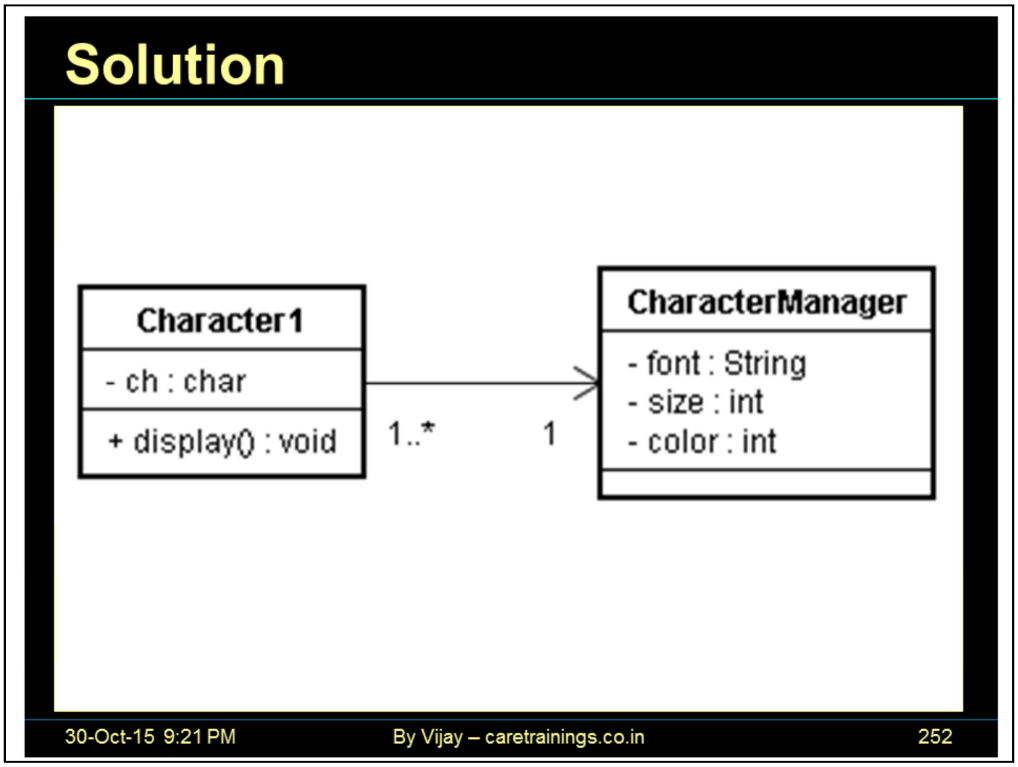
Flyweight DP

Problem

- We have a Character class. It stores information about one character.
  - Our document has a large number of characters.
  - Each character can have a different font and size.
  - How to reduce the memory requirements and runtime overheads?



Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

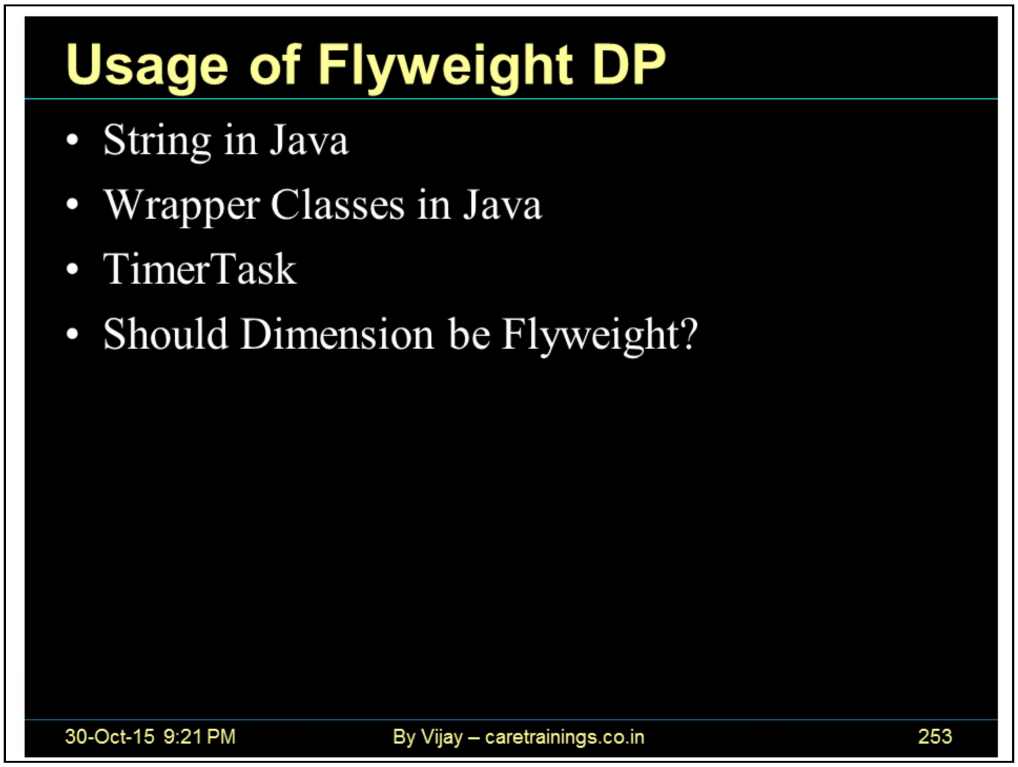
Flyweight performs “use sharing” to support large number of fine-grained objects efficiently.

A flyweight is a shared object that can be used in multiple contexts simultaneously.

==========

The Unicode of the characters can be stored in an array.

Many of the characters have the same attributes e.g. font, font size, color, etc. We can store these common attributes for a set of characters only once.

