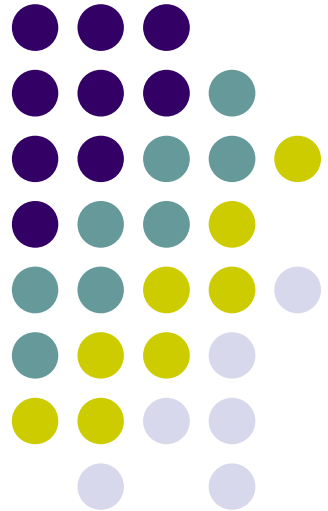
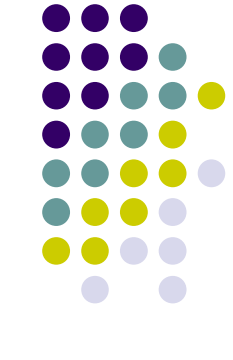
Singleton Design Pattern



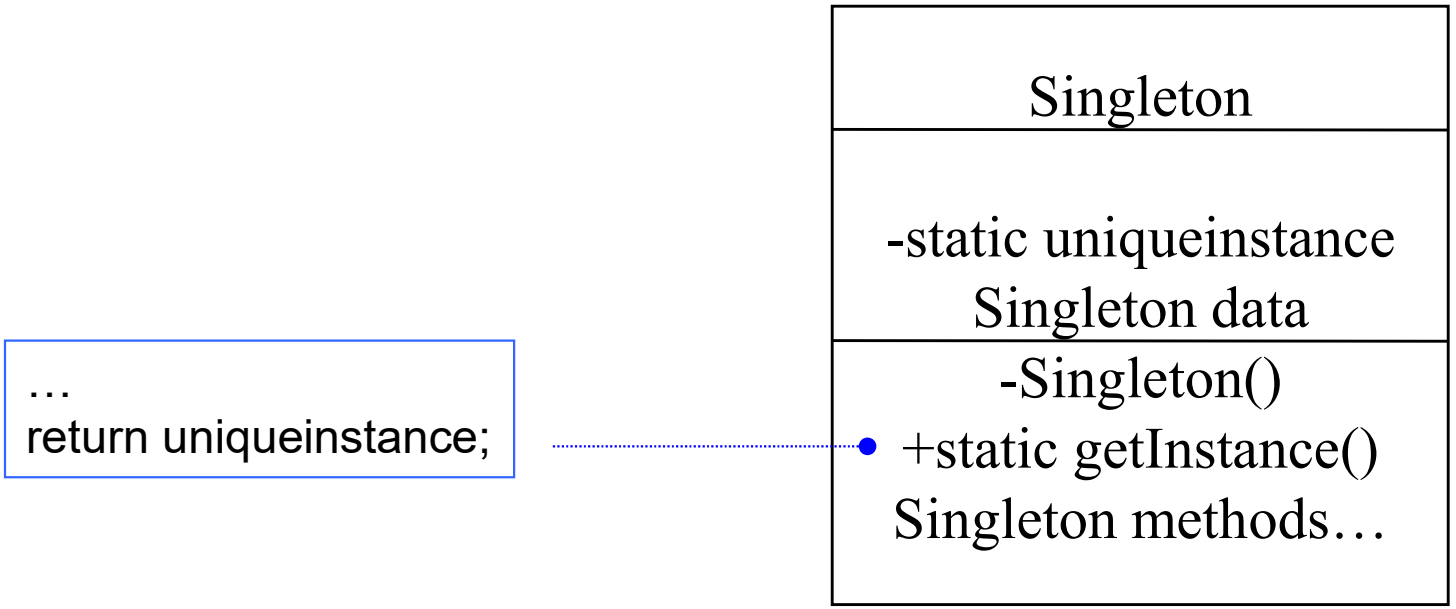
Singleton

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it
- Motivation
  - Important for some classes to have exactly one instance. E.g., although there are many printers, should just have one print spooler
  - Ensure only one instance available and easily accessible
    - global variables gives access, but doesn't keep you from instantiating many objects
  - Give class responsibility for keeping track of its sole instance

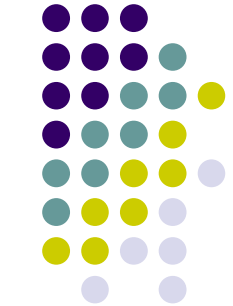
Design Solution



- Defines a getInstance() operation that lets clients access its unique instance
- May be responsible for creating its own unique instance



