Kotlin Museum Visit

COMP40009 - Computing Practical 1

12th – 16th February 2024

Aims

- To gain additional experience in object-oriented design using classes
- To practice writing simple interactive programs in Kotlin
- To gain experience writing concurrent programs in Kotlin
- To explore using locks to achieve mutual exclusion between threads

Introduction

This exercise involves modelling a museum comprising a number of exhibition rooms that visitors can move between. Exactly one of the exhibition rooms is the museum *entrance*. Rooms can be connected by *turnstiles*. An exhibition room has limited capacity, i.e., at most a certain number of visitors can be in the room at any time. Thus a turnstile can only be used by a visitor if the destination room of the turnstile has spare capacity. Some turnstiles lead to *outside* the museum.

Figures 1 and 2 illustrate this idea by showing layouts for two museums: an art gallery, comprising an entrance room and a single exhibition room, and an animal sanctuary with rooms containing various animal exhibits, as well as a gift shop. The capacity is shown for each room. The turnstiles are indicated by one-way arrows. The entrance to the museum, leading to its entrance room, is shown as a thin black rectangle.

Your first task will be to design suitable Kotlin classes to represent a museum, and methods to check that a museum meets certain structural requirements.

Then you will write a simple command line program to allow a museum to be explored, in the style of an old school text adventure game.[1]

Finally, you will do some concurrent programming. You will write a class that models a museum visitor who explores the rooms of a museum by randomly walking between them via turnstiles. Each museum visitor object will run in a separate thread, and the challenge for you will be to

[1] See, for example, Zork: `https://en.wikipedia.org/wiki/Zork`

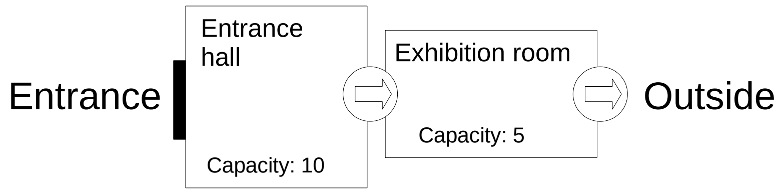

Figure 1: Layout and room capacities for an art gallery

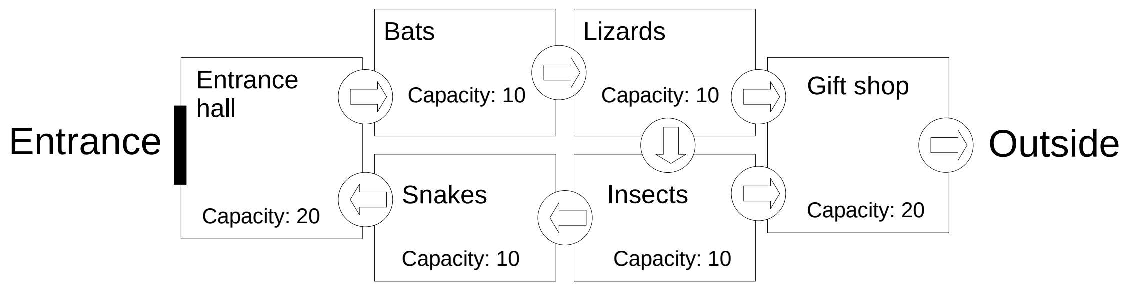

Figure 2: Layout and room capacities for an animal sanctuary

use Kotlin's support for locks to allow visitors to successfully navigate the museum concurrently, ensuring that the capacity constraints on rooms are respected and that visitor threads do not run into deadlock scenarios.

Note on `is` and `as` in Kotlin: Two Kotlin features that you may find useful during this lab which may not have been covered in the lectures yet are `is` and `as`. You can use `is` to ask whether, at runtime, the type of an expression (its "actual" type) is a particular subtype of its declared type (its "apparent" type). If, for some reason, you *know* that an expression will have a particular actual type that is a subtype of its apparent type, you can use `as` to *downcast* the expression to this type, so that you can use properties and methods that are specific to the subtype. Beware, though: if your use of `as` turns out to be wrong, your program will fail with an exception.

