

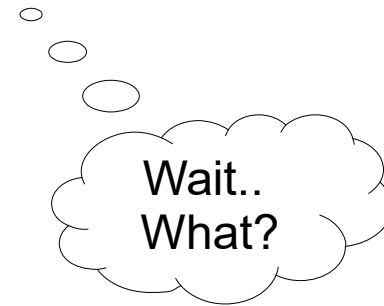


# Pattern / Anti-pattern: Singleton

# Topics

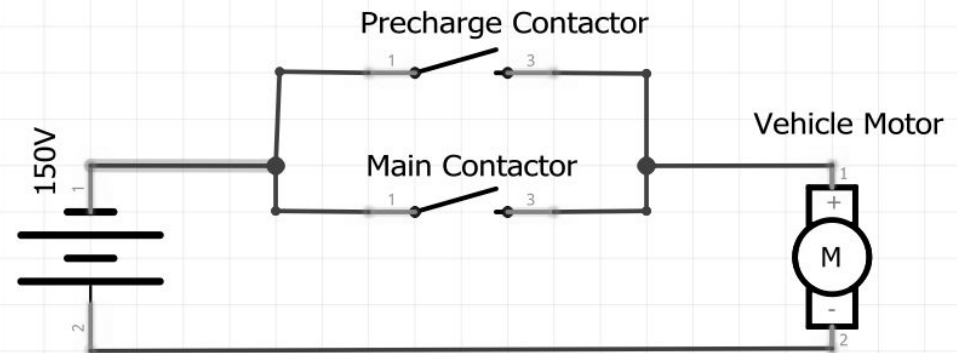
1) How can we:

- a) Create just one instance of a class?
- b) Allow all the code to share access to an object?
- c) Allow lazy initialization?
- d) Tightly couple all our code to one class



# Motivation

- Sometimes, it's critical that only one instance of a class exists
  - Ex: Logger, DB Connector, Thread pool, Launch control timer...
- Ex: Battery Contactor Controller (BCC)
  - Hardware to control power to an electric vehicle's motor
  - Software must control hardware carefully



# Battery Contactor Controller (BCC)

## Constructor

- check hardware connection & turn off (open) contactor
- *requires hardware sub-system to initialize first.*

## StartPreCharge()

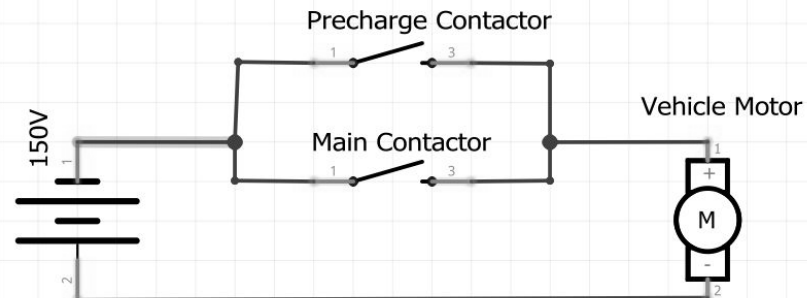
- Precharge contactor on for 10s, then turn on main contactor;
- Prevents voltage spike frying the system.

- turnOn() closes main contactor;
- turnOff() opens both contactors

## Safety mode

- Disables (opens) both contactors

```
class BCC {  
  
    public BCC() {  
        // Check HW  
        // Set to Off  
    }  
  
    .. startPreCharge();  
    .. turnOn();  
    .. turnOff();  
    .. activateSaftey();  
}
```

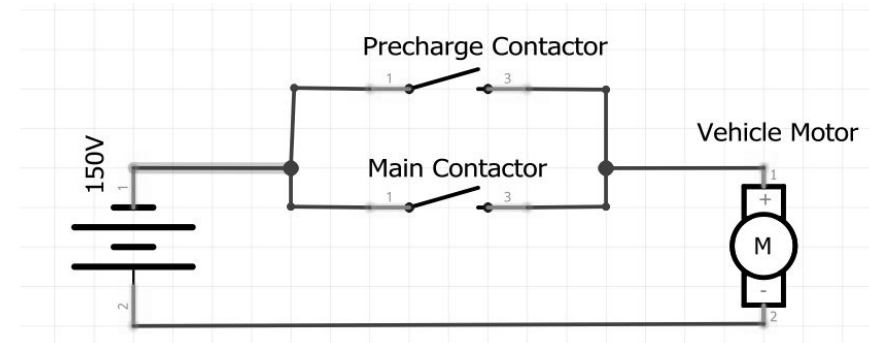


# Analysis of BCC

- What happen if more >1 BCC?
  - Constructor of 2nd would turn off contactors, interrupting whatever was happening 1st
  - Turing 1<sup>st</sup> “on” while 2<sup>nd</sup> pre-charging could damage hardware
  - Activating safety mode on 1<sup>st</sup> object irrelevant with 2<sup>nd</sup>!

