```
[]
```

```kotlin
package museumvisit

fun createArtGallery(): Museum {
    val entrance = MuseumRoom("Entrance hall", 20)
    val result = Museum("Art gallery", entrance)
    val exhibitionRoom = MuseumRoom("Exhibition room", 10)
    result.addRoom(exhibitionRoom)
    result.connectRoomTo(entrance, exhibitionRoom)
    result.connectRoomToExit(exhibitionRoom)
    return result
}

fun createAnimalSanctuary(): Museum {
    val entrance = MuseumRoom("Entrance hall", 20)
    val bats = MuseumRoom("Bats", 10)
    val lizards = MuseumRoom("Lizards", 10)
    val insects = MuseumRoom("Insects", 10)
    val giftShop = MuseumRoom("Gift shop", 10)
    val snakes = MuseumRoom("Snakes", 10)
    val result = Museum("Animal sanctuary", entrance)
    result.addRoom(bats)
    result.addRoom(lizards)
    result.addRoom(insects)
    result.addRoom(giftShop)
    result.addRoom(snakes)
    result.connectRoomTo(entrance, bats)
    result.connectRoomTo(bats, lizards)
    result.connectRoomTo(lizards, insects)
    result.connectRoomTo(insects, snakes)
    result.connectRoomTo(snakes, entrance)
    result.connectRoomTo(lizards, giftShop)
    result.connectRoomTo(insects, giftShop)
    result.connectRoomToExit(giftShop)
    return result
}

fun createAnimalSanctuaryWithUnreachableRooms(): Museum {
    val entrance = MuseumRoom("Entrance hall", 20)
    val bats = MuseumRoom("Bats", 10)
    val lizards = MuseumRoom("Lizards", 10)
    val insects = MuseumRoom("Insects", 10)
    val giftShop = MuseumRoom("Gift shop", 10)
    val snakes = MuseumRoom("Snakes", 10)
    val result = Museum("Animal sanctuary", entrance)
    result.addRoom(bats)
    result.addRoom(lizards)
    result.addRoom(insects)
    result.addRoom(giftShop)
    result.addRoom(snakes)
    result.connectRoomTo(bats, lizards)
    result.connectRoomTo(lizards, insects)
    result.connectRoomTo(insects, snakes)
    result.connectRoomTo(snakes, entrance)
    result.connectRoomTo(lizards, giftShop)
    result.connectRoomTo(insects, giftShop)
    result.connectRoomToExit(giftShop)
    return result
}

fun createAnimalSanctuaryWithRoomsThatDoNotLeadToExit(): Museum {
    val entrance = MuseumRoom("Entrance hall", 20)
    val bats = MuseumRoom("Bats", 10)
    val lizards = MuseumRoom("Lizards", 10)
    val insects = MuseumRoom("Insects", 10)
    val giftShop = MuseumRoom("Gift shop", 10)
    val snakes = MuseumRoom("Snakes", 10)
    val result = Museum("Animal sanctuary", entrance)
    result.addRoom(bats)
    result.addRoom(lizards)
    result.addRoom(insects)
    result.addRoom(giftShop)
    result.addRoom(snakes)
    result.connectRoomTo(entrance, bats)
    result.connectRoomTo(bats, lizards)
    result.connectRoomTo(lizards, insects)
    result.connectRoomTo(insects, snakes)
    result.connectRoomTo(lizards, giftShop)
    result.connectRoomToExit(giftShop)
    return result
}
```

```kotlin
package museumvisit

private fun getRoomNames(rooms: Set<MuseumRoom>): String =
    rooms.map { it.name }
        .sorted().joinToString()

class UnreachableRoomsException(private val unreachable: Set<MuseumRoom>) : Exception() {
    override fun toString(): String = "Unreachable rooms: " + getRoomNames(unreachable)
}

class CannotExitMuseumException(private val roomsThatDoNotLeadToExit: Set<MuseumRoom>) : Exception() {
    override fun toString(): String = "Impossible to leave museum from: " + getRoomNames(roomsThatDoNotLeadToExit)
}
```

```kotlin
package museumvisit

fun exploreMuseum(museum: Museum) {
    println("Welcome to ${museum.name}! Let’s explore.")
    var currentSite: MuseumSite = museum.enterIfPossible()!!
    currentSite.enter()
    while (currentSite is MuseumRoom) {
        println("You are in ${currentSite.name}")
        println("Have a good look around. Bored yet? Where do you want to go?")
        println("From here, you can go to:")
        val nextRoomNames = currentSite.exitTurnstiles.map { it ->
it.destination.name }
        for (name in nextRoomNames) {
            println("  $name")
        }
        val choice = readlnOrNull()
        if (choice == null) {
            println("You have had enough of this museum - taking you back to the
main menu.")
            return
        } else {
            if (choice in nextRoomNames) {
                currentSite = currentSite.exitTurnstiles.map { it.destination
}.filter { it.name == choice }[0]
            } else {
                println("I’m sorry, but that’s not one of the next places you can
go. Let’s try again.")
            }
        }
    }
    assert(currentSite == museum.outside)
    println("We hope you had a good time in the ${museum.name} museum - goodbye!!")
}

val museumsToExplore = listOf(createArtGallery(), createAnimalSanctuary())

fun main() {
    while (true) {
        println("Which museum would you like to explore?")
        println("  " + museumsToExplore.map { it.name }.joinToString())
        val choice = readlnOrNull()
        if (choice == null) {
            println("You have had enough of this game - what is wrong with you?
Goodbye.")
            return
        }
        val maybeMuseum = museumsToExplore.find { it.name == choice }
        if (maybeMuseum != null) {
            exploreMuseum(maybeMuseum)
        } else {
            println("I don’t know that museum, sorry.")
        }
    }
}
```

```kotlin
package museumvisit

import java.io.PrintStream

class ImpatientVisitor(name: String, printStream: PrintStream, museum: Museum) :
Visitor(name, printStream, museum) {
    override fun enterMuseum(): MuseumRoom {
        while (true) {
            val maybeResult: MuseumRoom? = museum.enterIfPossible()
            if (maybeResult != null) {
                return maybeResult
            }
            printStream.println("$name could not get into ${museum.name} but will
try again soon.")
            waitSomeTimeBeforeRetrying()
            printStream.println("$name is ready to try again.")
        }
    }

    override fun leaveRoom(currentRoom: MuseumRoom): MuseumSite {
        while (true) {
            val turnstile = currentRoom.exitTurnstiles.random()
            val maybeNextSite = turnstile.passToNextSiteIfPossible()
            if (maybeNextSite != null) {
                return maybeNextSite
            }
            printStream.println("$name failed to leave ${currentRoom.name} but will
try again soon.")
            waitSomeTimeBeforeRetrying()
            printStream.println("$name is ready to try leaving ${currentRoom.name}
again.")
        }
    }

    private fun waitSomeTimeBeforeRetrying() {
        Thread.sleep(10)
    }
}
```

```kotlin
package museumvisit

fun main() {
    val visitorNames = listOf(
        "Neha",
        "Alex",
        "Yi",
        "Jianyi",
        "Felix",
        "Oscar",
        "Amelia",
        "Noah",
        "Prakesh",
        "Satnam",
        "Susan",
        "Poppy",
        "Jaya",
        "Indy",
        "Lula",
        "Maximilian",
        "Minimilian",
        "Jacub",
        "Donald",
        "Liz",
        "Teresa",
        "Julia",
        "Parminda",
        "Xi",
    )
    val museum = createAnimalSanctuary()
    val visitors = visitorNames.mapIndexed { index, person ->
        Thread(ImpatientVisitor(person, System.out, museum))
    }
    visitors.forEach { it.start() }
    visitors.forEach { it.join() }
}
```

```kotlin
package museumvisit

import kotlin.IllegalArgumentException
import kotlin.concurrent.withLock

class Museum(val name: String, private val entrance: MuseumRoom) {

    var admitted: Int = 0
        private set

    val outside: OutsideMuseum = OutsideMuseum()

    private val rooms: MutableSet<MuseumRoom> = mutableSetOf(entrance)

    fun addRoom(room: MuseumRoom) {
        if (room in rooms) {
            throw IllegalArgumentException("A room should not be added to the museum multiple times.")
        }
        rooms.add(room)
    }

    fun connectRoomTo(startRoom: MuseumRoom, destinationRoom: MuseumRoom) {
        if (startRoom === destinationRoom) {
            throw IllegalArgumentException("Cannot connect a room to itself.")
        }
        if (startRoom !in rooms || destinationRoom !in rooms) {
            throw IllegalArgumentException("To add a connection between rooms, they must both be rooms of the museum.")
        }
        if (destinationRoom in startRoom.exitTurnstiles.map { it.destination }) {
            throw IllegalArgumentException("A turnstile from ${startRoom.name} to ${destinationRoom.name} already exists.")
