Java (Introduction) for C++ Programmers

Çağatay Sönmez

02.02.2021

Arcelik Electronics Plant Division, R&D Team – Software Design

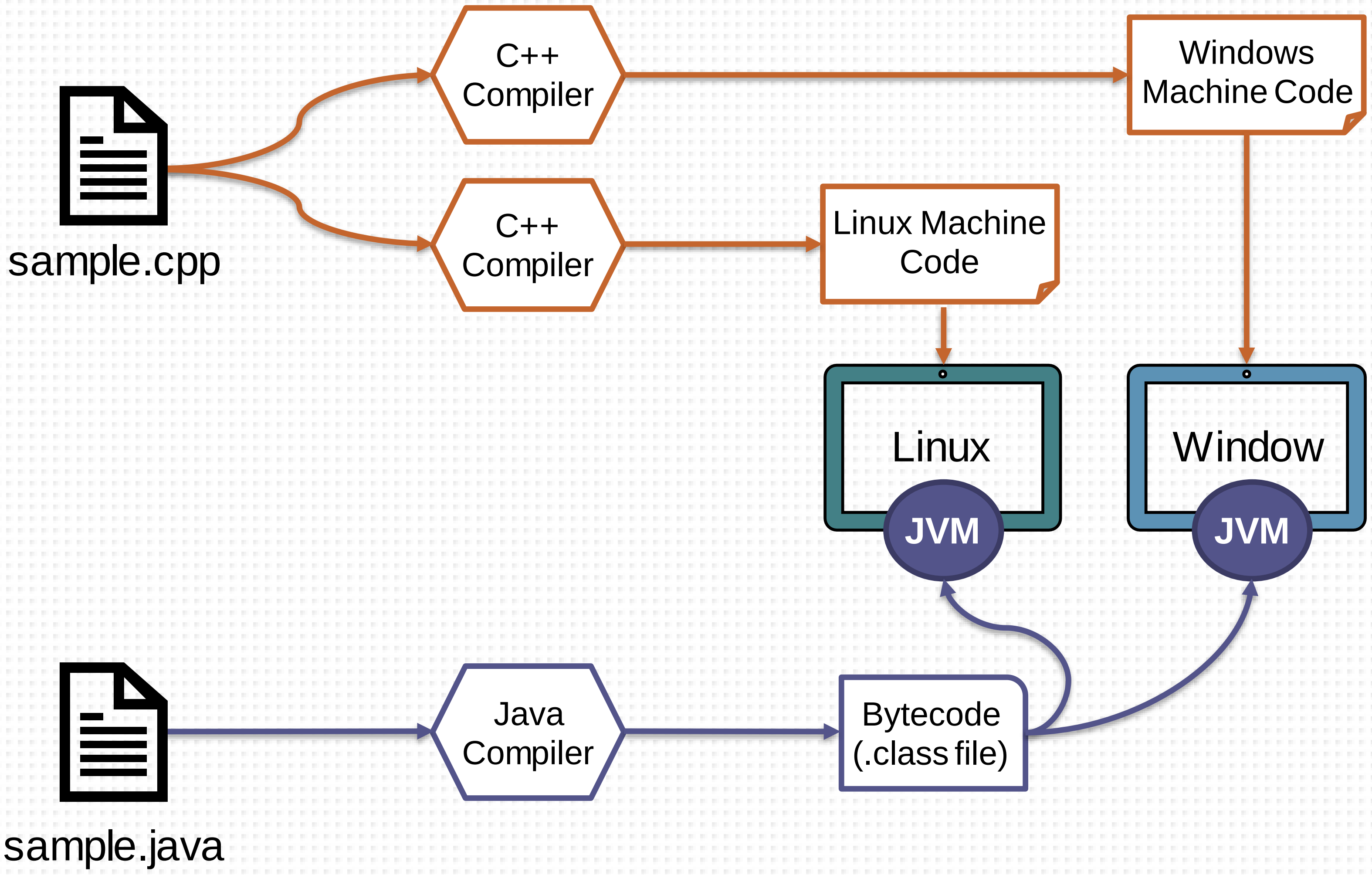
Agenda

- Java Execution Environment
- Java Types
- Java Memory Model
- Java Class Structure
- Inheritance/Abstract Class/Interface
- Java Pass-by-Value
- Packages
- Access Modifiers
- Design Patterns

The Java execution environment

- Java programs are compiled **like** C/C++ programs
- **Unlike** C/C++, Java source code is not converted to a platform-specific machine language
- Java programs are converted to a platform-independent language called **bytecode**
- Java virtual machine (**JVM**) runs bytecode to provide platform independency.
- There are many JVM implementations, **OpenJDK**'s HotSpot JVM is the most commonly used.

Comparison of execution environments



Compile & Run a Java Program

- Like C/C++, Java uses a **main** method, which has strict naming convention, to run a Java application:
  - *public static void main (String[] args)*

```
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello World!");
    }
}
```

Compillation:
$javac HelloWorld.java

Execution:
$Java HelloWorld

Types

- Java has two types of variables
  - Primitive types (byte, char, short, int, long, float, double, boolean)
  - Reference types (used to access objects)
- Primitive type always has a value, but reference type can be null
- Unlike C/C++, reference variables are **not** pointers but a handle to the object.

Wrapper Classes

- A Wrapper class is a class whose object wraps or contains a primitive data types

Primitive Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Char
float	Float
int	Integer
long	Long
short	Short

Creating Object

- Creating objects is similar to C++, objects are created via ‘**new**‘ operator
- Creating objects of a class is a two-step process
  - Declaration
  - Instantiation and Initialization

arrays are object in java!

```
// Declaration
int array[];

//Instantiation and Initialization
array = new int[5];
```

- The new operator instantiates a class, then call to a class constructor, which initializes the new object

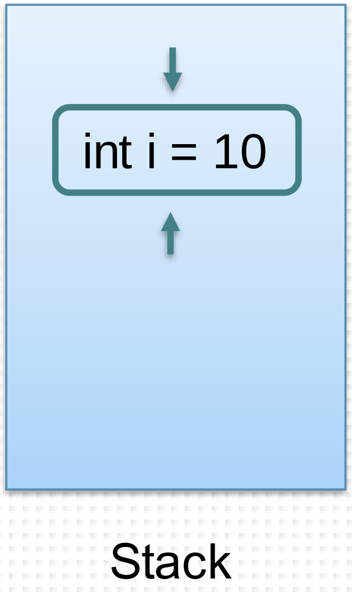
Java Memory Model

- Heap
  - Contains all allocated object instances
- Stack
  - Contains local primitives and reference to other objects
- Static
  - Contains static data/methods
- Code
  - Contains your bytecode

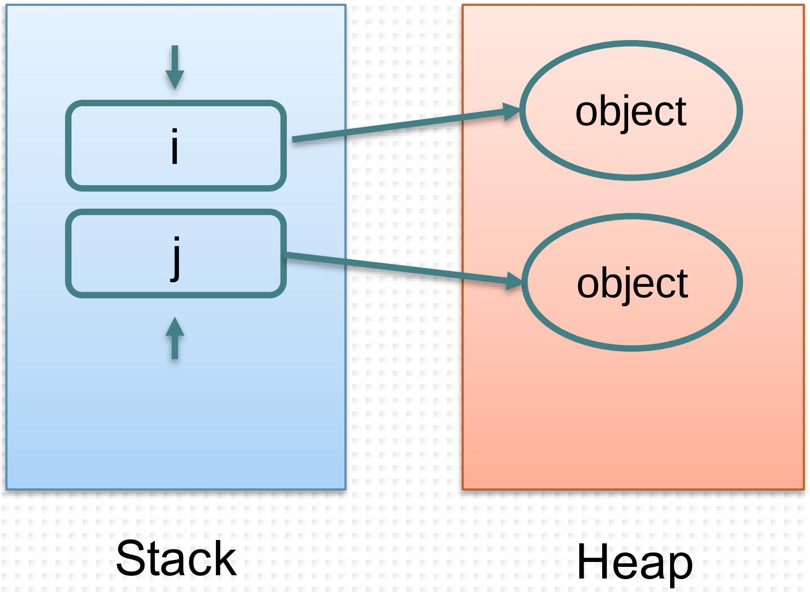
Unlike C++, objects **cannot** be stored on the stack; all Java objects have to be allocated on the heap!

Stack vs Heap

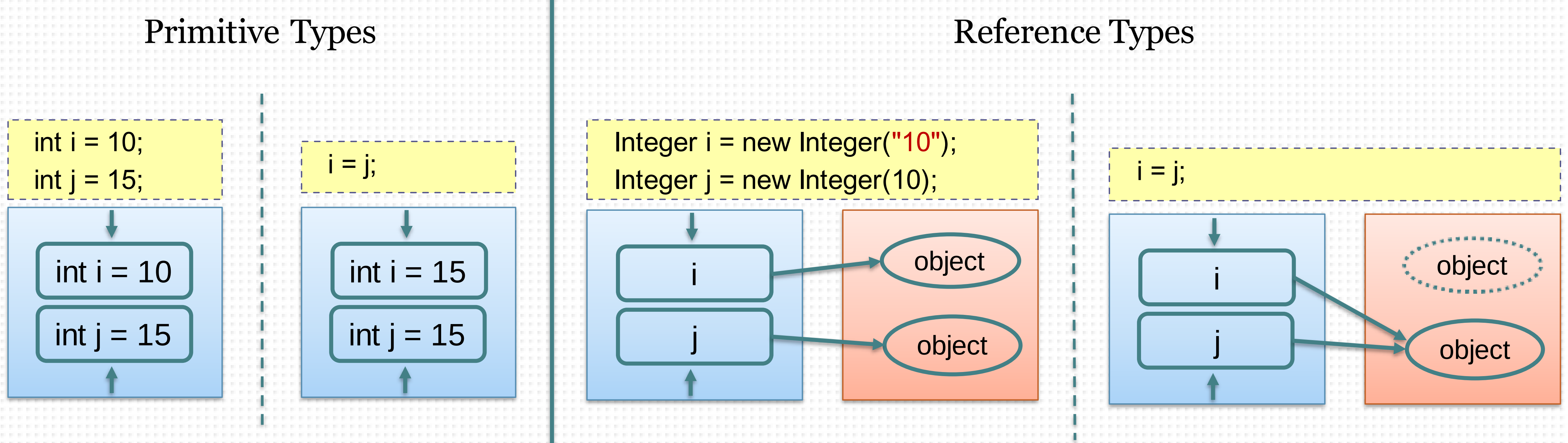
```
int i = 10;
```




```
Integer i = new Integer("10");
Integer j = new Integer(10);
```



Assignment Statement for Types



Garbage Collection

- Java uses garbage collector
- The memory of objects which are no longer referenced are automatically reclaimed by the collector.
- Therefore, java **does not** need malloc() and free() functions
- There is **no** "dangling reference" problem
- **System.gc()** method suggest JVM to run garbage collection
  - public static void gc()

Other Considerable Differences

Feature	C++	Java
pass by reference	yes	no
multiple inheritance	yes	no
operator overloading	yes	no
default arguments	yes	no
header files	yes	no
typedef	yes	no
struct	yes	no
union	yes	no
goto	yes	no

Flow Controls

```
switch(expression) {
    case value :
        //Statements
        break;
    default :
        //Statements
}
```

```
if (x == y)
    System.out.print("x is equal to y");
else
    System.out.print("x is not equal to y");
```

```
boolean flag = false;
while (flag == false) {
        System.out.println("flag is true");
        flag = true;
}
```

```
for (int i=0; i<10; i++) {
        System.out.println("i is: " + i);
}
```

```
boolean flag = true;
do {
        if (x == y)
            break;
} while (flag == true)
```

