Private Methods:

Private static methods will provide code reusability for public static methods.

Note: Interface static methods should be called by using interface name only even in implementation classes also.

Code:

```
public interface PrivateMethodsInterface {
 public static void m1() {
 m3();
 }

 public static void m2() {
 m3();
 }

 private static void m3(){
 System.out.println("Common Functionality of m1 and m2");
 }
}

public class PrivateMethodsImplemention implements PrivateMethodsInterface {

 public static void main(String[] args) {
 PrivateMethodsInterface.m1();
 PrivateMethodsInterface.m2();

 }
}
```

Output:

```
Common Functionality of m1 and m2

Common Functionality of m1 and m2
```

Try With Resource Enhancement:

We can use the resource reference variables which are created outside of try block directly in try block resources list. i.e. The resource reference variables need not be local to try block.

Code:

```
public class TryWithMultipleResources {

 public static void main(String[] args) {
 singleResource();
 System.out.println("Program Execution Multiple Resources With JDK9");
 multipleResource();
 }

 public static void singleResource() {
 MyResource r = new MyResource();
 try (r) {
 r.doProcess();
 } catch (Exception e) {
 System.out.println("Handling:" + e);
 }
 }

 public static void multipleResource() {
 MyResource r1 = new MyResource();
 MyResource r2 = new MyResource();
 MyResource r3 = new MyResource();
 MyResource r4 = new MyResource();
 try (r1; r2; r3; r4) {
 r1.doProcess();
 r2.doProcess();
 r3.doProcess();
 r4.doProcess();
 } catch (Exception e) {
 System.out.println("Handling:" + e);
 }
 }
}
```

Output:

Resource Creation...

Resource Processing...

Resource Closing...

Program Execution Multiple Resources With JDK9

Resource Creation...

Resource Creation...

Resource Creation...

Resource Creation...

Resource Processing...

Resource Processing...

Resource Processing...

Resource Processing...

Resource Closing...

Resource Closing...

Resource Closing...

Resource Closing...

@SafeVarargs Annotation:

Very few Var-arg Methods cause Heap Pollution, not all the var-arg methods. If we know that our method won't cause Heap Pollution, then we can suppress compiler warnings with @SafeVarargs annotation.

@SafeVarargs Annotation introduced in Java 7. Unitl Java 8, this annotation is applicable only for static methods, final methods and constructors. But from Java 9 onwards, we can also use for private instance methods also.

Code:

```
import java.util.*;

public class SafeVar {
 public static void main(String[] args) {
```

```
List<String> l1 = new ArrayList<>();
l1.add("A");
l1.add("B");
SafeVar sv = new SafeVar(l1);
sv.m1(l1);
sv.m2(l1);
sv.m3(l1);
}

@SafeVarargs
public SafeVar(List<String>... l) {
System.out.println("I am safevar const");
}

@SafeVarargs
public static void m1(List<String>... l) {
System.out.println("I am safevar m1");
}

@SafeVarargs
public final void m2(List<String>... l) {
System.out.println("I am safevar m2");
}

@SafeVarargs //valid in Java 9 but not in Java 8
private void m3(List<String>... l) {
System.out.println("I am safevar m3");
}
}
```

Output:

I am safevar const

I am safevar m1

I am safevar m2

I am safevar m3

Factory Methods:

Factory Methods of List, Set and Map, Up to 10 entries,it is recommended to use of() methods and for more than 10 items we should use ofEntries() method.

Code:

```
public class FactoryMethods {

 public static void main(String[] args) {

 List<String> beers = List.of("KF","KO","RC","FO");
 System.out.println(beers);

 Set<Integer> numbers=Set.of(10,20,30);
 System.out.println(numbers);

 Map<String,String> map=Map.of("A","Apple","B","Banana","C"
,"Cat","D","Dog");
 System.out.println(map);
 }
}
```

Output:

[KF, KO, RC, FO]

[20, 30, 10]

{C=Cat, B=Banana, A=Apple, D=Dog}

Process API Enhancement(Stream):

1.) takeWhile() -> Similar to filter() and when it got the false, the rest of remaining elements will stop processing.

2.) dropWhile() -> Opposite of takeWhile() method & It drops elements instead of taking them as long as predicate returns true. Once predicate returns false then rest of the Stream will be returned.

3.) Stream.iterate() -> It takes an initial value and a function that provides next value.

4.) Stream.ofNullable() -> If value is available then it will return that otherwise it will return null.

Code:

```
import java.util.*;
import java.util.stream.*;


public class StreamAPI {
 public static void main(String[] args) {
 ArrayList<Integer> l1 = new ArrayList<Integer>();
 l1.add(2);l1.add(4);l1.add(1);l1.add(3);l1.add(6);l1.add(5);l1.add(8);
 System.out.println("Initial List:" + l1);
 List<Integer> l2 = l1.stream().filter(i -> i % 2 == 0).collect(Collectors.toList());
 System.out.println("After Filtering:" + l2);
 List<Integer> l3 = l1.stream().takeWhile(i -> i % 2 ==
0).collect(Collectors.toList());
 System.out.println("After takeWhile:" + l3);
 List<Integer> l4 = l1.stream().dropWhile(i -> i % 2 ==
0).collect(Collectors.toList());
 System.out.println("After dropWhile:" + l4);
 List l8 = Stream.ofNullable(100).collect(Collectors.toList());
 System.out.println(l8);
 List l9 = Stream.ofNullable(null).collect(Collectors.toList());
 System.out.println(l9);
 Stream.iterate(1,x->x<5,x->x+1).forEach(System.out::println);
 //Stream.iterate(1,x->x+1).forEach(System.out::println);
 }}
```

Output:

```
Initial List:[2, 4, 1, 3, 6, 5, 8]

After Filtering:[2, 4, 6, 8]

After takeWhile:[2, 4]

After dropWhile:[1, 3, 6, 5, 8]

[100]

[]

1
```

2

3

4

Annoymous Class:

We can use Diamond Operator for Anonymous Classes also.

ArrayList<String> l = new ArrayList<>(){};

Code:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class DiamondOperatorAnnonymous {
 public static void main(String[] args) {
 String[] animals = {"Dog", "Cat", "Rat", "Tiger", "Elephant"};
 Iterator<String> iter = new Iterator<>() {
 int i = 0;

 public boolean hasNext() {
 return i < animals.length;
 }

 public String next() {
 if (!hasNext())
 throw new NoSuchElementException();
 return animals[i++];
 }
 };
 while (iter.hasNext()) {
 System.out.println(iter.next());
 }
 }
}
```

Output:

Dog

Cat

Rat

Tiger

Elephant