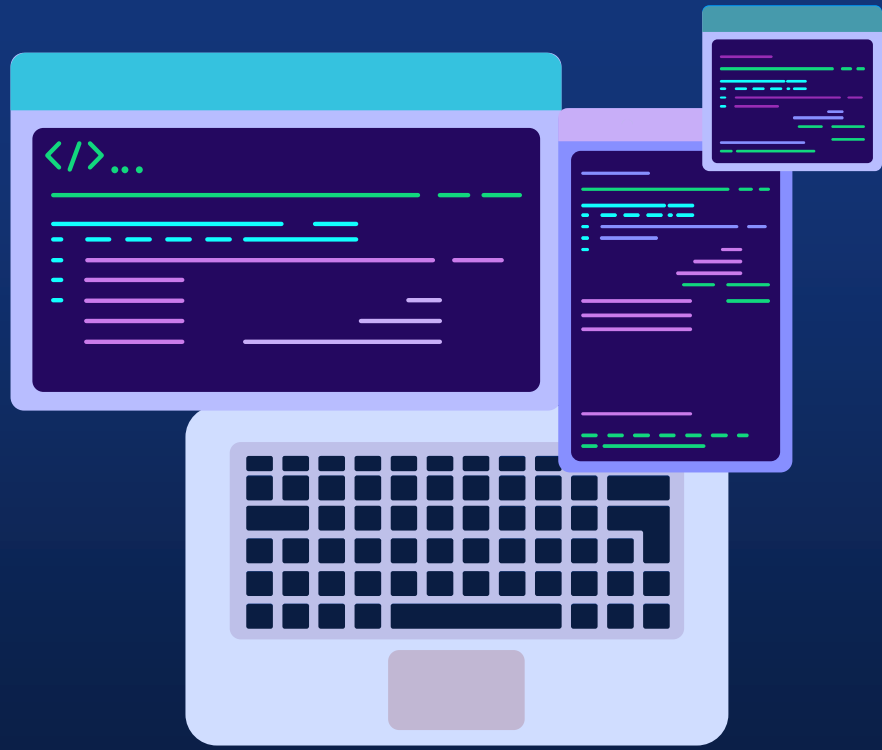
Software Design and Important concepts



Mentor: Einar Rocha

CONTENT

01
OOP Pillars

Inheritance, Polymorphism
Encapsulation, Abstraction

02
Clean Code

Meaningful Names,
Functions, Unit test
Code Smells...

03
SOLID

Single Responsiblity
Open closed
Liskov Substitution
Interface Segregation
Dependency Inversion

04
Design patterns

Singleton, Factory Method
Strategy, Observer
Builder...