Java Finally Block

- Unlike C++, Java uses finally keyword
- The finally block always executes when the try block exits.

```
try
{
    //do your job
}
catch
{
    //catch any exaption
}
finally
{
    //code block to run in any case
}
```

Java Class Structure

- Unlike C/C++, everything must be in a class
  - No global function, no global data

```
package example;

public class HelloWorld {
    private String name;

    public HelloWorld(String name){
        this.name = name;
    }

    public void SayHello() {
        System.out.println("Hello "  + name);
    }
}
```

Field(s)

Constructor

Method

Inheritance

- Unlike C/C++, Java **doesn't** support **multiple** inheritance!
- All methods are virtual by default

```
public class Base {
    void foo() {
        System.out.println("Base");
    }
}

public class Derived extends Base {
    @Override
    void foo() {
        System.out.println("Derived");
    }
}
```

Super Keyword

- Use **super** keyword to access the methods and fields of the parent class

```
public class Base {
    int i = 2;
}

public class Derived extends Base {
    int i = 1;
    void foo(int i) {
        System.out.println("i: " + i);
        System.out.println("i: " + this.i);
        System.out.println("i: " + super.i);
    }
}
…
new Derived().foo(0);
```

Prints 0

Prints 1

Prints 2

Abstract Classses

- Unlike C/C++, java has **abstract** keyword for abstract classes and methods

abstract class

abstract method
(pure-virtual method in C++)

non-abstract method
(virtual method in C++)

abstract methods have
to be overriden!

```
public abstract class Base {
  abstract void foo();
  void bar() {
     System.out.println("bar");
  }
}

public class Derived extends Base {
  @Override
  void foo() {
     System.out.println("foo");
  }
}
```

Final Keyword

- Unlike C/C++, java has **final** keyword for final classes, methods and fields
- Final classes and methods cannot be inherited
- Final fields cannot be assigned a value

final class

final field

final method

```java
final class Base {
  final int x = 10;
  final void foo() {
      System.out.println("foo");
  }
}
```

Interface

- Unlike C/C++, Java has **interface/implements** keywords
- Interface is a description of behavior without implementation

```java
interface Foo {
   void foo();
}
```

```java
interface Bar {
   void bar();
}
```

```java
class Base implements Foo, Bar {
   void foo() {
        System.out.println("foo");
   }

   void bar() {
        System.out.println("bar");
   }
}
```

Object Class

- Object class is the root of the class hierarchy.
- Every class has Object as a superclass.
- Provides methods that are common to all objects
  - boolean **equals**(Object o)
  - String **toString**()
  - int **hashCode**()
  - Object **clone**()
  - Class<?> **getClass**
  - protected void **finalize**()
  - …

Destructor in Java

- Unlike C/C++, Java **doesn't** have destructor
- Java garbage collector finalizes the objects
- The **finalize** method is called by the garbage collector
  - protected void finalize() throws Throwable
- Explicitly invoking the finalize method doesn't trigger the garbage collection immediately
- If you are expecting the finalize method to be called, you probably have a wrong design
- Not that finalize() is **deprecated** since Java 9

Passing Primitive Data Type Arguments

- Primitive arguments, such as int, double, are **passed into methods by value**
- Any changes to the values of the parameters exist only within the scope of the method

```
public static void foo(int i)
{
    i = 2;
}

public static void main(String args[]) {
    int i = 1;
    System.out.println("i = " + i);   //Result is 1
    foo(i);
    System.out.println("i = " + i);   //Result is 1
}
```

Passing Reference Data Type Arguments

- Reference data type parameters, such as objects, are also **passed into methods by value**
- However, the values of the object's fields can be changed in the method
- **Similar to passing by pointer in C++**
- **Java does not support pass by reference!**

C/C++ Pas by Pointer vs Reference

Pass by pointer in C++

```
static void update(string* arg){
    arg = new string("2");
}

int main(){
    string *str = new string("1");
    cout << *str << endl;
    update(str);
    cout << *str << endl;
    return 0;
}
```

```
Output:
1
1
```

Pass by reference in C++

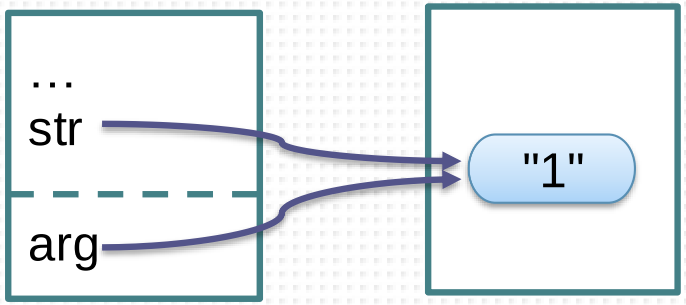
```
static void update(string& arg){
    arg = "2";
}

int main(){
    string str ="1";
    cout << str << endl;
    update(str);
    cout << str << endl;
    return 0;
}
```

```
Prints:
1
2
```

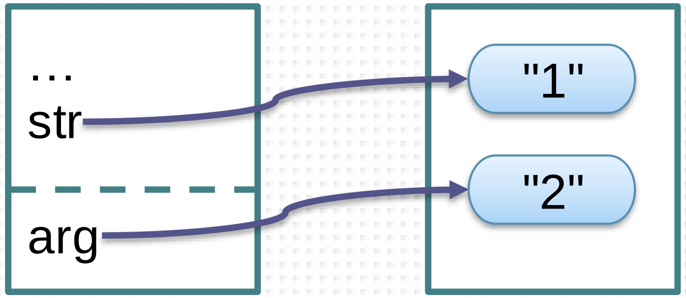
C/C++ Pass by Pointer Example I

```
#include <iostream>
using namespace std;

static void update(string* arg){
    arg = new string("2");
}

int main(){
    string *str = new string("1");
    cout << *str << endl;
    update(str);
    cout << *str << endl;
    return 0;
}
```

Just after the method call



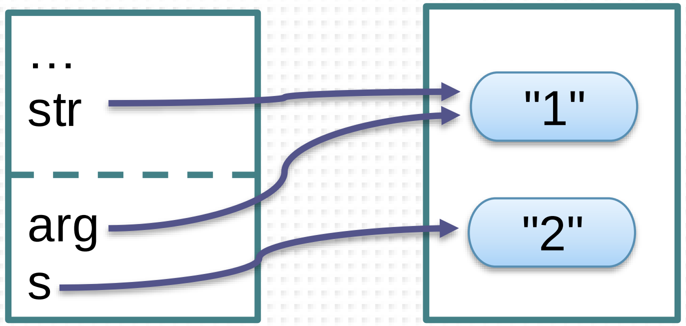
after *arg = new string("2");* line



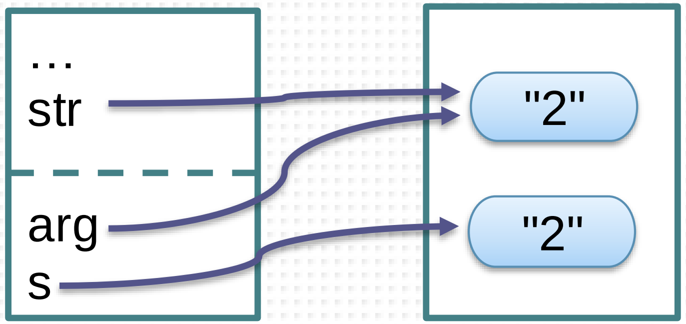
C/C++ Pass by Pointer Example II

```
#include <iostream>
using namespace std;

static void update(string* arg){
    string *s = new string("2");
    *arg = *s;
}

int main(){
    string *str = new string("1");
    cout << *str << endl;
    update(str);
    cout << *str << endl;
    return 0;
}
```

after *string *s = new string("2");*



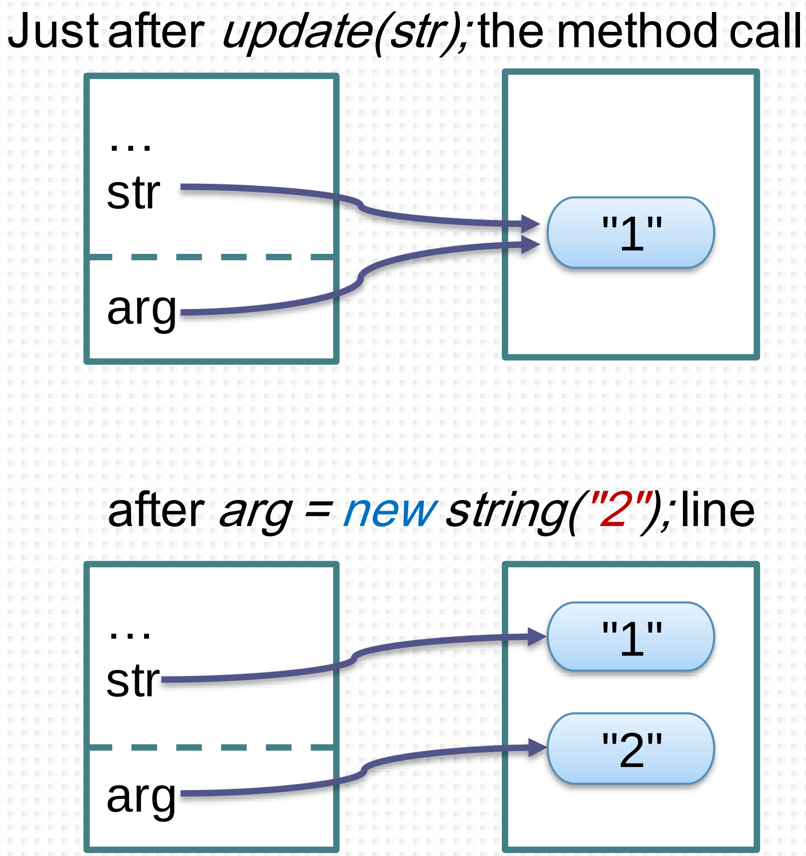
after **arg = s*;*



Java Pass by Value Example I

```
public class HelloWorld{

    public static void update(String arg){
        arg = new String("2");
    }

    public static void main(String args[]) {
        String str = new String("1");
        System.out.println(str);
        update(str);
        System.out.println(str);
    }
}
```

Output:
1
1

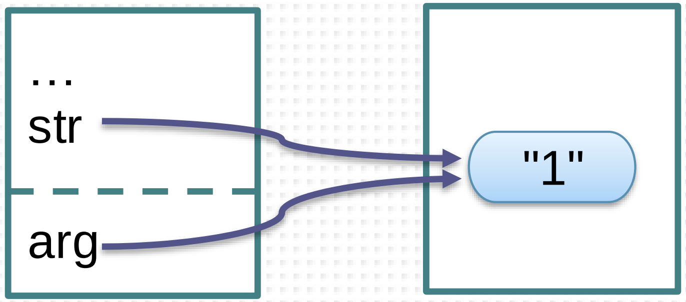


Java Pass by Value Example II

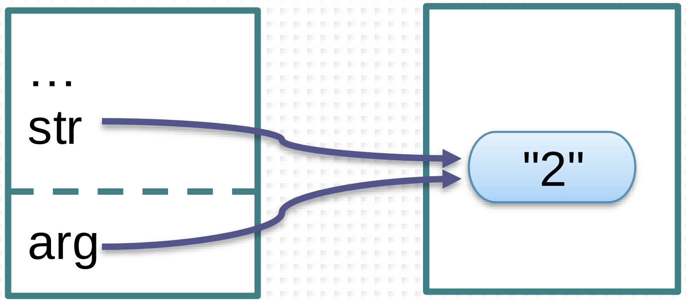
```java
public class HelloWorld{

    public static void update(StringBuffer arg){
        arg.delete(0,arg.length());
        arg.append("2");
    }

    public static void main(String args[]) {
        StringBuffer str = new StringBuffer ("1");
        System.out.println(str);
        update(str);
        System.out.println(str);
    }
}
```