Getting started

As per the previous exercises, use `git` to clone the repository with the skeleton files for this exercise, via

```
git clone git@gitlab.doc.ic.ac.uk:lab2324_spring/kotlinmusem_username.git
```

Modelling museums

Your first task is to write classes that model a museum. In designing the classes that follow, you should strive to maximise encapsulation: make properties and methods of these classes no more visible and no more mutable than is necessary to meet the given requirements.

The `MuseumRoom` class

Write a class `MuseumRoom` representing a room in a museum.

A museum room should be constructed via its name (a string) and its capacity (a positive integer, a precondition that should lead to an `IllegalArgumentException` being thrown if it is violated), recording the maximum number of visitors that can be in the room at any one time.

This is the service that a museum room should provide to its clients:

- Read access to the name of the room.
- Read access to `occupancy`, which records how many visitors are currently in the room. The occupancy is initially zero, and should never exceed the capacity of the room.
- A `hasCapacity()` method that returns true if and only if there is space for at least one more visitor in the room.

- Methods `enter()` / `exit()`, which throw an `UnsupportedOperationException` if the room is full / empty, respectively, and have the effect of incrementing / decrementing the room's occupancy otherwise.

A museum room is an example of a *museum site*. You will consider another kind of museum site below.

Test your code using the example tests provided in `MuseumRoomTest.kt`. Use the IntelliJ "Run tests with coverage" option to see the percentage of statement coverage of your `MuseumRoom` class that these tests achieve. Add additional tests so that 100% statement coverage is achieved. Ensure that the tests you write are meaningful: they should assert properties about the code that they exercise, so that if the code is wrong the tests will fail.

The `Museum` class

Write a class `Museum` representing a museum comprised of one or more rooms.

A museum should be constructed from a name (a string), and a museum room that serves as the entrance to the museum.

This is the service that a museum should provide to its clients:

- Read access to the name of the museum.
- Read access to `admitted`, which records how many visitors have been admitted to the museum since it was created. The value of `admitted` is initially zero, and should only ever be incremented.
- Read access to a special museum site representing *outside* the museum. Like a museum room, the outside area should have a name, which should be the fixed string "`Outside`", and an occupancy that should initially be zero. Like a museum room, it should be possible to enter the outside space, increasing its occupancy by one. Unlike a museum room, the capacity of the outside space is unlimited, and it should not be possible to exit the outside space (what visitors get up to once they have left a museum is beyond the scope of this exercise!)
- An `entranceHasCapacity` method that returns true if and only if the museum's entrance room has available capacity.
- An `enter` method that requires `entranceHasCapacity()` to hold. If it does not then an `UnsupportedOperationException` should be thrown. Otherwise, the count of visitors that have been admitted to the museum should be incremented, and the `enter` method of the museum's entrance room invoked.
- An `addRoom` method that takes a `MuseumRoom` parameter and adds this room to a set of rooms that form the museum. Initially this set of rooms should comprise the museum's entrance room. Adding a room to a museum when the museum already contains a room with the same name should lead to an `IllegalArgumentException` being thrown.
- A `connectRoomTo` method that takes two `MuseumRoom`s. They should both already belong to the museum; if not, an `IllegalArgumentException` should be thrown. Similarly, an `IllegalArgumentException` should be thrown if there is already a turnstile from the first room to the second room, or if the second room is the same room as the first room. Otherwise, the method should have the effect of connecting the first room to the second room via a turnstile.
- A `connectRoomToExit` method that takes a `MuseumRoom` parameter and adds a turnstile connecting this room to the space outside the museum. The room should be one of the museum's room, and there should not already be a turnstile connecting this room to the outside

space—an `IllegalArgumentException` should be thrown if either of these preconditions is violated.

It is up to you how you represent turnstiles between rooms. However, the turnstiles leading from a room should be ordered according to the order in which the connections between rooms were added to the museum.

A museum is considered *invalid* if one of the following holds:

- The museum contains a room that cannot be reached from the museum entrance by passing through turnstiles.
- The museum contains a room from which the outside space cannot be reached by passing through turnstiles.

