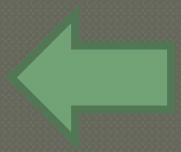
Design Patterns

- The Strategy Pattern
- The Factory Method
- Generics
- The Abstract Factory Pattern
- The State Pattern
- The Observer Pattern
- The Adapter Pattern
- The Composite Pattern
- The Iterator Pattern
- The Builder Pattern
- Fallen Patterns
  - **The Singleton Pattern**
  - The Visitor Pattern
- Command Pattern



The Fallen Patterns

- In this class, we've covered the most common and important object-oriented design patterns
- We haven't covered all of them!

- Some patterns have become obsolete, or discredited
- Despite this, you will still see them, and need to know them

In Particular

- Two discredited patterns are extremely common:
  - Singleton: Make sure there's only one, global instance of a class
  - Visitor: Kind of like an iterator but worse
- These patterns are often considered a "code smell"
  - Cranky experienced developers will complain when they see a singleton
- If you're in an interview, and they ask your favorite design pattern, never, ever say "singleton"
- Despite this, there are valid reasons to use them

Singleton

- Singleton pattern that ensures there will only be one instance of a particular class
- Instead of creating an object directly:
  - `Frabjurator frab = new Frabjurator(…);`
- We use a static method:
  - `Frabjurator frab = Frabjurator.getInstance();`
- `Frabjurator.getInstance()` **always returns the same instance**
- Therefore, is **different from a factory**

What's Wrong With It?

- Singletons are "anti-functional"
- They're essentially a fancy global variable
- Global variables are useful sometimes, but usually they're a terrible idea
  - They make refactoring more difficult
  - They make testing much more difficult
  - They make reasoning about the behavior more difficult
- You can't use the type system for initialization order
- Sometimes they are useful in spite of the above

Wikipedia

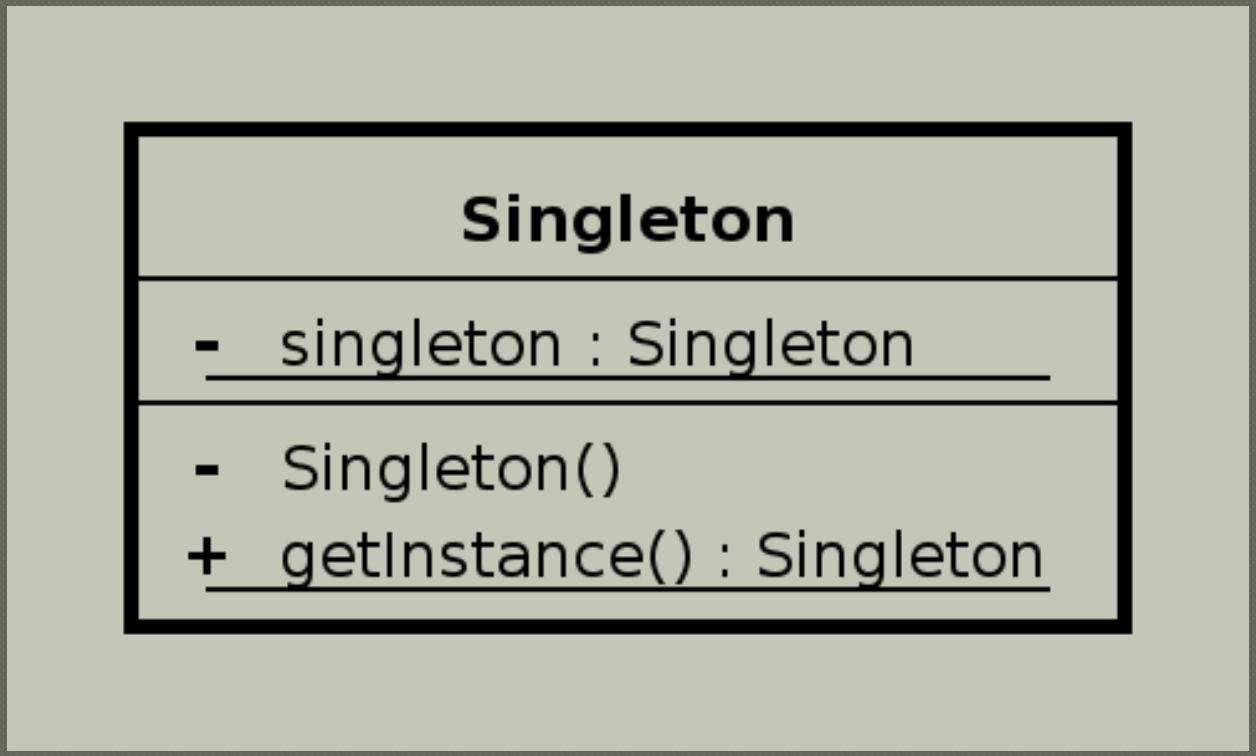
"There are some who are critical of the singleton pattern and consider it to be an anti-pattern in that it is frequently used in scenarios where it is not beneficial, introduces unnecessary restrictions in situations where a sole instance of a class is not actually required, and introduces global state into an application."

