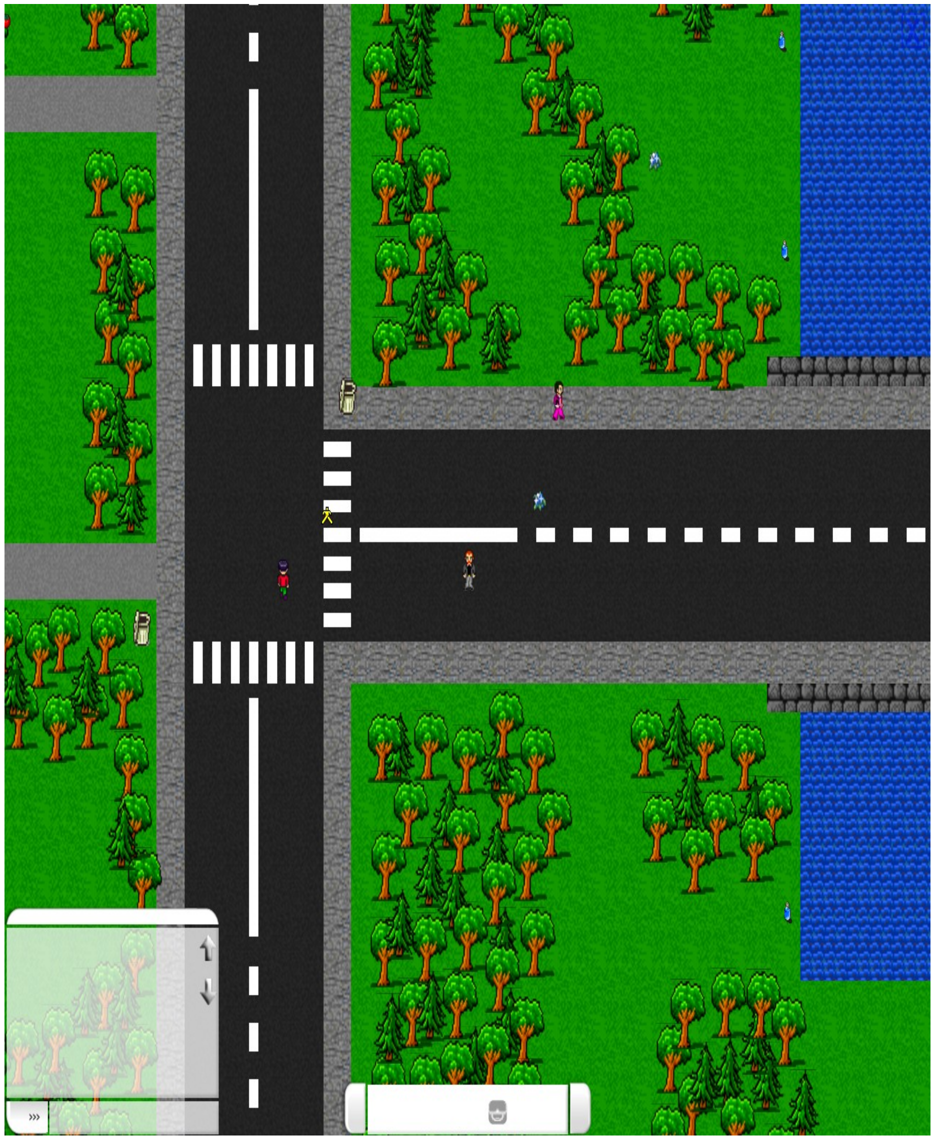
Diving in: the Singleton Design Pattern

Mammoth is a massively multiplayer game research framework.
`mammoth.cs.mcgill.ca`

The world of Mammoth is a 2D environment viewed from a 2D perspective.

The world contains a fixed number of game objects, some of which can be controlled by humans (players).

A player can move around in the game, examine objects, pick them up, and drop them again.



IDs?

Each object in the world (player, items, grass, etc) has a unique ID associated to it.

How do we hand out IDs, making sure that one never distributes a duplicate one?

ID Distributor

Mammoth uses unique identifiers (ID) to identify all the Game objects in the world.

These IDs are distributed by a single object.

If more than one distributor were used, duplicate IDs could be distributed.

The application needs global access to this distributor.

It would be very complicated/ugly to pass around the reference to the distributor throughout the application.

Problem

We need to make sure that only <u>one</u> instance of a class can be created.

