1. Introduction

Syntax

- $\alpha|\beta$: either α or β
- $[\alpha]$: α is optonal
- $\{\alpha\}$: α can occout zero or multiple times.
- $...$: Further arguments, options, . . . are possible.

What is Java (software platform)

- It is a high level, robust, secured and object-oriented programming language.
- Java comes with its own runtime enviroment (JRE) and API. Thus every hardware or software platform that supports this enviroment can run java programs.

What does it consist of?

- **Java Language**: specification of the programming language.
- **Java Virtual Machine (JVM)**: interpreting bytecode.
- **Java Library**: rich collection of standard APIs
  - **Java Standard Edition (SE)**: is the core Java programming platform java.lang, java.io, java.math, java.net, java.util, etc.
  - **Java Enterprise Edition (SE)**: large scale, distributed system built on top of Java SE e.g. libraries for database access, remote method invocation (RMI), web services, XML,. . .
  - **Java Micro Edition (SE)**: libraries for developing applications for mobile devices and embedded systems.

Features of Java

- Object Oriented Programing (OOP) language.
- Platform independent.
- Interpreted.
- Multithreaded.
- Secured: programs run inside virtual enviroment.
- Automatic Garbage Collection.

Why do we need yet another programming language?

The problem with C/C++/. . . is that they are designed to be compiled for a specific target.
Eventhough its possible to compile a C++ program for just any type of CPU, to do so requires a full C++ compiler targeted for that CPU.
Problem writing compilers is expensive and time-consuming.
Thus the goal was to create a **platform-independent language** that could be used to produce code that would run on a variety of CPUs under different enviroments.

Types of Java Applications

1. **Standalone Applications**: Desktop/window-based applications.
2. **Web Applications**: applications that run on the server side and create dynamic pages.
3. **Enterprise Applications**: are usually distributed, such as banking applications etc.
4. **Mobile Applications**: applications that are created for mobile devices e.g. Android.

2. Building a Java Program

Definition 2.1 Compiler: Is a computer program (or set of programs) that translates source code of a high-level programming language, e.g. C++ into a low level language (e.g. assembly language or direct into machine language).

Definition 2.2 Virtual Machine (VM): Is a software application that simulates a computer, but hides the underlying operating system and hardware from the programs that interact with the VM.

- **Source code** `file.java`: is first written in plain text files ending with a `.java` extension.
  Requirements
  1. Each source file can contain at most one public class.
  2. If there is a public class, then the class name and file name must match.
- **Bytecode** `file.class`: are created by the **java compiler** `javac.exec` from source code.
  Compiling source code:

```
javac [options] file.java
```

Options

- `-d destination_folder`: compiles file into the give destination folder.

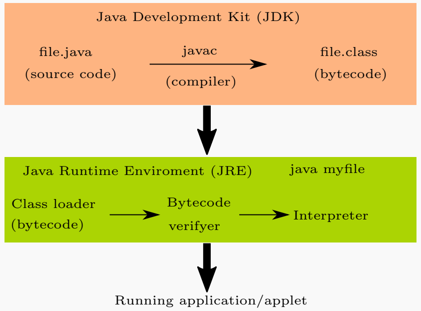
Notes

- Bytecode files are not files that can be read by processor of your platform yet.
- Bytecoes are platform-independent instructions. Thus Java's bytecode is highly portable and can run on any platform containg a JVM that supports the Java version of the bytecode.

Definition 2.3 Java Virtual Machine (JVM)/Interpreter: Is a platform independent runtime enviroment that reads and interprets the the bytecode `file.class` line by line in order to execute java programs.
Its main tasks are: Loading the bytecode, verifying the bytecode, executing the bytecode, garbage collection, thread synchronization,. . .
Running Java bytecode: `java file`

Definition 2.4 Java Runtime Environment (JRE):
JVM + Libraries: Provides the libraries, the Java Virtual Machine, and other components to run applets and applications written Java. As the JVM is just an virtual enviroment, the JRE is also known as the implementation of the JVM.
It is the minimum requirement to run (not creating) java programs.

Definition 2.5 Java Development Kit (JDK):
JRE + Development Tools: It consits of the JRE plus tools such as compilers or debuggers for developing applets and applications.
Thus is necessay in order to develope and running code.



Apllets vs. Applications

All Java programs can be classified as Applications and Applets. The striking differences are that applications contain main() method where as applets do not. One more is, applications can be executed at DOS prompt and applets in a browser. We can say, an applet is an Internet application.