        }
        startRoom.addExitTurnstile(destinationRoom)
    }

    fun connectRoomToExit(room: MuseumRoom) {
        if (room !in rooms) {
            throw IllegalArgumentException("To connect a room to the exit, the room must be part of the museum.")
        }
        if (outside in room.exitTurnstiles.map { it.destination }) {
            throw IllegalArgumentException("An exit turnstile from ${room.name} to already exists.")
        }
        room.addExitTurnstile(outside)
    }

    fun enterIfPossible(): MuseumRoom? {
        entrance.lock.withLock {
            if (entrance.hasCapacity()) {
                entrance.enter()
                admitted++
                return entrance
            }
            return null
        }
    }

    fun entranceHasCapacity(): Boolean = entrance.hasCapacity()

    fun enter() {
        if (!entrance.hasCapacity()) {
            throw UnsupportedOperationException("The museum entrance is full.")
        }
        admitted++
        entrance.enter()
    }

    // This is part of the extension
    fun enterByWaiting(): MuseumRoom {
        while (true) {
            entrance.lock.withLock {
                while (!entranceHasCapacity()) {
                    entrance.hasCapacityCondition.await()
                }
                enter()
                return entrance
            }
        }
    }

    fun checkWellFormed() {
        val reachableFrom = mutableMapOf<MuseumRoom, MutableSet<MuseumSite>>()
        for (room in rooms) {
            reachableFrom[room] = mutableSetOf(room)
        }
        var changed = true
        while (changed) {
            changed = false
            for (room in rooms) {
                for (turnstile in room.exitTurnstiles) {
                    changed = if (turnstile.destination is OutsideMuseum) {
                        changed or reachableFrom[room]!!.add(turnstile.destination)
                    } else {
                        changed or reachableFrom[room]!!.addAll(reachableFrom[turnstile.destination as MuseumRoom]!!)
                    }
                }
            }
        }

        val unreachableRooms: Set<MuseumRoom> = rooms - reachableFrom[entrance]!!.filterIsInstance<MuseumRoom>()
        if (unreachableRooms.isNotEmpty()) {
            throw UnreachableRoomsException(unreachableRooms)
        }
        val roomsThatDoNotReachExit: Set<MuseumRoom> =
            rooms.filter { outside !in reachableFrom[it]!! }.toSet()
        if (roomsThatDoNotReachExit.isNotEmpty()) {
            throw CannotExitMuseumException(roomsThatDoNotReachExit)
        }
    }

    override fun toString(): String {
        val result = StringBuilder()
        result.append("$name\n")
        val alreadyConsidered = mutableSetOf(entrance)
        val roomsToProcess = mutableListOf(entrance)
        while (roomsToProcess.isNotEmpty()) {
            val room = roomsToProcess.removeAt(0)
            result.append("${room.name} leads to: ")
            result.append(room.exitTurnstiles.map { it.destination.name }.joinToString())
            result.append("\n")
            for (nextRoom in room.exitTurnstiles.map { it.destination }.filterIsInstance<MuseumRoom>()) {
                if (nextRoom !in alreadyConsidered) {
                    alreadyConsidered.add(nextRoom)
                    roomsToProcess.add(nextRoom)
                }
            }
        }
        return result.toString()
    }
```

```kotlin
}
```

```kotlin
package museumvisit

class MuseumRoom(override val name: String, private val capacity: Int) :
MuseumSite() {

    init {
        if (capacity < 1) {
            throw IllegalArgumentException("A room must capacity for at least one
visitor.")
        }
    }

    val exitTurnstiles: List<Turnstile>
        get() = turnstileList

    private val turnstileList: MutableList<Turnstile> = mutableListOf()

    override fun hasCapacity(): Boolean = occupancy < capacity

    override fun enter() {
        if (!hasCapacity()) {
            throw UnsupportedOperationException("Room should never exceed its
capacity.")
        }
        occupancy++
    }

    fun exit() {
        if (occupancy == 0) {
            throw UnsupportedOperationException("Room should never go below
capacity 0.")