A museum will inevitably be invalid while it is being constructed, but a museum should be valid before it is accessed by visitors.

To support checking whether a museum is valid, write two custom exception classes that extend the `Exception` class (the superclass of all exceptions):

- `UnreachableRoomsException`, which should be constructed with a set of rooms that have been found to be unreachable from a museum's entrance. The `toString` method of this exception class should be overridden to return a string of the form:

  "`Unreachable rooms: `"

  followed by the names of the museums in the set, sorted alphabetically and separated by commas.

- `CannotExitMuseumException`, which should be constructed with a set of rooms from which it has been identified that the museum cannot be exited. Similar to `UnreachableRoomsException`, the `toString` method of this exception class should be overridden to return a string of the form:

  "`Impossible to leave museum from: `"

  followed by the names of the museums in the set, sorted alphabetically and separated by commas.

Do what you can to avoid duplication of code between these exception classes.

Armed with these custom exceptions, add a public method `checkWellFormed` to `Museum` that:

- Identifies all rooms that are unreachable in the museum, and throws an `UnreachableRoomsException` reporting all such rooms, if any exist.
- Otherwise, if all rooms are reachable, identifies all rooms in the museum from which exiting the museum is impossible, and throws a `CannotExitMuseumException` reporting all such rooms, if any exist.
- Otherwise, returns normally without throwing an exception.

Override `toString` in `Museum` according to the following specification:

- `toString` can assume that it is only called on a well-formed museum.
- The string that is returned should comprise multiple lines: a line giving the name of the museum and then lines describing each room.
- The museum name line should consist of just the museum's name.
- The line for a room should have the form

  "`Room` *name_of_room* `leads to: `"

followed by a comma-separated list of the museum sites that can be reached via a turnstile from the room, listed in turnstile order. As mentioned before, this order in which connections have been added to the museum.

- The order in which the lines should appear should correspond to the order in which the rooms would be visited in a breadth-first search of the museum, starting at its entrance room.

The tests that are provided to help you validate your solution provide additional clarity on exactly how exception strings and the string representation of a museum should be formatted.

Creating some example museums and testing your `Museum` class

In a new file, `ExampleMuseums.kt`, write two top-level functions, each taking no parameters and having return type `Museum`:

- `createArtGallery`, which should return a museum corresponding to the simple gallery of Figure 1.
- `createAnimalSanctuary`, which should return a museum corresponding to the animal sanctuary of Figure 2.

For two rooms in the animal sanctuary that have multiple exit turnstiles, the turnstile to the gift shop should be added *after* the other turnstile. (This is necessary so that your museum structures match what is expected by various tests.)

Add two additional methods to this file that create *invalid* museums:

- `createAnimalSanctuaryWithUnreachableRooms`: this should create a museum identical to the animal sanctuary, except that the turnstile connecting “Entrance hall” to “Bats” should not be added, making various rooms in the museum unreachable.
- `createAnimalSanctuaryWithRoomsThatDoNotLeadToExit`: this should create a museum identical to the animal sanctuary, except that the following turnstiles should not be added: “Insects” → “Gift shop”, “Snakes” → “Entrance hall”, so that it is impossible to exit the museum from certain rooms.

Test your code using the example tests provided in `Museum.kt`. Add additional tests so that you achieve 100% statement coverage of the `Museum` class, the exceptions that you created, and any other classes you may have created to support turnstiles between rooms. Again, ensure that the new tests you write are meaningful.

Writing an interactive program for museum exploration

Note: the next section, on concurrency, is orthogonal to this section on writing an interactive explorer. You are free to do these parts of the exercise in either order.

Your next task is to write an interactive command line program that allows a single visitor—the user—to explore a museum, in the style of a classic text adventure game. Write the program in a new file, `Explorer.kt`.

The program should allow the user to select (by typing) which of a number of museums they wish to explore, and should let them explore the museum until they reach its exit or enter an end-of-file character to indicate that they have had enough. They should then be given the chance to explore another museum. They can enter another end-of-file character to exit the explorer. On Linux you can use `Ctrl+D` to type an end-of-file character. On Windows it is `Ctrl+Z`. (On Mac, who knows—but you can probably figure it out!)

