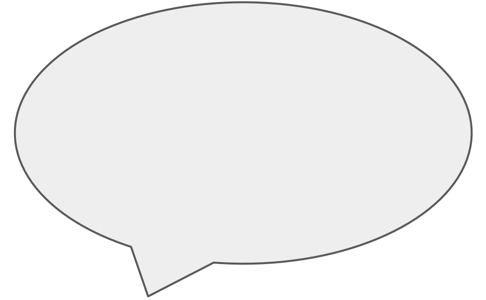
Object Oriented Programming Examples class 2

December 2021

Professor Andrew Rice

These slides are on the course website if you want to follow along.
(Some of the code is necessarily smaller than ideal for presenting.)

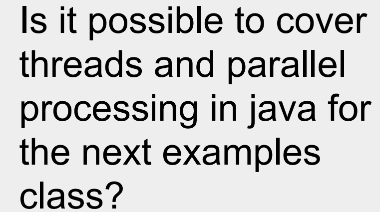
Chime exercise improvements



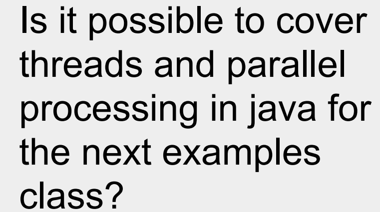
[FEEDBACK] The Matrices task is a little bit weird. We're asked to implement a rotation2D matrix that presumably rotates shapes and points in the anti-clockwise direction. The TextDrawing.plot method takes in a matrix with height 2 with the following specification "Elements in row 0 are y co-ordinates, elements in row 1 are x co-ordinates". This is a y-x column vector rather than an x-y column vector, so theoretically the matrix should not be the same as the rotation matrix used for an x-y column vector. But it turns out that it is the latter that is correct rather than the former, because the TextDrawing.plot method does not use the Cartesian coordinate system -> y increases in the downwards direction. So you have to substitute y with -y, and 'cancels' out the effect of using a y-x column vector: i.e. you can use the same matrix you would use with an x-y column vector and a Cartesian coordinate system. This took me quite a while to figure out and it might be helpful to clarify it in the problem description.

Too far off topic (sorry)

Good news! You do this in depth next year in Concurrent and Distributed Systems and the Further Java course.



Too far off topic (sorry)



Good news! You do this in depth next year in Concurrent and Distributed Systems and the Further Java course.

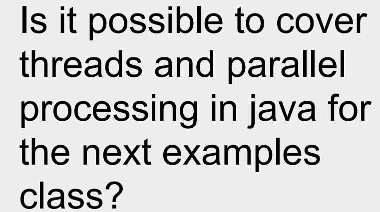
Great news! I co-wrote and co-teach Further Java.

Too far off topic (sorry)

Good news! You do this in depth next year in Concurrent and Distributed Systems and the Further Java course.

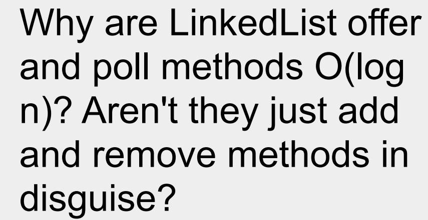
Great news! I co-wrote and co-teach Further Java.

Changing news! I'm not teaching it next year so the course might change.

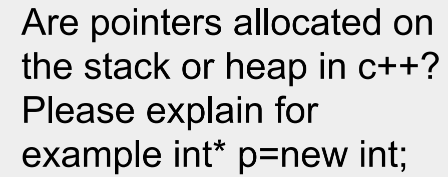


Errors in the notes

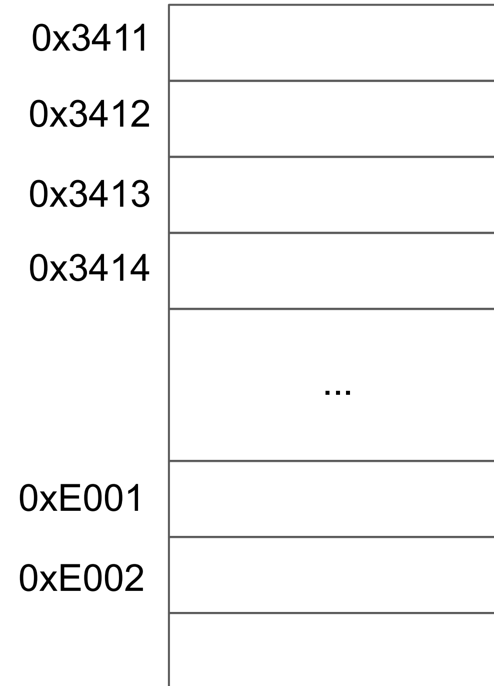
You are right. They should be O(1) and I got it wrong on the handout.



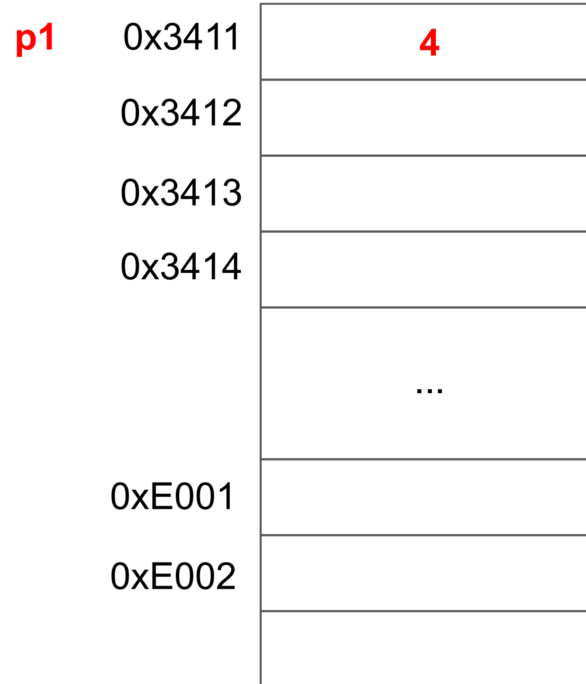
C++ pointers



```
int p1 = 4;
int* p2 = &p1;
int* p3 = new int;
*p3 = p1;
int** p4 = new int*;
*p4 = p2;
```



```
int p1 = 4;
int* p2 = &p1;
int* p3 = new int;
*p3 = p1;
int** p4 = new int*;
*p4 = p2;
```