Output:
1
2

Just after *update(str);* the method call



after *update(str);* the method call



Java Pass by Value Example III

```java
public static void foo(List _list){
    _list = new LinkedList <String>();
}

public static void bar(List _list) {
    _list.remove(0);
}

public static void main(String args[]) {
    List<String> list = new LinkedList<String>();
    list.add("element 0");
    System.out.println("List size = " + list.size());
    foo(list);
    System.out.println("List size = " + list.size());
    bar(list);
    System.out.println("List size = " + list.size());
}
```

passed-in reference (list) still references the same object!

values of passed-in reference can be updated

Result 1

Result 1

Result 0

== Operator & Object Equality

- Equality operator == compares values of primitive types but identities of objects
- Use **equals** method to compare two objects
  - Equals use equality operator by default, override this method to provide your own implementation

```
Integer i1 = new Integer("3");
Integer i2 = new Integer("3");

i1 == i1;          //Result is True
i1 == i2;          //Result is False
i1.equals(i2);     //Result is True

Integer i3 = i2;
i2 == i3 ;         //What is the result?
```

Packages

- A Java package is similar to namespace in C++
- Groups of related types are bundled into packages
- Packages avoid naming conflicts and provide Access control

/<project_root>/src/org/arcelik/view/Viev.java

```
package src.org.arcelik.view;

import src.org.arcelik.controller;
import java.swing.*;

public class View extends JPanel {
    //implementation
    src.org.arcelik.controller.C1 c1;
    C2 c2;
}
```

Creating package

Using package

Importing an Entire Package
(may lead to name ambiguity)

Using qualified name

Using package member

Class Visibility

- Public classes are visible to all packages
- Default classes are only visible in the same package

```
package P1;
public class Foo {
   //implementation
}
```

```
package P1;
class Bar {
 //implementation
}
```

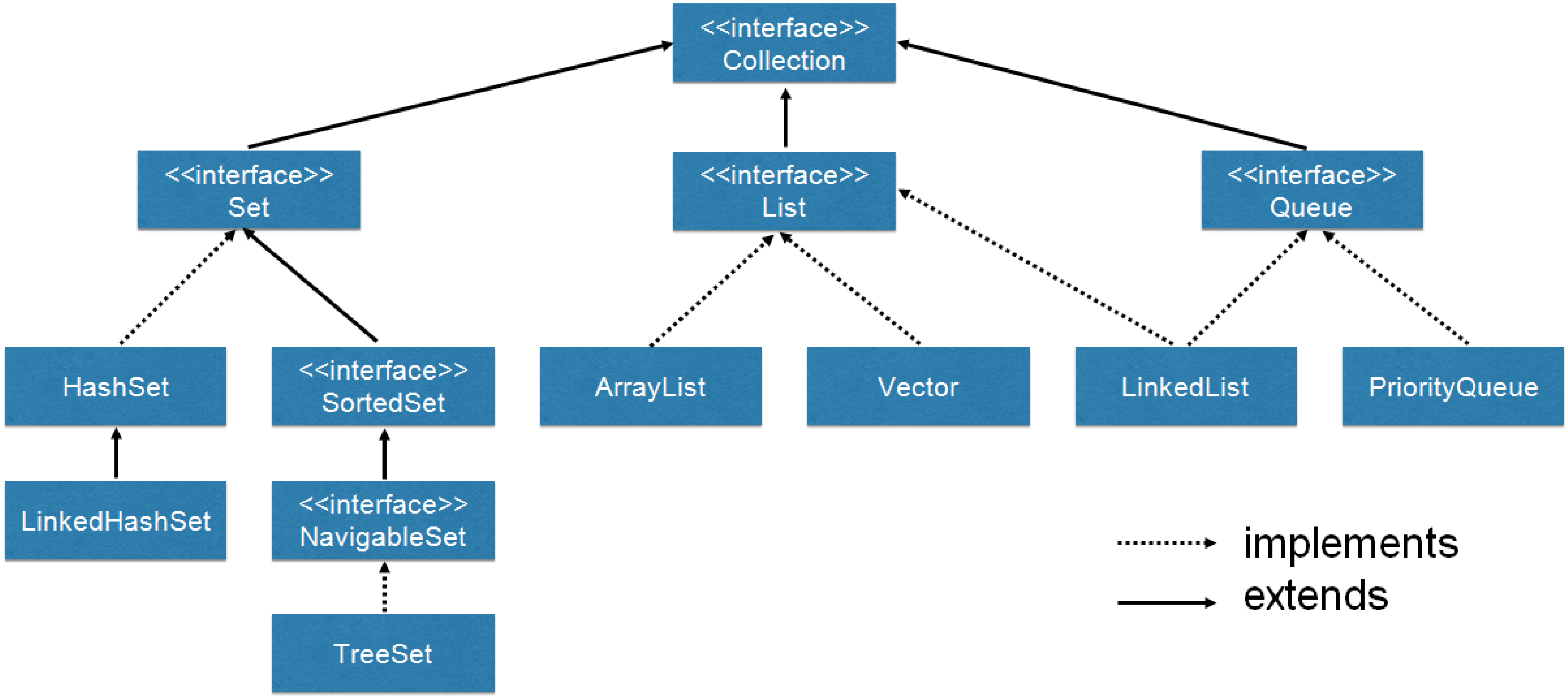
```
package P2;

import P1.*;

public class Test {
    void test() {
        Foo foo;     //ok
        Bar bar;     //compile error
    }
}
```

Access Modifiers (Visibility of Members)

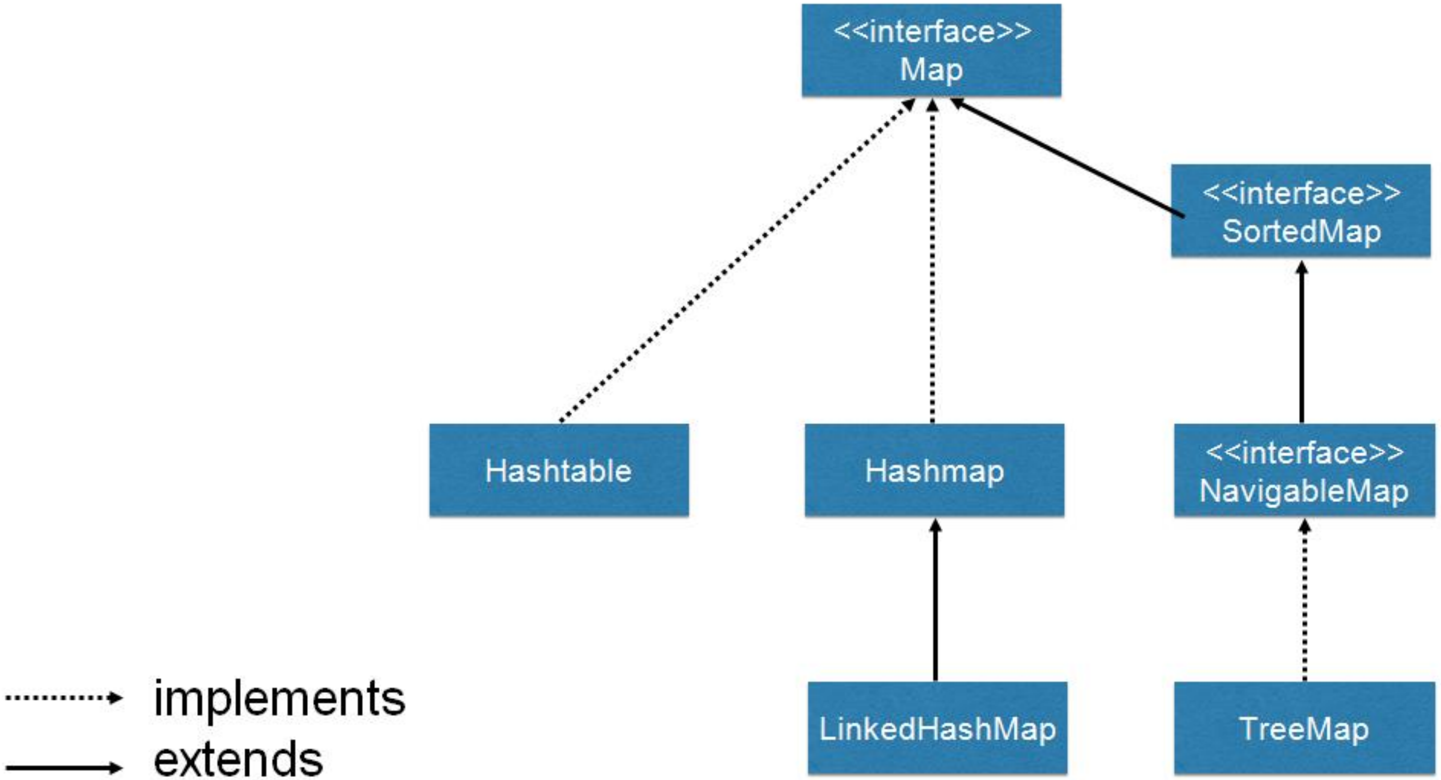
- Fields, constructors and methods can have one of four different Java access modifiers

Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

Collection Interfaces and Classes



Map Interfaces and Classes



Design Patterns

- Design Patterns are divided into three categories

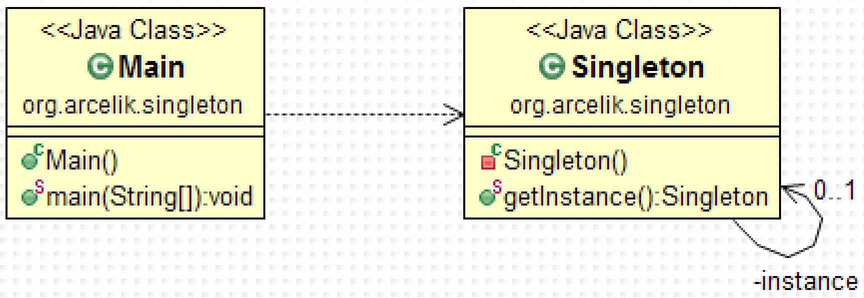
Creational	Structural	Behavioral
Singleton Pattern	Adapter Pattern	Mediator Pattern
Factory Pattern	Composite Pattern	Chain of Responsibility P.
Abstract Factory P.	Proxy Pattern	Observer Pattern
Builder Pattern	Flyweight Pattern	Strategy Pattern
Prototype Pattern	Facade Pattern	Command Pattern
	Bridge Pattern	State Pattern
	Decorator Pattern	Visitor Pattern
		Interpreter Pattern
		Iterator Pattern
		Memento Pattern

Why to Use Design Pattern

- Design patterns proposes a proven solution
- Provides minimal coupling among modules
- Isolating the variability that may exist in the system
- Making the overall system easier to understand and maintain
- Making communication between designers more efficient