When the user types an unknown museum or destination they should be asked to try again.

You are free to decide on exactly how the explorer program should work in terms of the messages it displays to the user. The appendix shows a possible session with the explorer, with the art gallery and animal sanctuary museums, where commands entered by the user are in green.

Hint: You may find Kotlin's `readlnOrNull` function useful in writing the explorer.

There are no automated tests for the explorer program—you should test it manually.

Concurrent visitors

Your final task (before the extension) is to write a class representing a visitor to a museum that can execute in its own thread. This allows you to simulate multiple visitors accessing a museum concurrently. The challenge associated with this is to ensure that the capacity constraints of the rooms in a museum are not violated by its visitors. Visitors should move between rooms without ever causing the maximum capacity of any room to be exceeded.

The visitor you are going to model is an *impatient* visitor. This visitor will spend some time in a room of the museum. They will then decide on which turnstile they want to go through to leave the room, choosing randomly among the available turnstiles. However, if they find that they cannot go through the chosen turnstile (because this would cause the capacity of the next room to be exceeded), then instead of waiting until the destination room has available capacity, they briefly pause and then choose a turnstile again at random. They are *impatient* due to their unwillingness to wait until a turnstile becomes usable. (If you do the extension, you will write an alternative *patient* visitor class, representing a visitor who is willing to wait.)

Supporting impatient visitors will require augmenting your museum classes with some state and methods to support concurrent visitors, and then writing a class that implements `Runnable` to represent an impatient visitor.

Adding locks to museum sites

A visitor thread that enters the museum, moves between rooms and exits the museum will need to *inspect* and *change* the state of the museum and its various sites. Because multiple visitors will be inspecting and changing this state *concurrently*, we will need to use *concurrency control* in the form of *locks* to ensure that they do so without compromising the state of the museum.

To illustrate why concurrency control is needed, suppose that two visitors A and B both wish to enter room X. Suppose the occupancy of X is 9 and its capacity is 10. If we are not careful, we can have a scenario where A and B both check, almost simultaneously, whether room X has spare capacity, and *both* find that it does. They then *both* enter room X, causing the occupancy of X to be incremented twice, so that the occupancy of X becomes 11 when the capacity of X is only 10.

To avoid such *race conditions*, you must make sure each class representing a museum site has a lock object.

Add a method to `Museum` called `enterIfPossible`. This method should first acquire the lock of the museum's entrance room to check whether the museum entrance has capacity. Then, if capacity is available, it can then use the existing service provided by `Museum`, and return a reference to the entrance room. If the museum entrance is found to be full, the method should return null. Thus the method's return type should be `MuseumRoom?`. Take care to ensure that, whatever happens, the lock on the entrance room has been released when the method returns.

To allow visitors to pass safely between rooms, add a method to an appropriate place in your museum code to model a visitor passing through a turnstile. Where you add this method, and its exact form, will depend on how you chose to implement turnstiles. Remember that a turnstile connects two sites—the starting room (the room that you exit via the turnstile), and the destination site (the room to which the turnstile leads, or the outside space if the turnstile exits the museum). The method that you write should acquire two locks: the lock associated with the starting room of the turnstile, and the room associated with the destination site. Once these locks are acquired, the method should check whether the destination site has available capacity. If it does, the method should call `exit` on the starting room and `enter` on the destination site, to model the transition between rooms. The method should return something that indicates whether the transition between rooms was successful or not. One option is to return a boolean; another is to return a nullable reference that will be null if the transition was not successful, or a reference to the destination site if the transition was successful. You will need to make an appropriate decision here based on how you have modelled turnstiles.