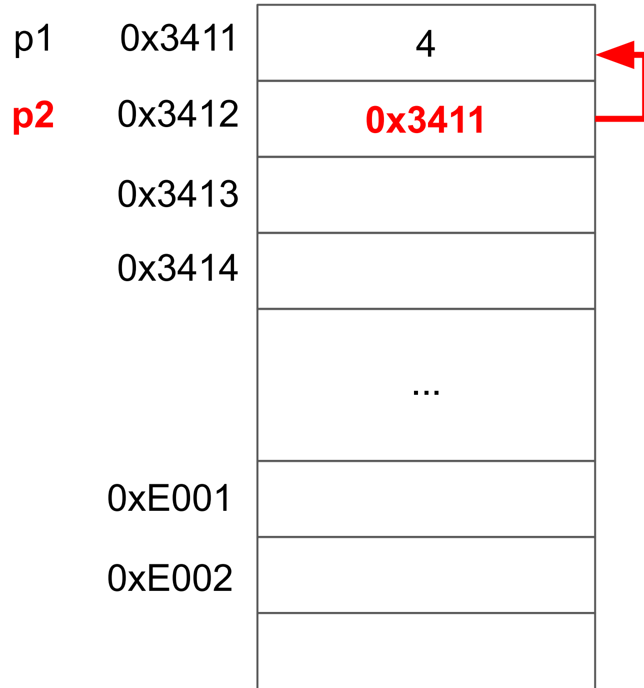


An asterisk on a type means 'pointer'

```
int* p1 = 4;
int* p2 = &p1;
int* p3 = new int;
*p3 = p1;
int** p4 = new int*;
*p4 = p2;
```

An ampersand on a r-value means 'take the address of'

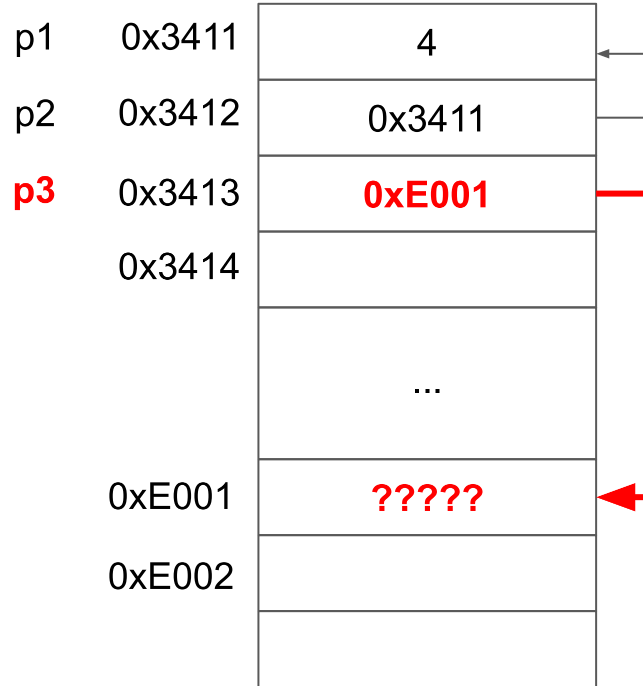
An ampersand on a type means '(C++) reference'



'new' means 'on the heap'. This is true in Java too...

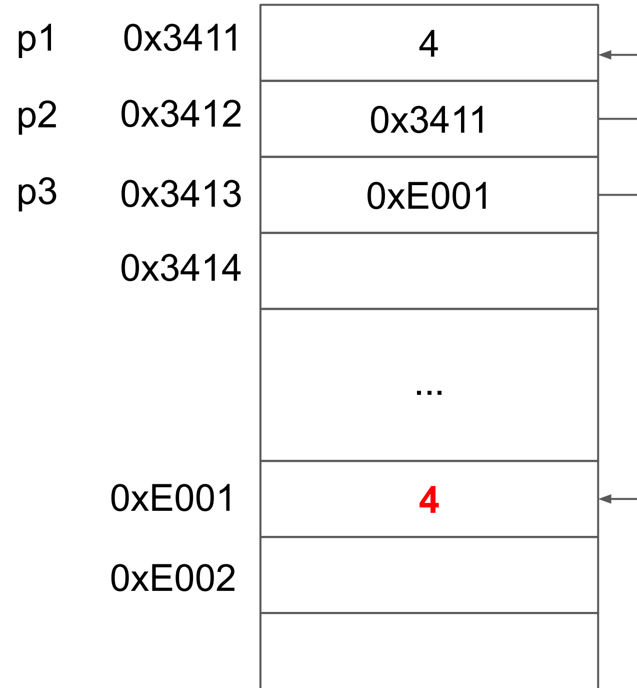
```
int p1 = 4;
int* p2 = &p1;
int* p3 = new int;
*p3 = p1;
int** p4 = new int*;
*p4 = p2;
```

Note the lack of brackets... we are not calling a constructor because this is not an object. Can't do this in Java.



```
int p1 = 4;
int* p2 = &p1;
int* p3 = new int;
*p3 = p1;
int** p4 = new int*;
*p4 = p2;
```

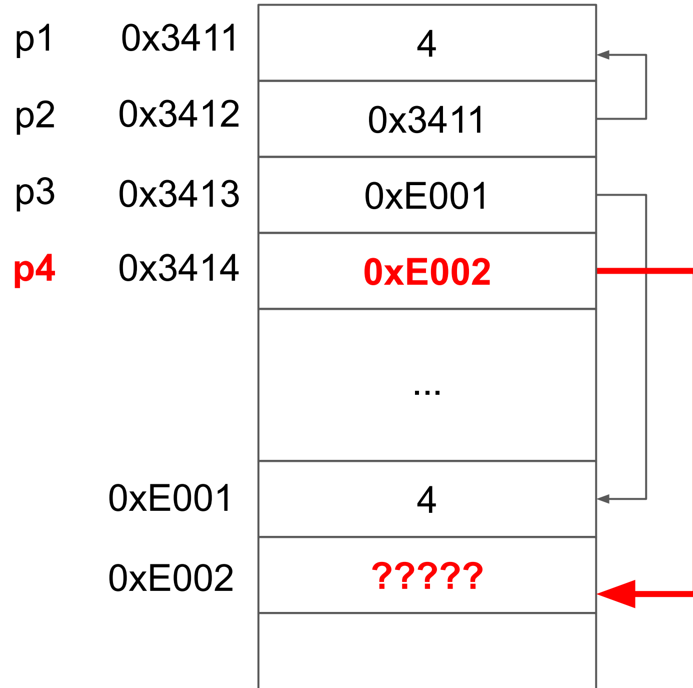
An asterisk on a variable means dereference (follow pointer)



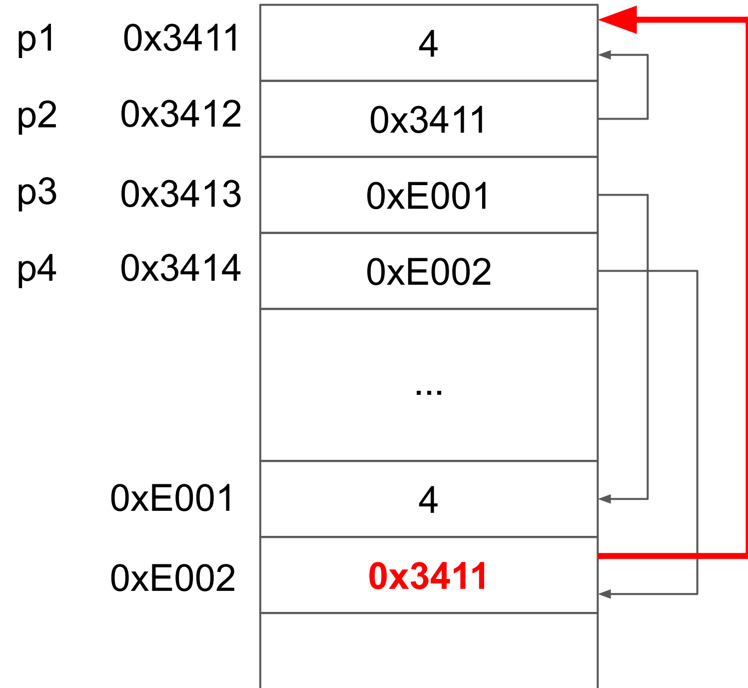
```
int p1 = 4;
int* p2 = &p1;
int* p3 = new int;
*p3 = p1;
int** p4 = new int*;
*p4 = p2;
```

p4 points to something on the heap and the thing it points to is another pointer which in turn points to an int.

p4 is an pointer to a pointer to an int.

