# Assumptions:

1. *"Your program should randomly generate seed data"* - It is assumed that seed data more specifically refers to the Events in the World and the Tickets that belong to these Events
2. *"Your program should operate in a world that ranges from -10 to +10 (Y axis), and -10 to +10 (X axis)"* - It is assumed these bounds are inclusive
3. *"Each event has zero or more tickets"* - It is assumed that if one of the closest Events has zero tickets, then an appropriate message be displayed that this event has no Tickets - rather than leaving it ambiguous e.g.

Event 5 at (1 , 2) - No tickets for this event - Distance: 5

Instead of:

Event 5 at (1 , 2) - null - Distance: 5

1. It is assumed that the "Example Program Run" is strict enough to assume that no dedicated GUI is required - therefore this is a console program
2. It is also assumed that, alongside a program that meets all the requirements and is *"idiomatic for the language...",* evidence of testing should also be included. Therefore, the JUnit framework has been used to provide tests for the code.

# Questions:

1. How might you change your program if you needed to support multiple events at the same location?

I made the decision that a World should store a list of Locations, and those Locations store a single event, rather than: a World storing a list of Events. This was precisely done with scalability in mind. So that If such a requirement were to arise then fewer changes would need to be made.

The required changes:

* Make a Location store an ArrayList of Events instead of a single Event
* When printing out the retrieved locations, add an inner loop to print out each Event at each of the closest locations
* Minor changes to toString() methods of Location to better display multiple Events (instead of one) OR, alternatively, implement a dedicated method which returns a list of the Location's Events as strings

2. How would you change your program if you were working with a much larger world size?

As mentioned in question 1, all algorithms and methods included in the program - to the best of my knowledge at least - have been designed to cater for scalability. Therefore, to address the issue of the greater need for efficiency with a larger world size, all algorithms run in O(n) time (worst-case) except for one:

The sorting algorithm for the Locations (Collections.sort()) uses a stable merge-sort implementation. Firstly, this means that equal elements will not be reordered as a result of the sort (hence, stable). Secondly, this means the algorithm runs in O( nlog(n) ) time - one of the fastest sorting algorithms available.

All current classes and methods are - according my planning and knowledge - as relevant as they would be, even if the world size was considerably larger.