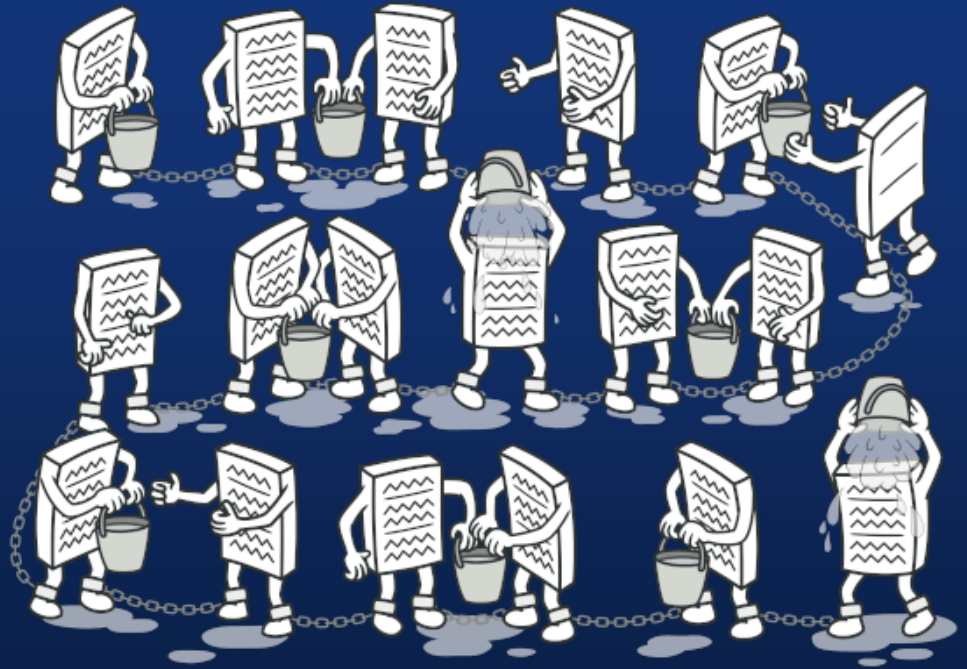
04

Design patterns

Agenda

- Chain of Responsibility — Example...
- Iterator — Example...