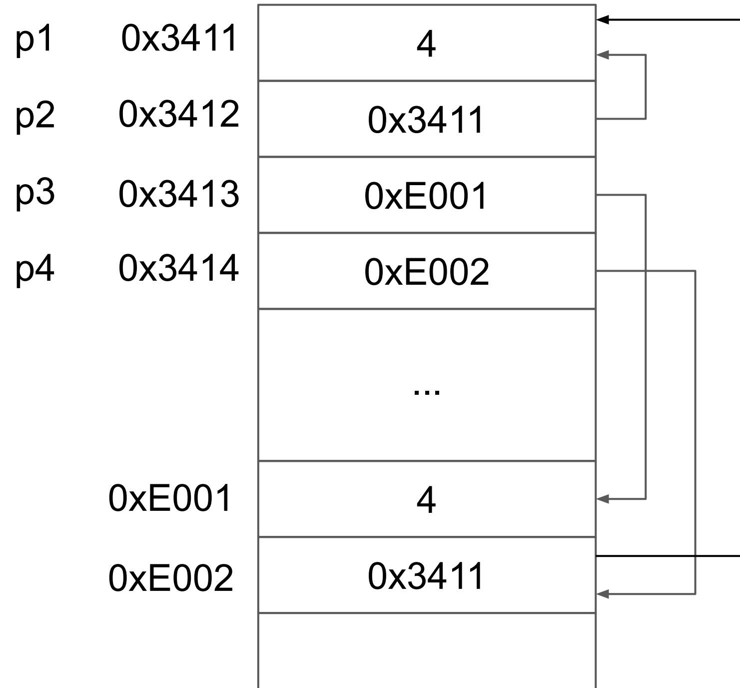


```
int p1 = 4;
int* p2 = &p1;
int* p3 = new int;
*p3 = p1;
int** p4 = new int*;
*p4 = p2;
```



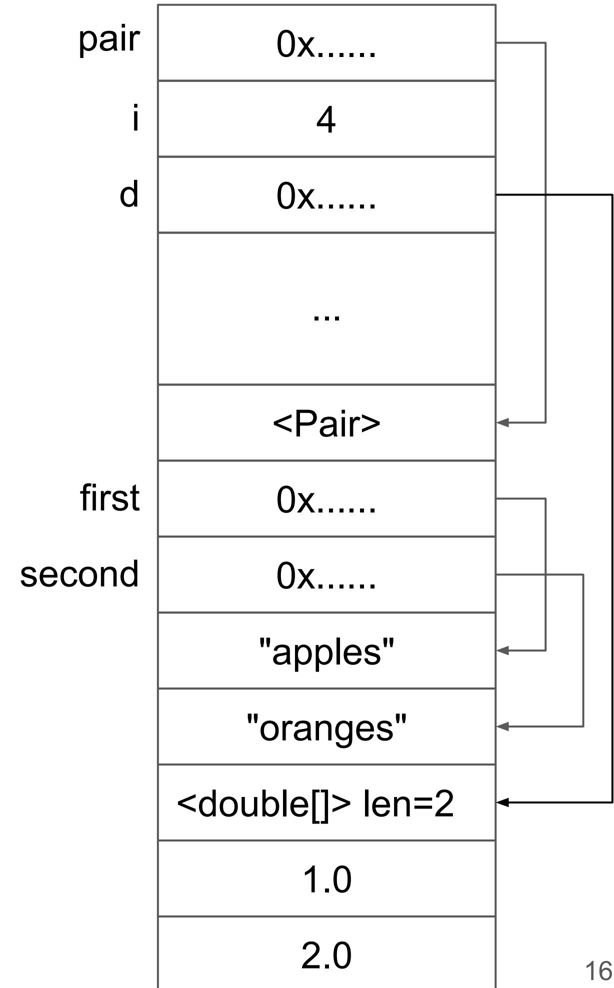
```
int p1 = 4;
int* p2 = &p1;
int* p3 = new int;
*p3 = p1;
int** p4 = new int*;
*p4 = p2;
```

****p4 == 4**



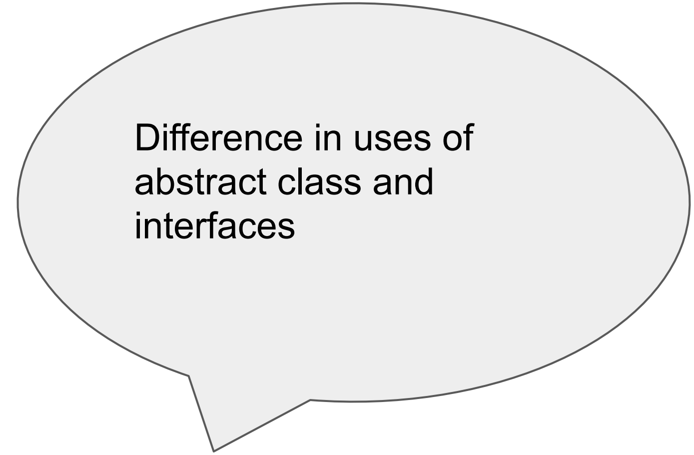
Java is more restrictive

	Java	**C++**
Objects	Heap only	Stack or Heap
Primitives	Stack only or on heap as field of an object or array element	Stack or heap
References / Pointers	Can only reference heap	Can point anywhere



Abstract classes and interfaces

	Abstract class	Interface
Multiple inheritance	No	Yes
Inherit code	Yes	Yes (default methods)
Inherit state (fields)	Yes	No (there is no state)
Inherit type	Yes	Yes



Comparable is a good example of an interface

```
class College implements Comparable<College> {
 private final String name;
 private final int foundingYear;

 College(String name, int foundingYear) {
   this.name = name;
   this.foundingYear = foundingYear;
 }

 public String name() { return name; }

 public int foundingYear() { return foundingYear; }

 @Override
 public int compareTo(College other) {
   return Comparator
       .comparing(College::name)
       .thenComparing(College::foundingYear)
       .compare(this, other);
 }
}
```