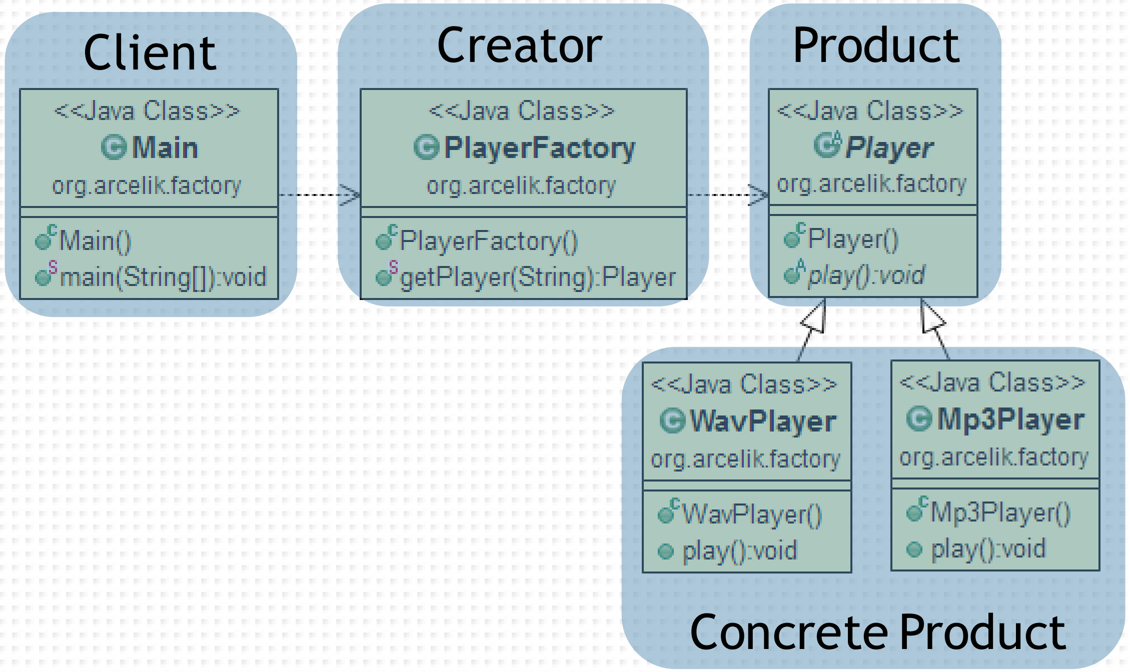
Singleton Design Pattern

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists



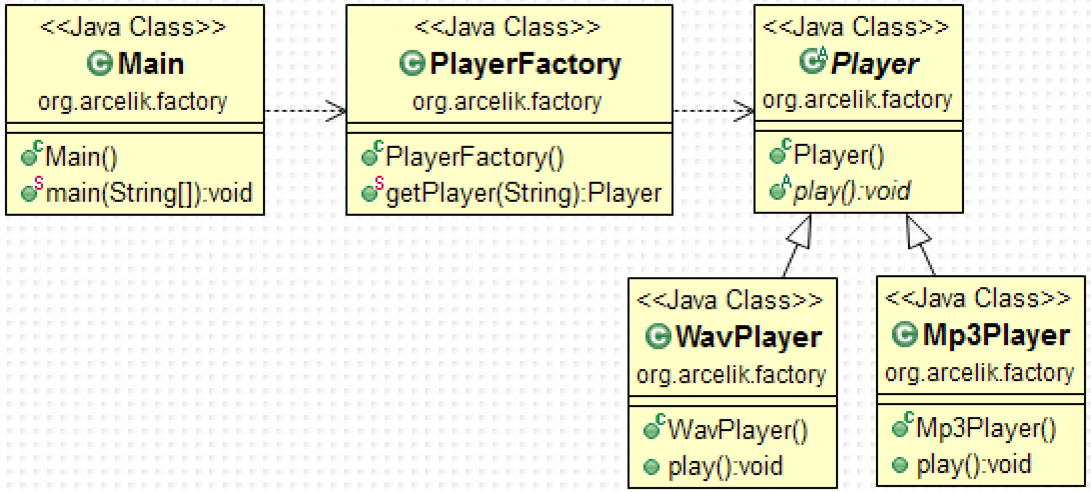
Factory Design Pattern I

- Factory pattern gives responsibility of instantiation of a class to the factory class instead of the client



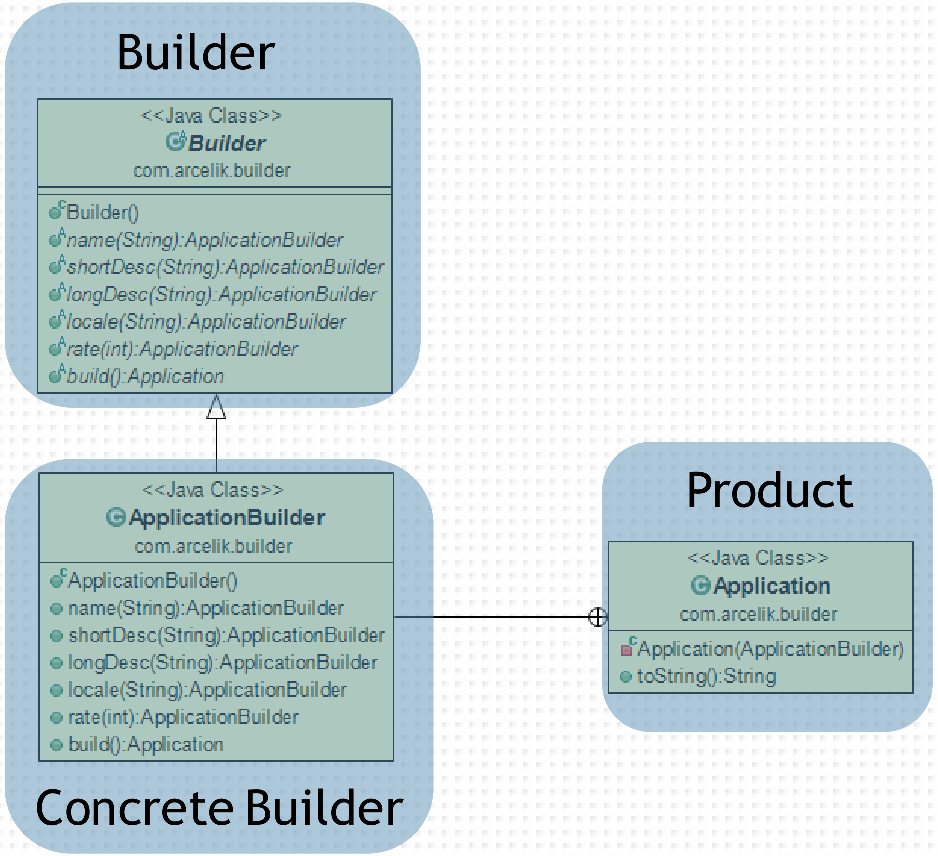
Factory Design Pattern II

- Click here to see the source code on GitHub



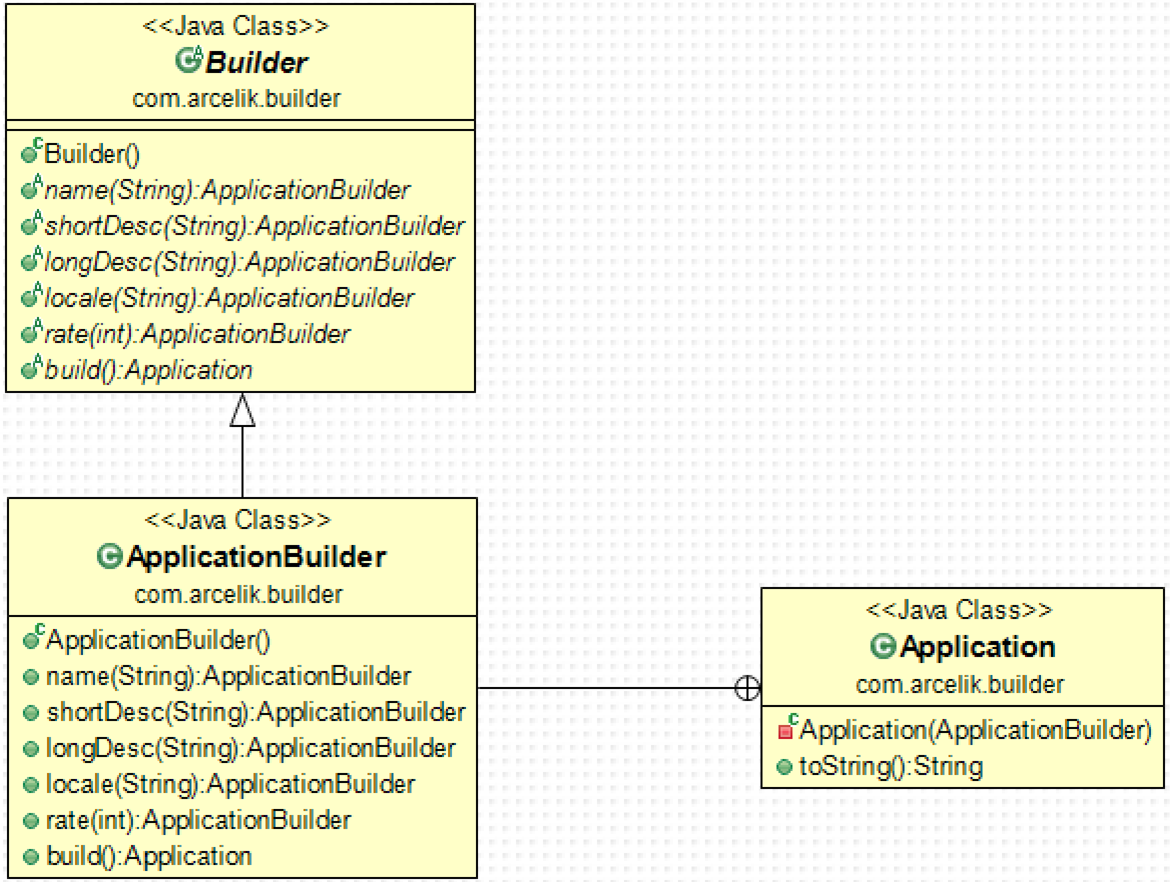
Builder Design Pattern I

- The purpose of the builder pattern is to separate the construction of a complex object from its representation.



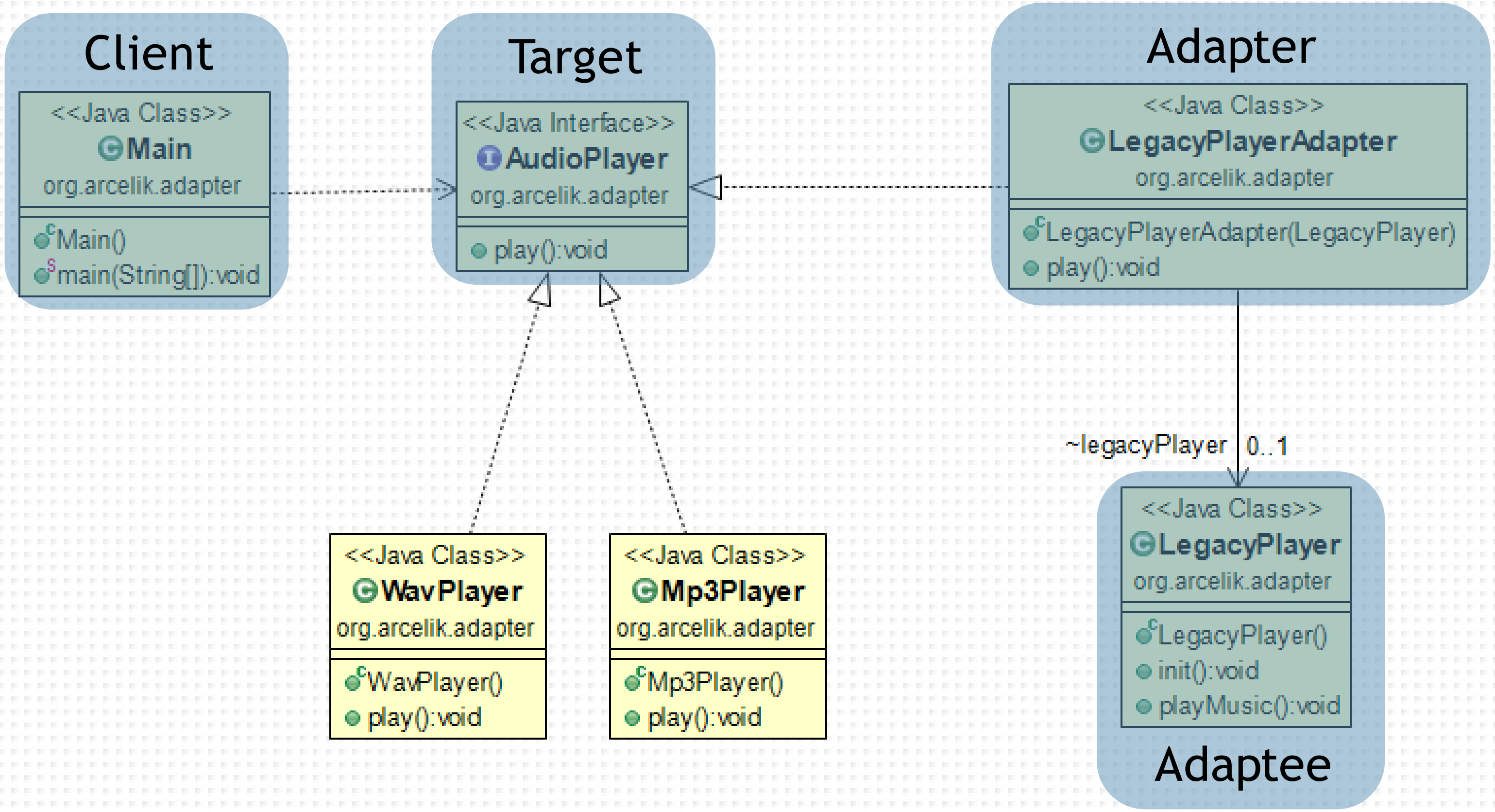
Builder Design Pattern II

- Click here to see the source code on GitHub



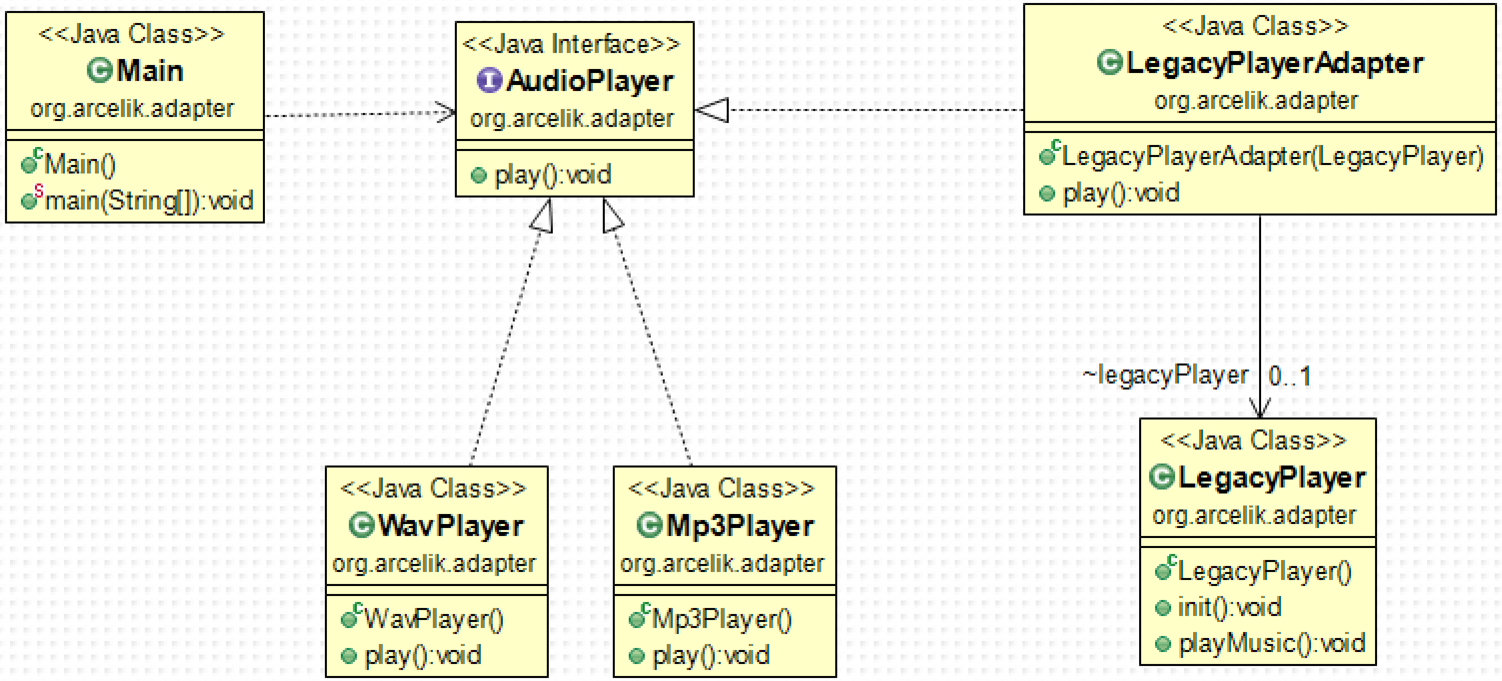
Adapter Design Pattern I

- Adapter pattern converts the interface of a class into another interface clients expect



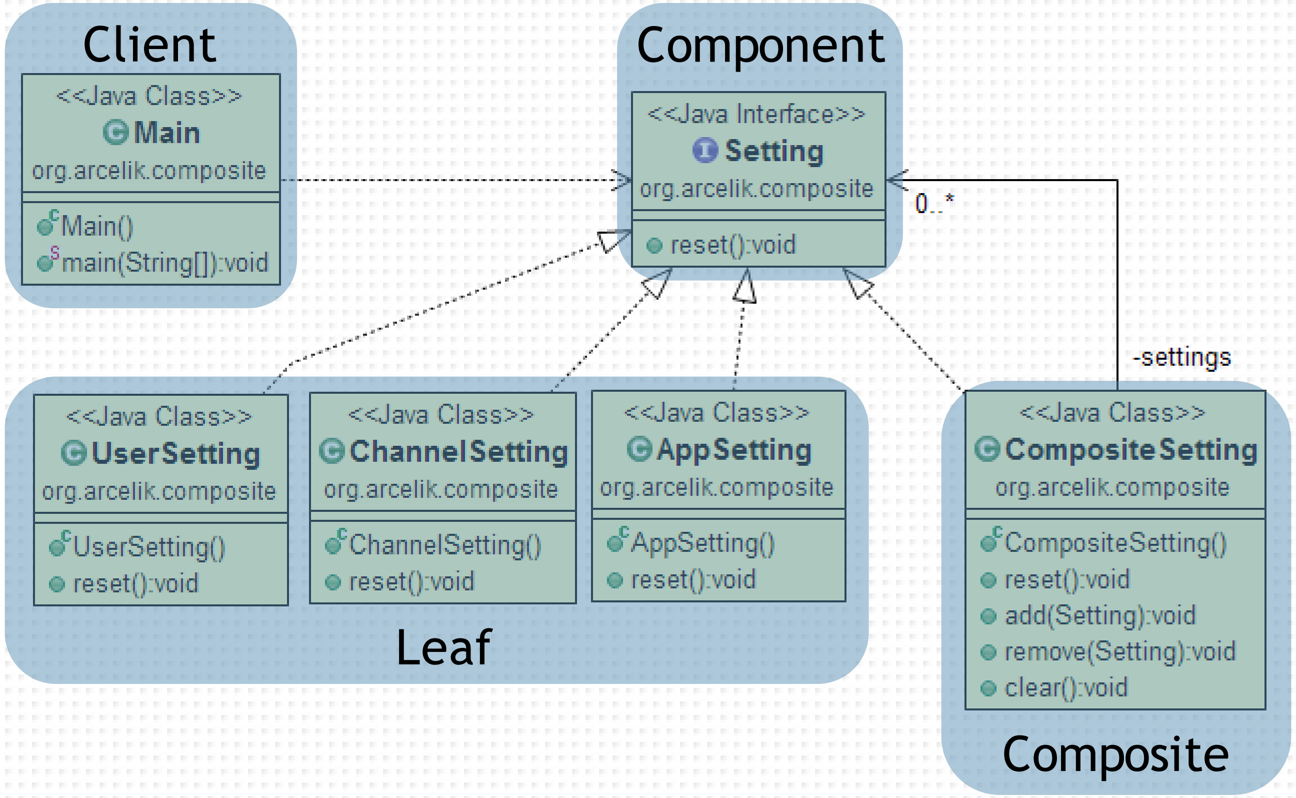
Adapter Design Pattern II

- Click here to see the source code on GitHub



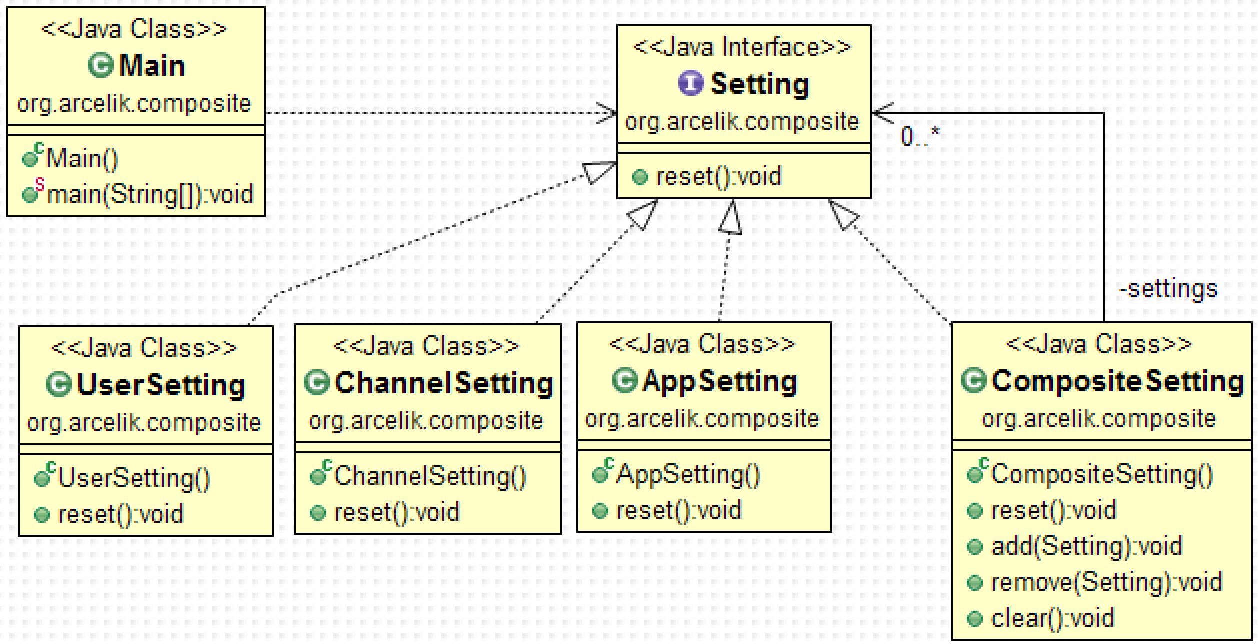
Composite Design Pattern I

- Composite pattern describes a group of objects that is treated the same way as a single instance of the same type



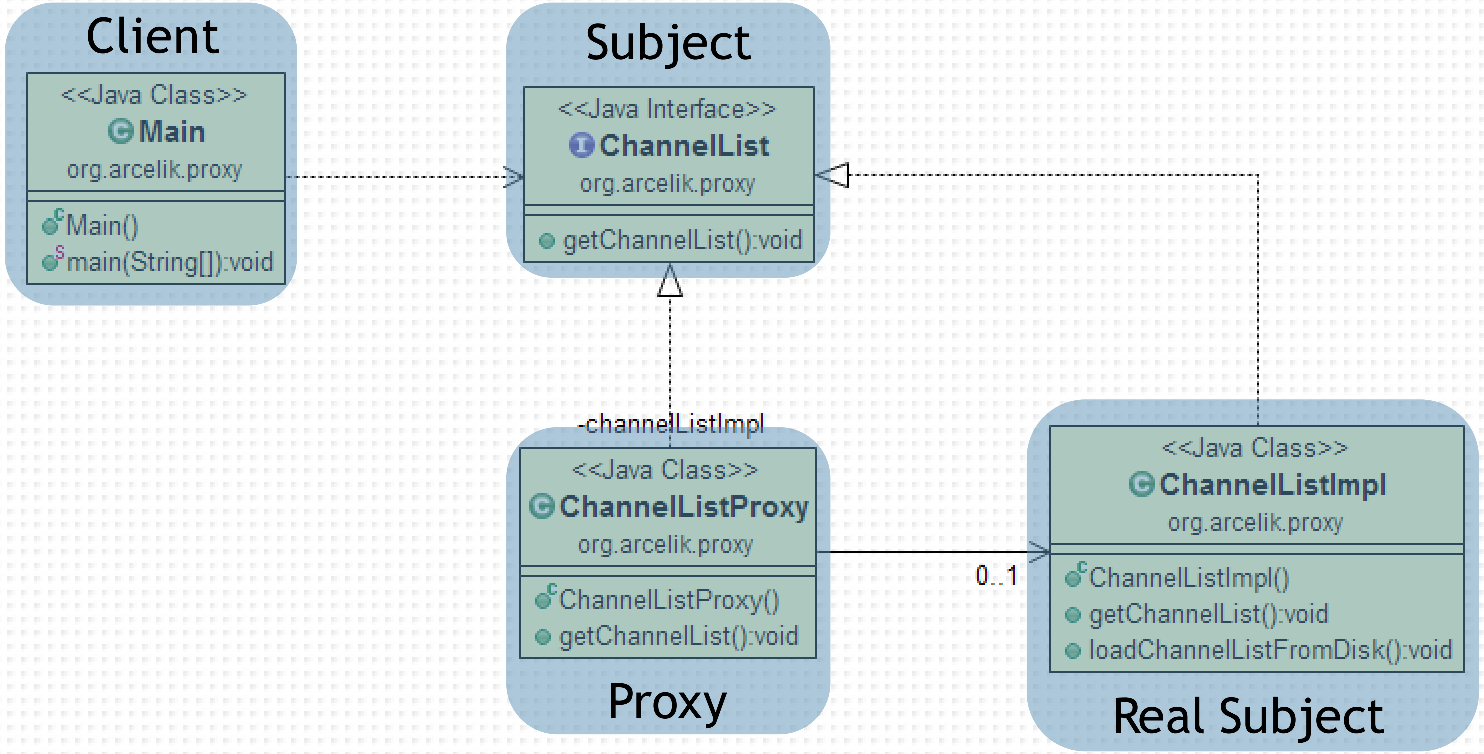
Composite Design Pattern II

- Click here to see the source code on GitHub



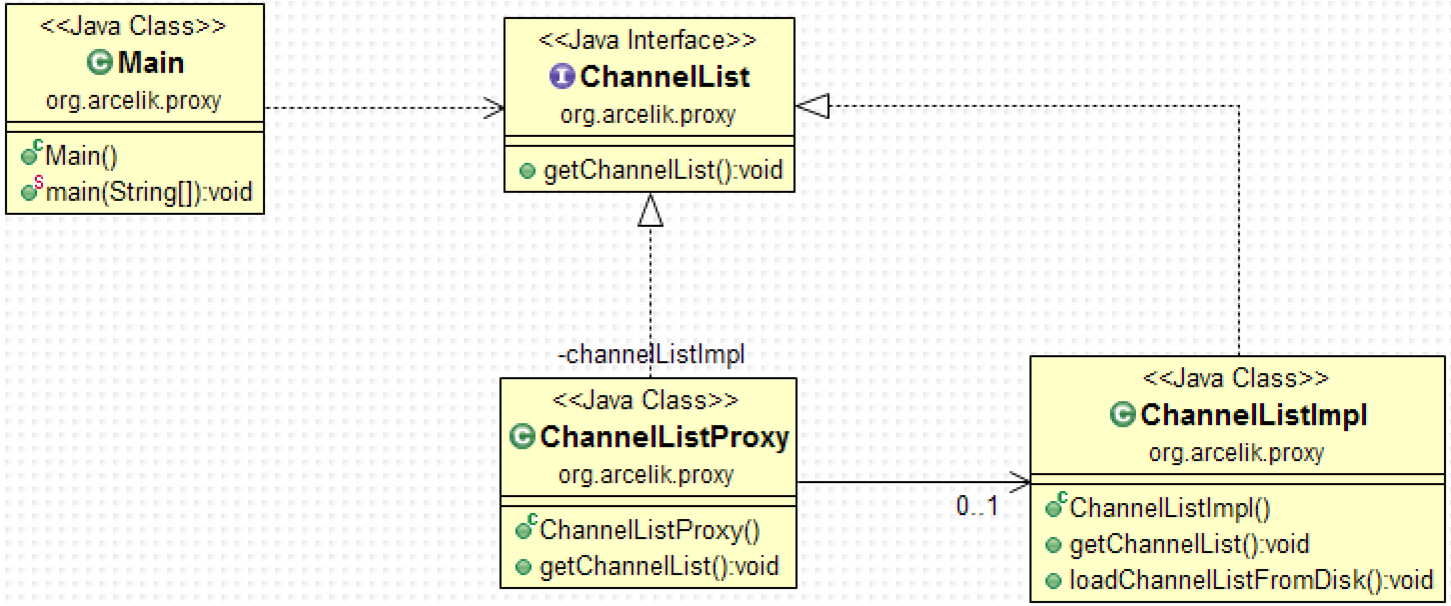
Proxy Design Pattern I

- Proxy pattern provides a substitute or placeholder for another object to provide controlled access of a functionality