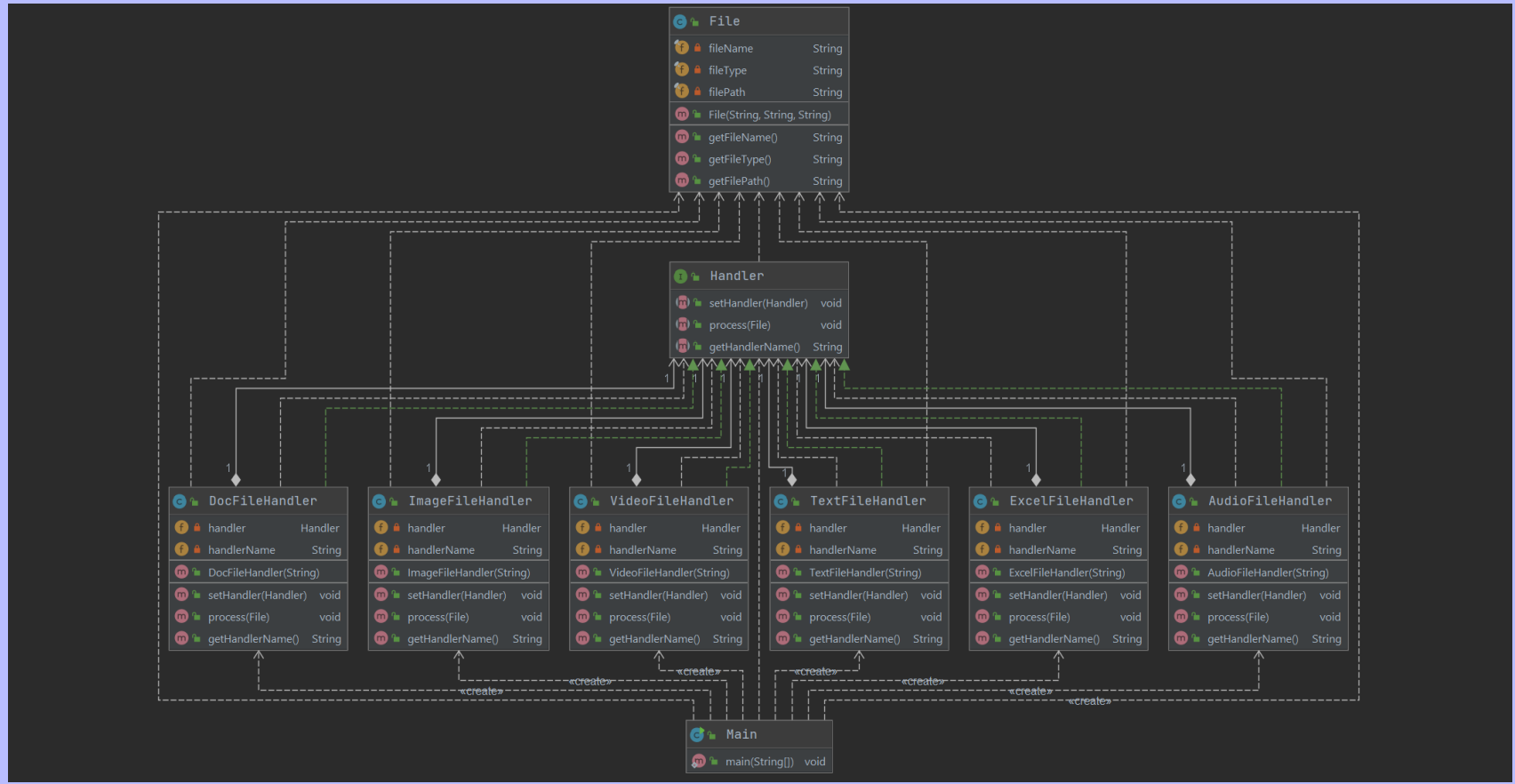
Chain of Responsibility



When to use?

- More than one objects may handle a request, and the handler isn’t known a priori.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically

Chain of Responsibility



CONS

Some requests may end up unhandled.

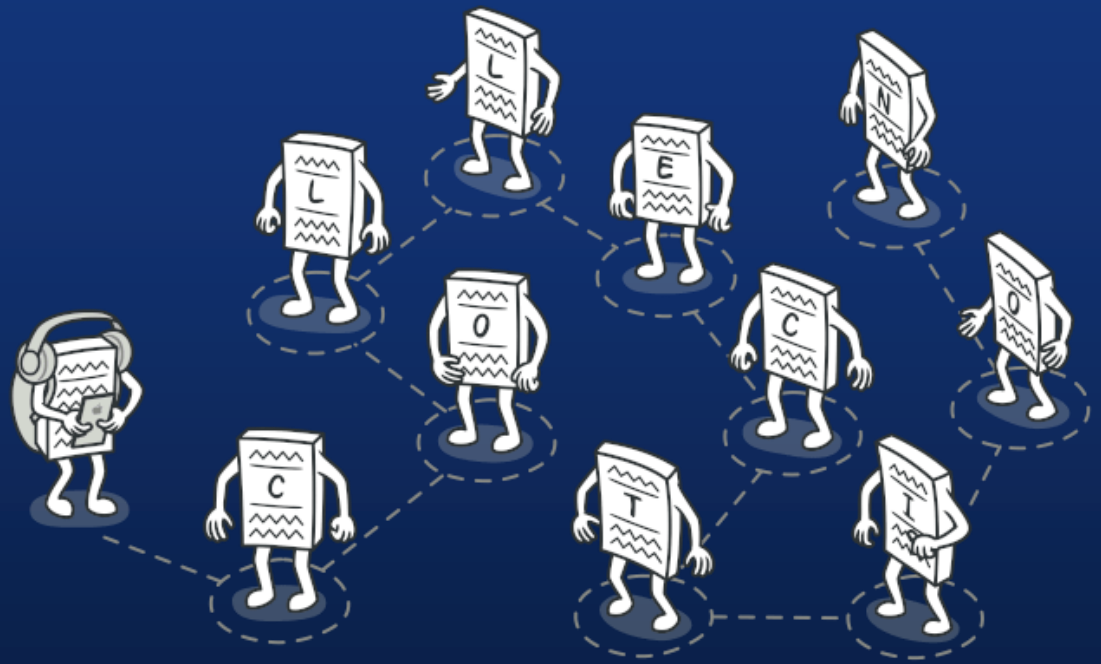
PROS

You can control the order of request handling.

Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform operations.

Open/Closed Principle. You can introduce new handlers into the app without breaking the existing client code.