Comparable is a good example of an interface

```
List<College> colleges = ...
Collections.sort(colleges);
```

Comparable is a good example of an interface

```
public class Collections {
  // ...
  public static <T extends Comparable<? super T>> void sort(List<T> list) {
     // ...
  }
}
```

T must implement Comparable<? super T>

For example, if T == College then it can implement

- Comparable<College>
- Comparable<Object>

```java
class College implements Comparable<Object> {
  private final String name;
  private final int foundingYear;

  College(String name, int foundingYear) {
    this.name = name;
    this.foundingYear = foundingYear;
  }

  public String name() { return name; }

  public int foundingYear() { return foundingYear; }

  @Override
  public int compareTo(Object other) {
    return Comparator
        .comparing(Object::toString)
        .compare(this, other);
  }
}
```

```java
List<College> list = ...
College c1 = list.get(0);
College c2 = list.get(1);
if (c1.compareTo(c2) < 0) {
 // ...
}
```

Collection is a good example of default methods

Collection specifies some methods which can be

a) widely used
b) built by calling other methods in collection..no state is needed...

```
default boolean removeIf(Predicate<? super E> filter) {
   Objects.requireNonNull(filter);
   boolean removed = false;
   final Iterator<E> each = iterator();
   while (each.hasNext()) {
       if (filter.test(each.next())) {
           each.remove();
           removed = true;
       }
   }
   return removed;
}
```

Calendar is an example of an abstract class

```
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.YEAR, foundingYear);
calendar.add(Calendar.DAY_OF_YEAR, -1);
Date time = calendar.getTime();
```

Calendar contains state and a lot of functionality

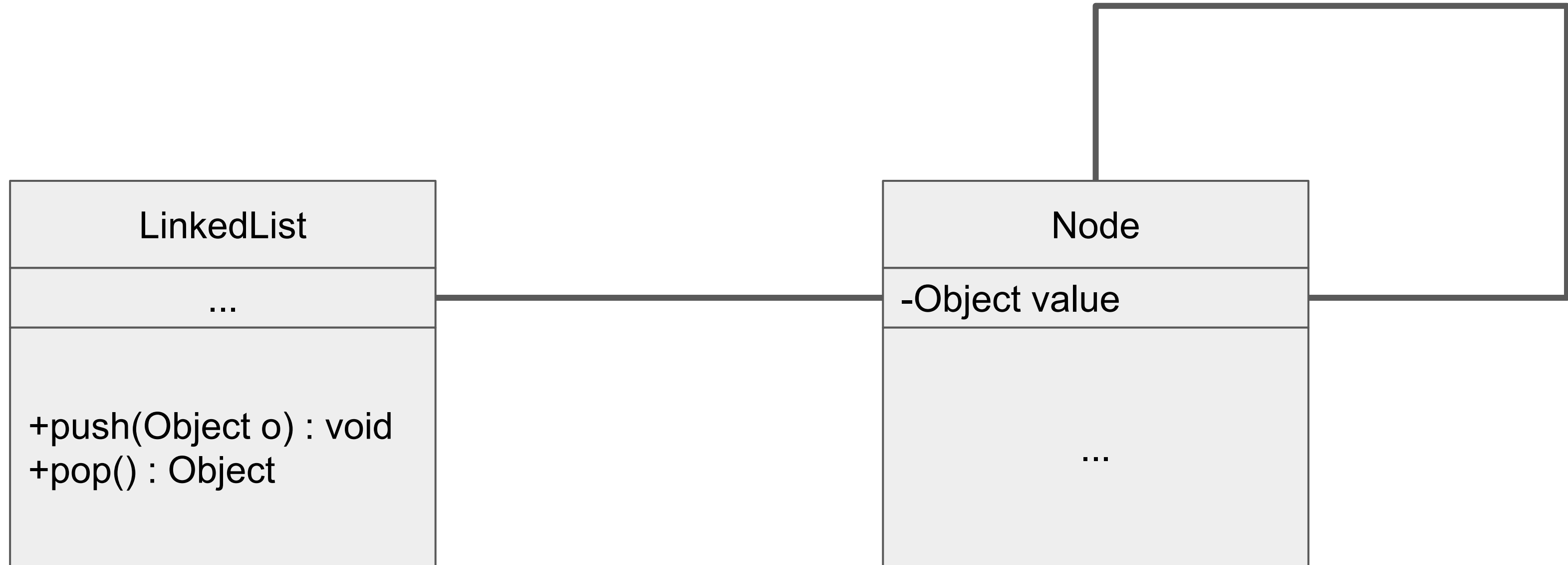
But there are details that vary between calendars

Subclasses for GregorianCalendar and JapaneseImperialCalendar

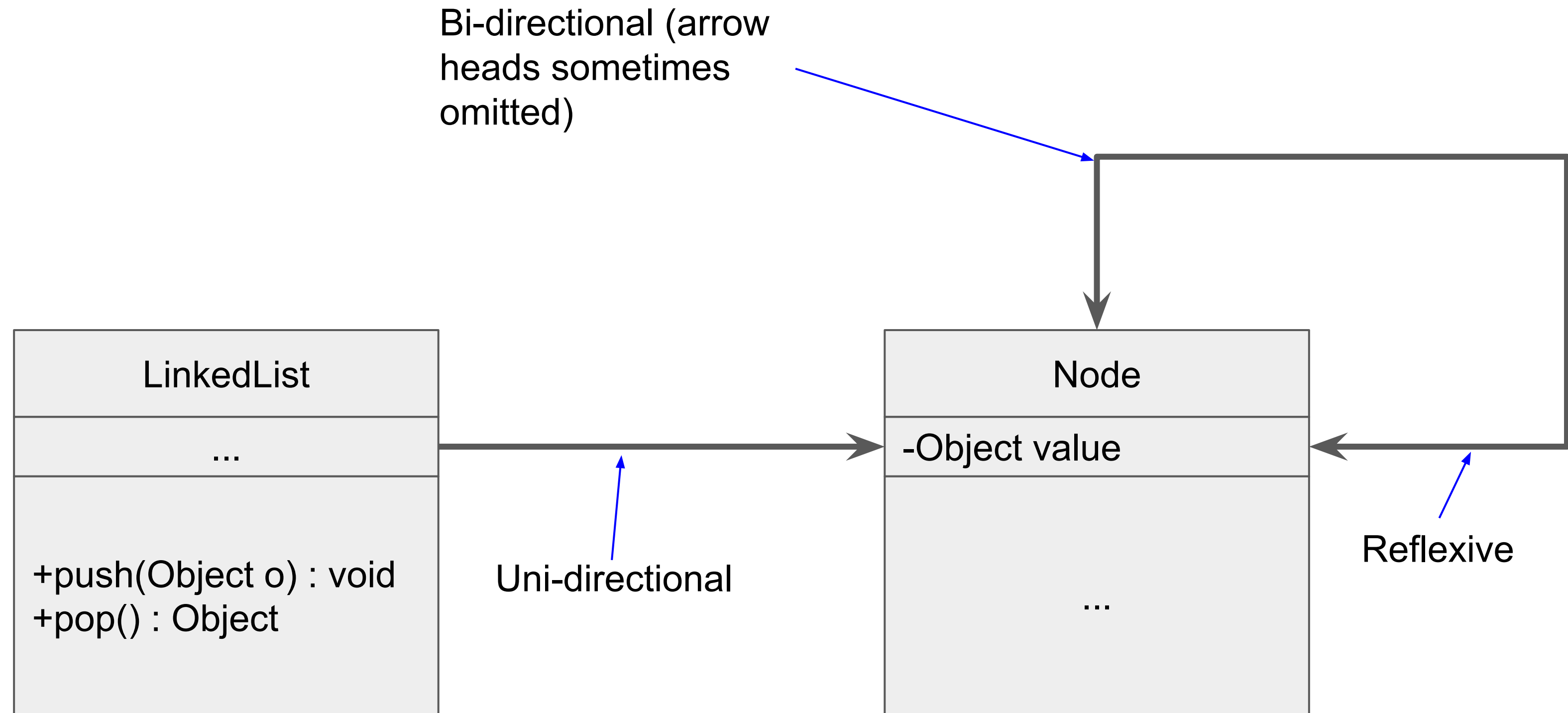
Creates close coupling of implementations