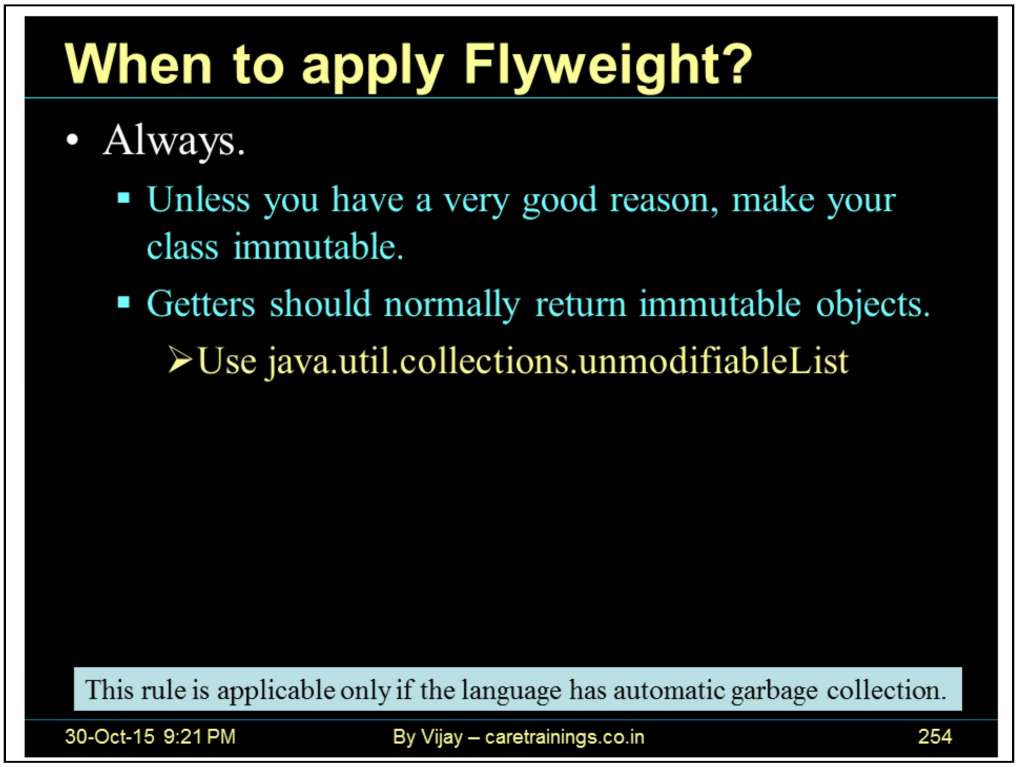


Java uses the Flyweight pattern to manage the String objects used to represent string literals.

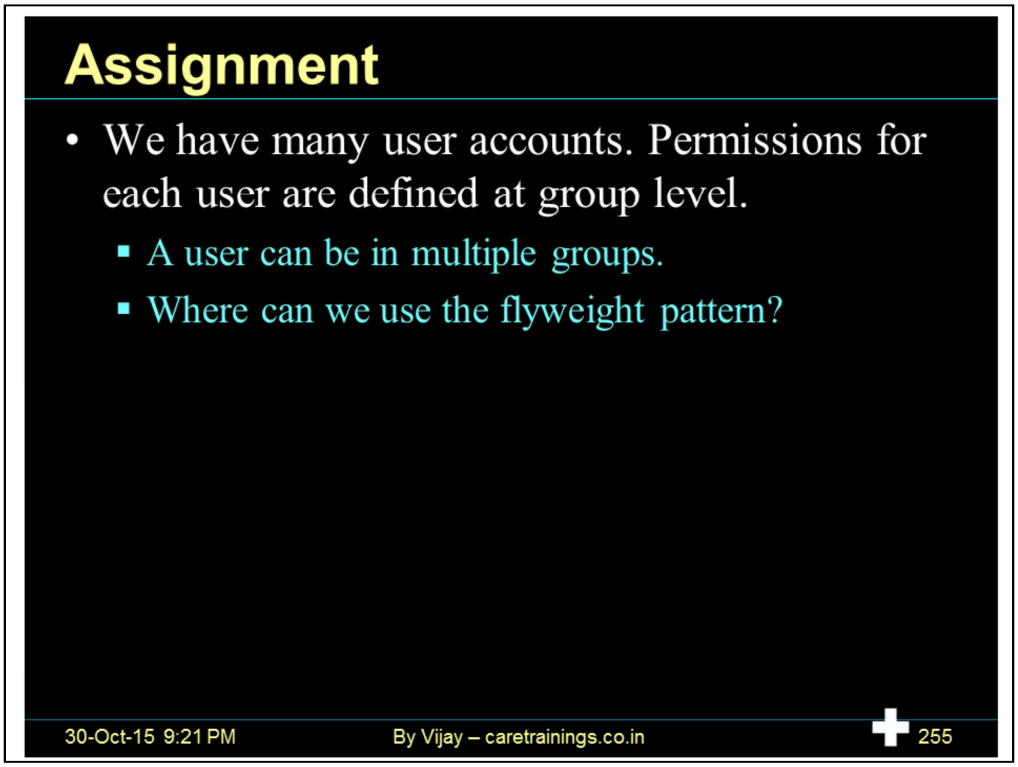
All wrapper classes are Flyweight

TimerTask

Dimension class should have been a Flyweight



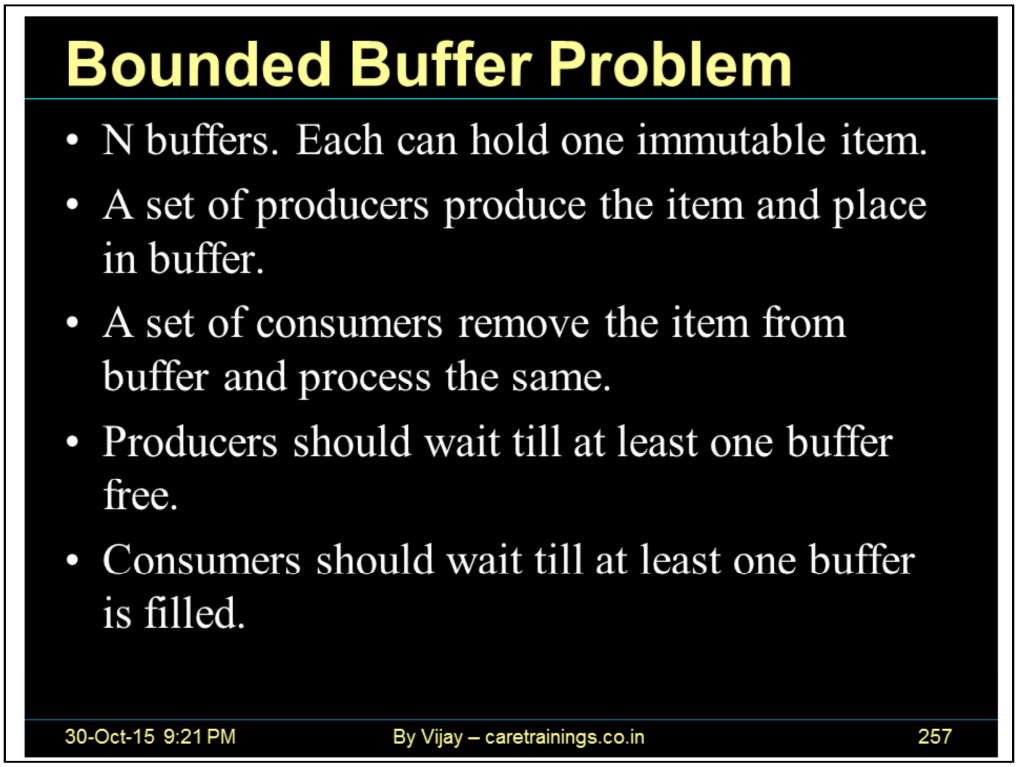
If you cannot make the entire class immutable, make maximum part immutable.
Limit mutability as much as you can.
The less mutable the better.
Final is the new private.



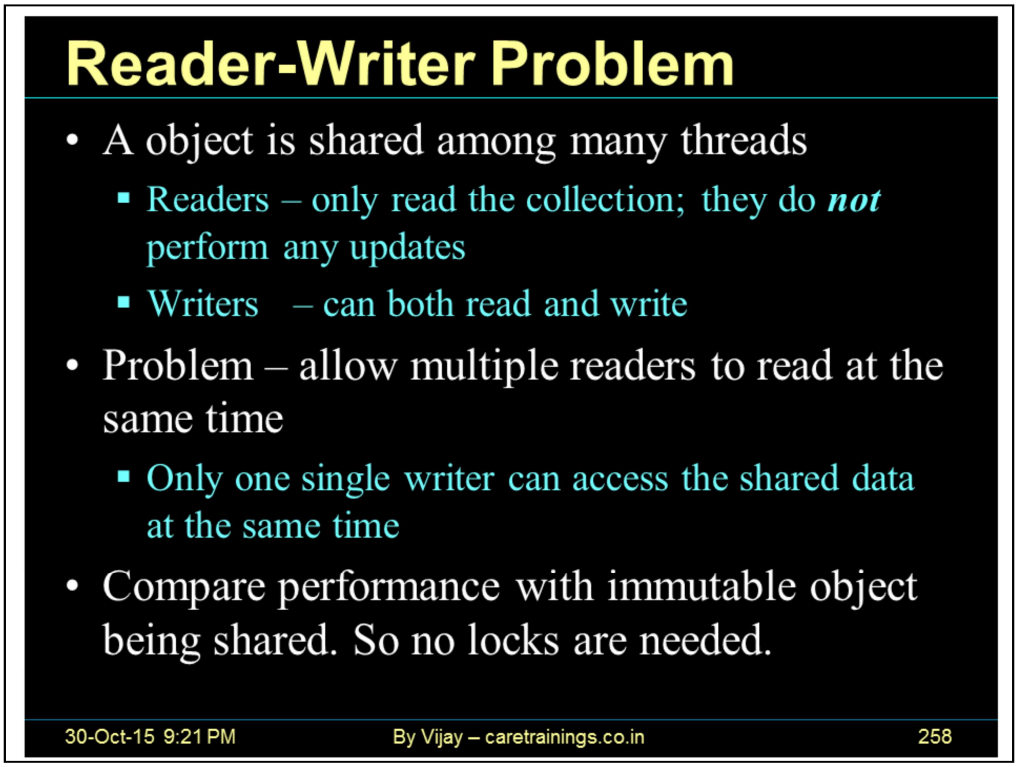
OOAD classes: Different classes (Immutable), Prefer Value Objects, Rules for Concurrency,

