Singleton Pattern

514770-1
Fall 2026
10/13/2026
Kyoung Shin Park
Computer Engineering
Dankook University

Singleton Pattern

- "Ensure a class only has one instance, and provide a global point of access to it."
- Sometimes we need to have exactly one instance of our class, e.g. a printer spooler (where we only need one print managing the work list), or a single database connection (shared by multiple objects).
- java.lang.Runtime#getRuntime()
- java.awt.Desktop#getDesktop()
- java.lang.System#getSecurityManager()

Singleton Pattern

- The most popular approach is to implement Singleton:
  - **A private default constructor**
  - **A static field containing the singleton instance**
  - **A static factory method for obtaining the singleton instance**
  - While this is a common approach, it's important to note that it **can be problematic in multithreading scenarios**

Singleton Pattern

	Description
Pattern	Singleton
Problem	Sometimes we need to have exactly one instance of our class, e.g. printer spooler, DB connection, configuration manager, etc
Solution	It was created as a solution to classes that need to be instantiated only once.
Result	Consistent state because there is only one instance.

Classic Implementation of Singleton Pattern

- Classic Implementation of Singleton Pattern
  - Private default constructor
  - Create static field containing its only instance
  - Create static factory method for obtaining the instance
- This code can be a problem for multi-threaded programs (the solution is described later).

Classic Implementation of Singleton Pattern

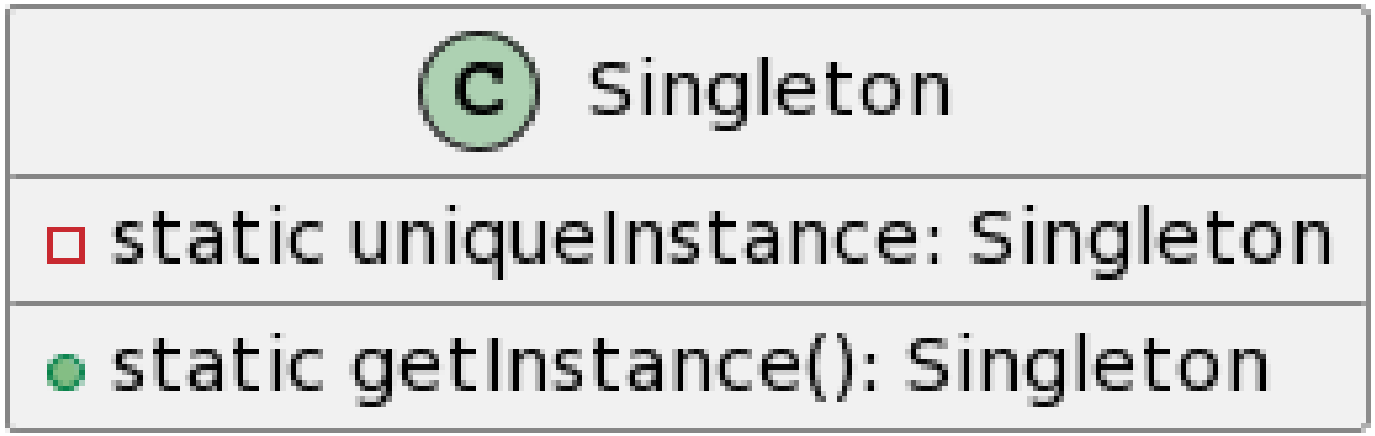
```
public class Singleton {
  // static field containing its only instance
  private static Singleton uniqueInstance;

  // other member fields ..

  // private default constructor
  private Singleton() { }
  // static factory method for obtaining the
instance
  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  // other member methods ..
}
```

Classic Implementation of Singleton Pattern

- Class Diagram



Chocolate Factory (HFDP Ch. 5)

- Chocolate factories have computer controlled chocolate boilers.
- The job of the boiler is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars.
- The controller class for Choc-O-Holic, Inc.'s industrial strength Chocolate Boiler.
  - Ensure that bad things don't happen, like draining 500 gallons of unboiled mixture, or filling the boiler when it's already full, or boiling an empty boiler.