Where you see it

- Singleton is associated with factories:
  - The factory itself can be a singleton (configuration managers)
- It is common when accessing hardware resources:
  - Mouse.getInstance();
  - Screen.getInstance();

Singleton Class Diagram

- Singleton has the easiest class diagram



3 Things Worth Noting

1. The constructor is private
   - Why?
   - Because if it weren't, the user could create more than one instance, defeating the purpose
2. The instance field is static and private
   - Why?
   - Static because it's an easy way to ensure that there is exactly one, globally accessible field
   - Private to force users to use `getInstance`
3. The only way to access the instance is through the `getInstance` method (which is also static)
   - This is where we return the one instance, or create it if it doesn't exist

Example

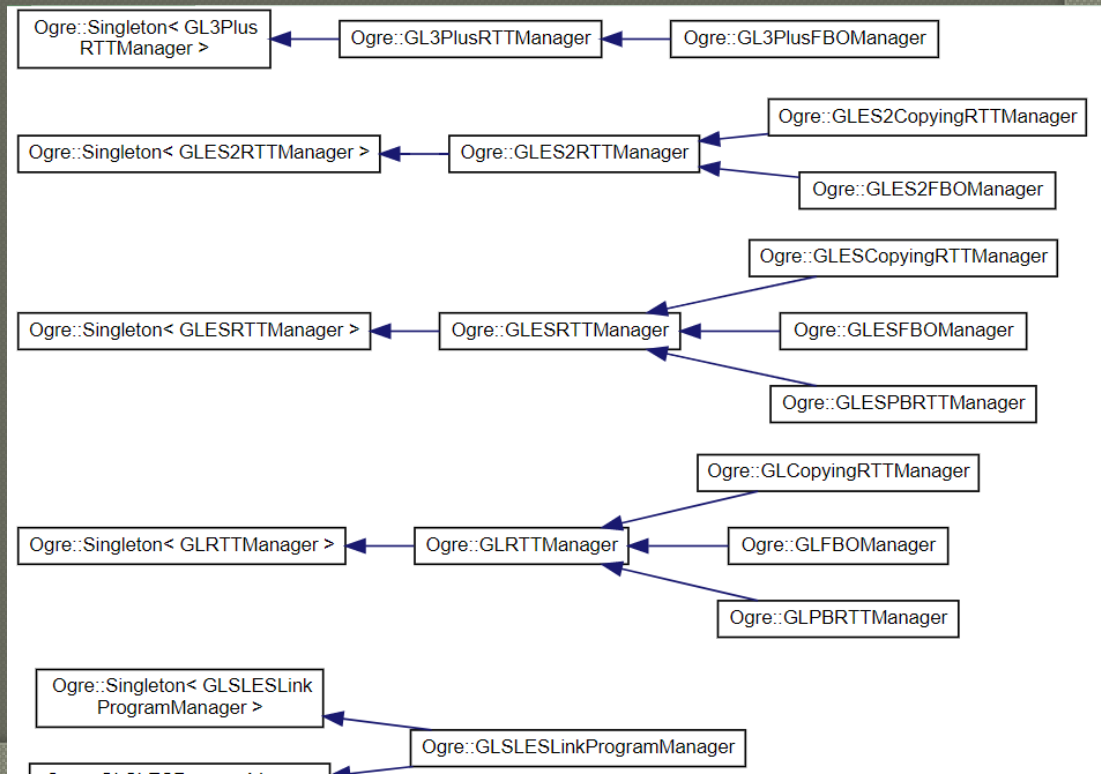
```
class GraphicsManager {
  private static GraphicsManager instance;

  private GraphicsManager(…) { … }

  public static GraphicsManager getInstance() {
    if( instance == null )
      instance = new GraphicsManager(…);
    return instance;
  }
}
```

Great Example of Misuse

- The Ogre3D engine
- Tons of "managers" and "singletons" ("manager" is a smell too)
- How can we replace the singletons?



What's the Alternative?

- Use the type system to your advantage:

```
var engine = new GameEngine();

//ensures one instance
var video = engine.createVideoSystem();

var screen = video.createScreen();
```

Type Systems

- If a class has a package default visibility constructor, it can't be created by users of your class outside the package
- But if the class is public, it can still be used by users outside the package
- Use the type system to force you to initialize in the right order
- C++ has friend classes instead

Benefits to the Alternative

- Correct order of initialization is forced, so it can't be wrong
- The instances are also not global, which reduces coupling
- Basically, this is what you should do instead of using a singleton in almost every circumstance

Other Alternatives

- Need to avoid making multiple instances of a class?
- You can just not do that