        }
        occupancy--
    }

    fun addExitTurnstile(destination: MuseumSite) {
        turnstileList.add(Turnstile(this, destination))
    }
}
```

```kotlin
package museumvisit

import java.util.concurrent.locks.Condition
import java.util.concurrent.locks.Lock
import java.util.concurrent.locks.ReentrantLock

abstract class MuseumSite {
    val lock: Lock = ReentrantLock()
    val hasCapacityCondition: Condition = lock.newCondition()

    abstract val name: String
    var occupancy: Int = 0
        protected set

    abstract fun hasCapacity(): Boolean

    abstract fun enter()
}
```

```kotlin
package museumvisit

class OutsideMuseum : MuseumSite() {

    override val name: String = "Outside"

    override fun hasCapacity(): Boolean = true

    override fun enter() {
        occupancy++
    }
}
```

```kotlin
package museumvisit

import java.io.PrintStream

class PatientVisitor(name: String, printStream: PrintStream, museum: Museum) : Visitor(name, printStream, museum) {
    override fun enterMuseum(): MuseumRoom = museum.enterByWaiting()

    override fun leaveRoom(currentRoom: MuseumRoom): MuseumSite {
        while (true) {
            val turnstile = currentRoom.exitTurnstiles.random()
            val maybeNextSite = turnstile.passToNextSiteWithPatienceAndPersistence(5, 100)
            if (maybeNextSite != null) {
                return maybeNextSite
            }
        }
    }
}
```

```kotlin
package museumvisit

import java.util.concurrent.TimeUnit
import kotlin.concurrent.withLock

class Turnstile(val origin: MuseumRoom, val destination: MuseumSite) {

    fun passToNextSiteIfPossible(): MuseumSite? {
        // The following condition establishes a total order relation over the MuseumSites: assuming the names of the
        // rooms are unique, it is possible with the following condition to establish an order such that rooms with
        // larger names (in alphabetical order) are always locked first. Any ordering can be used (including comparing
        // the hashcodes of the MuseumSite instances) as long as it is total, i.e., can order any possible pair of
        // MuseumSites. Establishing an order is critical to avoid circular dependencies (i.e., deadlocks).
        val (firstLock, secondLock) = if (destination.name.compareTo(origin.name) >= 0) {
            Pair(destination.lock, origin.lock)
        } else {
            Pair(origin.lock, destination.lock)
        }
        firstLock.withLock {
            secondLock.withLock {
                if (destination.hasCapacity()) {
                    origin.exit()
                    destination.enter()
                    origin.hasCapacityCondition.signalAll()
                    return destination
                }
                return null
            }
        }
    }

    // This is part of the extension.
    fun passToNextSiteWithPatienceAndPersistence(
        attempts: Int,
        timeoutPerAttempt: Long,
    ): MuseumSite? {
        for (attempt in 1..attempts) {
            val maybeNextSite = passToNextSiteIfPossible()
            if (maybeNextSite != null) {
                return maybeNextSite
            }
            destination.lock.withLock {
                if (!destination.hasCapacity()) {
                    destination.hasCapacityCondition.await(timeoutPerAttempt, TimeUnit.MILLISECONDS)
                }
            }
        }
        return null
    }
}
```

```kotlin
package museumvisit

import java.io.PrintStream

abstract class Visitor(protected val name: String, protected val printStream: PrintStream, protected val museum: Museum) : Runnable {

    abstract fun enterMuseum(): MuseumRoom

    abstract fun leaveRoom(currentRoom: MuseumRoom): MuseumSite

    override fun run() {
        var currentSite: MuseumSite = enterMuseum()
        printStream.println("$name has entered ${museum.name}.")
        while (currentSite != museum.outside) {
            assert(currentSite is MuseumRoom)
            printStream.println("$this has entered ${currentSite.name}.")
            val randomVisitTimeInMillis = (Math.random() * 200).toInt() + 1
            Thread.sleep(randomVisitTimeInMillis.toLong())
            printStream.println("$this wants to leave ${currentSite.name}.")
            val newSite = leaveRoom(currentSite as MuseumRoom)
            printStream.println("$this has left ${currentSite.name}.")
            currentSite = newSite
        }
        printStream.println("$this has left ${museum.name}.")