Chocolate Factory (HFDP Ch. 5)

```
public class ChocolateBoiler {
  private boolean empty;
  private boolean boiled;
  public ChocolateBoiler() {
    empty = true; // empty boiler
    boiled = false;
  }
  public boolean isEmpty() {
    return empty;
  }
  public boolean isBoiled() {
    return boiled;
  }
  public void fill() {
    if (isEmpty()) { // to fill, it must be empty
      empty = false;
      boiled = false;
      // fill boiler with milk/chocolate mixture
  }
```

Chocolate Factory (HFDP Ch. 5)

```
  // to drain, it must be full(non empty) and boiled
  // once it is drained, we set empty back to true
  public void drain() {
    if (!isEmpty() && isBoiled()) {
      // drain the boiled milk and chocolate
      empty = true;
    }
  }
  // to boil, it must be full and not already boiled
  public void boil() {
    if (!isEmpty() && !isBoiled()) {
      // bring the contents to a boil
      boiled = true;
    }
  }
}
```

Chocolate Factory (HFDP Ch. 5) – Using Singleton Pattern

```
public class ChocolateBoiler {
  private static ChocolateBoiler uniqueInstance;
  private boolean empty;
  private boolean boiled;

  private ChocolateBoiler() {
    empty = true; // empty boiler
    boiled = false;
  }

  public static ChocolateBoiler getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new ChocolateBoiler();
    }
    return uniqueInstance;
  }
  // rest of code..
}
```

Thread-Safe Singleton

- The main problem with the classic implementation of Singleton is that it is **not thread safe**.
- Using **synchronized** makes sure that **only one thread at a time** can execute **getInstance().**

```
public class Singleton {
  private static Singleton uniqueInstance;
  private Singleton() { }
  // only one thread can execute this at a time
  public static synchronized Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
  // rest of code..
}
```

Thread-Safe Singleton (Eager Initialization)

- The main disadvantage of this method is that using synchronized every time while creating the singleton object is expensive and may **decrease the performance** of your program.
- **Eager Instantiation**
  - If performance of getInstance() is not critical for your application,
  - Here we have created **instance of singleton in static initializer**.

```
public class Singleton {
  // static initializer
  private static Singleton inst = new Singleton();
  private Singleton() { }
  public static Singleton getInstance() {
    return inst;
  }
  // rest of code..
}
```

Thread-Safe Singleton (Lazy Initialization with DCL)

- **DCL(Double Checked Locking)** to reduce the use of synchronization in getInstance()
  - If you notice carefully once an object is created, synchronization is no longer useful because now object will not be null and any sequence of operations will lead to consistent results.
  - So **we will only acquire lock on the getInstance() once, when the object is null**. This way we only synchronize the first way through, just what we want.

Thread-Safe Singleton (Lazy Initialization with DCL)

```
public class Singleton {
  // Double Checked Locking
  private static volatile Singleton inst;
  private Singleton() { }
  public static Singleton getInstance() {
    // we only synchronize the first time
    if (inst == null) {
      synchronized (Singleton.class) {
        if (inst == null) {
          inst = new Singleton();
        }
      }
    }
    return inst;
  }
  // other member methods..
}
```

Thread-Safe Singleton

- volatile (since Java5)
  - The volatile keyword marks a **variable** that always goes to the **main memory, for both reads and writes**, of the multiple threads accessing it (and **not just to the CPU cache**).
  - The volatile keyword guarantees visibility of changes to variables across threads.
- synchronized
  - The synchronized keyword will cause all modifications guarded by considered **lock to synchronize** with main memory and adds **mutual exclusion**.
  - Mutual exclusion prevents an object from being seen in an inconsistent state by one thread while some other thread is updating that object.

Singleton (Inner Static Class)

- When Singleton loads at first by JVM, since there is no static data member in the class; SingletonHolder does not loads or creates inst.
- This will happen only when we invoke getInstance(). JLS(Java Language Specification) guaranteed the sequential execution of the class initialization; that means thread-safe.

```
public class Singleton {
  // inner static class
  private static class SingletonHolder {
    static final Singleton inst = new Singleton();
  }
  private Singleton() { }
  public static Singleton getInstance() {
    return SingletonHolder.inst;
  }
}
```

Singleton vs Static Class

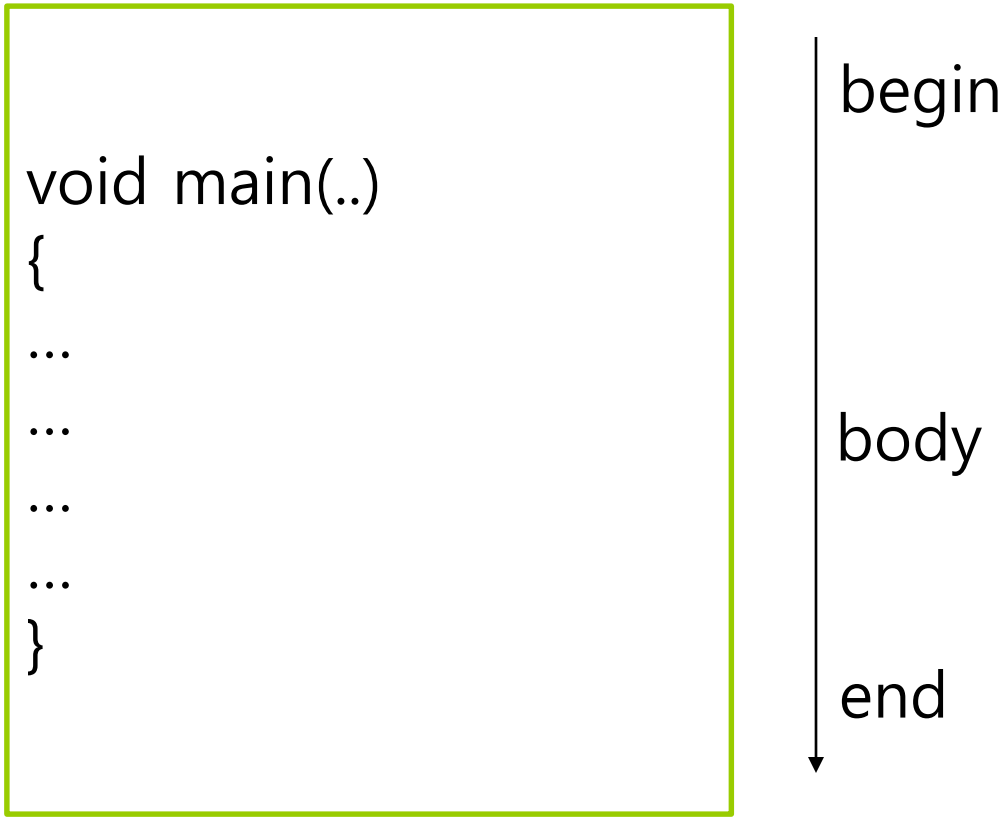
- Singleton provides only one instance during application life cycle.
- Static class is a class which only contains static methods.
- Java supports static variables, static methods, static block and static classes. Java allows nested classes. A static nested class may be instantiated without instantiating its outer class.
- Both Singleton pattern (e.g. **java.lang.Runtime**) and static class (e.g. **java.lang.Math**) can be used without creating object and both provide only one instance.
- The fundamental difference between **Singleton** pattern and **static class** is, one represent an **object** while other represent a **method**.

Singleton vs Static Class

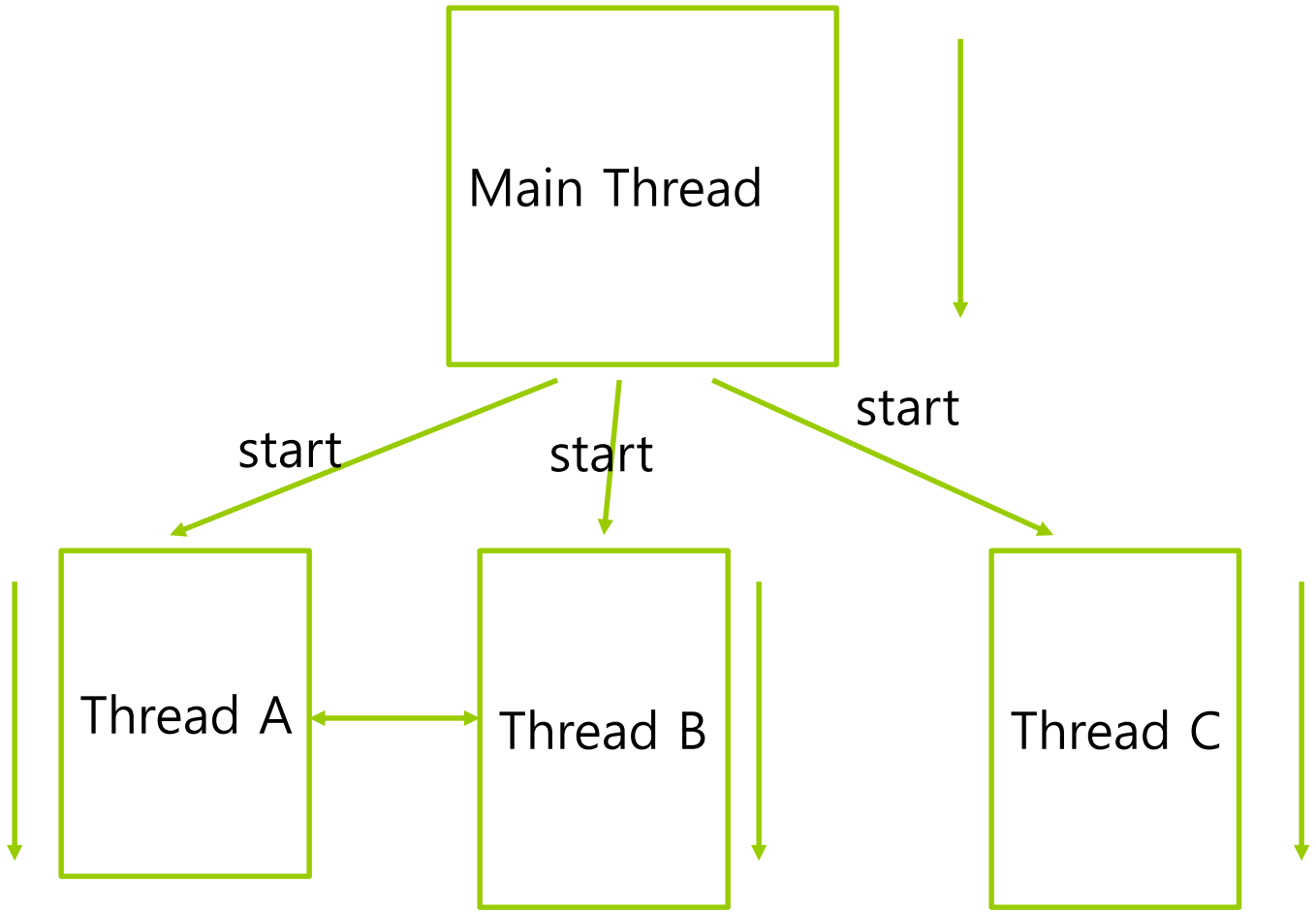
- If Singleton is not maintaining any state, and just providing global access to methods, than consider using static class.
- Singleton uses inheritance and polymorphism to extend a base class, and implements an interface.
  - E.g. In java.lang.Runtime, getRuntime() method returns different implementations based on different JVM, but **guarantees only one instance** per JVM.

