COSC 1P03 Assignment 2
Life's full of randomness. But, like, not really

Objective:
To create implementations of an ADT, as well as test them.

Background:
Have you ever given much thought into how a computer can generate “random” numbers?
Remember, *everything* a computer does is deterministic by its very nature. Randomness doesn't exist in a computer (we're going to set aside the discussion over whether it exists *at all*). In spite of this, we often find ourselves needing random numbers.
Or do we?
Technically, we typically only need *chaotic* values: generated values, such that it's impractical to anticipate the next number in the sequence. So long as we have that, it'll suffice for most applications.
And those applications are numerous:

- Myriad forms of cryptography rely on them
- Games can be less predictable
- Multiple simulations of the same premises can have varying outcomes

We typically achieve this via a PRNG — Pseudo-Random Number Generator
A PRNG relies on three components:

- Some **internal state**, that is updated with each number generated, such that successive calls will be different from each other
- Some **seed**, to create the *initial* version of that internal state
- A mathematical formula, of some sort, to create numbers based on that internal state (which also might be what updates it)

An interesting side effect of how these operate is that, if you create two instances of the same type of PRNG, with the same starting seed, and make the same requests, they always produce the same results.

- In case you've ever wondered what a 'seed' is for procedurally-generated games: that's why
- This can be handy if you ever need to re-play the results of something

If you want to get *different* results, just use a different seed!

- It's very common to incorporate the 'current time' into the seed, as later re-plays would presumably occur at different starting times

A craptacular example:
Let's design an absolutely terrible PRNG:

- Our starting seed will *always* be the number 0
- Whenever we request a 'random' integer, we'll increment the state, and return that value
- Floating-point values are more complicated, so we'll generate the next 'state integer', and then divide by some sufficiently large value to scale it to the range of [0..1)

You can find an example of such here:
https://www.cosc.brocku.ca/~efoxwell/private/1P03/2025s/CrappyRNG.java
Note that this is **not** for this assignment. This just uses some notion of 'state' to generate values.

The actual task:

We'll define a PRNG as such:

- It is part of the `chaotic` package
- It is presumed to have some sort of starting seed
  - We have the ability to request it (via `getSeed`)
  - The seed should be of the `long` type
  - If none is provided, the current system time in milliseconds should be used
- It can yield a 'random' `int`eger
  - If no additional details are provided, the value will be anywhere within the range of 0 to the maximum integer value
  - You can request a different upper boundary, keeping 0 as the lower-bound
  - You can specify *both* boundaries
  - Ranges are subject to the following caveats:
    - They are *inclusive* on the lower-bound, and *exclusive* on the upper-bound
    - The upper-bound *must* always be higher than the lower-bound
    - Negative values are possible, so long as they don't violate the above requirement
    - Requesting invalid ranges throws a `PRNGException`
- It can yield a floating-point (double) value, from 0.0 (inclusive) to 1.0 (exclusive)

Of course, this means an *interface*.

You'll also create *three* different implementations of this PRNG:

- An incrementing PRNG that simply increments its internal state by half its seed for each request
  - Of course, this will *not* look very random! (It's for confirming process; not results)
- A *Timely PRNG* that also keeps track of the passage of *nanoseconds*, as a source of entropy:
  - Use the `System.nanoTime` function
  - Unless a starting seed is explicitly provided for initialization, just use that 'nano time'
  - Each time a value is generated, update its internal state based on the number of nanoseconds that have passed since the *previous* call
  - (To confirm: this is the one PRNG that *won't* be repeatable, even with the same seed)
- A basic (traditional) PRNG:
  - Assuming some *state* (initially the *seed*), first generate an integer, and then take the modulus of that if you need to fit within a particular range
  - For each such call, you also need to update that state:
    - $state=(1103515245 \times state+12345) \bmod 2147483647$
    - There are other numbers you *could* use; these are just the classic C defaults
    - (btw, 2147483647 is just `Integer.MAX_VALUE`, if you prefer. It's the highest positive value you can represent for a signed integer)

Of course, all three implementations are part of the same (`chaotic`) **package**.

Finally, you'll write two *client* programs, both under the `client` **package**:

- A very simple test harness, to demonstrate their usage
  - You should write methods to test a `PRNG`, and then call them with different *implementations*
  - Make sure to test that exceptions are thrown (do I need to say your client must catch them?)
- A basic text-based game, that relies on randomness
  - e.g. the "high-low" game:
    - The game picks a random number from 1 to 100, and the user needs to guess it
    - If the user guesses too high, it says to try 'lower'; and vice versa
    - The user may repeat the game, etc.

Let the user decide which PRNG to use for the latter.

Tips and special requirements:

- Please note that everything you need is already provided to you; there's nothing to 'look up'
  - But please do note that the requirements are *very* specific
- You'll need at least two constructors per implementation:
  - One that lets you assign an explicit seed, and one that uses the current system time
  - You realize you shouldn't write these completely independently... right?
  - Remember that, when calling `getSeed`, that yields the *original* seed, not the current state
- Even though the 'crappy rng' was mostly worthless, it still contains a reasonable approach for generating the floating-point value
- The seeds must be of the `long` type. That's actually a *good* thing:
  - It makes the values easier to work with, because you're generating `int`s, but starting with `long`s. Values in the integer (32-bit) range won't overflow when stored in the long integer (64-bit) range (though obviously `long` values can/will still overflow!)
  - The 'current system time' is already provided as a `long` value
  - (Similarly, you'll want to maintain the 'internal state' as a `long` as well)
  - Some marks will be dedicated to the 'appropriateness' of the implementation. As a hint of one such factor: you'll have three versions of `nextInt`, right?
- This should be very clear by now, but you're required to use two separate packages:
  - The testing code is not RNG code
  - The RNG code is not for testing, and *certainly* not for printing
  - If you ignore this, please don't ask why you received a zero
- This should be easy to implement in IntelliJ
  - **You can have me help you with this, considering there are two main classes!**
  - Remember: JDK 11
- You're still submitting a .zip file. Only. Not .rar. Not .7z
- You're welcome to use BasicIO, but beyond that you will not need a single `import` (beyond your own other package!)

Submission:

Bundle up your solution, data files, and a sample execution (pretty much everything the marker needs to easily grade your work) into a .zip file, and submit it through Brightspace.

Some reminders:

- .zip means .zip
- You're **not** writing your solution in one IDE and *then* converting to IntelliJ