    }

    override fun toString(): String {
        return name
    }
}
```

```kotlin
package museumvisit

import org.junit.Test
import java.io.ByteArrayOutputStream
import java.io.PrintStream
import kotlin.test.assertEquals

class ExtensionTest {

    private fun testMuseumVisit(museumUnderTest: MuseumUnderTest, people: List<String>, allPatient: Boolean) {
        val byteOutputStreams = people.map {
            ByteArrayOutputStream()
        }
        val printStreams = byteOutputStreams.map {
            PrintStream(it)
        }
        assertEquals(0, museumUnderTest.museum.admitted)
        assertEquals(0, museumUnderTest.museum.outside.occupancy)
        for (room in museumUnderTest.rooms) {
            assertEquals(0, room.occupancy)
        }
        val visitors = people.mapIndexed { index, person ->
            Thread(
                if (allPatient || (index % 2 == 0)) {
                    PatientVisitor(person, printStreams[index], museumUnderTest.museum)
                } else {
                    ImpatientVisitor(person, printStreams[index], museumUnderTest.museum)
                },
            )
        }
        visitors.forEach { it.start() }
        visitors.forEach { it.join() }
        assertEquals(people.size, museumUnderTest.museum.admitted)
        assertEquals(people.size, museumUnderTest.museum.outside.occupancy)
        for (room in museumUnderTest.rooms) {
            assertEquals(0, room.occupancy)
        }
        byteOutputStreams.forEachIndexed { index, byteArrayOutputStream ->
            if (allPatient || (index % 2 == 0)) {
                checkPatientOutput(people[index], byteArrayOutputStream.toString(), museumUnderTest)
            } else {
                checkImpatientOutput(people[index], byteArrayOutputStream.toString(), museumUnderTest)
            }
        }
    }

    @Test
    fun ‘two patient visitors to small museum‘() {
        testMuseumVisit(createSmallMuseumUnderTest(), listOf("Ally", "Chris"), true)
    }

    @Test
    fun ‘one patient and one impatient visitor to small museum‘() {
        testMuseumVisit(createSmallMuseumUnderTest(), listOf("Ally", "Chris"), false)
    }

    @Test
    fun ‘many patient visitors to small museum‘() {
        testMuseumVisit(createSmallMuseumUnderTest(), lotsOfPeople, true)
    }

    @Test
```

```kotlin
    fun ‘many mixed-patience visitors to small museum‘() {
        testMuseumVisit(createSmallMuseumUnderTest(), lotsOfPeople, false)
    }

    @Test
    fun ‘two patient visitors to aquarium‘() {
        testMuseumVisit(createAquariumMuseumUnderTest(), listOf("Ally", "Chris"), true)
    }

    @Test
    fun ‘one patient and one impatient visitor to aquarium‘() {
        testMuseumVisit(createAquariumMuseumUnderTest(), listOf("Ally", "Chris"), false)
    }

    @Test
    fun ‘many patient visitors to aquarium‘() {
        testMuseumVisit(createAquariumMuseumUnderTest(), lotsOfPeople, true)
    }

    @Test
    fun ‘many mixed-patience visitors to aquarium‘() {
        testMuseumVisit(createAquariumMuseumUnderTest(), lotsOfPeople, false)
    }
}
```

```kotlin
package museumvisit

import org.junit.Test
import java.io.ByteArrayOutputStream
import java.io.PrintStream
import kotlin.test.assertEquals

class ImpatientVisitorsTest {

    private fun testMuseumVisit(museumUnderTest: MuseumUnderTest, people:
List<String>) {
        val byteOutputStreams = people.map {
            ByteArrayOutputStream()
        }
        val printStreams = byteOutputStreams.map {
            PrintStream(it)
        }
        assertEquals(0, museumUnderTest.museum.admitted)
        assertEquals(0, museumUnderTest.museum.outside.occupancy)
        for (room in museumUnderTest.rooms) {
            assertEquals(0, room.occupancy)
        }
        val visitors = people.mapIndexed { index, person ->
            Thread(ImpatientVisitor(person, printStreams[index],
museumUnderTest.museum))
        }
        visitors.forEach { it.start() }
        visitors.forEach { it.join() }
        assertEquals(people.size, museumUnderTest.museum.admitted)
        assertEquals(people.size, museumUnderTest.museum.outside.occupancy)
        for (room in museumUnderTest.rooms) {
            assertEquals(0, room.occupancy)
        }
        byteOutputStreams.forEachIndexed { index, byteArrayOutputStream ->
            checkImpatientOutput(people[index], byteArrayOutputStream.toString(),
museumUnderTest)
        }
    }

    @Test
    fun ‘two visitors to small museum‘() {
        testMuseumVisit(createSmallMuseumUnderTest(), listOf("Ally", "Chris"))
    }

    @Test
    fun ‘many visitors to small museum‘() {
        testMuseumVisit(createSmallMuseumUnderTest(), lotsOfPeople)
