CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2018

Doc 5 Iterator & Command
Sep 13, 2018

Iterator Pattern

Provide a way to access the elements of a collection sequentially without exposing its underlying representation

Java Iterators

External

Iterator
- hasNext()
- next()
- remove() Optional
- forEachRemaining

ListIterator
- add(), remove(), set() Optional
- hasNext(), hasPrevious()
- next(), previous()
- nextIndex(), previousIndex()

SplitIterator
- forEachRemaining() + others
- For concurrent processing

Internal

forEach

Java Iterator

```
LinkedList<Strings> strings =  new LinkedList<Strings>();

code to add strings

Iterator<String> list = strings.iterator();
while (list.hasNext()){
    String element = list.next();
    if (element.size % 2 == 0)
    System.out.println(element);
    }
}
```

```
for (String element : strings) {
    if (element.size % 2 == 0)
        System.out.println(element);
}
```

Syntax sugar for above

Python Iterator

```
a = ['house', 'car', 'bike']

for x in a:
      print(x)

items_iterator = iter(a)
print( next(items_iterator))
print( next(items_iterator))
print( next(items_iterator))          #error raised here
```

Java 8

New Features
- New Time, Date & Calendar classes
- Improvements to Cryptographic classes
- Nashorn JavaScript Engine
- Concurrency Improvements
  - Accumulators, Adders
- Default Methods
- Functional language features
  - **Lambda Expressions**
  - **Collection Streams (internal iterators)**

Lambda & Closure

Lambda

Function without a name

Closure

Store the environment with the function

Lambda Expression - Python

```
inc = lambda n : n + 1
result = inc(11)
print(result)              # 12


multi_args = lambda a, b : a + b
result = multi_args(1,2)
print(result)              # 3
```

```
def adder(n):
      return lambda k : k + n

add5 = adder(5)
add9 = adder(9)
result = add5(1)
print(result)              # 6

result = add9(1)
print(result)              # 10
```

adder shows that Python lambdas are also closures

Motivating Example - Sorting

```
a = ['house', 'car', 'bike']

a.sort()
print(a)
```

['bike', 'car', 'house']

```
a.sort(key = lambda x: len(x))
print(a)
```

['car', 'bike', 'house']

```
a.sort(key = len)
```

Lambda - Swift

```
let inc = {x in x + 1}
inc(2)                          // 3


let multiArgs = {(x, y) -> Int in x + y}
multiArgs(1,2)                              //3
```

```
func adder(n:Int) -> (Int) -> Int {
    return {x in x + n}
}

let add5 = adder(n:5)
add5(1)                             // 6
```

```
func rollingSum(n:Int) -> (Int) -> Int {
    var sum = n
    return {x in
        sum = sum + x
        return sum
    }
}

let add = rollingSum(n:0)
add(5)                              // 5
add(2)                              // 7
```