Extra content on UML

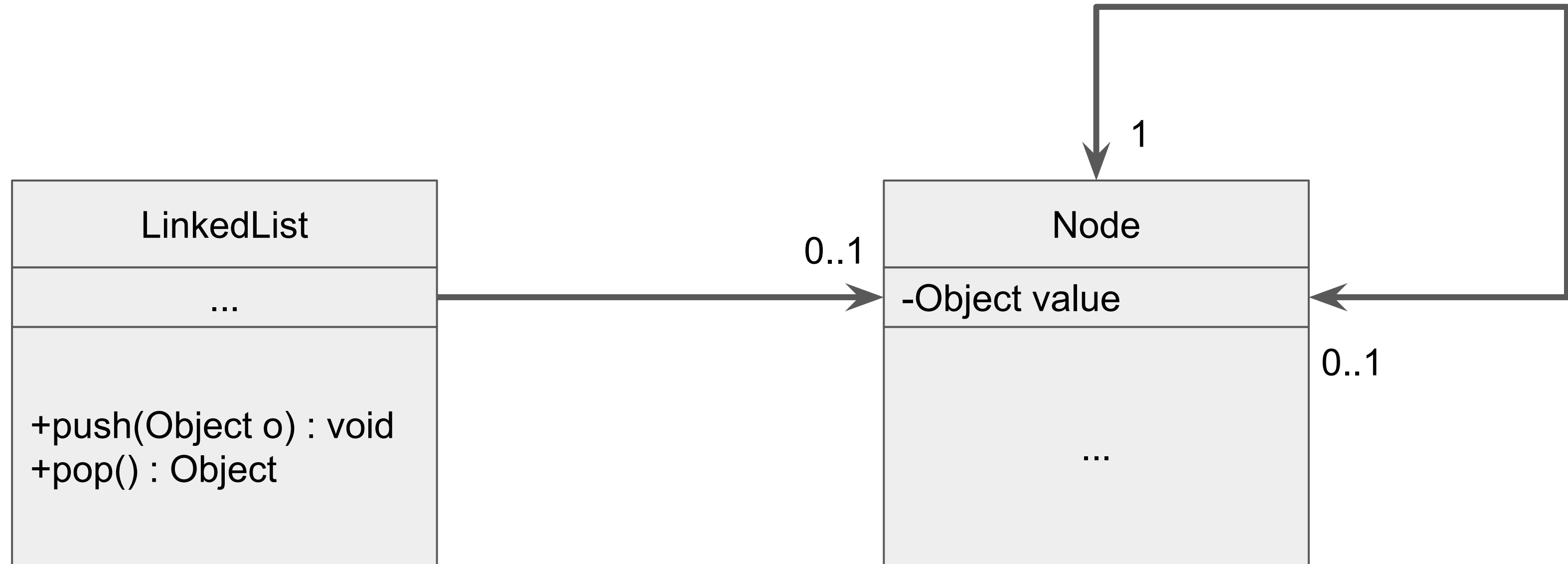
Any line means 'some relationship'



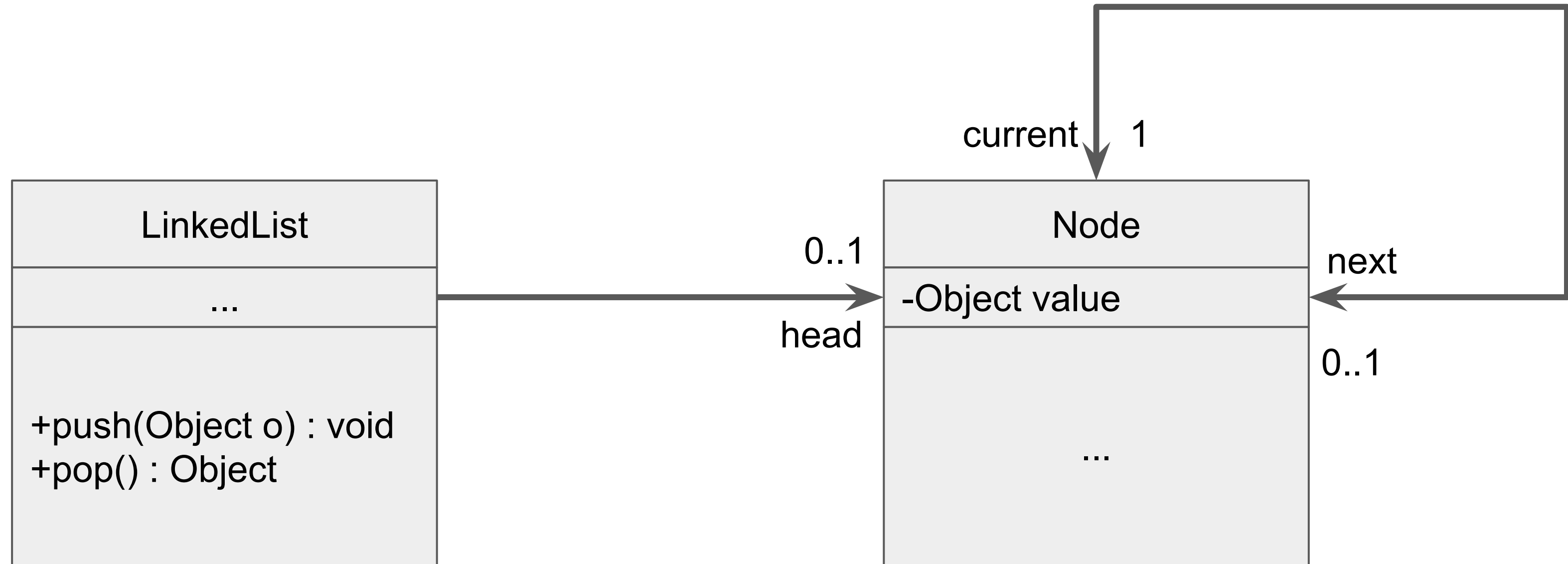
Open arrow means Association or 'knows about'