Avoiding deadlocks

Suppose that visitor X is attempting to move from room A to room B at the same time that visitor Y is attempting to move from room B to room A via a turnstile in the opposite direction. This has the possibility of leading to deadlocks due to concurrent calls to the room transition method described above. The thread for visitor X transitioning from A to B must acquire both A's lock and B's lock. Suppose it acquires A's lock, but has not yet acquired B's lock. The thread for visitor Y transitioning form B to B must also acquire both A's lock and B's lock. Suppose it acquires B's lock, but has not yet acquired A's lock. The threads are now in deadlock: each needs to acquire a lock that the other holds. More generally a deadlock can occur whenever there is a circular dependency between threads with respect to the locks that they hold and the locks that they require.

To avoid deadlocks, you must ensure that locks on museum sites are consistently acquired according to some *total order* on museum sites.

Writing an `ImpatientVisitor` class

Now that you have put in place facilities for visitors to enter the museum and transition between rooms without invalidating the state of the museum, it is time to implement a class for our impatient visitors!

Write a class, `ImpatientVisitor`, that implements `Runnable`. An impatient visitor should be constructed from a name, a print stream (to which it will write output), and the museum to be visited.

The `run` method (required by `Runnable`) should operate as follows:

- The visitor should enter the museum by repeatedly calling `enterIfPossible` until a non-null `MuseumRoom` reference is returned (a reference to the museum's entrance room). Whenever null is returned, the visitor should sleep for 10 milliseconds before trying again. Remark: we'll use milliseconds as the (arbitrary) time unit in order to speed up the simulation.
- Once in the museum, the visitor should keep track of which room it is in. The visitor should spend a random period of time in the room—between 1–200 milliseconds—before attempting to leave the room.
- When it is time for the visitor to leave a room, it should choose one of the turnstiles from which the room can be exited at random, and call the method that you added as per the instructions above to safely transition to another museum site via a turnstile.
- If transitioning through a turnstile succeeds and leads to another room, the visitor should update its current room to be this new room, spend some time in the room, and then leave the room, etc. If transitioning through a turnstile leads to the outside of the museum, the visitor should terminate—i.e. the `run` method should return.

- If transitioning through a turnstile fails, due to the site on the other side of the turnstile being at capacity, the visitor should pause for 10 milliseconds. The visitor should then try to leave the room, again by choosing a turnstile at random (possibly choosing the same turnstile that the visitor just failed to go through, as the site on the other side may now have available capacity).

A visitor should use its print stream to print lines that record its progress, according to the following scheme:

- On failing to enter the museum:

  "*visitor_name* `could not get into` *museum_name* `but will try again soon.`"

- Before trying to enter the museum again:

  "*visitor_name* `is ready to try again.`"

- On successfully entering the museum:

  "*visitor_name* `has entered` *museum_name*`.`"

- On entering a room (including entering the entrance room for the first time):

  "*visitor_name* `has entered` *room_name*`.`"

- When ready to leave a room:

  "*visitor_name* `wants to leave` *room_name.*"

- On failing to leave a room:

  "*visitor_name* `failed to leave` *room_name* `but will try again soon.`"

- Before trying to leave the room again:

  "*visitor_name* `is ready to try leaving` *room_name* `again.`"

- On successfully leaving a room:

  "*visitor_name* `has left` *room_name*`.`"

- On leaving the museum:

  "*visitor_name* `has left` *museum_name*`.`"

The method `checkImpatientOutput` in the provided `TestHelpers.kt` provides code that the automated tests use to validate the output produced by visitors. If in doubt, consult this file for precise details on the output that is expected.

Testing your solution

Before running the automated tests, it is worth manually confirming that your visitor threads can successfully navigate some museums.

