Exercise 1: factorial
Write a stream that computes the factorial of \( n \).

Exercise 2: Fibonacci tuples series (iterate)
The numbers in the following sequence are part of the Fibonacci series: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... The first two numbers of the series are 0 and 1, and each subsequent number is the sum of the previous two.

The series of Fibonacci tuples is similar; you have a sequence of a number and its successor in the series: (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), ...

Generate the first 10 elements of the series of Fibonacci tuples using the iterate method.

The first problem is that the iterate method takes a UnaryOperator as argument, and you need a stream of tuples such as (0, 1). We use an array of two elements to represent a tuple. For example, new int[]{0, 1} represents the first element of the Fibonacci series (0, 1). This will be the initial value of the iterate method:

```java
Stream.iterate(new int[]{0, 1}, ???)
          .limit(10)
          .map(t -> t[0])
          .forEach(System.out::println);
```

You need to figure out the ??? in the code. Remember that iterate will apply the given lambda successively.

Exercise 3: Fibonacci tuples series (generate)
Complete the following code to obtain the same result with generate.

```java
IntSupplier fib = new IntSupplier() {
    ?
    public int getAsInt() {
        ???
    }
};
IntStream.generate(fib)
          .limit(10)
          .forEach(System.out::println);
```

Exercise 4: joining strings with reducing
In the following exercises, you have access to the methods getType and getName from class Dish that return both a String. The list of all dishes is in menu.

Which of the following statements using the reducing collector are valid replacements for this joining collector? Answer the question by carefully analyzing the type of each step and function.

```java
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```
String shortMenu = menu.stream().map(Dish::getName)
    .collect(reducing((s1, s2) -> s1 + s2)).get();

String shortMenu = menu.stream()
    .collect(reducing((d1, d2) -> d1.getName() + d2.getName()))
    .get();

String shortMenu = menu.stream()
    .collect(reducing("", Dish::getName, (s1, s2) -> s1 + s2));

Exercise 5: groupingBy
We would like to generate a list of dishes for each dish type. This would lead to the following output:

{MEAT=[pork, beef, chicken],
 FISH=[prawns, salmon],
 OTHER=[french fries, rice, season fruit, pizza]}

Write the code to achieve this objective.

Exercise 6: flatMapping
A list of tags is associated to each dish name as follows (we assume that each possible dish appears in this map):

```java
Map<String, List<String>> dishTags = new HashMap<>();
dishTags.put("pork", asList("greasy", "salty"));
dishTags.put("beef", asList("salty", "roasted"));
dishTags.put("chicken", asList("fried", "crisp"));
dishTags.put("french fries", asList("greasy", "fried"));
dishTags.put("rice", asList("light", "natural"));
dishTags.put("season fruit", asList("fresh", "natural"));
dishTags.put("pizza", asList("tasty", "salty"));
dishTags.put("prawns", asList("tasty", "roasted"));
dishTags.put("salmon", asList("delicious", "fresh"));
```

We would like to generate a list of all unique tags of only the dishes contained in a given list `submenu`. This would lead to the following output:

{{salty, greasy, roasted, fried, crisp, tasty, fresh, delicious, natural, light}}

Write the code to achieve this objective.

Exercise 7: collector composition
We would like to generate a list of unique tags for each dish type. This would lead to the following output:

{MEAT=[salty, greasy, roasted, fried, crisp],
 FISH=[roasted, tasty, fresh, delicious],
 OTHER=[salty, greasy, natural, light, tasty, fresh, fried]}

Write the code to achieve this objective.

Exercise 8: iterate
Rewrite the following without the predicate in iterate:
```java
IntStream.iterate(0, n -> n < 100, n -> n + 4).
    .forEach(System.out::println);
```

**Exercise 9: maximum collector alternative**

Write an alternative to:

```java
Comparator<Dish> dishCaloriesComparator = Comparator.comparing(Dish::getCalories);
int maxCalories = menu.stream()
    .collect(maxBy(dishCaloriesComparator))
    .get()
    .getCalories();
```

**Exercise 10: summing collector alternative**

Write an alternative to:

```java
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

**Exercise 11: using partitioningBy**

Like the `groupingBy` collector, the `partitioningBy` collector can be used in combination with other collectors. In particular, it could be used with a second `partitioningBy` collector to achieve a multilevel partitioning. What will be the result of the following multilevel partitionings and what is the type of each returned value? Answer the question by carefully analyzing the type of each step and function.

1. `menu.stream() .collect(partitioningBy(Dish::isVegetarian, partitioningBy(d -> d.getCalories() > 500)));

2. `menu.stream() .collect(partitioningBy(Dish::isVegetarian, partitioningBy(Dish::getType)));

3. `menu.stream() .collect(partitioningBy(Dish::isVegetarian, counting()));`

**Exercise 12: simulating `takeWhile` in Java 8**

The `takeWhile` method was introduced in Java 9, so unfortunately you cannot use this solution if you are still using Java 8. How could you work around this limitation and achieve something similar in Java 8?