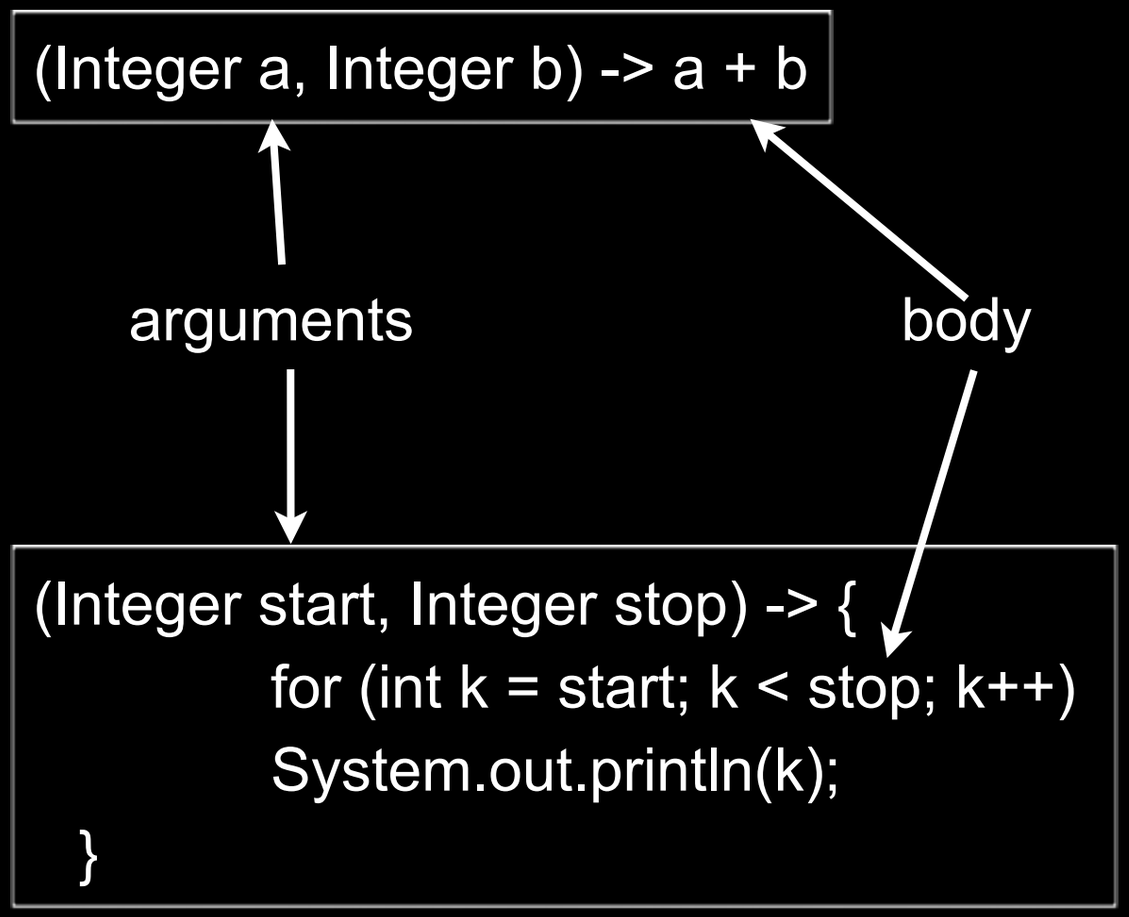
Documentation

Java lambda Tutorial
http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Java 8 Lambdas, Warburton, O'Reilly Media, 2014
http://libproxy.sdsu.edu/login?url=http://proquest.safaribooksonline.com/

Lambda Expression

Anonymous Function



Short Version of Lambda Syntax

(String text) -> text.length();

text -> text.length();

(Integer a, Integer b) -> a + b

(a, b) -> a + b

Using Lambdas

Function<String,Integer> length = **text -> text.length();**

int nameLength = **length.apply**("Roger Whitney");

BiFunction<Integer,Integer,Integer> adder = **(a, b) -> a + b;**

int sum = **adder.apply**(1, 2);

```
import java.util.function.Function;

public class MainExample {
    public static void main(String[] args) {
        Function<String,Integer> length =  text -> text.length();
        System.out.print(length.apply("Roger"));
    }
}
```

Other Types of Lambdas

```
Predicate<Integer> isLarge =  value -> value > 100;
if (isLarge.test(59))
    System.out.println("large");


Consumer<String> print = text -> System.out.println(text);
print.accept("hello World");

    int size = xxx;
Supplier<List> listType = size > 100 ? (()-> new ArrayList()): (() -> new Vector());
List elements = listType.get();
System.out.println(elements.getClass().getName());
```

Lambda Types

New - See java.util.function Interfaces

- Predicate<T> -- a boolean-valued property of an object
- Consumer<T> -- an action to be performed on an object
- Function<T,R> -- a function transforming a T to a R
- Supplier<T> -- provide an instance of a T (such as a factory)
- UnaryOperator<T> -- a function from T to T
- BinaryOperator<T> -- a function from (T, T) to T

Pre-existing

- java.lang.Runnable
- java.util.concurrent.Callable
- java.security.PrivilegedAction
- java.util.Comparator
- java.io.FileFilter
- java.beans.PropertyChangeListener
- etc.

Functional Interfaces

Interface with one method

Can be used to hold a lambda

java.lang.Runnable

void run()

Runnable Example

```
Runnable test = () -> System.out.println("hello from thread");
Thread example = new Thread(test);
example.start();
```

OnClickListener Example

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View source) {
        makeToast();
    }
});
```

```
button.setOnClickListener( event -> makeToast());
```

Internal Iterator - forEach

```
String[] rawData = {"cat", "can", "bat", "rat"};
List<String> data = Arrays.asList(rawData);
data.forEach( word ->System.out.println(word) );
```

Java Iterators

External

Iterator

ListIterator

When you don’t need all items

When complicated logic

```
Iterator<String> list = strings.iterator();
while (list.hasNext()){
    String element = list.next();
    if (element.size % 2 == 0)
    System.out.println(element);
    }
}
```

Internal

forEach

When you want all items

Easier to implement

Pattern Parts

Intent

Motivation

Applicability

Structure

Participants

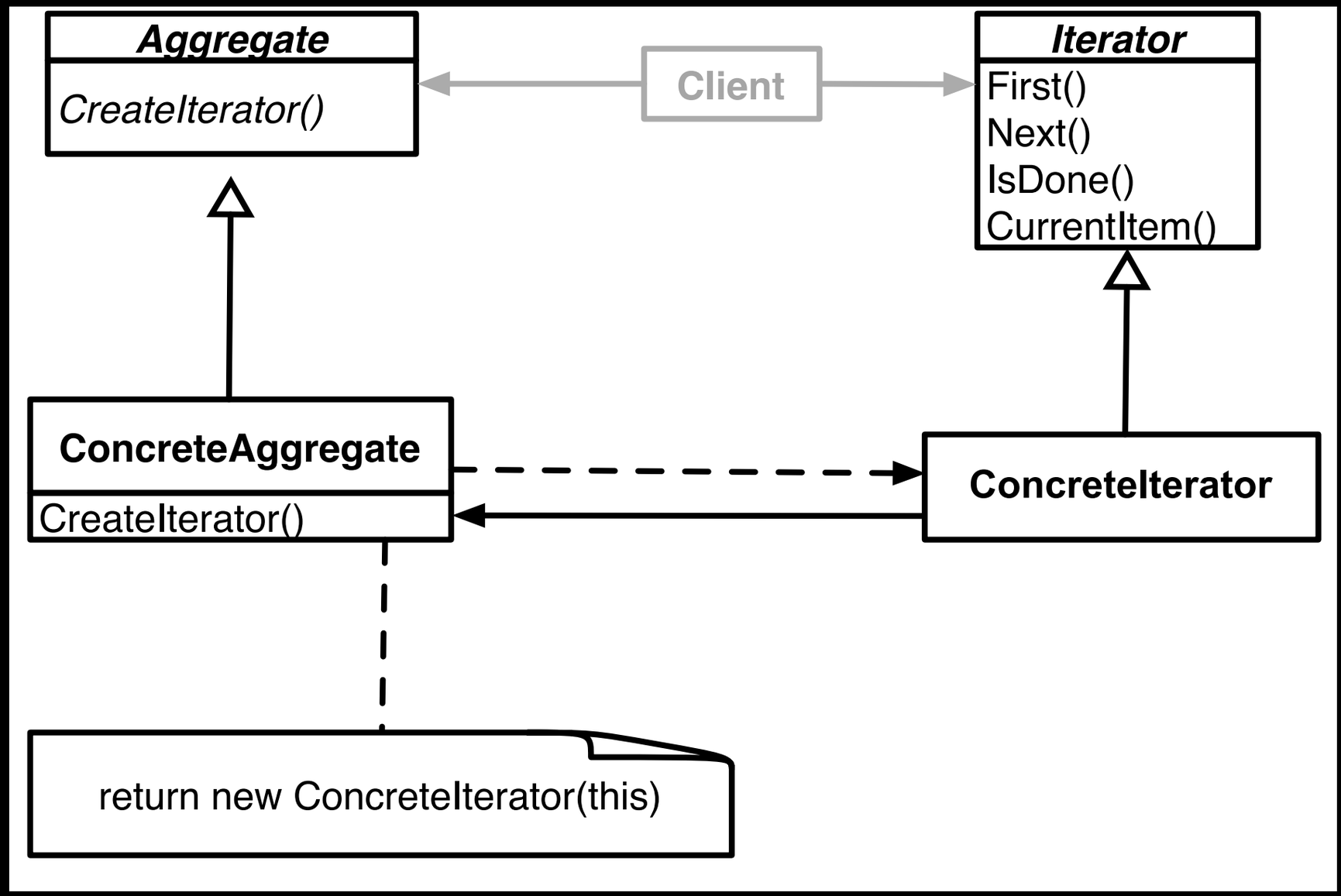
Collaborations

Consequences

Implementation

Sample Code

Iterator Structure

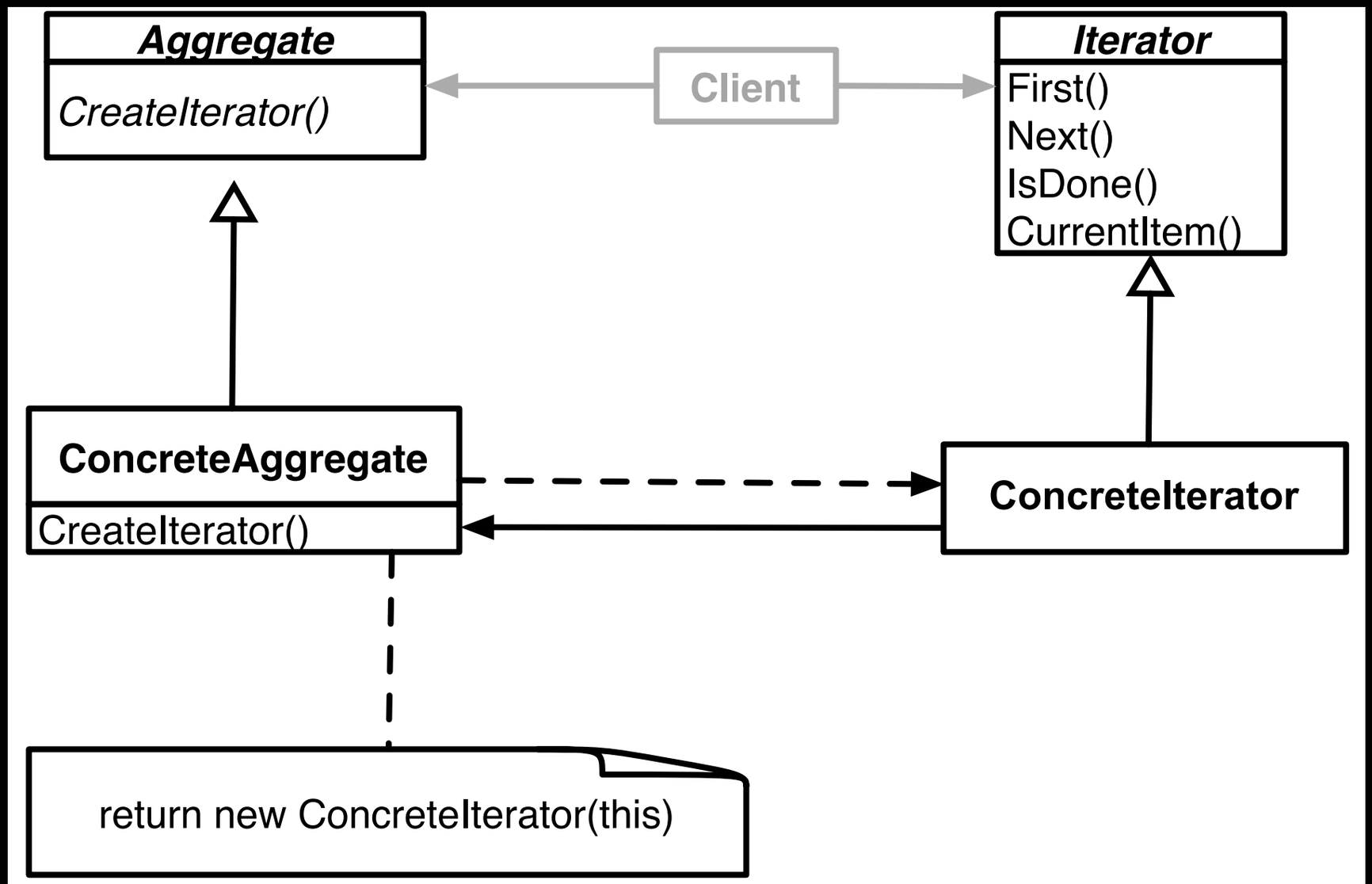


Iterator Structure & Names

Aggregate, ConcreteAggregate, Client, ConcreteIterator

Roles that classes will perform

Classes will not have those names



Issues - Concrete vs. Polymorphic Iterators

Concrete