Uncomment the code in supplied file `ImpatientVisitorsDemo.kt`. This file has a main function that launches a large number of visitor threads into the animal sanctuary. If things are working properly, the output of this main function should look something like this (but the output will change on each execution depending on the order in which threads interleave):

```
Minimilian has entered Animal sanctuary.
Jacub has entered Animal sanctuary.
Julia could not get into Animal sanctuary but will try again soon.
Susan has entered Animal sanctuary.
Neha has entered Animal sanctuary.
Maximilian has entered Animal sanctuary.
```

```
Donald has entered Animal sanctuary.
Yi has entered Animal sanctuary.
Donald has entered Entrance hall.
Xi could not get into Animal sanctuary but will try again soon.
Oscar has entered Animal sanctuary.
Maximilian has entered Entrance hall.
Amelia has entered Animal sanctuary.
Jaya has entered Animal sanctuary.
Noah has entered Animal sanctuary.
Noah has entered Entrance hall.
Teresa could not get into Animal sanctuary but will try again soon.
Poppy has entered Animal sanctuary.
Parminda could not get into Animal sanctuary but will try again soon.
Felix has entered Animal sanctuary.
Satnam has entered Animal sanctuary.
Satnam has entered Entrance hall.
Liz has entered Animal sanctuary.
Indy has entered Animal sanctuary.
Yi has entered Entrance hall.
Prakesh has entered Animal sanctuary.
...
Felix wants to leave Entrance hall.
Felix failed to leave Entrance hall but will try again soon.
Amelia is ready to try leaving Entrance hall again.
Amelia failed to leave Entrance hall but will try again soon.
Parminda is ready to try leaving Entrance hall again.
Parminda failed to leave Entrance hall but will try again soon.
Liz wants to leave Entrance hall.
...
```

It is unlikely your solution will work first time—you will likely have to spend some time debugging race conditions and deadlocks. The nondeterministic nature of concurrency makes debugging far more challenging than it is for sequential programs. If your solution is not working, consider temporarily adapting the main function to have a smaller number of visitors explore a simpler museum, as it will probably be easier to debug your solution in a simpler setting.

Once you believe this main function is working correctly, turn to the tests in `ImpatientVisitorsTest.kt`, which depend on `TestHelpers.kt`—you should uncomment the code in both of these files. These tests check that the output produced by each visitor navigating a museum is as it should be—i.e. that your visitors stick precisely to the above specification regarding output. Some of these tests may take a while to run. However, no individual test should take more than a minute or two to run. If you find that a test does not terminate, this probably means that there is a bug in your code that has caused your visitor threads to deadlock.

Extension

Condition variables

To implement this extension you will need to learn about *condition variables*. The `Lock` interface has a `newCondition()` method that returns a `Condition` object associated with the lock. In the concurrency literature such objects are called *condition variables*, or just *conditions* for short.

A condition provides the following three key methods, each of which should only be called when a thread holds the lock from which a condition was created:

- `await()`: This causes the thread to be put to sleep and added to a queue of *waiters* associated with the condition. When the thread is put to sleep, it releases the lock associated with the condition. A thread will remain asleep until it is *woken*. A thread gets woken when it receives a signal. This can either come accidentally from the operating system (in which case the thread is said to have been *spuriously* woken), or deliberately from another thread signalling via the condition by calling `signal` or `signalAll`, which are discussed next. When a thread is woken, it will acquire the lock associated with the condition again before returning from `await`.
- `signal()`: This signals one thread currently waiting on the condition (or is a no-op if no threads are waiting).
- `signalAll()`: This signals all threads currently waiting on the condition (so it is a no-op if no threads are waiting).

The `await` method on a condition also has an overload that takes parameters indicating the maximum amount of time for which the thread should wait. This is expressed via a `long` value, and a time unit (e.g. `TimeUnit.MILLISECONDS` for milliseconds). This overload of `await` has the same behaviour as described above, except that if a thread has not been woken after the specified period of time has elapsed, then the thread will be woken even though it has not been signalled.

Condition variables are normally used in the following way. Suppose an object has a `lock` property (of type `Lock`), and a `condition` property of type `Condition` such that `condition` was created from `lock`. Suppose that a thread cannot proceed until a particular boolean expression holds, and that the boolean expression refers to the state of a shared resource that is protected by the lock. The thread can handle this situation via the following pattern:

```
lock.withLock {
  // The thread now has exclusive access to the resource
  while (!propertyOfResource) {
    // The thread must wait for some other thread to do some work
    // that will lead to propertyOfResource holding. By calling
    // await(), the thread releases the lock so that other threads
    // can access the resource. Those other threads must signal the
    // condition when they are done. This will lead to the waiting
    // thread being woken up and re-acquiring the lock. It can then
    // check whether propertyOfResource now holds.

    condition.await()

    // At this point, the thread holds the lock again: it does not
    // return from await until it has managed to re-acquire the
    // lock. However, we cannot assume that propertyOfResource
    // holds. The thread might have been spuriously woken. If the
    // thread was signalled by some other thread, that thread might
    // not have made propertyOfResource true. Even if it did, it
    // may have woken up many threads that were waiting on the
    // condition, and a different thread may have accessed the
    // shared resource in a way that has changed
    // propertyOfResource. For this reason, condition.await() is
    // normally used in a loop.
  }
  // Use resource now that the required property holds.
  ...
}
```