    }

    @Test
    fun ‘two visitors to aquarium‘() {
        testMuseumVisit(createAquariumMuseumUnderTest(), listOf("Ally", "Chris"))
    }

    @Test
    fun ‘many visitors to aquarium‘() {
        testMuseumVisit(createAquariumMuseumUnderTest(), lotsOfPeople)
    }
}
```

```kotlin
package museumvisit

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.assertFalse
import kotlin.test.assertTrue
import kotlin.test.fail

class MuseumRoomTest {

    @Test
    fun ‘test name‘() {
        assertEquals("Dali paintings", MuseumRoom("Dali paintings", 10).name)
    }

    @Test
    fun ‘test capacity positive‘() {
        try {
            MuseumRoom("Dali paintings", 0)
            fail("An IllegalArgumentException should have been thrown.")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
        try {
            MuseumRoom("Dali paintings", -100)
            fail("An IllegalArgumentException should have been thrown.")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
    }

    @Test
    fun ‘successful enter and exit‘() {
        val room = MuseumRoom("Sharks", 5)
        assertTrue(room.hasCapacity())
        room.enter()
        assertTrue(room.hasCapacity())
        room.enter()
        assertTrue(room.hasCapacity())
        room.enter()
        assertTrue(room.hasCapacity())
        room.enter()
        assertTrue(room.hasCapacity())
        room.enter()
        assertFalse(room.hasCapacity())
        room.exit()
        assertTrue(room.hasCapacity())
        room.exit()
        assertTrue(room.hasCapacity())
        room.exit()
        assertTrue(room.hasCapacity())
        room.exit()
        assertTrue(room.hasCapacity())
        room.exit()
        assertTrue(room.hasCapacity())
    }

    @Test
    fun ‘exception on enter if room gets full‘() {
        val room = MuseumRoom("Sharks", 3)
        assertTrue(room.hasCapacity())
        room.enter()
        assertTrue(room.hasCapacity())
        room.enter()
        assertTrue(room.hasCapacity())
        room.enter()
        assertFalse(room.hasCapacity())
        try {
            room.enter()
            fail("An UnsupportedOperationException should have been thrown.")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception expected
        }
    }

    @Test
    fun ‘exception on exit if room is emtpy‘() {
        val room = MuseumRoom("Sharks", 3)
        assertTrue(room.hasCapacity())
        try {
            room.exit()
            fail("An UnsupportedOperationException should have been thrown.")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception expected
        }
    }
}
```

```
package museumvisit

import org.junit.Test
import kotlin.test.assertEquals
import kotlin.test.assertFalse
import kotlin.test.assertTrue
import kotlin.test.fail

class MuseumTest {

    @Test
    fun ‘test toString art gallery‘() {
        assertEquals(
            """
            Art gallery
            Entrance hall leads to: Exhibition room
            Exhibition room leads to: Outside

            """.trimIndent(),
            createArtGallery().toString(),
        )
    }

    @Test
    fun ‘test toString animal sanctuary‘() {
        assertEquals(
            """
            Animal sanctuary
            Entrance hall leads to: Bats
            Bats leads to: Lizards
            Lizards leads to: Insects, Gift shop
            Insects leads to: Snakes, Gift shop
            Gift shop leads to: Outside
            Snakes leads to: Entrance hall

            """.trimIndent(),
            createAnimalSanctuary().toString(),
        )
    }

    @Test
    fun ‘test well formed art gallery‘() {
        createArtGallery().checkWellFormed()
    }

    @Test
    fun ‘test well formed animal santuary‘() {
        createAnimalSanctuary().checkWellFormed()
    }

    @Test
    fun ‘test animal santuary with unreachable rooms‘() {
        try {
            createAnimalSanctuaryWithUnreachableRooms().checkWellFormed()
            fail("An UnreachableRoomException should have been thrown")
        } catch (exception: UnreachableRoomsException) {
            assertEquals(
                """
                Unreachable rooms: Bats, Gift shop, Insects, Lizards, Snakes
                """.trimIndent(),
                exception.toString(),
            )
        }
    }

    @Test
    fun ‘test animal santuary with rooms that do not lead to exit‘() {
        try {
```

```kotlin
            createAnimalSanctuaryWithRoomsThatDoNotLeadToExit().checkWellFormed()
            fail("An CannotExitMuseumException should have been thrown")
        } catch (exception: CannotExitMuseumException) {
            assertEquals(
                """