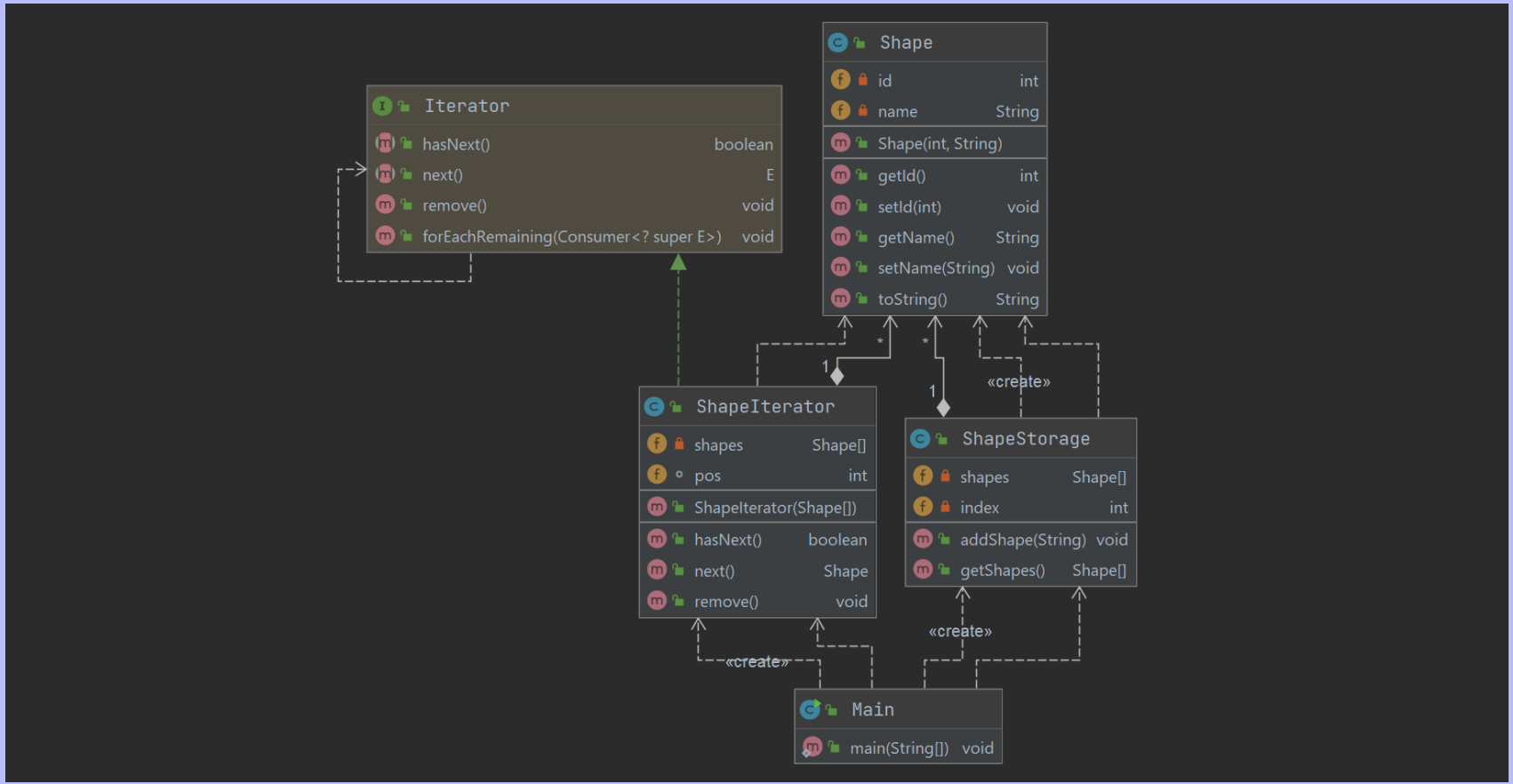
Iterator



When to use?

- To access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures.

Iterator


Iterator
hasNext() boolean
next() E
remove() void
forEachRemaining(Consumer<? super E>) void
Shape
id int
name String
Shape(int, String)
getId() int
setId(int) void
getName() String
setName(String) void
toString() String
ShapeIterator
shapes Shape[]
pos int
ShapeIterator(Shape[])
hasNext() boolean
next() Shape
remove() void
ShapeStorage
shapes Shape[]
index int
addShape(String) void
getShapes() Shape[]
Main
main(String[]) void
1
*
1
*
«create»
«create»
«create»

CONS

Applying the pattern can be an overkill if your app only works with simple collections.

Using an iterator may be less efficient than going through elements of some specialized collections directly.

PROS

Single Responsibility Principle. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.

Open/Closed Principle. You can implement new types of collections and iterators and pass them to existing code without breaking anything.

You can iterate over the same collection in parallel because each iterator object contains its own iteration state.