A thread that updates the resource in a manner that might be of interest to waiting threads should signal those threads—either one of them or all of them, depending on the scenario. Here is what this looks like in the case where a single thread is signalled:

```
lock.withLock {
   // Modify the shared resource
   ...
   // Inform one thread waiting on the condition that the resource
   // has changed.
   condition.signal()
}
```

Patient visitors

Adapt your museum classes so that every museum room has a condition property, created from its lock property. This condition should be signalled each time a visitor leaves the room. You should decide when you implement the functionality below whether `signal` or `signalAll` is appropriate here.

Now add a new kind of visitor: a *patient visitor*. Unlike an impatient visitor, who keeps trying to enter the museum until they succeed, a *patient* visitor should wait patiently until it is informed that it can enter the museum. You should allow this by adding an analogue of the `enterIfPossible` method to museum, called e.g. `enterByWaiting`.

Similarly, when passing through a turnstile a patient visitor should wait until the next site becomes free.

First, try implementing a patient visitor that will wait however long it takes in order to pass to the next room. You should find that this does not work! Think of a situation where this can lead to deadlock.

As a compromise, a patient visitor should use the time-bounded version of `await` to wait for up to a given period of time when attempting to pass through a turnstile. If the patient visitor does not succeed, it should try again a few more times up to some maximum number of attempts. Only if these repeated attempts fail should the patient visitor consider selecting a different turnstile to pass through.

You should ensure that your visitor and museum code supports patient and impatient visitors visiting the museum simultaneously.

Use the tests in `ExtensionTest.kt` to help guide you.

Appendix

An example of what a session with the interactive explorer might look like.

```
Which museum would you like to explore?
  Art gallery, Animal sanctuary
Art gallery
Welcome to Art gallery! Let's explore.
You are in Entrance hall
Have a good look around. Bored yet? Where do you want to go?
From here, you can go to:
  Exhibition room
Exhibition room
You are in Exhibition room
Have a good look around. Bored yet? Where do you want to go?
```

```
From here, you can go to:
  Outside
Outside
We hope you had a good time in the Art gallery museum - goodbye!!
Which museum would you like to explore?
  Art gallery, Animal sanctuary
Animal sanctuary
Welcome to Animal sanctuary! Let's explore.
You are in Entrance hall
Have a good look around. Bored yet? Where do you want to go?
From here, you can go to:
  Bats
Bat
I'm sorry, but that's not one of the next places you can go. Let's try again.
You are in Entrance hall
Have a good look around. Bored yet? Where do you want to go?
From here, you can go to:
  Bats
Bats
You are in Bats
Have a good look around. Bored yet? Where do you want to go?
From here, you can go to:
  Lizards
Lizards
You are in Lizards
Have a good look around. Bored yet? Where do you want to go?
From here, you can go to:
  Insects
  Gift shop
Insects
You are in Insects
Have a good look around. Bored yet? Where do you want to go?
From here, you can go to:
  Snakes
  Gift shop
Gift shop
You are in Gift shop
Have a good look around. Bored yet? Where do you want to go?
From here, you can go to:
  Outside
Outside
We hope you had a good time in the Animal sanctuary museum - goodbye!!
Which museum would you like to explore?
  Art gallery, Animal sanctuary
^D
You have had enough of this game - what is wrong with you? Goodbye.
```

Note: your repository containing the skeleton for this lab can be found at *`https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinmusem_username`*. ***As always, you should use LabTS to test and submit your code.***

Assessment

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```