Singleton Example (Java)



- Database

Database
static Database* DB instance attributes…
static Database* getDB() instance methods…

```
public class Database {
private static Database DB;
    ...
private Database() { ... }
public static Database getDB() {
    if (DB == null)
      DB = new Database();
    return DB;
}
    ...
}
```

In application code…

```
Database db = Database.getDB();
db.someMethod();
```

Implementation

- Declare all of class's constructors private
  - prevent other classes from directly creating an instance of this class
- Hide the operation that creates the instance behind a class operation (getInstance)
- Variation: Since creation policy is encapsulated in getInstance, it is possible to vary the creation policy

Singleton Consequences

- Ensures only one (e.g., Database) instance exists in the system
- Can maintain a pointer (need to create object on first get call) or an actual object
- Can also use this pattern to control fixed multiple instances

Example1

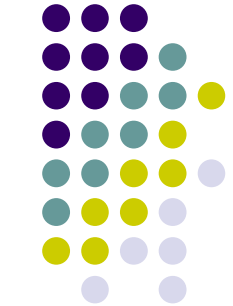
```
package SingleTon.Tel;

public class SingleTon {

    public static void main(String[] args) {

        abc obj1 = abc.getInstance();
        abc obj2 = abc.getInstance(); // new abc() is not possible as constructor is private

        //  abc obj3 = new abc();   // When constructor abc() is private this is not possible
        //  obj3.getInstance();     //

    }

}

class abc{

    public static abc obj = new abc(); // creating static object of class abc
    private abc() {  //creating the constructor as private so that only one instance can be created

    }

// to achieve Singleton pattern we have to create Static object and private constructor

    public static abc getInstance() {
        System.out.println("Hello World!");
        return obj;
    }

}
```

Eager SingleTon

```
public static abc obj = new abc();
```

- In reference to the above object creation, obj has been created and it will be on memory as it is ‘static’.
- Even the object ‘obj’ is not being used still it will be there in memory.
- Hence if ‘obj’ is heavily loaded with lots of data, high amount of memory and processing power will be consumed.
- This type of object creation is called as **‘Eager Initialization’** of SingleTon pattern.

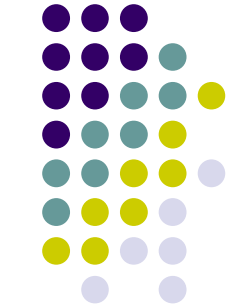
Eager -> Lazy SingleTon



- Let us try to make it ‘Lazy’…

```
package SingleTon.Lazy.Tel;

public class SingleTonLazyDemo {

    public static void main(String[] arg) {
        abc obj1 = abc.getInstance(); //to make Lazy do it first

    }

}

//to achieve Singleton pattern we have created Static object and private constructor

class abc{

    public static abc obj; // instead of creating it, just initialize here and create within getInstance()
    private abc() {
        System.out.println("Hello World!");
    }

    public static abc getInstance() {
        obj = new abc(); // create the object here...now you can say it is lazy
        return obj;
    }

}
```

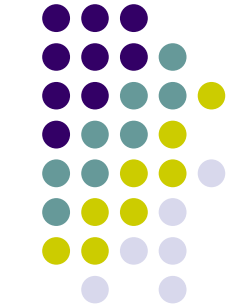
Problem…?



- In this situation you will be able to create more than one object, hence concept of SingleTon makes no sense…

```
package SingleTon.Lazy.Tel;

public class SingleTonLazyDemo {

    public static void main(String[] arg) {
        abc obj1 = abc.getInstance(); //to make Lazy do it first
        abc obj2 = abc.getInstance();
    }

}

//to achieve Singleton pattern we have created Static object and private constructor

class abc{

    public static abc obj; // instead of creating it, just initialize here and create within getInstance()
    private abc() {
        System.out.println("Hello World!");
    }

    public static abc getInstance() {
        obj = new abc(); // create the object here...now you can say it is lazy
        return obj;
    }

}
```