How many Flyweights?

- Java 8 has a new package java.time
  - What percentage of classes are Immutable?
  - If a mutable class exists, then justify its design.



Code this problem in Java. Assume a set of 3 threads produce ten million integers. A set of 2 threads consume these integers. Output should be how many evens and how many odds.



Shared Data

Data set

Semaphore **`rw_mutex`** initialized to 1

Semaphore **`mutex`** initialized to 1

Integer **`read_count`** initialized to 0

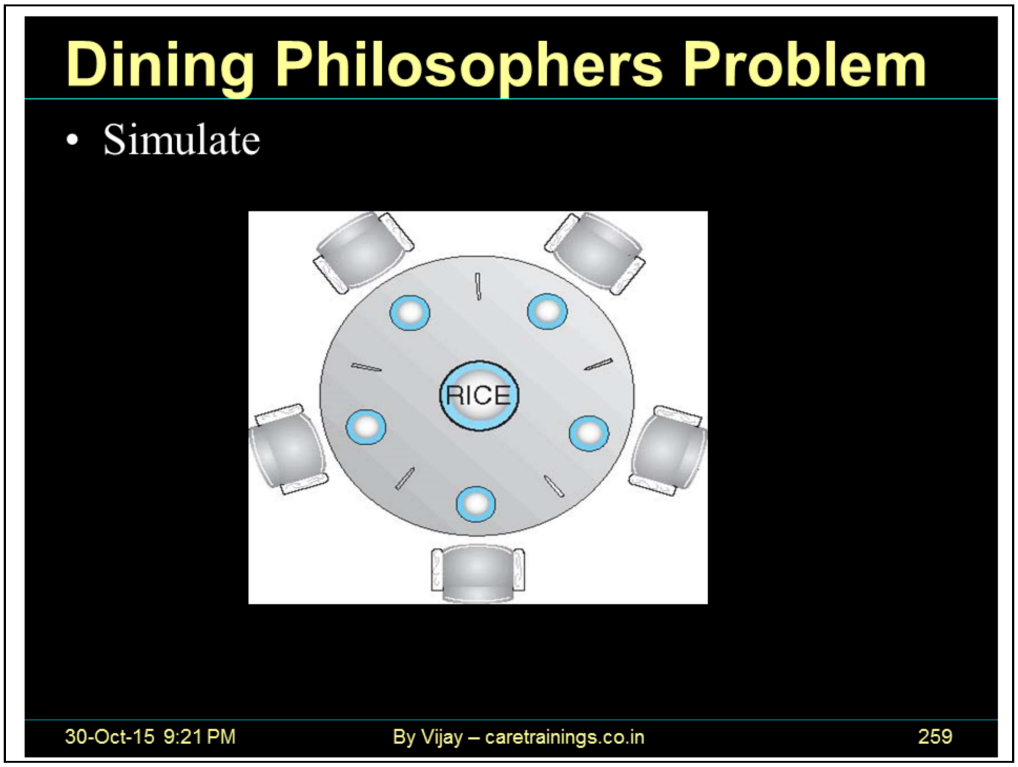
The structure of a writer process

```
do {
   wait(rw_mutex);
      ...
   /* writing is performed */
      ...
   signal(rw_mutex);
} while (true);
```