3. Basics

In Java, every variable, constant, and function (*including main*) must be inside some class.
A Java Program: is a class which contains a main method:

```
public static void main(String[] args) { ... }
```

- `pulic`: accessible from everywhere.
- `static`: defined on the class level (not bound to instances).
- `void`: does not return a result (⇒ procedure)
- `args`: argument, array of command-line arguments

1) Packages

Programmers can define their own packages to bundle group of classes/interfaces, etc.
Packages create a new namespace thus there won't be any name conflicts with similar identifiers from other packages.

Definition 3.1 Package Statement: Identifies the package that a Java program belongs to. The package statement should be the first line in the source file (there can be only one package statement per source file).
Package Statement: `package package-name;`

Compiling source files with package statements

Using the `-d destination_fodler` option will create a folder with the given package name *in the package statement* will be created in the given destination folder (if not existing), and place the complied source file into it.

```
javac -d destination_folder file.java
```

Using Packages

- `package_name.identifier`: use fully qualified names.
- `import package_name.*`: import the whole package.
- `import package_name.identifier`: import certain identifiers.

The default package

- If a program does not include a package statement it belongs to the so called default package, which is basically an default, unnamed package.
  When developing small or temporary applications e.g. for testing purposes, its ok not to include a package statement.
  But in order to avoid name conflicts, all java source files belonging to a program should contain a package statement.
- **Convention**: use your transposed internet domain name for uniqueness, if you have one.
- **Lower Cases**: use lower case letters for packages in orde to avoid any conflicts with the names of classes and interfaces.

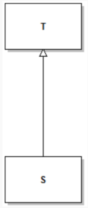
4. Java

Definition 4.1 Liskov Substitution Principle: If S is a subtype of T, then objects of type T may be replaced with objects of type S **without** altering the correctness of the program

In other words:

- Whenever you work with an instance of type T, you should not be surprised if you effectively work with an instance of type S
- An instance of type S can be used at all places where an instance of type T is expected

Consequences: Overriding methods need to satisfy (at least) the rules specified by the base class



Definition 4.2 Variance: Is a term applied to the expected behavior of subtypes in a class hierarchy containing complex types.

5. Java

6. Basics

Definition 6.1 Resolution: Is a rule of inference.

Definition 6.2 Type: defines a behavior but no implementation.
(Java: Types are defined by classes)

Definition 6.3 Subtyping: Are specializations of Types and define a is-a relationship.
(Java: Subtyping is defined by subclassing, i.e. each subclass also defines a subtype)

Definition 6.4 Java Method Signature: Is the method name and the number, type and order of its parameters:

```
methodName(Type1, Type2,...)
```

Note

Return types, name of the arguments and thrown exceptions are not considered to be a part of the method signature.

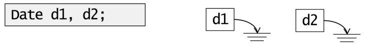
Definition 6.5 Method Declaration: Is a declaration of a function i.e. declares an identifier and its types,...

```
visibility [static] returnType methodName (args);
```

Definition 6.6 Variable/Reference Declaration: In java the only way to access an object is through a reference variable.

```
StaticType reference;
```

A reference is not an object, thus no memory is allocated for an object of the Type `StaticType`. of the reference.



Note

- Do not confuse C++ references with java references in Java variables are called (by convention) references.
- Reference variables are sort of C++ pointers but we can not do pointer arithmetic's on it e.g. ptr++
- Reference Variables are of size:
  - 32-bit on a 32-bit JVM
  - 32-bit or 64-bit on a 64-bit JVM, depending on the configuration

Definition 6.7 Static Type: A reference variable is declared to be of a specific type and that type, known as static type can never be changed.
Static type = type of reference variable

Corollary 6.1 Guarantees of Static Type: When a variable is declared as being of a particular type, then we have a language-enforced guarantee that any *object* referenced by that reference variable will have (at least) all the features of that *type*.
⇒ *dynamic type* needs to provide methods of *static type*.

Definition 6.8 Instanciation new: The new operator instantiates a class by dynamically allocating memory (=allocation at run time on the heap) for a new object and returns a reference to that memory.

```
new MyClass();
```

This reference can then be stored in/assigned to a (reference definition 6.6) variable.

```
Type ref = new MyClass();
```

Note

In Java, all class objects must be dynamically allocated.

