# About the Player movements

There is no spoon!
What if the GUI doesn't check for connecting rooms?

# Sprint 1 design decision re: the Player's moving

In sprint 1, we decided that the Player should have a behavior:

```
void move(Room.Direction) { ... }
```

It was up to the GUI to first check if a direction was legal, and disable the navigation buttons for the invalid directions.

# Sprint 1 design decision re: the Player's moving

How did the GUI check whether to enable or disable a navigation button?

We knew that

- A Room had four connecting rooms
- A connecting room with the value null signified "No connection there"
- E.g. if currentRoom.getConnectingRoom(NORTH) was null, there was no exit to the North

# Can we trust the GUI designers?

What if a sloppy GUI designer doesn't care to check all the connecting rooms from the Player's current room? What would happen then?

The GUI would pass null to the Player's `go(Room.Direction)` method!

The Player's method `go(Room.Direction)` would detect this and throw an `IllegalArgumentException`

# Exceptions - recap

There are two main types of Exception classes in Java:

- Those who inherit from RuntimeException or any of its descendents (unchecked!)
- Those who don’t and merely inherit from Exception (or any of its descendents but not via RuntimeException) (Checked exceptions!)

Do you remember the implications for handling the two different types of Exception?

# Exceptions - recap - Runtime exceptions

Runtime exceptions are said to be *Unchecked.*

That means, there is no obligation to do anything particular when calling a method which could throw a runtime exception (like illegal argument).

So what would happen if we were a sloppy GUI designer and called

```
player.go(null)
```

???

# Exceptions - recap - Runtime exceptions

The go() method would throw an `IllegalArgumentException` (according to specification in the javadoc we gave you).

The GUI probably (since the sloppy designer didn't care to read the documentation) prepare for this, and the application would crash with at stack trace.

# So what should we do with sloppy coders?

We could, of course, force them to handle the exception if they happen to send e.g. null to the players go() method!

Enter Checked exceptions!

# Exceptions - recap - Checked exceptions

We could change the implementation of the Player's go() method so that it instead throws a checked exception. Let's create a checked exception class:

```
public class IllegalMoveException extends Exception {
  public IllegalMoveException(String message) {
    super(message);
  }
}
```

# Exceptions - recap - Checked exceptions

We could change the implementation of the Player's go() method so that it instead throws a checked exception. Let's create a checked exception class:

```
public class IllegalMoveException  extends Exception {
  public IllegalMoveException(String message) {
    super(message);
  }
}
```

Make sure it's a checked exception (don't inherit RuntimeException!)

# Exceptions - recap - Checked exceptions

We could change the implementation of the Player's go() method so that it instead throws a checked exception. Let's create a checked exception class:

```
public class IllegalMoveException extends Exception {
  public IllegalMoveException(String message) {
    super(message);
  }
}
```

Make it possible to create an instance and passing a message, and forward that to the super class' constructor.

# Exceptions - recap - Checked exceptions

Change the go() method to throw this new exception instead:

```
/**
 * Moves the player in given direction.
 * @param direction the direction in which to move the player.
 * @throws IllegalMoveException - if the room in direction does not exist.
 */
public void go(Direction direction) throws IllegalMoveException {
  Room newRoom = currentRoom.getRoom(direction);
  if (newRoom == null) {
    throw new IllegalMoveException("Room not existing in direction:  "
                                   + direction);
  }
  currentRoom = newRoom;
}
```

# Exceptions - recap - Checked exceptions

Change the go() method to throw this new exception instead:

```
/**
 * Moves the player in given direction.
 * @param direction the direction in which to move the player.
 * @throws IllegalMoveException - if the room in direction does not exist.
 */
public void go(Direction direction)  throws IllegalMoveException {
  Room newRoom = currentRoom.getRoom(direction);
  if (newRoom == null) {
    throw new IllegalMoveException("Room not existing in direction:  "
                                   + direction);
  }
  currentRoom = newRoom;
}
```

# Exceptions - recap - Checked exceptions

Checked exceptions need to be - eh - checked - or handled. The code for the GUI now **has to** become:

```
try {
  player.go(Direction.NORTH);
  updateGui();
} catch (IllegalMoveException e) {
  messages.setText("Bad direction - should have disabled that button!");
}
```

The checked exception forces the sloppy coder to handle it, or he/she can’t compile. That’s a good thing.

# UML for throwing exceptions

This is one way of representing in UML that a method throws an exception: