State

Slide Series 3

Content

Handle state

Fundamentals

No state

Immutability

Inner classes

Freeze/reduce state

Readonly

Singleton design pattern

Packages

Class invariants

Strategies

Some strategies to handle state

- Fundamental techniques
- Runtime support
- No state
- Immutable state
- Frozen state
- Readonly
- Limit number of objects
- Controlling references
- Proving, reasoning, identify restriction on state.
- Defining and preserving valid state
- Runtime support

Fundamental techniques

- No redundant instance variables
- If possible move instance variables to local variables
- Prefer side effect free function
- Differentiate between accessors and mutators
- Remove unused objects (accepted usage of null)
  - Example: Unused elements in arrays

Runtime Support

Java runtime support for state

"All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an ArrayIndexOutOfBoundsException to be thrown."

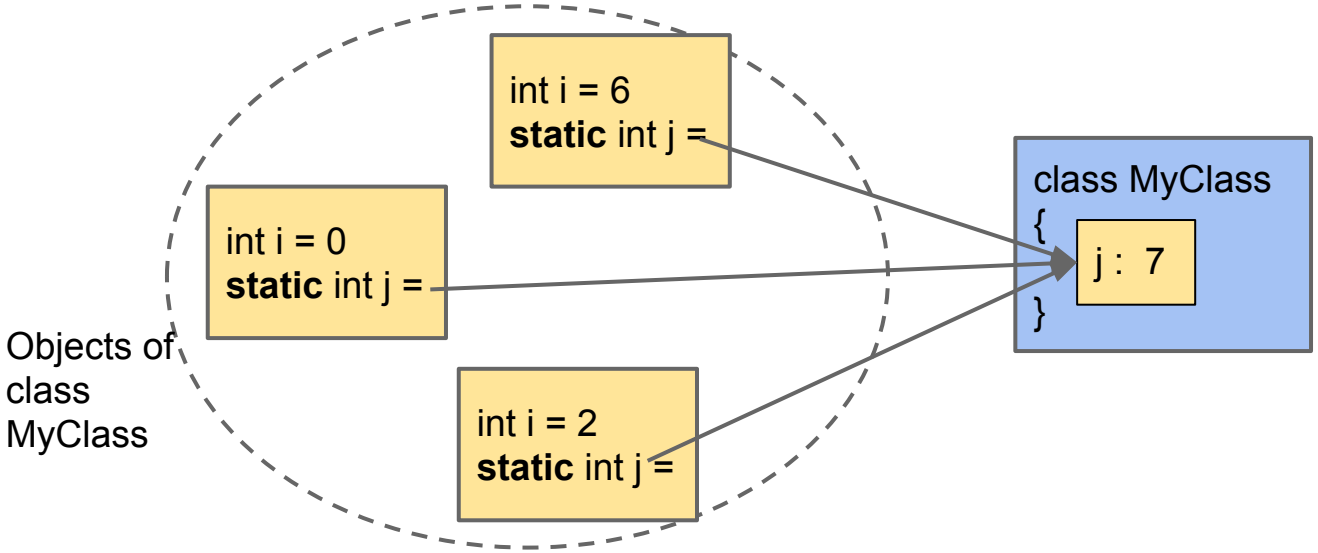
// JLS 10.4

- Best Practices: During development program should crash fast and loud
- Exception better than invalid state (possible shows up much later, harder to find)

Instance and Class variables

An instance variable belongs to an object, each object has it's own "copy". Part of state for object.

A **class variable** (**static**) is shared among all instances



Instance and Class methods*

Class methods can only access class variables

- No need for objects (possible object reference not used)

Instance methods can access instance and class variables

- Must have object

Static Classes*

Possible with "static" classes

- No writable attributes (so no side effects)
- All methods static (i.e. class methods)
- Private constructor (can't create instances)

... no state!

Usage for Static Classes*

Static classes are an exception, avoid! Possible usage

- Collections of "universal" functions, like java.util.Math
- Static factories, classes responsible for constructing other objects, more to come...

Objections

- Implementation fixed (can't override)
- No object to pass around (as parameter) i.e. direct dependency on implementation
- ... also we normally need state

Static Override and Overload*

As note, no override for static (override is about instances)

Overload possible

- Also possible mixing overloaded instance and class methods

There's also something called **hiding**, see inheritance

Immutability*

Some possibilities (in general)

- Immutable classes, all instances are immutable
- Immutable objects, some instances may be mutable others not
- Immutable references, can only be used to read
  - The C++ keyword **const**

Only immutable classes supported by Java (in this course we don't distinguish between immutable classes/objects...)

final

*"Fields declared **final** are initialized once, but never changed under normal circumstances" //* JLS 17.5

Note: If a reference variable is final it's possible to update the referenced object!

```
// Funny C++ example (const similar to final)
const int* const someMethod(const int* const&) const;
```

Immutable Class*

Must obey

- Any fields that contain references to mutable objects, such as arrays, collections, or mutable classes like Date:
  - Are private final (no mutators)
  - Are never returned or otherwise exposed to callers
  - Are the only reference to the objects that they reference
  - Do not change the state of the referenced objects after construction (some exceptions to this)
- The class is declared final (can't inherit from class, more later)
- No super class can change state (or super is Object)

Immutable Instances*

Immutable instances are created with an initial (valid) state which never changes

- Each object act as a value (has value semantics)
- No (or very limited) problems with shared state
- More advantages later

"Classes [instances] should be immutable unless there's a very good reason to make them mutable ... If a class cannot be made immutable, limit its mutability as much as possible."// Josh Bloch (Java guru)

Defensive Copies*

Immutable class must not expose references to mutable objects. Must create **"defensive" copies**

Incoming references

- References passed in via constructor

Outgoing references

- Methods returning references

Immutability and Efficiency*

Each value an object. Many values => many objects, possible heavy memory use

- Reuse, cache, share objects
- Integer.valueOf(4), will get a cached instance
- Avoid +-operator for strings (lot's of copying)

Example: Word Processor

- If each character in document has formatting info..??
- ... better let characters share formatting-info objects (immutable glyph class)

Java Collections*

Possible with immutable collections

- Use Collections.unmodifiable to get immutable copy
- Arrays.asList() will return immutable list
- Elements in the collection not immutable (have to make them immutable if need…)

Interesting design. Use Navigate > Go to Source in NetBeans to inspect code

Object Initialization*

Must give immutable objects well defined state at instantiation

Order of initialization

1. static attributes, in written order
2. static initializers
3. instance attributes, in written order
4. instance initializers (usage: if class anonymous, no constructor name, use instance initializer)
5. constructor … more to say, see inheritance…

Inner Classes*

Inner class

- Class declared inside an enclosing (outer) class
- Have hidden reference to enclosing instance, there must be an enclosing instance (the inner class object must be associated with an instance of the outer class)
- Instances created using the enclosing instance
- private or public
- Many, many details, JLS 8.1.3

… more later, see dependencies …

Inner Classes, cont

- The enclosing class can instantiate as many number of inner class objects as it wishes, inside it's code.
- If the inner class is public and the outer class as well, then code in some other unrelated class can as well create an instance of the inner class.
- No inner class objects are automatically instantiated when outer class instantiated
- Visibility: Everything in enclosing and (possible more) inner class(es) is visible all over

Nested Classes*

Nested (top-level) class = Inner class declared with **static**

- Ordinary class, no hidden reference
- If the inner class is static, then static inner class can be instantiated without an outer class instance,

Inner and nested classes will show up as files like;

- outer$inner.class when compiled (or possible
- outer$2.class if inner is anonymous, more later...)

Usage of Inner Classes

When class is used only in enclosing class (Node class in Linked List)

When class is very closely related (in some way) to enclosing class (inner enums)

In more advanced design situations, ... see next slide

Freezable Objects*

Possible an objects can't be fully initialized at construction time (i.e. using the constructor)

If not ... no really good solution

Possible semi-solution: Freezable objects

- Initialize attributes after construction, using a helper object, a "creator"
- Object frozen with (final) call to "freeze"-method
- Uses : Inner classes

Read Only Objects*

As stated we don't have immutable references

Alternative

- If object immutable no problem else...
- ... return copies (same as defensive copies) or
- ... return readonly-interface (possible to cast and modify) or ...
- ... return readonly wrapper object

Limit Number of Objects

By limiting the number of objects the state space is reduced

How to limit number of objects?

- One possibility : Use design pattern "Singleton"

So what is a **design pattern**?

Design Patterns

*"A **design pattern** in architecture and computer science is a formal way of documenting a solution to a design problem in a particular field of expertise. '//*Wikipedia

A reusable OO solution (language independent) to some design problem

- For now: How to limit number of instances?

Design patterns have names, possible to communicate at a higher level. Started with the GOF-book

The Singleton Design Pattern*

Will ensure there is only one instance of a class

- Will expose a global access point to instance (could be a risky, anything global is suspicious, don't overuse!)
- There is severe criticism of Singleton
- If not read-only, data should go to the Singleton not from (a sink)!
- Different implementations, we'll inspect a few

Usage Singleton

Some reasonable usages

- Logger,

- Read only singletons storing some global state (user language, help file path, application path)

- A **service locator,** an object that knows how to access other things …

- Swing or other client-side UIs

Hiding State

Of course we hide state at the class level

- All attributes should be private
- Other access (protected, public) is special, avoid!

Possible to hide state at a higher level (hiding classes) using **packages**

- Also a way of creating a structure, organizing application, avoiding name clashes and more...

Packages

"[Java] Programs are organized as sets of packages [paket]. Each package has its own set of names for types" // JLS 7

"The members of a package are its subpackages and all the top level class types [class, enum] and top level interface types declared in of the package." // JLS 7.1

"A package may not contain two members of the same name, or a compile-time error results." // JLS 7.1

Packages Declaration

"A package declaration ... specifies the name of the package to which the compilation unit [class, enum or interface] belongs" // JLS 7.4.1

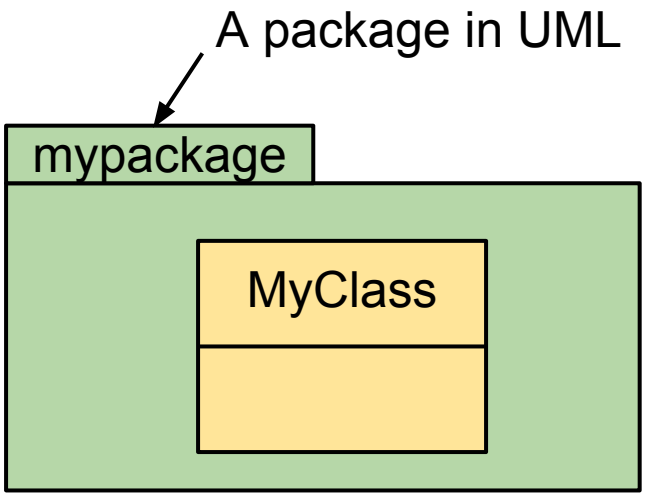
```
package mypackage; //Declaration first in file
...
// Qualified name of class is mypackage.MyClass
public class myClass {
...
}
```

Package names uses lowercase (not camelCase)

If no package declaration: "unnamed package" (default). Forbidden!!! (can't have sub packages)

Packages UML

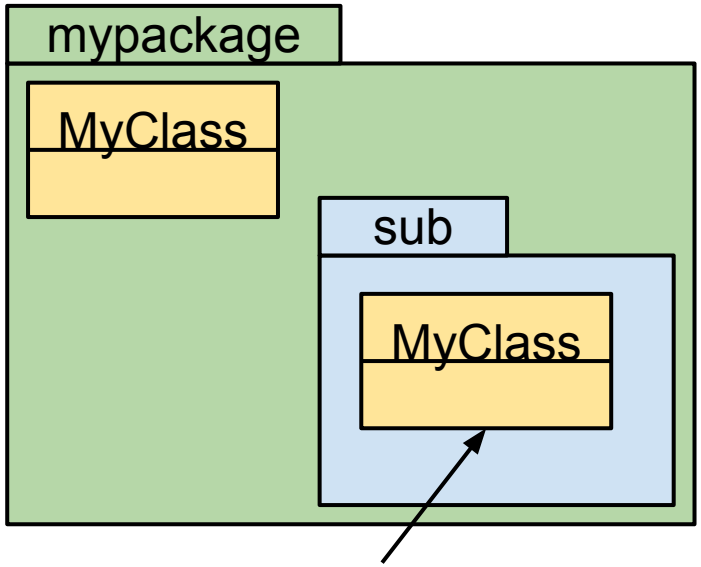
```
package mypackage;
import ...
public class MyClass {
    ...
}
```



Package Hierarchies

```
package mypackage;
import ...
public class MyClass {
        ...
}

package mypackage.sub;
public class MyClass {
    ...
}
```



Qualified name: mypackage.sub.MyClass

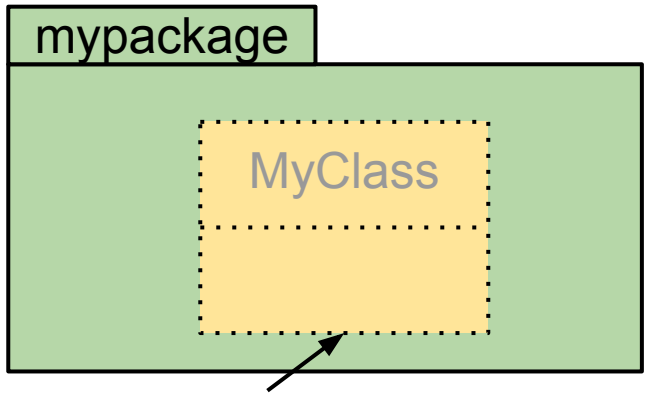
Sub packages have no special relationship (privileges), just a way to organize

Packages and Visibility*

"A top level type [class, enum, interface] is accessible outside the package that declares it only if the type is declared public."
// JLS 7

... hiding state!

```
package mypackage;
// No "public" here
class myClass {
    ...
}
```



Can't access from outside

Package Internal Access

If leaving out private, protected or public => **default access**, all classes in package can access

```
package mypackage;
class myClass {
    // Oh, oh, all classes can use! Probably bad!
    (nothing here) int secret = ..
}
```

Be careful!

Import

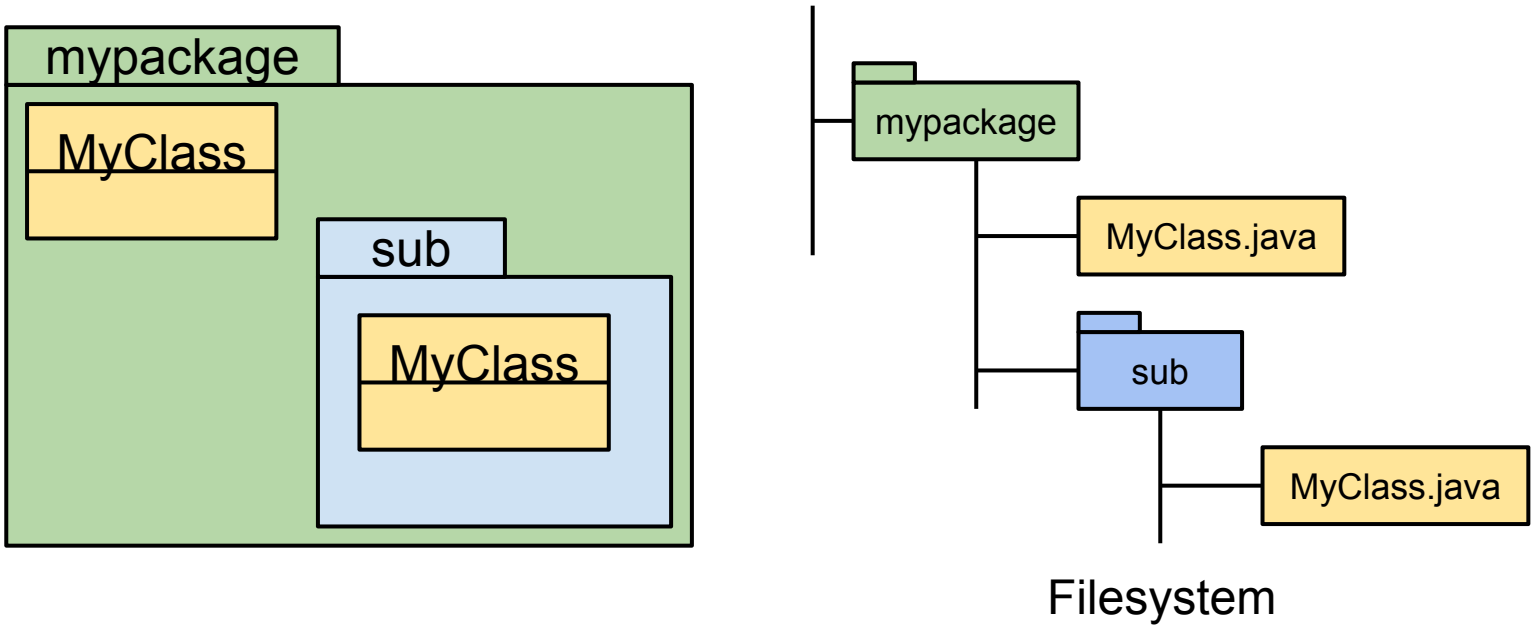
*"An import declaration allows ... a named type to be referred to by a **simple name** ... Without the use of an appropriate import declaration, the only way to refer to a type declared in another package ... is to use a fully qualified name"* // JLS 7.5

```
import java.util.Vector;
...
// Possible to use simple name
Vector<String> v = new Vector<>();
```

java.lang.*; automagically imported

Package Representation

Packages represented as folders in file system



Unique Package Names

"Developers should take steps to avoid the possibility of two published packages having the same name by choosing unique package names for packages that are widely distributed" // JLS 7.7

Use "reversed" internet domain as prefix packages, then GU id, then module/application name, then subpackages

```
// This should be unique
se.gu.cid.myapp.mypackage.MyClass

(No classes in se.gu.cid packages)
```

Command Line Execution

Must give fully qualified name of class with main-method

The JVM must be able to find all *.class files
Use: **classpath (-cp)**

- Folder containing top level package (se)
- Fully qualified name of class with main-method
- Arguments to program: main(String args[])

```
// Must be in project folder
$ java -cp ./bin se.chalmers.hajo.myapp.Main Pelle
```

Still We Have State

We have reviewed some techniques to reduce the problem with state, still we will have state ... so

.... try some reasoning...

Class Invariants*

A class invariant captures the idea of valid state for a class

- If invariant false class is in invalid state

```
 //LinkedList class
public class LinkedList {
    // Invariant: head != null
    private Node head = ...
     ...
}
```

If invariant violated we have (probably) lost all nodes

Reasoning with Class Invariants*

1. If class invariants established at object construction...
2. ... and invariants holds before and after any call to a mutator...
3. ..and we have no representation exposure ... (no leaking references)
4. ...then; Class is always i valid state (trivially true for immutable classes)

Using Class Invariants for Proofs

Very complicated, we don't...

Anyway we can try to informally reason

- Find class invariants
- Check constructors
- Check mutators preserves the invariant
  - Possible broken during method but established before return
- Check accessors (for exposure)
  - Return readonly result

Where do Class Invariants Belong?

Internals of class should not be exposed

- User of class should know nothing of the internals... more later...
- Some class invariants are strictly for the programmer (maintainer of class code). Written directly in code.

External class invariants (not revealing any internals) possible useful for others

- Class invariant in documentation

Class Invariant Example

The java.util.Collection .add (...) - method

"Ensures that this collection contains the specified element (optional operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)"

"If a collection refuses to add a particular element for any reason other than that it already contains the element, it must throw an exception (rather than returning false). This preserves the invariant that a collection always contains the specified element after this call returns."

... We don't need to check!

Failure Atomicity*

To preserve the invariants ensure any exception is thrown before the state change

- I.e. if exception state is unchanged

```
// Failure atomicity
public void doIt( int i ){
    // Possible exception here
    result = doSomeCall(i);
    // If no exception change state
    this.somevalue = result;
}
```

Summary

We try to fight the state

- No state
- Immutable
- Freezable
- Read only
- Reduce state, the Singleton DP
- Hide state using private and packages

Keep state valid

- Invariants, failure atomicity