Object Oriented Programming (OOP)

1) Principles

Definition 6.9 Encapsulation:

- Methods and data are combined in classes
- Not unique to OOP

2) Class Basics

1. Static

Definition 6.10 Static Fields: Per class fields that exist only per class (≠ per object), accessible via class (or instance)

- Bound to class (available even if no instance has been created)
- Only one copy of the attribute (identical for all instances)
- Initialization per default with zero (0 / 0.0 / false / null)

```
static Type ref;
```

Constants Declare constants as `static final` fields.

Definition 6.11 Static Methods: Per Class methods, accessible via class (or instance):

```
static ResultType Name(args){ Body }
```

- No `this`
- No access to instance attributes & methods

Do not use instances to access static fields or methods.

2. Class Initialization

Upon loading o the class

3) Inheritance

Definition 6.12 (Implementation Aspect) **Code Inheritance/Extension/Subclassing:**

- Subclasses add new fields & methods
- Subclasses have more (specialized) attributes & operations
- Implementation aspect

Class: defines a type **and** implementation

Definition 6.13 (Design Aspect) **Interface Inheritance/Specialization/Subtyping:**

- Types define a set of objects
- Subtypes are specializations of this Type
- Inheritance of behavioral aspects

Interface: defines type

Definition 6.14 extends:
Define new class by extending existing classes.
In Java only one base class can be specified to inherit from.
Extension inherits all fields and methods from the base class.

1. Polymorphism

Definition 6.15 Type Checking: The process of verifying and enforcing the constraints of types

Definition 6.16 Statically Typed Languages: Is a language where the type of a variable is known at compile time.
For some languages this means that we as programmer must specify of what type each variable is.
Advantage: can do type-checking during compile time by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.
Examples: C, C++, Java

Definition 6.17 Dynamically Typed Languages: If the type is associated with/depends on run-time **values**, and not named variables/fields/etc
Advantage: we do not have to specify types every time.
Drawback: need to do type checking at run-time

Definition 6.18 Polymorphism: Polymorphism allows operations to be performed on objects without needing to know which class the object belongs to, provided that we can guarantee that the class implements the specified type.

Definition 6.19 Polymorphic Assignment: Instances of an extended class can be assigned to references of type base class:

```
BaseClass ref = new extendedClass();
```

Definition 6.20 Dynamic Type: Is the type of the object assigned to a reference variable.
Dynamic types of reference variables may change with every assignment.
Dynamic Type ⊆ Static Type as the dynamic type must full fill at least the guarantees of the static type.

Definition 6.21 Static Binding: Is type resolution based on the static type of a variable reference.

Definition 6.22 Static Binding and Overloading: If we overload a method in Java, the compiler will produce a version for each overloaded function signature.
The resolution of the method signature (not to the actual implementation) of the method is done during compile time and does hence depend on the static type passed to the method ⇒ static binding

```
StaticTypeOfa a = new DynamicTypeOfa();
a.methodToResolveTo(StaticTypeOfb b);
```

`staticTypeOfb` decides which overloaded method to call.

Check if thats correct

Definition 6.23 Dynamic Type Binding and Overloading: The runtime chooses a function implementation

1. Based on the function signature chosen at compile time (static binding)
2. Depending on the dynamic type of the object referenced by a in order to chose an actual implementation of

```
DynamicTypeOfa.methodToResolveTo(StaticTypeOfb b)
```

Check if thats correct

Note

For non-overloaded methods only the dynamic type of the reference variable decides which method to call.

Check if thats correct

Listening 6.1 : *Dynamic Type Inference*

```
if( ref instanceof Type)
```

check if this is only for conceret or also base type

Note

A runtime error is thrown if the dynamic type of `ref` is not a Type or extension thereof. (Type) ref;

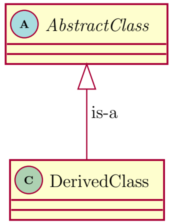
4) Abstract Classes

Definition 6.24 Abstract Methods: Are methods that only define a signature/are only a declaration:

```
visibility abstract retrunType methodName();
```

- Define methods to be implemented in subclasses
- Can only be declared in `abstract` classes
- Cannot be `private` as private methods cannot be overridden
- Cannot be `static` as static methods cannot be overridden

Definition 6.25 Abstract Classes: Are classes that stand in an "is-a" relationship with their subclasses:



- Cannot be instantiated
- Usually have one or more abstract methods
- May have attributes, constructors, non-abstract methods

```
visibility abstract class Name{ Body }
```

Note

An abstract class must not necessarily have an abstract method.
Useful to declare classes abstract that cannot/may not be initiated but are supposed to be extended.

Note: Derived Classes

- Have to override (implement) all `abstract` methods
- Or have to be declared abstract as well, if not all abstract methods are overridden.
- Can only extend one Class.

Note

Arrays of an abstract base type may be instantiated sine no instances are created (only reference variables)

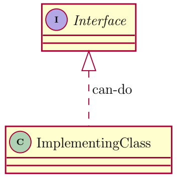
```
ContainerType[] name = new ContainerType[Number]
```

Usage

- Want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

5) Interfaces

Definition 6.26 Interfaces=pure abstract class: Have to be implemented by the class that uses the interface and represent a "can-do" relationship:



- Have only `public` and `abstract` methods
- Attributes are by default `public`,`static` and `final`
- No constructors ⇒ instantiated (≠ declared)

Definition:

```
public intreface ClassName{ Body; }
```

Implementation:

```
class ClassName implements InteraceName{ Body; }
```

- All methods defined in the declared interface have to be implemented unless its another interface adding more functionality
- A class may implement multiple interfaces

Note: Java 8 default methods

Can be implemented inside interfaces and are able to access other methods. Allows to extend interfaces without breaking existing classes that implement the interface.

Use `@override` to implement the methods.

Usage

- You expect that unrelated classes would implement a piece of functionality.
- Want to specify the behavior of a particular data type, but not concerned about who implements its behavior.

Interfaces as function arguments

Methods with interfaces as function argument can be called with any class that implements the interface.

Interface Reference

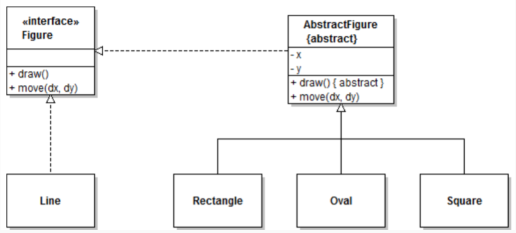
Interfaces can be used as reference variable for all subclasses **but** be care-full if the reference object implements another interface we will not be able to call its functions unless we use an implicit cast.
Check if referenced object implements an interface: `ref instanceof aInterface`

Listening 6.2 : *Interaface Variable*

```
TypeImplementingAandB ref = new TypeImplementingAandB()
InterfaceTypeA refA = ref;
InterfaceTypeB refB = ref;
refA instanceof InterfaceTypeB // True as ref implements B
refA.methodOfTypeImplementingAandB() // works
refA.methodOfB() // does not work
((InterfaceTypeB)refA).methodOfB() // works
```

Intrefaces and Abstract Classes

- `Interface`: provides a type
- `abstract Class` semi finished component that contains default implementations
  - which can be used in subclasses
  - which can be overriden in subclasses



Abstract Classes vs. Interfaces

- Abstract Classes
  - A class may extend only one abstract class
  - Abstract classes may contain attributes & concrete implementations
- Interfaces
  - A class may implement several interfaces
  - Interfaces may contain no implementations

Definition 6.27 Marker Interface: Is an empty interface, that can be used to add a certain attribute/characteristic to a class that can be checked with `instanceof` e.g. RandomAcess

Definition 6.28
Functional Interface `@FunctionalInterface`: Is an interface with a single abstract method:

```
@FunctionalInterface
public interface InterfaceName{
  visibility returnType methodName(args);
}
```

Notes

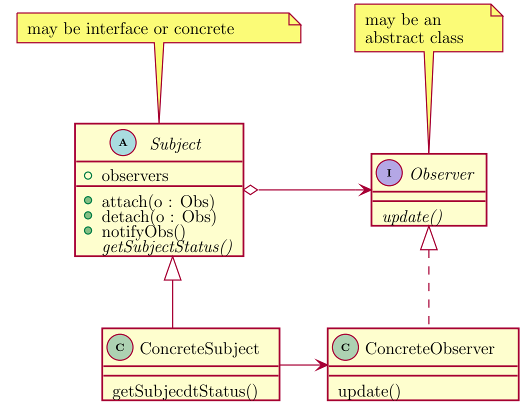
- The annotation `@FunctionalInterface` allows compilers to generate an error if the interface does not satisfy the conditions of a functional interface.
- Default methods are not abstract and do not count.

Lambda Expressions and Functional Interfaces

7. Behavioral Patters

Definition 7.1 Behavioral Patterns: These design patterns are specifically concerned with communication between objects.

1) Observer/Listener/Publish-Subscriber



Intent
- Consistency assurance between cooperating objects without connecting them too much.
- One-to-many relation between objects which allows to inform the dependent objects about state changes

Listening 7.1 : *Observer*

```
interface Observer{
  void update ();
}
```

Listening 7.2 : *Subject*

```
class Subject {
  private List <Observer > observers = new ArrayList <Observer >();

  public void addObserver(Observer o) {
    // method of List
    observers.add(o);
  }

  public void removeObserver(Observer o) {
    // method of List
    observers.remove(o);
  }

  protected void notifyObservers () {
    // inform all Observers that we have changed
    for(Observer obs : observers) {
      // Let (concrete) observers handle it
      obs.update ();
    }
  }
}
```

Listening 7.3 : *Concrete Subject*

```
class ConcSubject extends Subject {
  private int state;
  public int getState (){
    return state;
  }

  public void setState(int val){
    state = val;
    notifyObservers();
  }
}
```

Listening 7.4 : *Concrete Observer*

```
class ConcObserver implements Observer {
  private ConcSubject s;
  ConcObserver (ConcSubject s){
    this.s = s;
    s.addObserver(this);
  }

  public void update (){
    // take appropriate steps
  }
}
```

Participants
- **Observer**: typically an interface that declares methods to handle notifications.
- **Subject**: maybe concrete or interface knows its observers over the observer interface (guarantee that the method update exists).
- **Concrete Observer**:
  - Handles state changes of its observable
  - Implements the observer interface to keep its state consistent with the subject
  - May maintain a reference to a concrete subject (pull-model)

  Check if ref is really because of pull model
- **Concrete Subject**: Implements/extends Subject and hence can:
  - attach observers
  - detach observers
  - call the method notify if its state changes (e.g. in a set method)

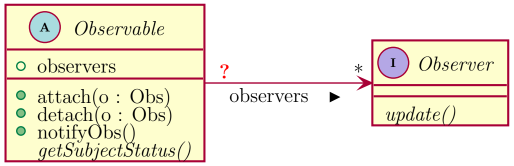
Consequence
- Subject does not care about the number of observers ⇒ notifications are broadcast.
- A simple state change/operation may lead to a (unwanted) cascade of updates.
  → beware of cyclic dependencies see

  add cref

Note
- Thus it is up to the observer to handle or **ignore** notifications.
- When should `notifyObservers` be called?
  - **Automatically**: at every state change
    ⇒ many updates
  - **Explicitly**: after a set of changes
    ⇒ responsibility e.g.
    - Asynchronous at idle times
    - Use a boolean flag `changed`, that can be set in order to test if object has changed see section 3

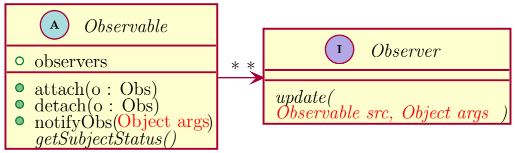
    clarify why or what for?

1. One or Many Observables



0. . . 1 : *
- Subject/Observable is usually passed to constructor of the Concrete Observer (⇒ reference).
-

*** : ***
- If we want that observers can observe multiple subjects and if these observers may be able to ignore certain events from those subjects, then the observer need to know the source of an update event.
  Moreover if arguments are passed to the Observables, then the observer needs to know:
  - The arguments
  - The source of the event



Listening 7.5 : *Many vs Many*

```
protected void notifyObservers(Object args){
    for(Observer obs : observers){
        obs.update(this, args);
    }
}
```

2. Push vs Pull Model

Push-Model
- The subject (i.e. the Observable) sends the observer on notification all the data it will need. The observer doesn't need to query the subject for information.
- Observable must know which information is needed by the observers in order for them to act appropriately (or Observable sends all data)

Pull-Model
- In the pull model, the subject merely notifies the observer that something happened, and the observer queries the subject based on that, to get the information it needs.
- Observable send only information about what has changed
- Observer has to access the Observable state (via reference)

maybe add pros and cons

Pitfalls

3. Java.util.Observable

Provides an implementation of an Observable that can be used.
But it is never used!
Problem: multiple inheritance is in java not possible, thus `Java.util.Observable` can only be implemented if extension is not defined in its own inheritance hierarchy.
Solution: composite twin class (??).
⇒ code duplication ⇒ `Java.util.Observable` is not used.