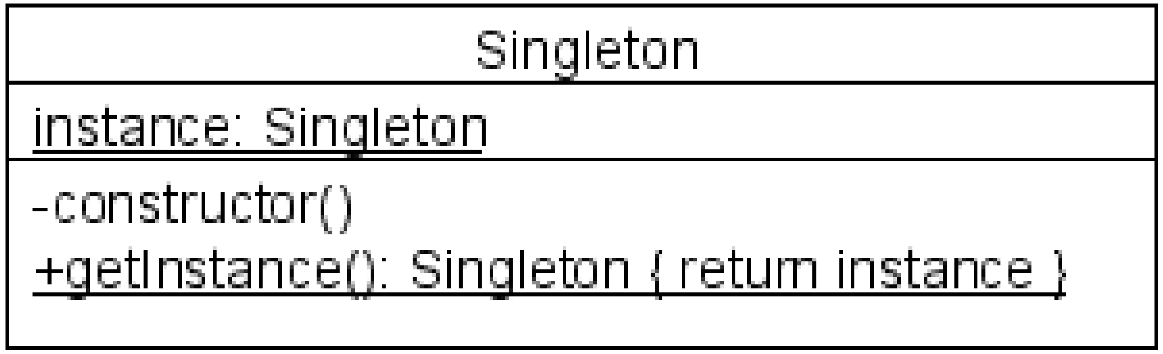
We want that instance to be easy to access anywhere in the application.

Singleton

Ensure a class only has one instance,

and provide a global point of access to it.

Class Diagram



Code Structure

```
public class Singleton {

   private static Singleton instance = new Singleton();

   private Singleton() { }

   public static Singleton getInstance() {
      return Singleton.instance;
   }
}
```

Consequences

You are assured that only one instance can be created.

Global access to that instance without the use of a global variable (less pollution)

Can be modified to allow a fixed number of instances.

Singletons can be sub-classed.

ID Distributor Example

```
public class IdDistributor {

    private static IdDistributor instance = new
      IdDistributor();
    private long lastId;

    private IdDistributor() {
        this.lastId = -1;
    }
    public static IdDistributor getInstance() {
        return IdDistributor.instance;
    }
    public long getId() {
        this.lastId++;
        return this.lastId;
    }
}
```

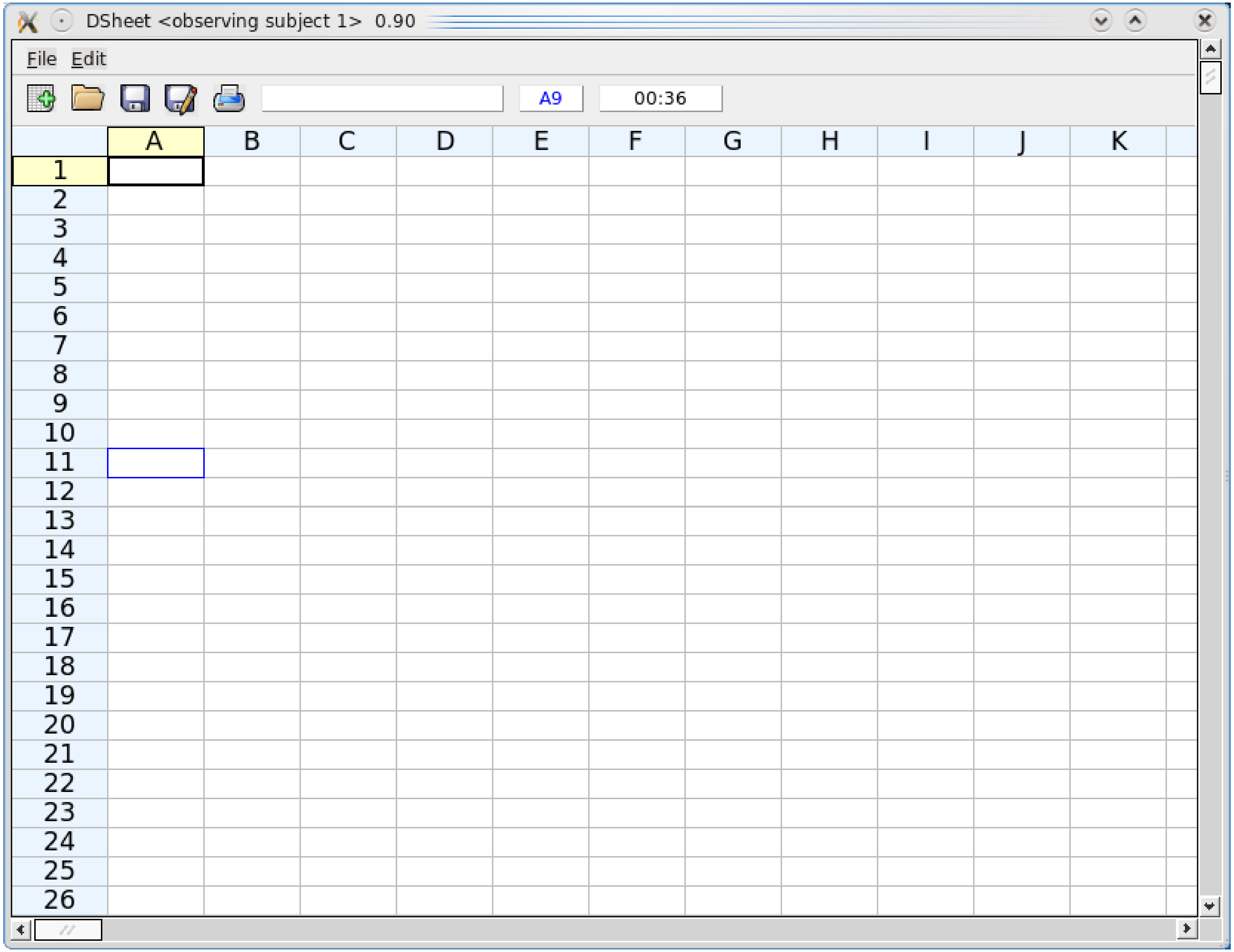
Lazy Initialization

```
public class Singleton {

   private static Singleton instance;

   private Singleton() { }

   public static Singleton getInstance() {
      if (Singleton.instance == null) {
         Singleton.instance = new Singleton()
      }

      return Singleton.instance;
   }
}
```