Solution…

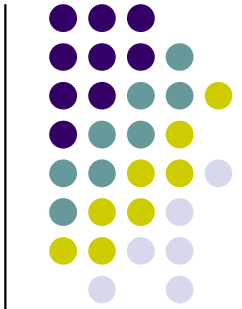


- Keep a condition checking before creating the object…

```
package SingleTon.Lazy.Tel;

public class SingleTonLazyDemo {

    public static void main(String[] arg) {
        abc obj1 = abc.getInstance(); //to make Lazy do it first
        abc obj2 = abc.getInstance();
    }

}

//to achieve Singleton pattern we have created Static object and private constructor

class abc{

    public static abc obj; // instead of creating it, just initialize here and create within getInstance()
    private abc() {
        System.out.println("Hello World!");
    }

    public static abc getInstance() {
        if(obj == null) {// this condition checking confirms that the object has not been created earlier
        obj = new abc(); // create the object here...now you can say it is lazy
        }
        return obj;
    }

}
```

Is it Enough…???

Synchronized getInstance…

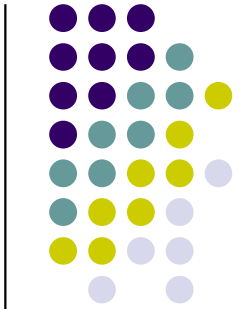


- Keep a condition checking before creating the object…

```
package SingleTon.Synchro.Tel;

public class SynchronizedGetInstance {

    public static void main(String[] arg) {
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                abc obj = abc.getInstance();
            }

        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                abc obj = abc.getInstance();
            }

        });
        t1.start();
        t2.start();
    }

}

//to achieve Singleton pattern we have created Static object and private constructor

class abc{

    public static abc obj; // instead of creating it, just initialize here and create within getInstance()
    private abc() {
        System.out.println("Hello World!");
    }

    public static abc getInstance() {
        if (obj == null) {
        obj = new abc(); // create the object here...now you can say it is lazy
        }
        return obj;
    }

}
```

Is it Enough…???

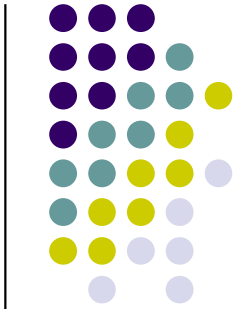
Synchronized getInstance…



- Again two object can be created here…How to solve?

```
package SingleTon.Synchro.Tel;

public class SynchronizedGetInstance {

    public static void main(String[] arg) {
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                abc obj = abc.getInstance();
            }

        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                abc obj = abc.getInstance();
            }

        });
        t1.start();
        t2.start();
    }

}

//to achieve Singleton pattern we have created Static object and private constructor

class abc{

    public static abc obj; // instead of creating it, just initialize here and create within getInstance()
    private abc() {
        System.out.println("Hello World!");
    }

    public static synchronized abc getInstance() {
        if (obj == null) {
        obj = new abc(); // create the object here...now you can say it is lazy
        }
        return obj;
    }

}
```

Double-checked Locking



- getInstance() is seemed to be heavily loaded…

```
package SingleTon.Synchro.Tel;

public class SynchronizedGetInstance {

    public static void main(String[] arg) {
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                abc obj = abc.getInstance();
            }
        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                abc obj = abc.getInstance();
            }
        });
        t1.start();
        t2.start();
    }
}

class abc{

    public static abc obj; // instead of creating it, just initialize here and create within getInstance()
    private abc() {
        System.out.println("Hello World!");
    }
    //public static synchronized abc getInstance() {
    public static abc getInstance() { //Double checked Locking -- remove synchronized from here and perform a double check
        if (obj == null) {
            synchronized (abc.class) { //put Synchronize here
                if (obj == null)
                obj = new abc(); // create the object here...now you can say it is lazy
            }
        }
        return obj;
    }
}
```

Enum SingleTon Pattern

- By default, the Enum instance is thread-safe, and you don't need to worry about double-checked locking.

```
package Singleton.Enum.Tel;

public class SingletonEnum {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        abc obj1 = abc.INSTANCE;
        System.out.println(obj1.getI());

        obj1.setI(2);
        System.out.println(obj1.getI());

        abc obj2 = abc.INSTANCE;
        obj2.setI(5);
        System.out.println(obj2.getI());

    }
}

enum abc {

    INSTANCE;
    int i;

    public int getI() {
        return i;
    }

    public void setI(int i) {
        this.i = i;
    }

}
```