Numbers indicate multiplicity

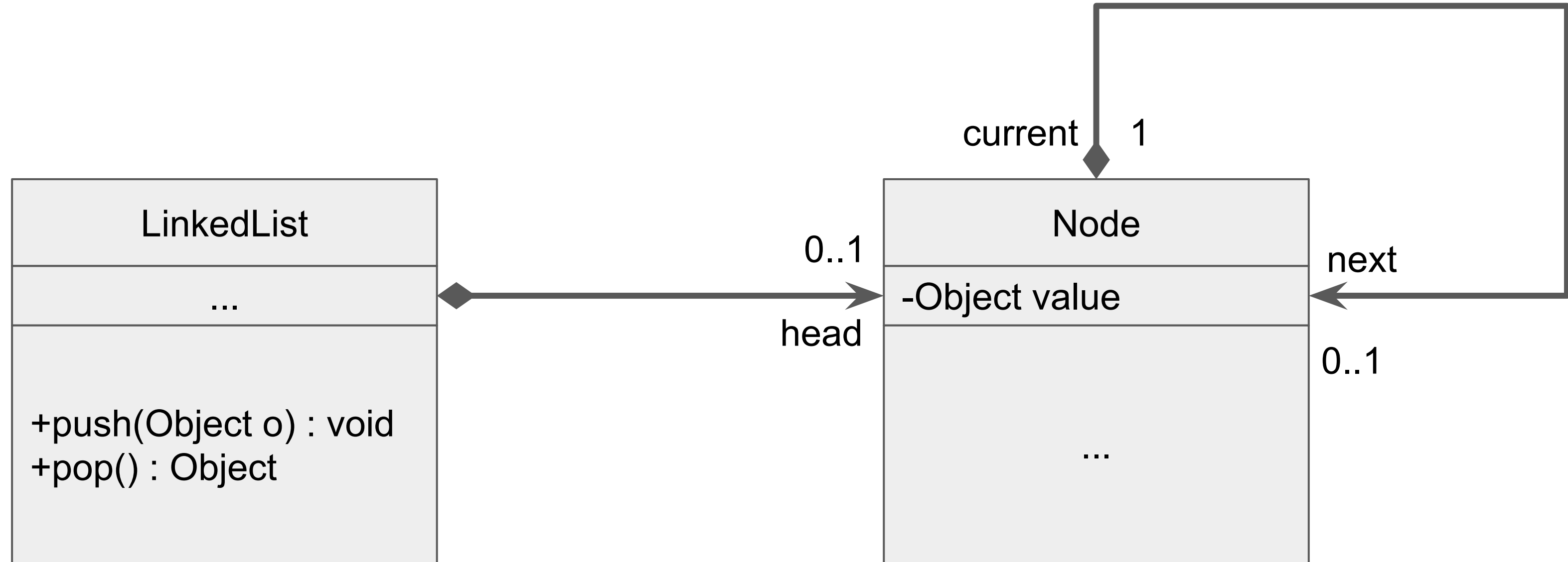


You can also indicate role

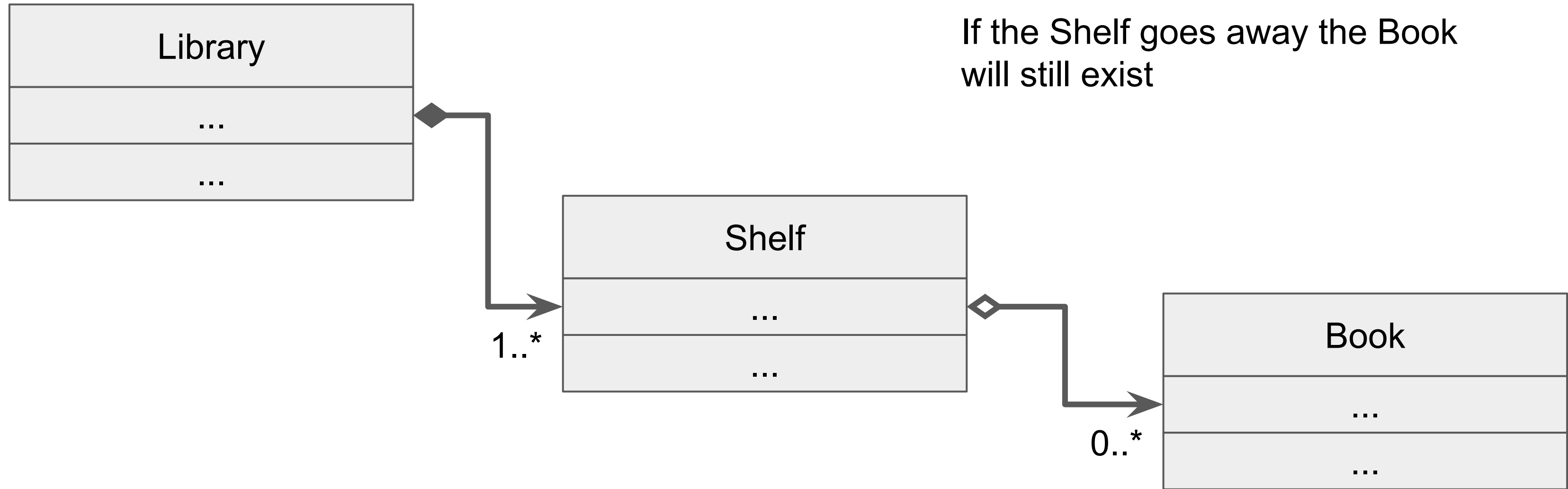


Solid diamond means Composition or 'owns a'

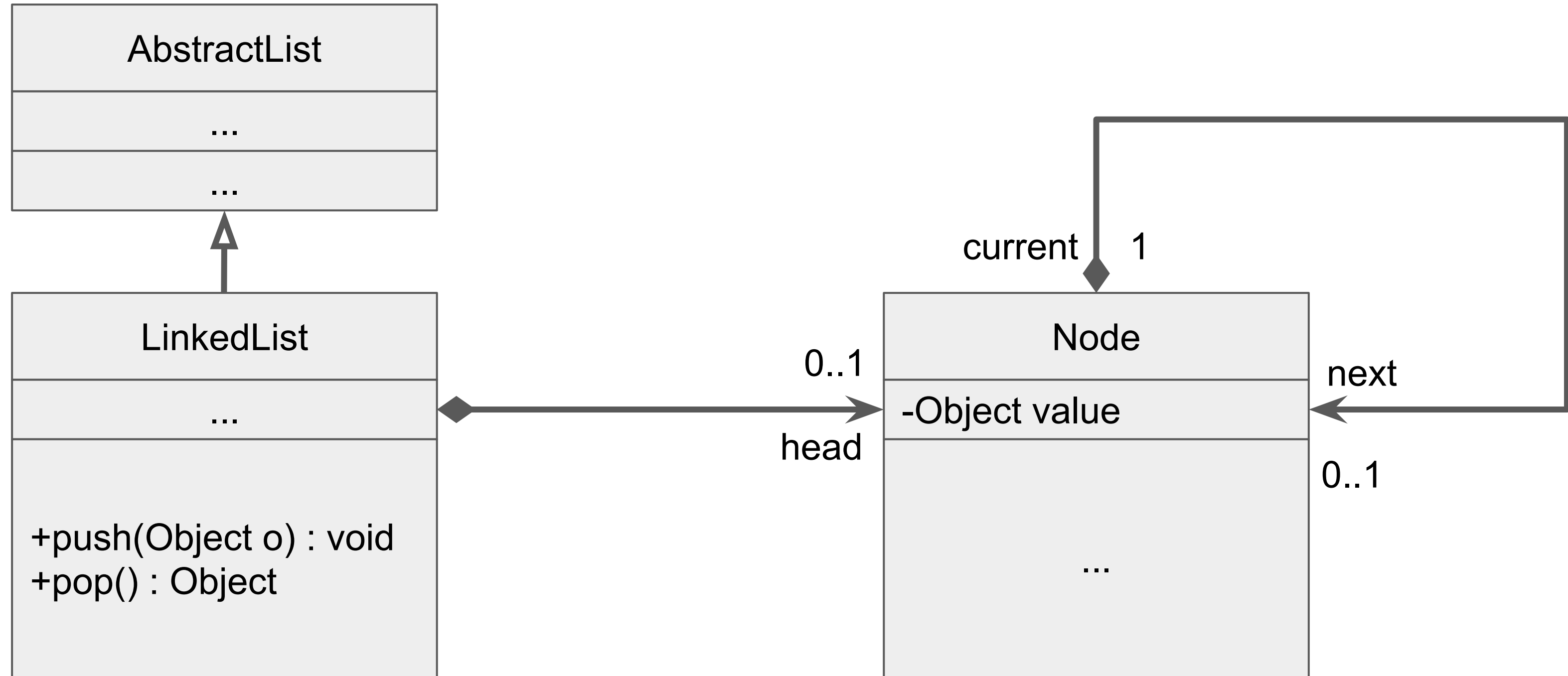
When the instance of LinkedList
is destroyed so is the Node



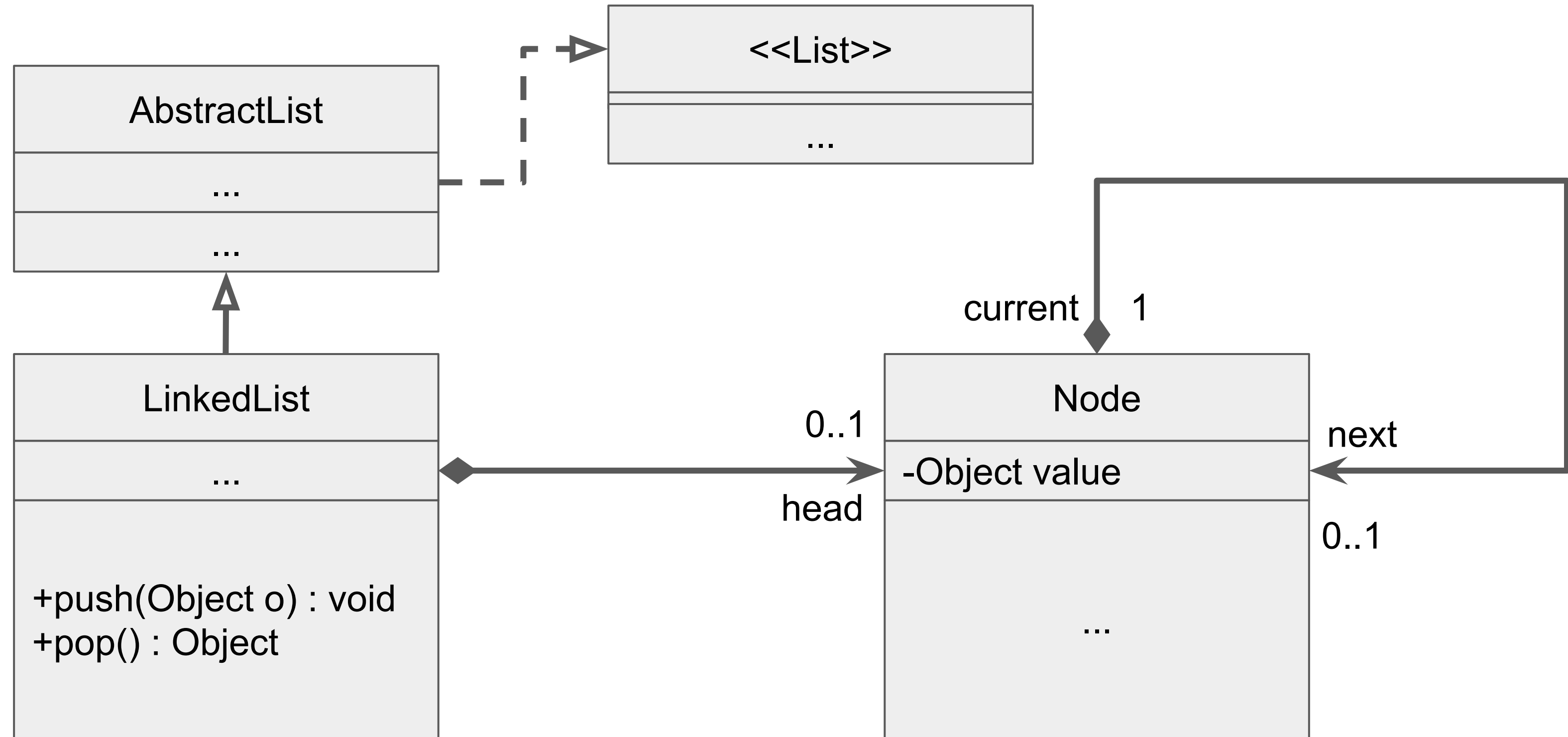
Empty diamond means Aggregation or 'has a'



Empty triangle means Generalisation or 'extends'