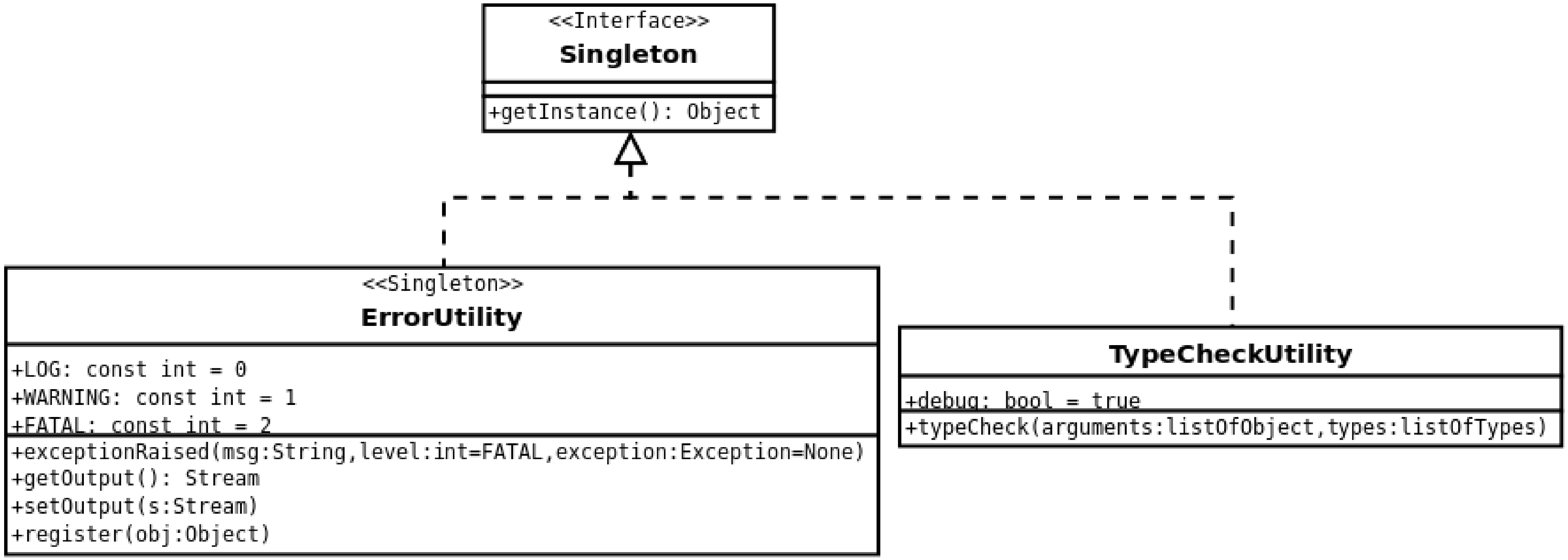
Lazy Initialization (Better)

```
public class Singleton {

   private static Singleton instance;

   private Singleton() { }

   public static synchronized Singleton getInstance() {
      if (Singleton.instance == null) {
         Singleton.instance = new Singleton()
      }

      return Singleton.instance;
   }
}
```

Dsheet


DSheet <observing subject 1> 0.90
File
Edit
A9
00:36
A
B
C
D
E
F
G
H
I
J
K
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

Singleton



exception is the object
representing the error
that has just occured.
Depending on the implementation
language, exception as classes instances
could be built-in or not.
This is why it is optional