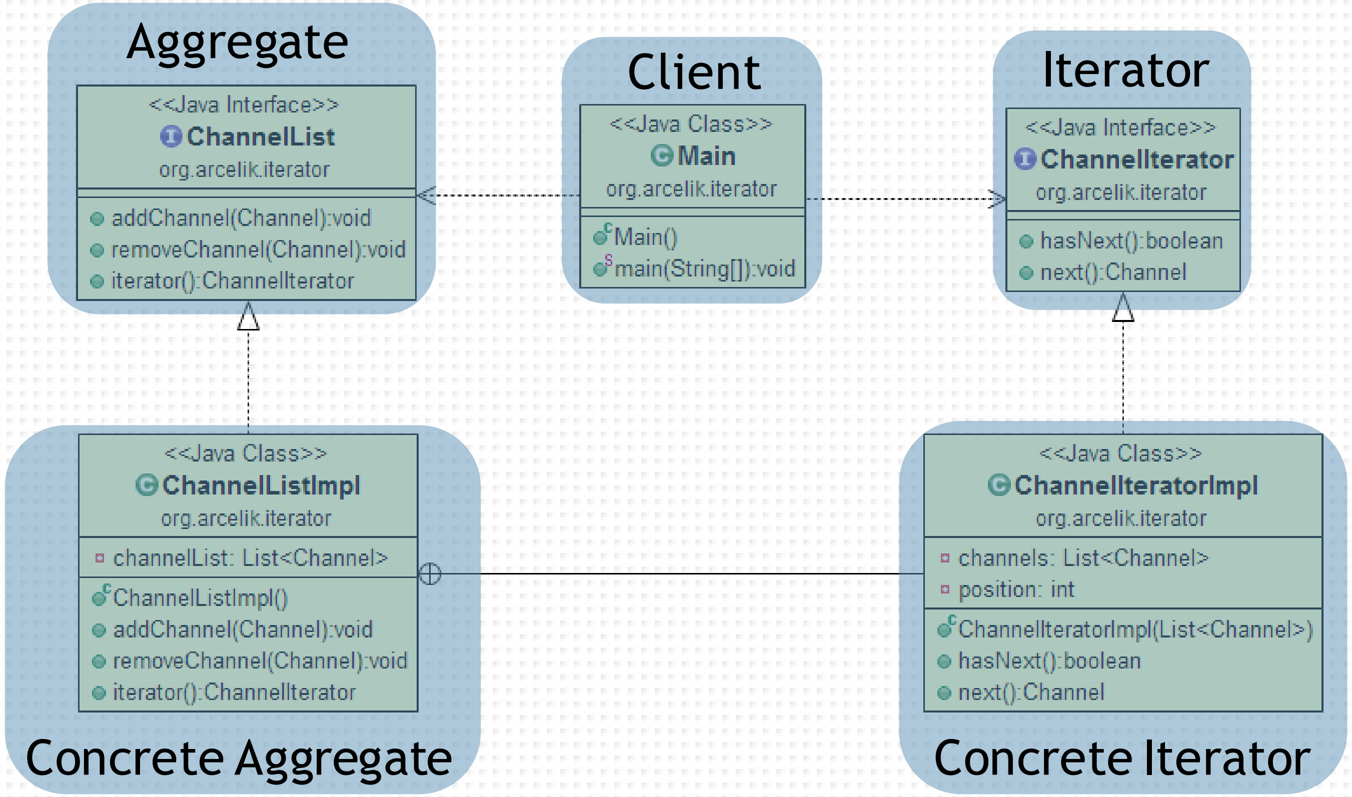
Proxy Design Pattern II

- Click here to see the source code on GitHub



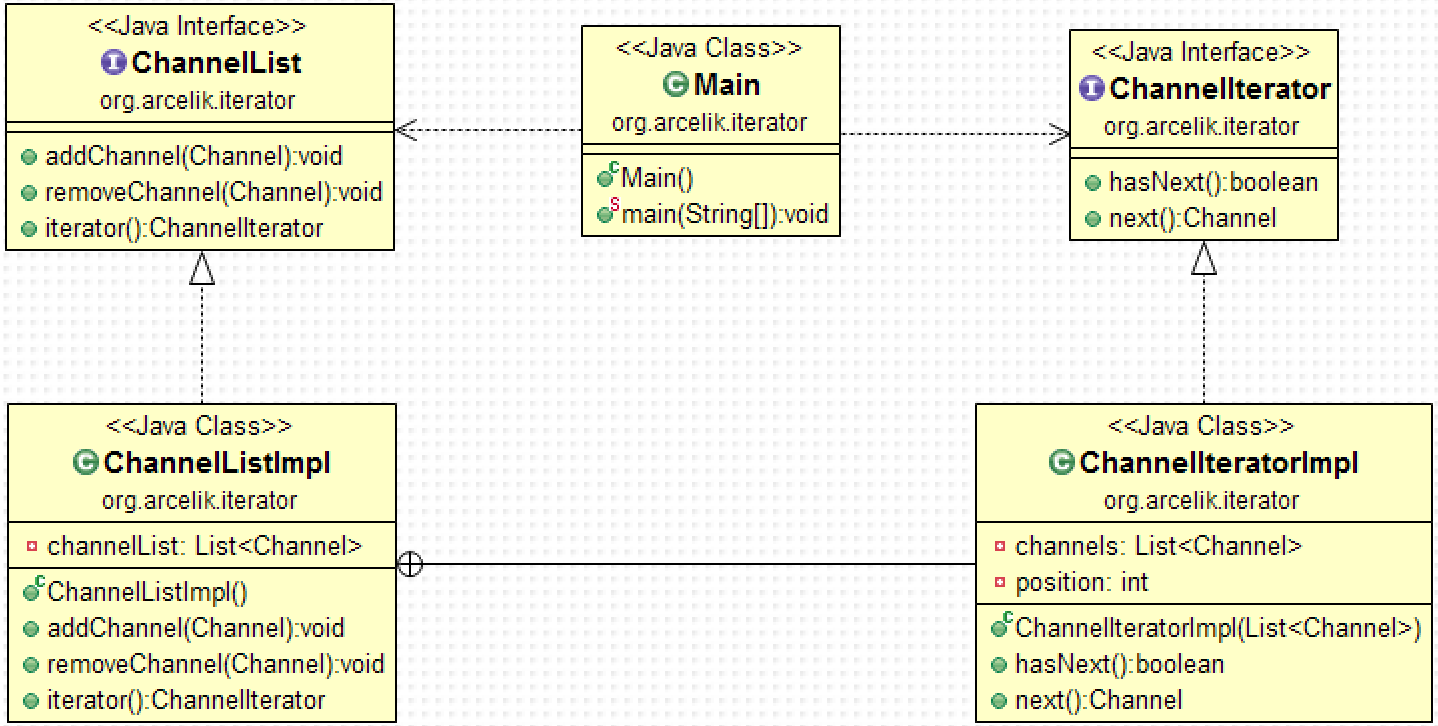
Iterator Design Pattern I

- Iterator pattern provides a way to access the elements of an aggregate object sequentially



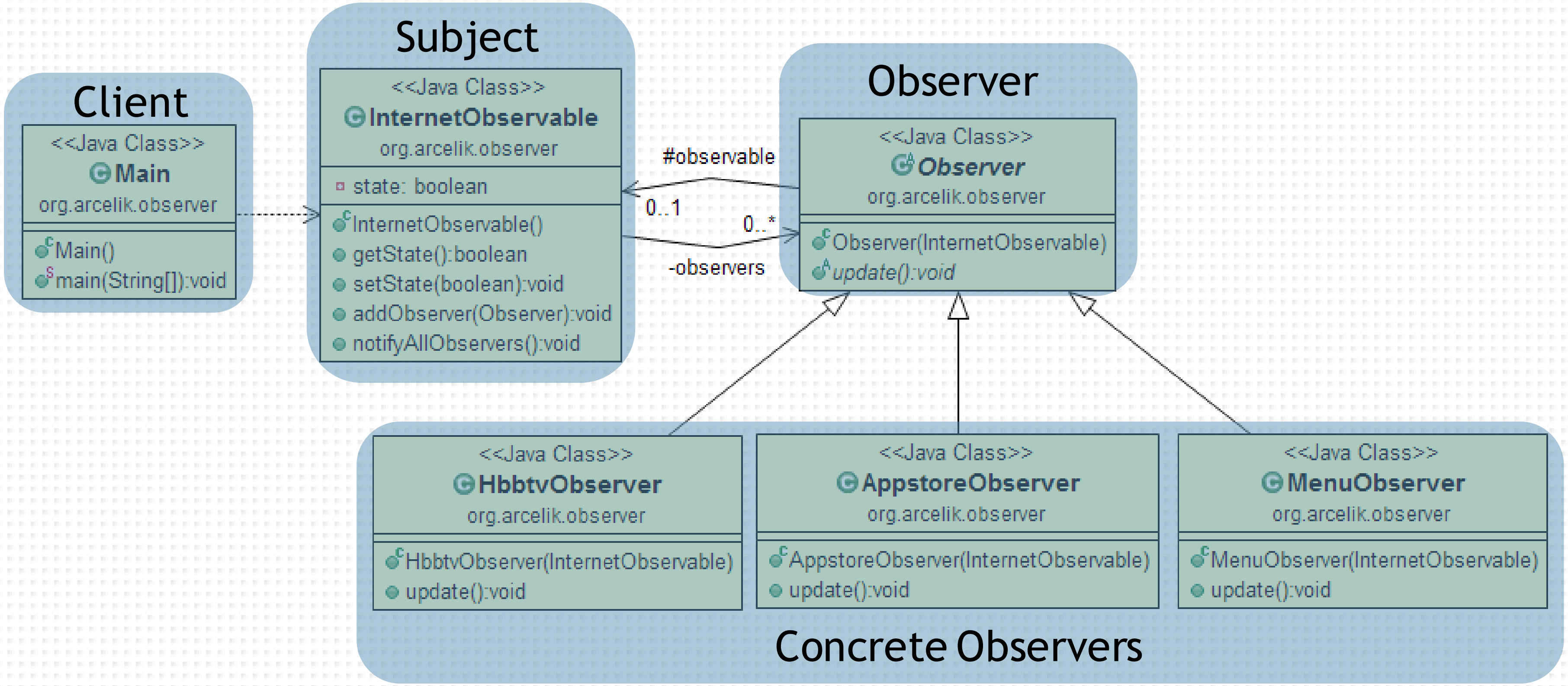
Iterator Design Pattern II

- Click here to see the source code on GitHub



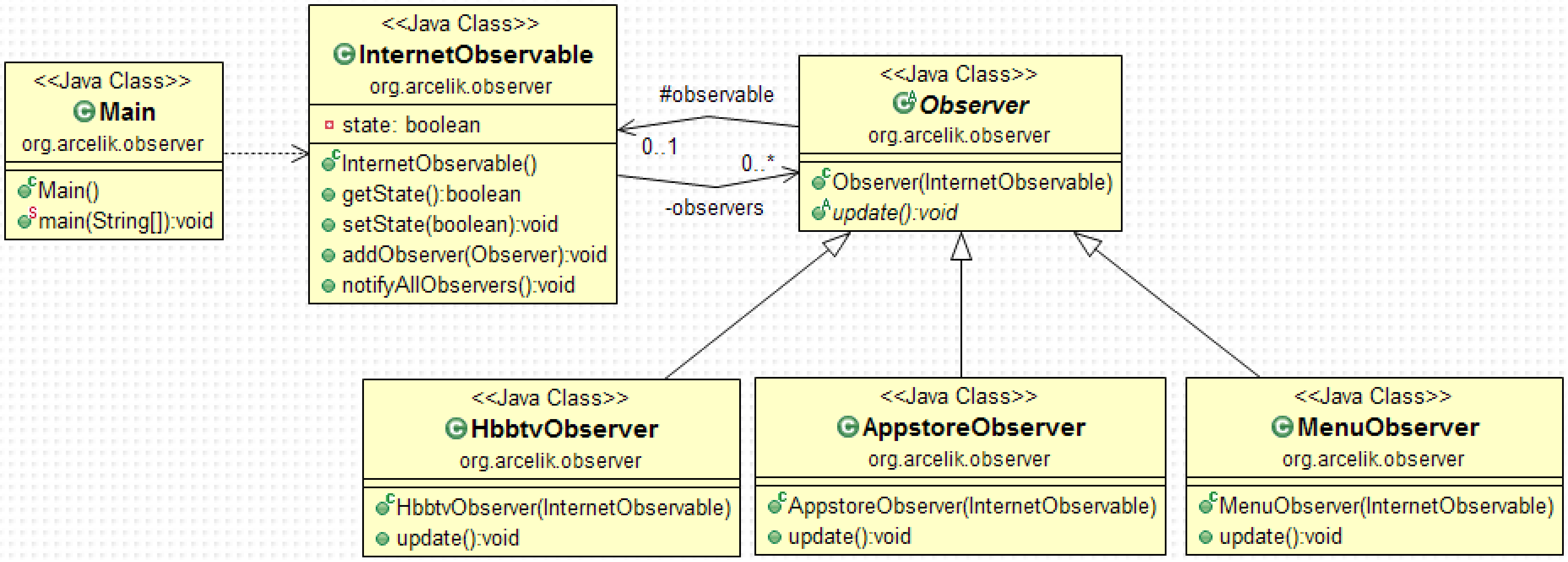
Observer Design Pattern I

- Observer pattern is useful if you are focusing the state of an object and want to get notified whenever there is any change



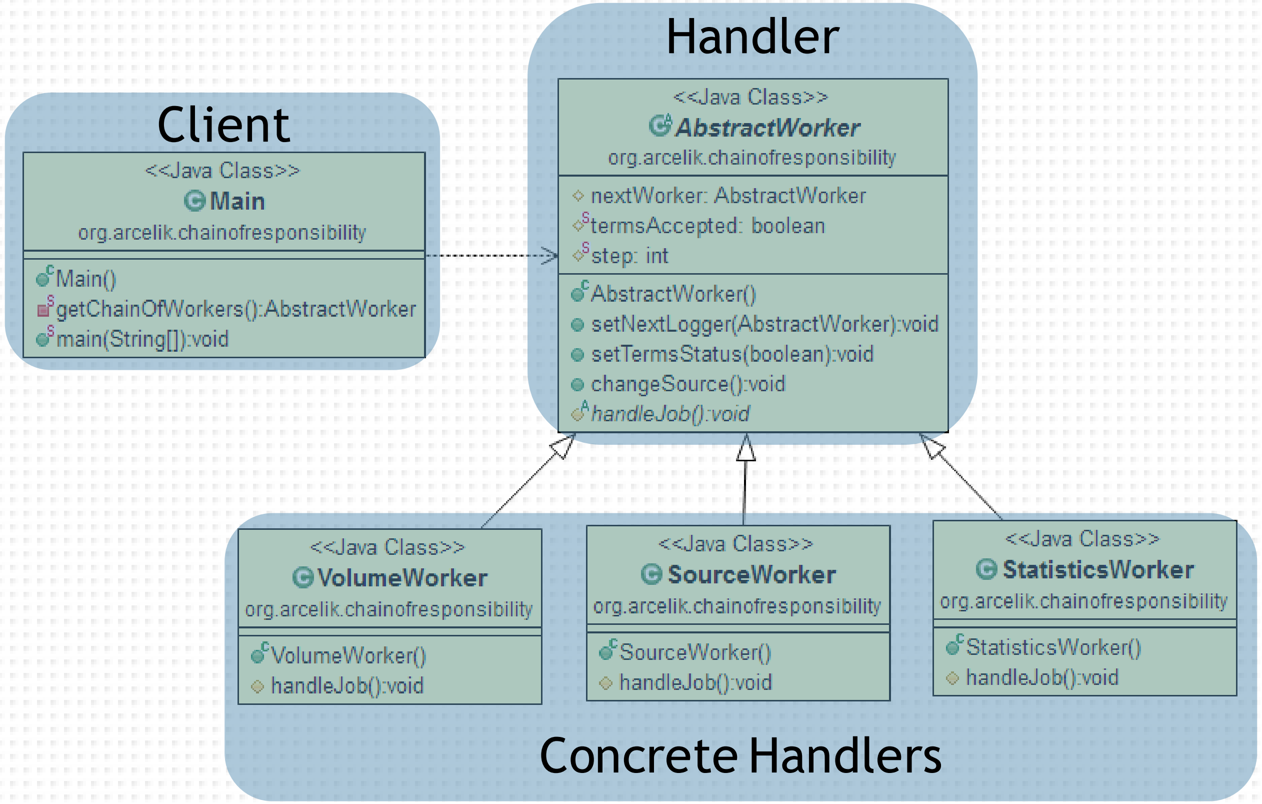
Observer Design Pattern II

- Click [here]() to see the source code on GitHub



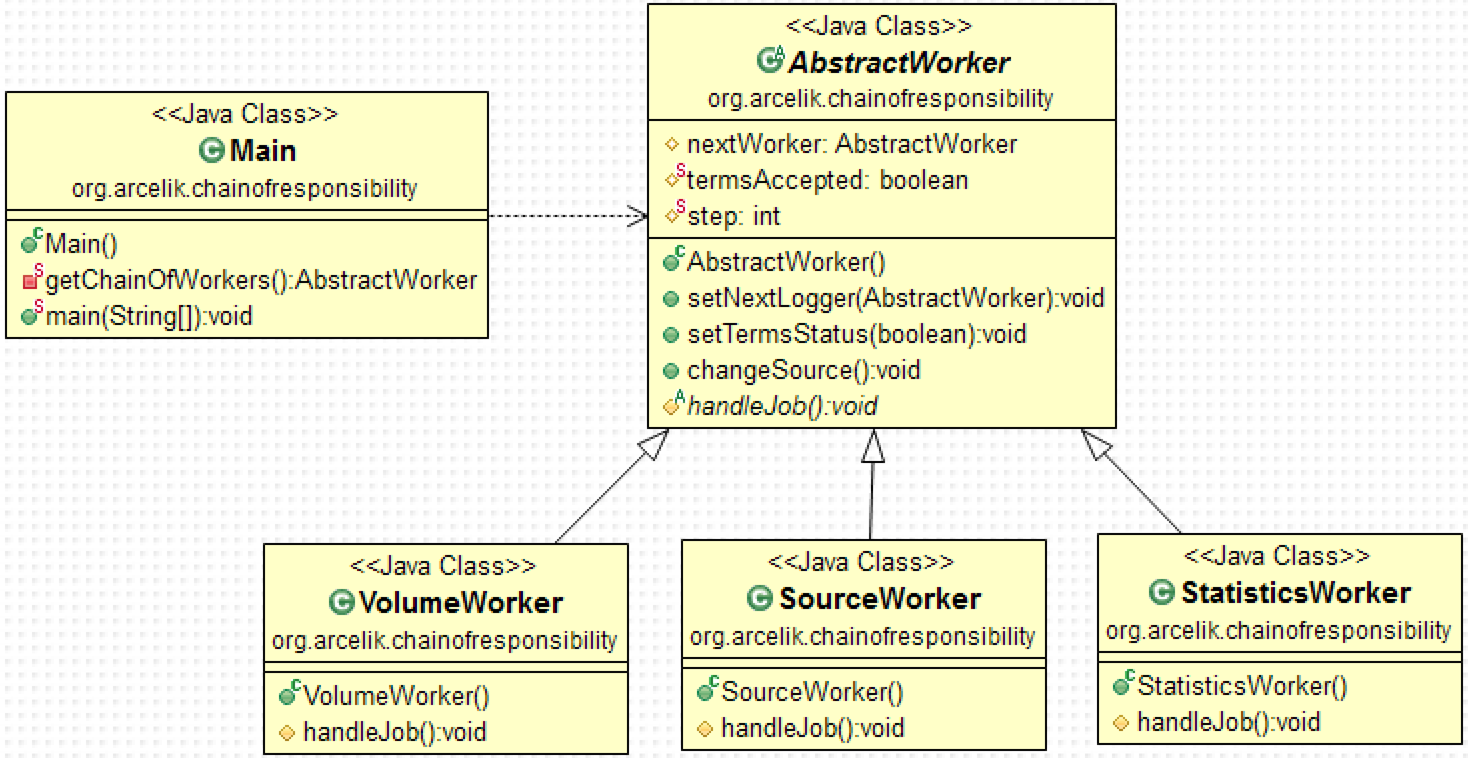
Chain of Responsibility Design Pattern I

- Chain of responsibility pattern is used when a request from client should be passed to a chain of objects to process them



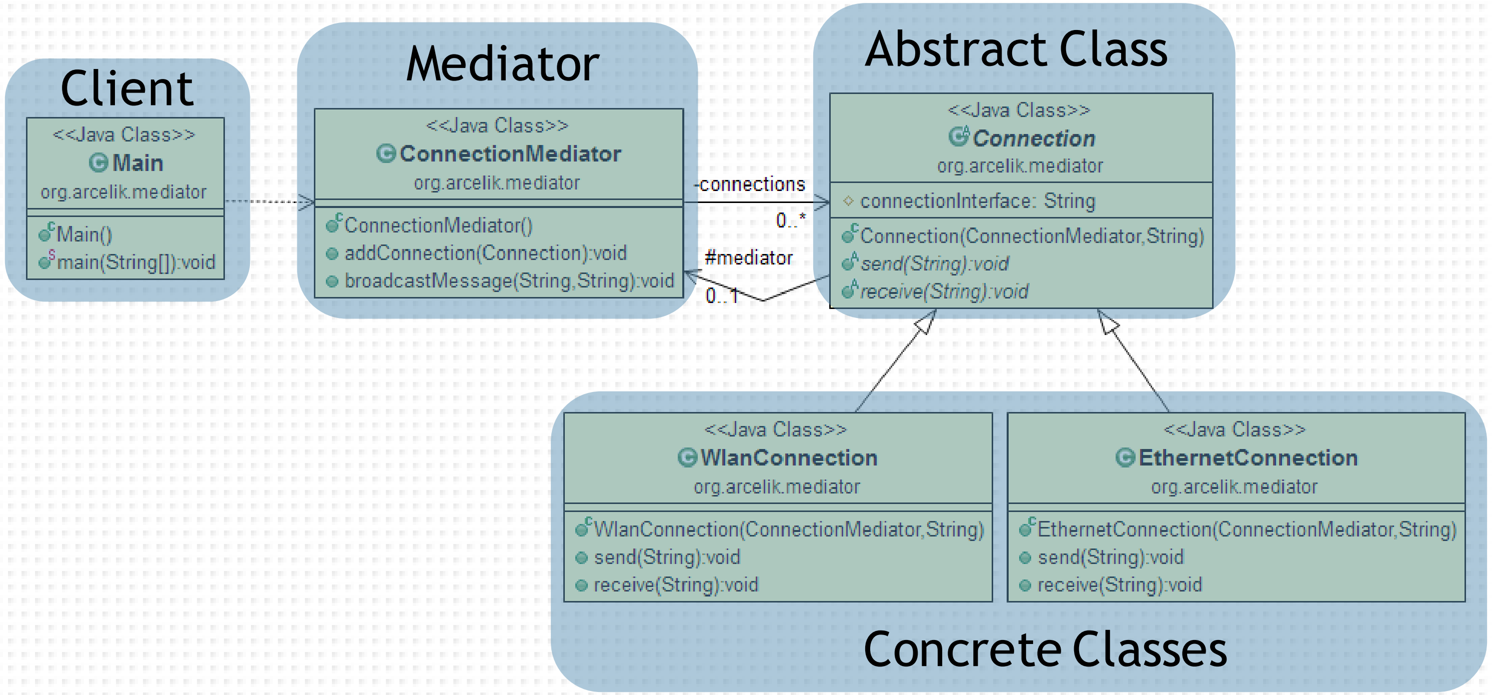
Chain of Responsibility Design Pattern II

- Click here to see the source code on GitHub



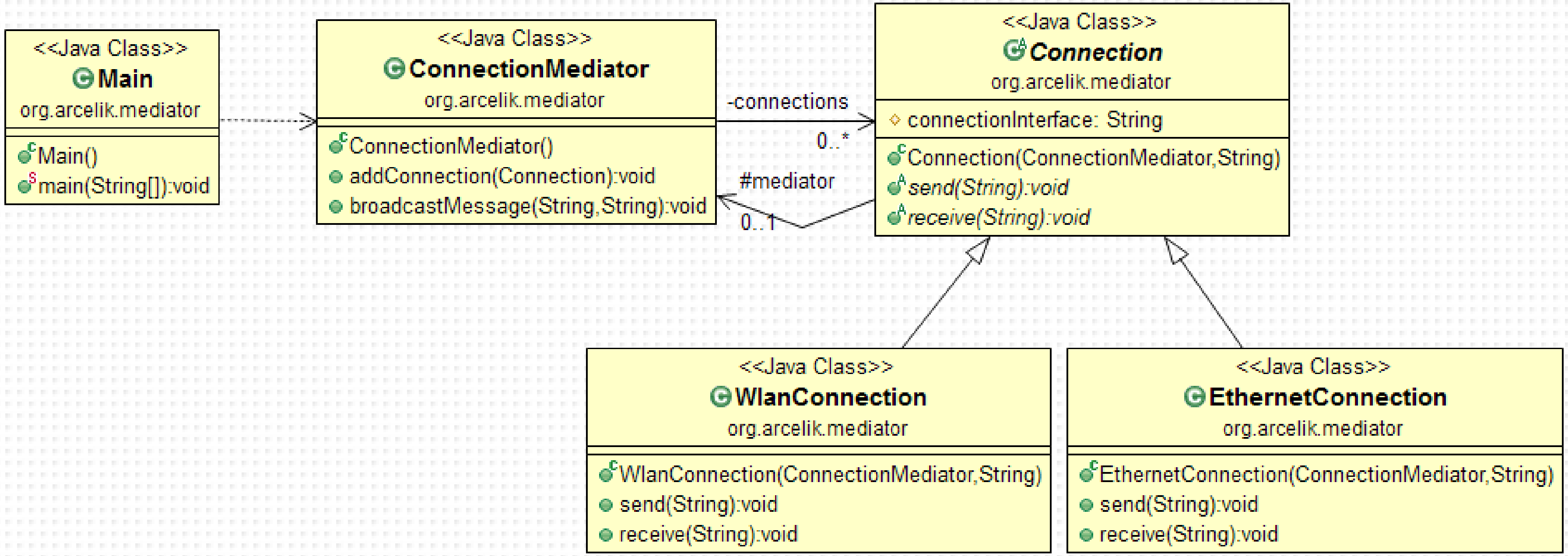
Mediator Design Pattern I

- Mediator pattern provides a centralized communication medium between different objects in a system



Mediator Design Pattern II

- Click here to see the source code on GitHub



QUESTIONS?