                Impossible to leave museum from: Insects, Snakes
                """.trimIndent(),
                exception.toString(),
            )
        }
    }

    @Test
    fun ‘cannot connect unknown room to exit‘() {
        val museum = Museum("Some museum", MuseumRoom("Entrance", 5))
        try {
            museum.connectRoomToExit(MuseumRoom("Some room", 3))
            fail("Expected IllegalArgumentException to be thrown")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
    }

    @Test
    fun ‘cannot connect rooms if first is unknown‘() {
        val entrance = MuseumRoom("Entrance", 5)
        val museum = Museum("Some museum", entrance)
        try {
            museum.connectRoomTo(MuseumRoom("Some room", 3), entrance)
            fail("Expected IllegalArgumentException to be thrown")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
    }

    @Test
    fun ‘cannot connect rooms if second is unknown‘() {
        val entrance = MuseumRoom("Entrance", 5)
        val museum = Museum("Some museum", entrance)
        try {
            museum.connectRoomTo(entrance, MuseumRoom("Some room", 3))
            fail("Expected IllegalArgumentException to be thrown")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
    }

    @Test
    fun ‘cannot add room multiple times‘() {
        val entrance = MuseumRoom("Entrance", 5)
        val museum = Museum("Some museum", entrance)
        try {
            museum.addRoom(entrance)
            fail("Expected IllegalArgumentException to be thrown")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
    }

    fun ‘cannot add room with same name‘() {
        val entrance = MuseumRoom("Entrance", 5)
        val museum = Museum("Some museum", entrance)
        try {
            museum.addRoom(MuseumRoom("Entrance", 6))
            fail("Expected IllegalArgumentException to be thrown")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
```

```kotlin
    }

    @Test
    fun ‘cannot connect room to same room multiple times‘() {
        val entrance = MuseumRoom("Entrance", 5)
        val exhibitionRoom = MuseumRoom("Exhibition room", 3)
        val museum = Museum("Some museum", entrance)
        museum.addRoom(exhibitionRoom)
        museum.connectRoomTo(entrance, exhibitionRoom)
        try {
            museum.connectRoomTo(entrance, exhibitionRoom)
            fail("Expected IllegalArgumentException to be thrown")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
    }

    @Test
    fun ‘cannot connect room to exit multiple times‘() {
        val entrance = MuseumRoom("Entrance", 5)
        val museum = Museum("Some museum", entrance)
        museum.connectRoomToExit(entrance)
        try {
            museum.connectRoomToExit(entrance)
            fail("Expected IllegalArgumentException to be thrown")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
    }

    @Test
    fun ‘cannot connect room to self‘() {
        val entrance = MuseumRoom("Entrance", 5)
        val museum = Museum("Some museum", entrance)
        try {
            museum.connectRoomTo(entrance, entrance)
            fail("Expected IllegalArgumentException to be thrown")
        } catch (exception: IllegalArgumentException) {
            // Good: exception expected
        }
    }

    @Test
    fun ‘test museum has capacity‘() {
        val museum = createArtGallery()
        for (i in 0..<20) {
            assertTrue(museum.entranceHasCapacity())
            assertEquals(i, museum.admitted)
            museum.enter()
        }
        assertFalse(museum.entranceHasCapacity())
        try {
            museum.enter()
            fail("An UnsupportedOperationException should have been thrown.")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception expected
        }
    }
}
```

```kotlin
package museumvisit

import kotlin.test.assertEquals
import kotlin.test.assertTrue
import kotlin.test.fail

class MuseumUnderTest(val museum: Museum, val entrance: MuseumRoom, val rooms: List<MuseumRoom>)

fun createSmallMuseumUnderTest(): MuseumUnderTest {
    val entrance = MuseumRoom("Entrance", 1)
    val exhibition1 = MuseumRoom("Room 1", 1)
    val exhibition2 = MuseumRoom("Room 1", 1)

    val rooms = listOf(entrance, exhibition1, exhibition2)

    val museum = Museum("Small museum", entrance)
    museum.addRoom(exhibition1)
    museum.addRoom(exhibition2)
    museum.connectRoomTo(entrance, exhibition1)
    museum.connectRoomTo(exhibition1, entrance)
    museum.connectRoomTo(exhibition1, exhibition2)
    museum.connectRoomTo(exhibition2, exhibition1)
    museum.connectRoomToExit(exhibition2)
    museum.checkWellFormed()

    return MuseumUnderTest(museum, entrance, rooms)