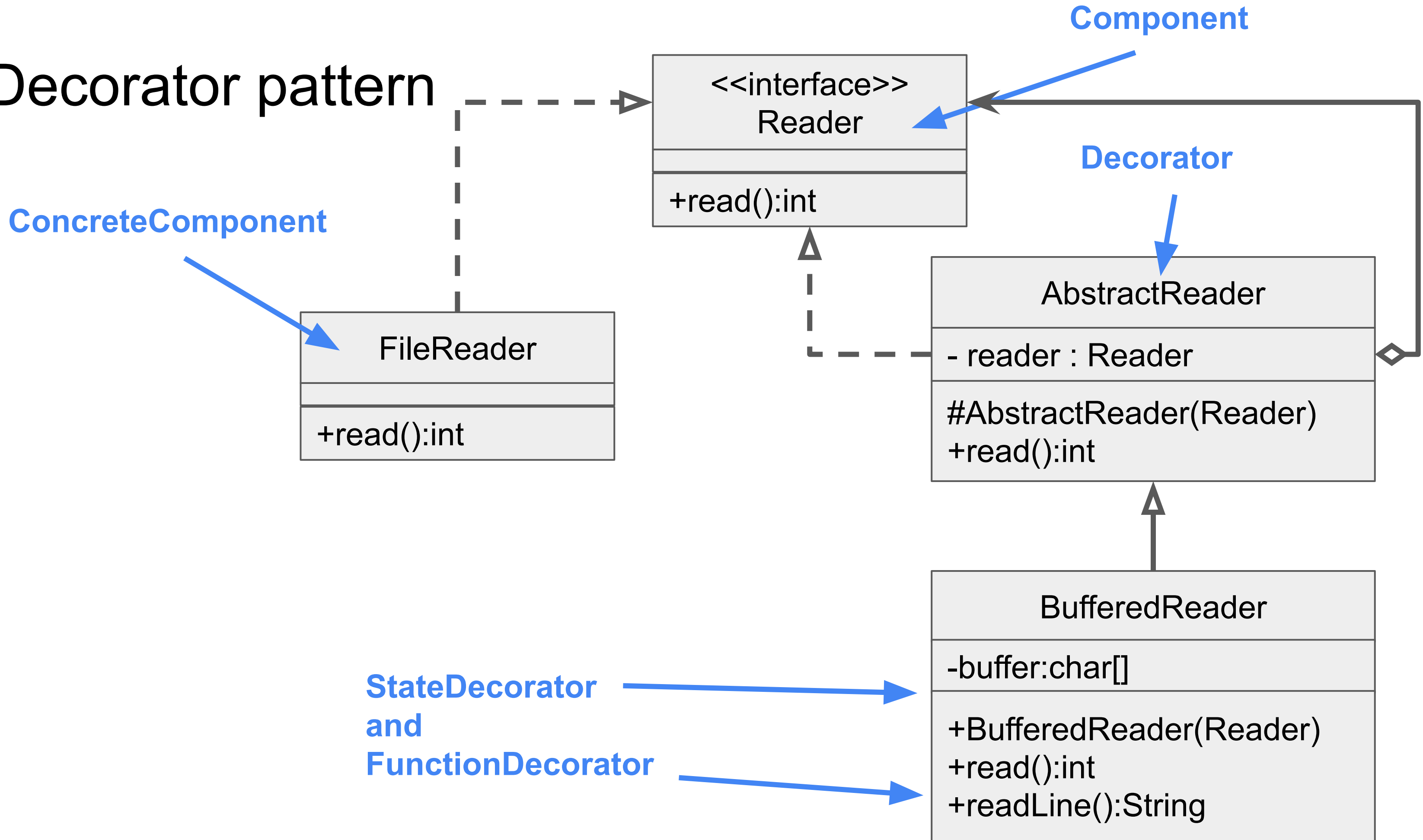
Dashed open triangle means Realises or 'implements'



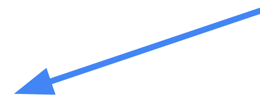
UML Example: decorator pattern

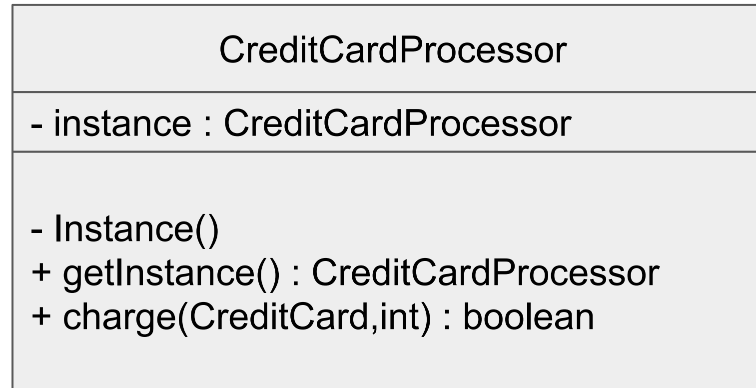
```
try (BufferedReader r =
            new BufferedReader(
                    new FileReader("/tmp/andy.txt"))) {
 String line;
 while ((line = r.readLine()) != null) {
   System.out.println(line.toUpperCase());
 }
}
```

Decorator pattern

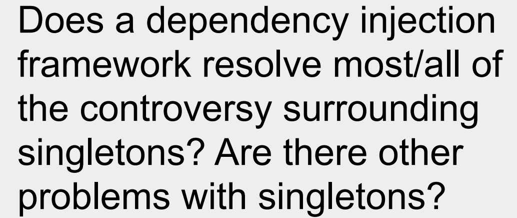


Singleton Pattern





Singleton Pattern



Problems

Restricts you to a single instance and a single implementation

Relies on global state (the static field)

Violates single-responsibility principle

I would argue these are all problems. The controversy is about how bad they are for your program compared to alternative approaches

Dependency injection

```java
public class Timer {

 private final Clock clock;
 private Instant startTime;

 Timer() {
   this.clock = Clock.systemUTC();
 }

 void start() {
   startTime = clock.instant();
 }

 Duration elapsed() {
   return Duration
      .between(startTime,
               clock.instant());
 }
}
```

```java
public class Timer {

 private final Clock clock;
 private Instant startTime;

 Timer(Clock clock) {
   this.clock = clock;
 }

 void start() {
   startTime = clock.instant();
 }

 Duration elapsed() {
   return Duration
      .between(startTime,
               clock.instant());
 }
}
```

Not examinable

Dependency Injection Framework

```
@Singleton
public class Timer {

 private final Clock clock;
 private Instant startTime;

 @Inject
 Timer(Clock clock) {
   this.clock = clock;
 }

 void start() {
   startTime = clock.instant();
 }

 Duration elapsed() {
   return Duration
      .between(startTime,
               clock.instant());
 }
}
```

```
class TimerModule extends AbstractModule {
 @Override
 protected void configure() {
   bind(Clock.class)
     .toInstance(Clock.systemUTC());
 }
}
```

```
Injector injector =
   Guice.createInjector(new TimerModule());
Timer t1 = injector.getInstance(Timer.class);
//...
Timer t2 = injector.getInstance(Timer.class);
```

Singleton Pattern with Dependency Injection Framework

Problems

Restricts you to a single instance and a single implementation

Solved through use of different modules and different injectors

Relies on global state (the static field)

Solved through use of (multiple) instances of injectors

Violates single-responsibility principle

Solved through use injection framework managing lifecycle

But: new problems created...e.g. hard to understand control flow of object instantiation, cycles in your graph will cause crashes at runtime

Course summary

How to write a Java program

Access modifier rules, extends, implements, abstract classes, interfaces, static, final, unit testing framework

OOP Concepts

Abstraction, Encapsulation, Inheritance, Polymorphism, UML

How to write a 'good' program

Testability, Dependency injection, Immutability, Open-closed principle, Design patterns