Singleton Pattern	Static Class
Lazy loaded	Eagerly loaded (static binding in compile-time)
OOP, inheritance, polymorphism	Static methods cannot be overridden
Slow performance	Fast performance by static binding
Easier test	Hard to test
Heap memory	Stack memory

A Single Threaded program



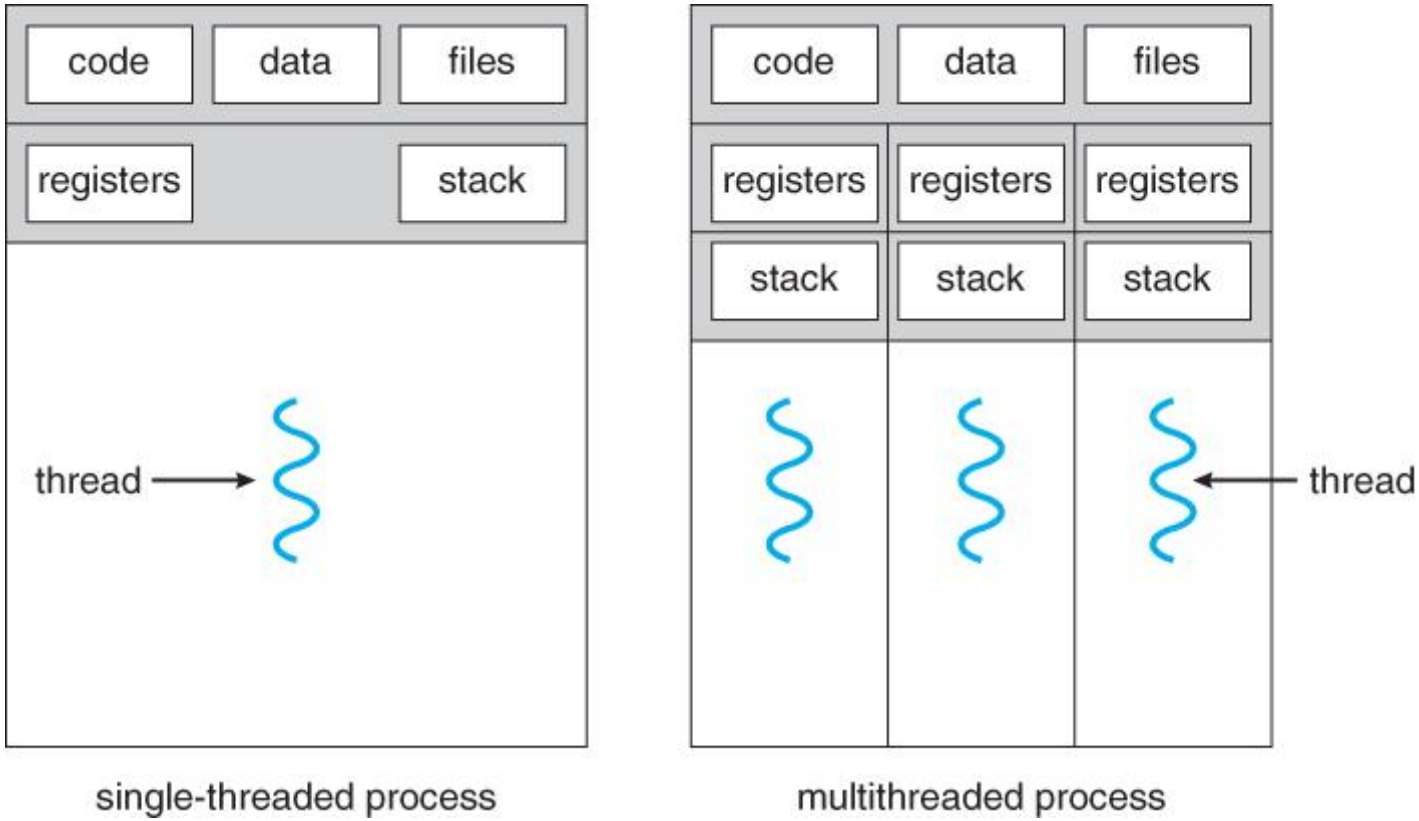
A Multithreaded Program



Threads may switch or exchange data/results

Single vs Multithreaded Process

Threads are light-weight processes within a process



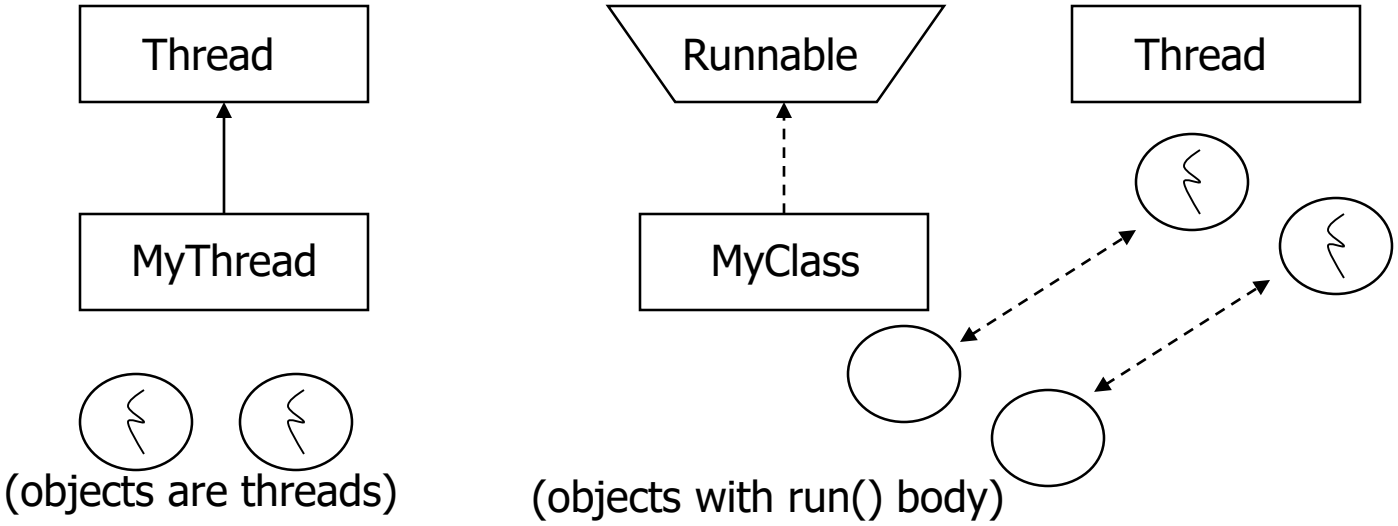
https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Java Threads

- Java has built in support for Multithreading
- Synchronization
- Thread Scheduling
- Inter-Thread Communication:
  - currentThread
  - yield
  - sleep
  - resume
  - start
  - run
  - stop
  - setPriority
  - getPriority
  - suspend
- Java Garbage Collector is a low-priority thread

Java Threads

1. Create a class that **extends** the **Thread** class
2. Create a class that **implements** the **Runnable** interface



1. Extending the Thread Class

- Create a class by extending Thread class and override run() method:

```
class MyThread extends Thread
{
        public void run()
       {
          // thread body of execution
       }
}
```

- Create a thread:

```
MyThread thr1 = new MyThread();
```

- Start Execution of threads:

```
thr1.start();
```

- Create and Execute:

```
new MyThread().start();
```

1. Extending the Thread Class

```
class MyThread extends Thread {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
}

class ThreadEx1 {
        public static void main(String [] args  ) {
          MyThread t = new MyThread();
           t.start();
       }
}
```

2. Threads by implementing Runnable interface

- Create a class that implements the interface Runnable and override run() method:

```
class MyThread implements Runnable
{
  .....
  public void run()
  {
     // thread body of execution
  }
}
```

- Creating Object:

```
MyThread myObject = new MyThread();
```

- Creating Thread Object:

```
Thread thr1 = new Thread( myObject );
```

- Start Execution:

```
thr1.start();
```

2. Threads by implementing Runnable interface

```
class MyThread implements Runnable  {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
}

class ThreadEx2 {
        public static void main(String [] args  ) {
                Thread t = new Thread(new MyThread());
                 t.start();
        }
}
```

Life Cycle of Thread