The structure of a reader process

```
do {
   wait(mutex);
   read count++;
   if (read_count == 1)
```

```
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```



Philosophers spend their lives thinking and eating

Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl

Need both to eat, then release both when done

In the case of 5 philosophers

Shared data

Bowl of rice (data set)

Semaphore chopstick [5] initialized to 1

The structure of Philosopher *i*:

```
do  {
        wait (chopstick[i] );
          wait (chopStick[ (i + 1) % 5] );

                // eat

          signal (chopstick[i] );
          signal (chopstick[ (i + 1) % 5] );
```

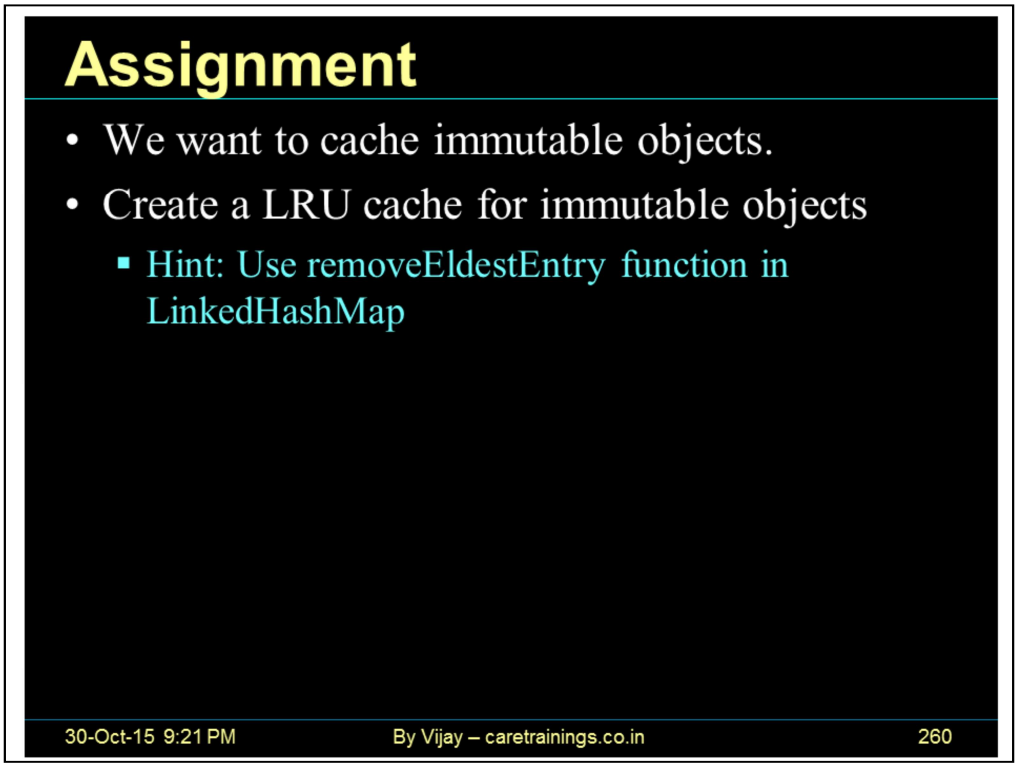
```
                                    //  think

            } while (TRUE);
```

What is the problem with this algorithm?

The last philosopher must take spoons in reverse order, otherwise there is a deadlock.

Moral: Rice is used only one thread. So no locking. Spoon is like a mutable object; only one thread can use it.

A large application will have some mutable objects shared. First try to make it immutable. If that is not possible, then ensure locks are acquired in same order.



Java Solution in Patterns project FlyweightCache.