}

fun createAquariumMuseumUnderTest(): MuseumUnderTest {
    val entrance = MuseumRoom("Aquarium entrance", 20)
    val crustaceans = MuseumRoom("Crabs and lobsters", 4)
    val sharks = MuseumRoom("Sharks", 4)
    val rays = MuseumRoom("Rays", 6)
    val seahorses = MuseumRoom("Seahorses", 3)
    val smallFish = MuseumRoom("Small fish", 9)
    val bobbits = MuseumRoom("Bobbit worms", 1)

    val rooms = listOf(crustaceans, sharks, rays, seahorses, smallFish, bobbits, entrance)

    val museum = Museum("Ally’s Grand Aquarium", entrance)
    museum.addRoom(crustaceans)
    museum.addRoom(sharks)
    museum.addRoom(rays)
    museum.addRoom(seahorses)
    museum.addRoom(smallFish)
    museum.addRoom(bobbits)
    museum.connectRoomTo(entrance, crustaceans)
    museum.connectRoomTo(crustaceans, sharks)
    museum.connectRoomTo(sharks, rays)
    museum.connectRoomTo(rays, seahorses)
    museum.connectRoomTo(seahorses, smallFish)
    museum.connectRoomTo(smallFish, bobbits)
    museum.connectRoomTo(bobbits, entrance)
    museum.connectRoomTo(sharks, smallFish)
    museum.connectRoomTo(smallFish, sharks)
    museum.connectRoomToExit(entrance)
    museum.connectRoomToExit(rays)
    museum.checkWellFormed()

    return MuseumUnderTest(museum, entrance, rooms)
}

val lotsOfPeople = listOf(
    "Neha",
    "Alex",
    "Yi",
```

```kotlin
    "Jianyi",
    "Felix",
    "Oscar",
    "Amelia",
    "Noah",
    "Prakesh",
    "Satnam",
    "Susan",
    "Poppy",
    "Jaya",
    "Indy",
    "Lula",
    "Maximilian",
    "Minimilian",
    "Jacub",
    "Donald",
    "Liz",
    "Teresa",
    "Julia",
    "Parminda",
    "Xi",
)

fun checkImpatientOutput(person: String, output: String, museumUnderTest: MuseumUnderTest) {
    val lines = output.split("\n")
    var index = 0
    while (lines[index] != "$person has entered ${museumUnderTest.museum.name}.") {
        assertEquals("$person could not get into ${museumUnderTest.museum.name} but will try again soon.", lines[index])
        index++
        assertEquals("$person is ready to try again.", lines[index])
        index++
    }
    index++
    while (index < lines.size) {
        val personEnteredRegex = """$person has entered ([a-zA-Z0-9 ]+)\.""".toRegex()
        val personEnteredRegexMatchResult = personEnteredRegex.find(lines[index])!!
        val (roomName) = personEnteredRegexMatchResult.destructured
        assertTrue(roomName in museumUnderTest.rooms.map { it.name }, "Unknown room name $roomName")
        assertTrue(index < lines.size - 1)
        index++
        assertEquals("$person wants to leave $roomName.", lines[index])
        assertTrue(index < lines.size - 1)
        index++
        while ("$person has left $roomName." != lines[index]) {
            assertEquals("$person failed to leave $roomName but will try again soon.", lines[index])
            assertTrue(index < lines.size - 1)
            index++
            assertEquals("$person is ready to try leaving $roomName again.", lines[index])
            assertTrue(index < lines.size - 1)
            index++
        }
        assertTrue(index < lines.size - 1)
        index++
        if (lines[index] == "$person has left ${museumUnderTest.museum.name}.") {
            assertEquals(lines.size - 2, index)
            assertEquals("", lines[lines.size - 1])
            return
        }
    }
    fail("Expected to see $person leaving the museum.")
}

```

```kotlin
fun checkPatientOutput(person: String, output: String, museumUnderTest: MuseumUnderTest) {
    val lines = output.split("\n")
    var index = 0
    assertEquals("$person has entered ${museumUnderTest.museum.name}.", lines[index])
    assertTrue(index < lines.size - 1)
    index++
    while (index < lines.size) {
        val personEnteredRegex = """$person has entered ([a-zA-Z0-9 ]+)\.""".toRegex()
        val personEnteredRegexMatchResult = personEnteredRegex.find(lines[index])!!
        val (roomName) = personEnteredRegexMatchResult.destructured
        assertTrue(roomName in museumUnderTest.rooms.map { it.name }, "Unknown room name $roomName")
        assertTrue(index < lines.size - 1)
        index++
        assertEquals("$person wants to leave $roomName.", lines[index])
        assertTrue(index < lines.size - 1)
        index++
        assertEquals("$person has left $roomName.", lines[index])
        assertTrue(index < lines.size - 1)
        index++
        if (lines[index] == "$person has left ${museumUnderTest.museum.name}.") {
            assertEquals(lines.size - 2, index)
            assertEquals("", lines[lines.size - 1])
            return
        }
    }
    fail("Expected to see $person leaving the museum.")
}
```