Listening 7.6 : *Java.util.Observable*

```
public interface Observer {
  void update(Observable o, Object arg);
}

public class Observable {
  private boolean changed = false;

  // Marks this Observable as having been changed
  protected void setChanged() { changed = true; }

  // Indicates that this object has no longer changed,
  // or that it has already notified all of its observers
  // of its most recent change.
  protected void clearChanged() { changed = false; }

  // Tests if this object has changed.
  public boolean hasChanged() { return changed; }

  // If this object has changed, as indicated by the
  // hasChanged method, then notify all of its observers
  // and then call the clearChanged method to indicate
  // that this object has no longer changed.
  public void notifyObservers(Object arg) {
    if (!changed) return;
    // See section on ConcurrentModificationException
    Object[] copy = obs.toArray();
    clearChanged(); // before notification!
    for (int i = copy.length-1; i>=0; i--)
      ((Observer)copy[i]).update(this, arg);
  }
}
```

Add observable twin class solution after we studied again composite

4. ConcurrentModificationException

Definition 7.2 ConcurrentModificationException: We cannot change a list e.g. add/remove while iterating over it.
Example: Once Observer and notify.

Definition 7.3 OnceObserver: Is an observer that only observers for a single notification.
⇒ removes itself afterwards.

Listening 7.7 : *OnceObserver*

```
public class OnceObserver implements Observer {
                         Conc. Observerable
  public void update(Observable source, Object arg) {
        // take appropriate steps once
        source.removeObserver(this);
  }
}
```

Listening 7.1 : *Problem* *java*

```
protected void notifyObservers(Object arg){
  for(Observer obs : observers){
    obs.update(this, arg);
  }
}
```

Problem: we cannot remove an observer from `obs` while iterating over it

Solutions

1. Perform the notifications on a copy of the observer list.
2. Delay add/remove Observer calls.
3. Copy the observer list upon modification.

Definition 7.4 Java list to array: Returns an array containing all of the elements in this list in proper sequence (from first to last element).

`java.util.ArrayList.toArray(a T[])`

a: is the array into which the elements of the list are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.

understand better how this works

Listening 7.8 : *Copy Solution*

```
protected void notifyObservers(){
  Observer[] copy;
  copy = observers.toArray(new Observer[observers.size()]);
  for (Observer obs : copy){
    obs.update(this);
  }
}
```

Listening 7.9 : *Alternative*

```java
protected void notifyObservers(){
  for(Observer o : new ArrayList<>(observables)){
    o.update(this);
  }
}
```

maybe add Mutation or gamble, must likly not important

Definition 7.5 Copy on write list: Thread-safe variant of ArrayList in which all mutative operations are implemented by making a fresh copy of the underlying array.

Listening 7.10 : *Best Solution*

```java
private List<Observer> observers = new CopyOnWriteArrayList<>();
public void addObserver(Observer o) { observers.add(o); }
public void removeObserver(Observer o) { observers.remove(o); }

protected void notifyObservers(Object arg) {
  for(Observer obs : observers) {
    obs.update(this, arg);
  }
}
```

5. Cyclic dependencies

add picture

Problem

If we have a cyclic dependency we will end up in an endless loop unless we take appropriate measures.

finish section

6. Causality of Changes

Problem

How can we make sure that notifications of the model will be handled by the observers in the same order as they were applied to the models.

Solution

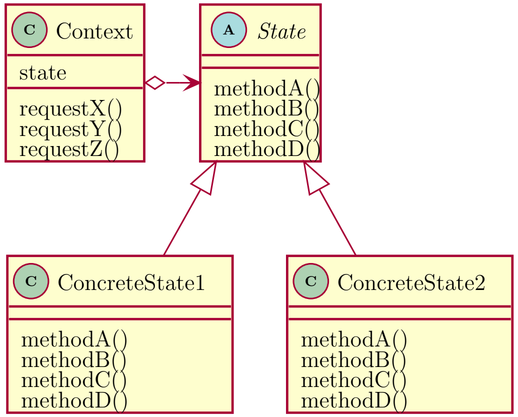
- Queuing of state changes: model becomes asynchronous. . . difficult to program
- Queuing of notifications
- Prohibit state changes during notification

7. Memory Management

Is automatic in java if no more reference to an object exists. ⇒ need to detach all observers/listeners for clean up.

add twin classes

2) State Patterns



Intent Context related behavior depending on the current state

8. Structural Patters

9. Creational Patters

Add: Method type bindining exercise 2/slides/more research

Add: Nested Classes/Anonymous functions

$https : //www.tutorialspoint.com/java/java_innerclasses.htm$

Add: Java generics, generics with questionmark and extends.