System.out.println("New HashMap: " + map); // New HashMap: {os1=OperatingSystem(id=10,name=Microsoft,Desc=active),

// os2=OperatingSystem(id=50,name=Valoores,Desc=null)}

**ComputeIfPresent(Key, Function) Method**

This method computes a specified mapping function for the given key and its associated value, and then updates the value for that key if the value for the specified key is present and non-null

**Example 1:**

HashMap<Integer, String> cityMap = new HashMap<>();

cityMap.put(101, "Varanasi");

cityMap.put(102, "Paris");

cityMap.put(103, null);

cityMap.computeIfPresent(102, (k, v) -> null);

System.out.println(cityMap); // {101=Varanasi, 103=null} -- 102 is removed

cityMap.computeIfPresent(103, (k, v) -> "Beirut");

System.out.println(cityMap); // {101=Varanasi, 103=null}

cityMap.put(104, "Beirut");

String newVal = cityMap.get(104);

cityMap.computeIfPresent(104, (k, v) -> newVal.concat("-Tripoli"));

System.out.println(cityMap); // {101=Varanasi, 103=null, 104=Beirut-Tripoli}

**Example 2:**

public OperatingSystem updateOS(OperatingSystem os)

{

os.setDesc(“updated”);

return os;

}

OperatingSystem os1 = new OperatingSystem()

os1.setId(10);

os1.setName(“Microsoft”);

os1.setDesc(“active”);

System.out.println("HashMap: " + map); // HashMap: {os1=OperatingSystem(id=10,name=Microsoft,Desc=active)}

map.computeIfPresent(“os1”, (k, v) -> updateOS(v));

System.out.println("New HashMap: " + map); // New HashMap: {os1=OperatingSystem(id=10,name=Microsoft,Desc=update)}

## Default and Static methods

Before interfaces could have only public abstract methods. It was not possible to add new functionality to the existing interface without forcing all implementing classes to create an implementation of the new methods, nor it was possible to create interface methods with an implementation.

Now interfaces can have ***static*** and **default** methods that, despite being declared in an interface, have a defined behavior.

**Static Method**

Consider an interface called Vehicle with the following method

static String producer()

{

return "N&F Vehicles";

}

The static producer() method is available only through and inside of an interface. It can't be overridden by an implementing class.

To call it outside the interface, the standard approach for static method call should be used:

String producer = Vehicle.producer();

**Default Method**

Default methods are declared using the new default **keyword**. These are accessible through the instance of the implementing class and can be overridden.

Let's add a default method to our Vehicle interface, which will also make a call to the static method of this interface:

default String getOverview()

{

return "ATV made by " + producer();

}

For executing the default method (in class VehicleImpl) an instance of this class should be created:

Vehicle vehicle = new VehicleImpl();

String overview = vehicle.getOverview();

## Optional<t>

Before developers had to carefully validate values they referred to, because of a possibility of throwing the NullPointerException.

Optional<T> class can help to handle situations where there is a possibility of getting the NullPointerException. It works as a container for the object of type T. It can return a value of this object if this value is not a null. When the value inside this container is null, it returns an empty object.

**Example:**

**In DAOImpl** public Optional<ProfileVO> getProfile(ProfileSC profileSC) throws SoftSolDAOException  
 {  
   return Optional.ofNullable(getSqlSessionTemplate().selectOne("USM\_Profile.getProfileDetail", profileSC.getId()));  
 }  
     
  **In BOImpl**  
  public Optional<ProfileVO> getProfile(ProfileSC profileSC) throws SoftSolException  
 {  
   Optional<ProfileVO> theVO = profileDAO.getProfile(profileSC);  
   return theVO;  
 }  
  
 **In Action** protected ActionForward select(ActionMapping actionMapping, ActionForm form, HttpServletRequest request,

HttpServletResponse response, String actionType)  
 {  
  .........  
   profileVO = profileBO.getProfile(criteria);  
   profileVO.ifPresent(theVO ->   
          {  
             copyProperties(profileForm, theVO, getLocale(request));  
             request.getSession().setAttribute("applicationId", profileForm.getApplication\_id());  
             request.setAttribute("applicationId", profileForm.getApplication\_id());  
          });  
   ....  
 }