• ..

```
class BCC {  
  
    public BCC() {  
        // Check HW  
        // Set to Off  
    }  
  
    .. startPreCharge();  
    .. turnOn();  
    .. turnOff();  
    .. activateSafety();  
}
```



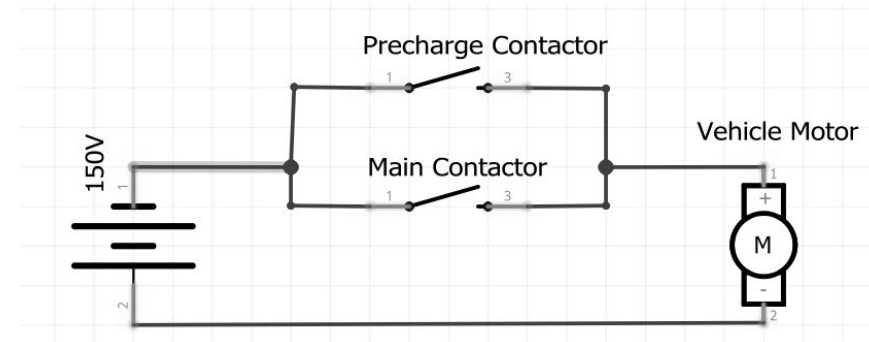
# Requirements of BCC

- Requirement

- ..
- ..
- ..

```
class BCC {  
  
    public BCC() {  
        // Check HW  
        // Set to Off  
    }  
  
    .. startPreCharge();  
    .. turnOn();  
    .. turnOff();  
    .. activateSaftey();  
}
```

Global access  
to the BCC  
a bad design?



# Ideas that Don't Work

# Easy things that don't work

- Bad idea 1: ..
  - gives everyone access to BCC
  - but..
- Bad idea 2: ..
  - everything static (static class w/ static member functions and variables)
  - ..

## Requirement

- 1) At most one copy of BCC
- 2) Construction *after* other subsystems initialized
- 3) Any code can get access to the BCC



# Singleton

# Limit Construction

- new executes a constructor; ..
  - Can instantiate an object from inside the class
- Create a..

```
public class BCC {  
    private static BCC instance;  
  
    private BCC() {  
        // Check HW  
        // Set to Off  
    }  
  
    public static BCC getInstance() {  
        if (instance == null) {  
            instance = new BCC();  
        }  
        return instance;  
    }  
    public void startPreCharge() {}  
    public void turnOn() {}  
    public void turnOff() {}  
    public void activateSaftey() {}  
}
```

# Singleton Pattern

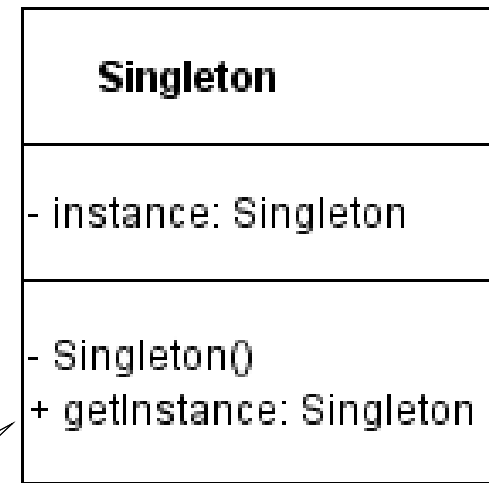
- Singleton Pattern

..

- To get an instance of this class, you have to go through this class.
- Public static method to get an instance so anyone can access it
- Allows lazy initialization.

- Exercise

Changed RedHeadDuck into singleton  
HeadFirst sample code.  
(Would not **want** to, though!)



Statics

// Private constructor

# Multi-Threaded

- What if singleton used in multithreaded application?

- two threads enter the ==null check at once; instantiating two BCC's.

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

- 1st one through gets an orphaned copy of the BCC, thus causing havoc!

- Solution

- ..

- Poor Solution

- If not needing lazy / late initialization:  
private static instance = new BCC();  
... getInstance() { return instance; }

- hard to track down bugs: construction happens at application launch; and initialization order dependency.

# Problems with Singletons

- Inheriting from a singleton class is problematic
  - have to make constructor protected, and then can end up with multiple of them!
- ..
  - GS makes it hard to understand the system because
    - ..
    - things happen outside normal flow of execution.
  - Components accessing GS
    - ..
    - (mock and driver objects)
- Google code talk on global state  
<https://www.youtube.com/watch?v=-FRm3VPhsel>

# How to avoid globals?

- What design principle/technique can we use to avoid this?
- ..  
pass it a reference to the required “global” object(s)
  - ..
  - Testable: Client code able to select which objects it wants other code to use (good for mock'ing)
  - Explicitly force the initialization order at compile time.

# Guideline

- When to use dependency injection (DI)?
  - ..  
(things that don't store the data, but process it)
    - should use Dependency Injection
  - ..  
(store data; things you'd save)
    - need not use dependency injection;  
just instantiate the object as needed

# Summary

---

- Singleton Pattern for
  - Limit instantiation of a class to 1
  - Global access to that object
- Supports lazy initialization
- Anti-pattern: It creates global state
  - cannot test with it
  - tight coupling to all classes that use it
  - hidden dependencies