Collectors.teeing() in Stream API:

A teeing collector has been exposed as a static method Collectors::teeing. This collector forwards its input to two other collectors before merging their results with a function.

teeing(Collector, Collector, BiFunction) accepts two collectors and a function to merge their results. Every element passed to the resulting collector is processed by both downstream collectors, then their results are merged using the specified merge function into the final result.

Code:

```
public class Person {
 String name;
 int age;

 public Person(String name, int age) {
 this.name = name;
 this.age = age;
 }

 @Override
 public String toString() {
 return "Person{" +
 "name='" + name + '\'' +
 ", age=" + age +
 '}';
 }
}
```

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

```
import static java.util.stream.Collectors.counting;
import static java.util.stream.Collectors.summingDouble;

public class CollTeeing {
 public static void main(String[] args) {
 List<Person> personList = Arrays.asList(new Person("bob", 34), new
Person("bob", 43),
 new Person("mary", 81), new Person("john", 12), new Person("bob", 22));

 System.out.println("list of person objects - \n" + personList);
 Stream<Person> personStream = personList.stream();

 Map<String, List<Person>> result = personStream.collect(Collectors.teeing(
 Collectors.filtering(p -> p.age % 2 == 0, Collectors.toList()),
 Collectors.filtering(p -> p.age % 2 != 0, Collectors.toList()),
 (res1, res2) -> {
 Map<String, List<Person>> map = new HashMap<>();
 map.put("EvenAgedPersons", res1);
 map.put("OddAgedPersons", res2);
 return map;
 }));
 System.out.println("Result of applying teeing -\n " + result);

 double mean = Stream.of(1, 2, 3, 4, 5)
 .collect(Collectors.teeing(
 summingDouble(i -> i),
 counting(),
 (sum, n) -> sum / n));
 System.out.println(mean);
 }
}
```

Output:

```
list of person objects -

[Person{name='bob', age=34}, Person{name='bob', age=43}, Person{name='mary', age=81},
Person{name='john', age=12}, Person{name='bob', age=22}]

Result of applying teeing -
```

{OddAgedPersons=[Person{name='bob', age=43}, Person{name='mary', age=81}], EvenAgedPersons=[Person{name='bob', age=34}, Person{name='john', age=12}, Person{name='bob', age=22}]}

3.0

String API:

a.) String.indent(): The indent method helps with changing the indentation of a String. We can either pass a positive value or a negative value depending on whether we want to add more white spaces or remove existing white spaces.

b.) String.transform(): The transform() method takes a String and transforms it into a new String with the help of a Function.

c.) String constants: Since Java 12, String class implements two additional interfaces java.lang.constant.Constable and java.lang.constant.ConstantDesc.

String class also introduces two additional low-level methods describeConstable() and resolveConstantDesc(MethodHandles.Lookup).

Code:

```
import java.lang.invoke.MethodHandles;
import java.util.ArrayList;
import java.util.List;
import java.util.*;

public class StringNewMethodsJava12 {
 public static void main(String[] args) {
 String result = "foo\nbar\nbar2".indent(4);
 System.out.println(result);

 List<String> names = List.of(" Alex", "brian ");

 List<String> transformedNames = new ArrayList<>();
 for (String name : names) {
 String transformedName = name.transform(String::strip)
 .transform(String::toUpperCase);
 transformedNames.add(transformedName);
 }
 System.out.println(transformedNames);
```

```
 String str = "JavaHungry";
 Optional optional = str.describeConstable();
 System.out.println(optional);
 System.out.println(optional.get());

 String constantDesc = str.resolveConstantDesc(MethodHandles.lookup());
 System.out.println(constantDesc.equals(str));
 System.out.println(constantDesc);
 }
}
```

Output:

```
foo

bar

bar2


[ALEX, BRIAN]

Optional[JavaHungry]

JavaHungry

true

JavaHungry
```

Compact Number Formatting:

Large numbers rendered by a user interface or a command-line tool are always difficult to parse. It is far more common to use the abbreviated form of a number. Compact number representations are much easier to read and require less space on the screen without losing the original meaning.

E.g. 3.6 M is very much easier to read than 3,600,000.

Java 12 introduces a convenient method called NumberFormat.getCompactNumberInstance(Locale, NumberFormat.Style) for creating a compact number representation.

Code:

```
import java.text.NumberFormat;
import java.util.Locale;

public class CompactNumberFormatting {

 public static void main(String[] args) {
 System.out.println("Compact Formatting is:");
 NumberFormat upvotes = NumberFormat
 .getCompactNumberInstance(new Locale("en", "US"),
NumberFormat.Style.SHORT);
 upvotes.setMaximumFractionDigits(1);
 System.out.println(upvotes.format(2592) + " upvotes");

 NumberFormat upvotes2 = NumberFormat
 .getCompactNumberInstance(new Locale("en", "US"),
NumberFormat.Style.LONG);
 upvotes2.setMaximumFractionDigits(2);
 System.out.println(upvotes2.format(2011) + " upvotes");
 }
}
```

Output:

Compact Formatting is:

2.6K upvotes

2.01 thousand upvotes

Switch Expressions:

This change extends the switch statement so that it can be used as either a statement or an expression.

Instead of having to define a break statement per case block, we can simply use the arrow syntax. The arrow syntax semantically looks like a lambda and separates the case label from the expression.

Code:

```
import static java.util.Calendar.*;

public class SwitchExpression {
 public static void main(String[] args) {
 int day = 1;
 boolean isWeekend = switch (day) {
 case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> false;
 case SATURDAY, SUNDAY -> true;
 default -> throw new IllegalStateException("Illegal day entry :: " + day);
 };
 System.out.println(isWeekend);
 }
}
```

Output:

true

Files.mismatch(Path, Path):

Sometimes, we want to determine whether two files have the same content. This API helps in comparing the content of files.

mismatch() method compares two file paths and return a long value. The long indicates the position of the first mismatched byte in the content of the two files. The return value will be '–1' if the files are “equal.”

Code:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class FilesMisMatch {
 public static void main(String[] args) throws IOException {
 Path filePath1 = Files.createTempFile("file1", ".txt");
 Path filePath2 = Files.createTempFile("file2", ".txt");
 Files.writeString(filePath1, "JournalDev Test String");
```

```
 Files.writeString(filePath2, "JournalDev Test String");

 long mismatch = Files.mismatch(filePath1, filePath2);
 System.out.println("File Mismatch position... It returns -1 if there is no
mismatch");
 System.out.println("Mismatch position in file1 and file2 is >>>> " + mismatch);

 filePath1.toFile().deleteOnExit();
 filePath2.toFile().deleteOnExit();

 Path filePath3 = Files.createTempFile("file3", ".txt");
 Path filePath4 = Files.createTempFile("file4", ".txt");
 Files.writeString(filePath3, "JournalDev Test String");
 Files.writeString(filePath4, "JournalDev.com Test String");

 long mismatch2 = Files.mismatch(filePath3, filePath4);
 System.out.println("Mismatch position in file3 and file4 is >>>> " + mismatch2);

 filePath3.toFile().deleteOnExit();
 filePath4.toFile().deleteOnExit();
 }
}
```

Output:

File Mismatch position... It returns -1 if there is no mismatch

Mismatch position in file1 and file2 is >>>> -1

Mismatch position in file3 and file4 is >>>> 10