```
Reader iterator = new StringReader( "cat");
int c;
while (-1 != (c = iterator.read() ))
   System.out.println( (char) c);
```

Polymorphic

```
Vector listOfStudents = new ArrayList();

// code to add students not shown

Iterator list = listOfStudents.iterator();
while ( list.hasNext() )
   System.out.println( list.next() );
```

Memory leak issue in C++, Why?

Issue - Who Controls the Iteration?

External (Active)

```
var numbers = new LinkedList();

code to add numbers

Vector evens = new Vector();
Iterator list = numbers.iterator();
   while ( list.hasNext() ) {
    Integer a = (Integer) list.next();
    int b = a.intValue();
    if ((b % 2) == 0)
        evens.add(a);
}
```

Internal (Passive)

```
numbers  = LinkedList.new

code to add numbers

evens = numbers.find_all { |element| element.even? }
```

Issue - Who Defines the Traversal Algorithm

Object being iterated

Iterator

Issue - Robustness

What happens when items are added/removed from the iteratee while an iterator exists?

```
Vector listOfStudents = new Vector();

// code to add students not shown

Iterator list = listOfStudents.iterator();
listOfStudents.add( new Student( "Roger") );

list.hasNext();              //What happens here?
```

Java Stream

Stream java.util.stream.Stream

Sequence of values

Operations on the values

Operations are chained together into pipelines

Example

```
String[] words = {"a", "ab", "abc", "abcd", "bat"};
List<String> wordList = Arrays.asList(words);
List<String> longWords
longWords = wordList.stream()
                        .filter( s -> s.length() > 2)
                        .filter( s -> s.charAt(0) == 'a')
                        .map( s -> s.toUpperCase())
                        .collect( Collectors.toList());
System.out.println(longWords);
```

Lazy Evaluation

```
String[] words = {"a", "ab", "abc", "abcd", "bat"};
List<String> wordList = Arrays.asList(words);
List<String> longWords
longWords = wordList.stream()
                    .filter( s -> s.length() > 2)
                    .filter( s -> s.charAt(0) == 'a')
                    .map( s -> s.toUpperCase())
                    .collect( Collectors.toList());
System.out.println(longWords);
```

Only One pass of List to do all operations

4.0 gpa

```
List<Student>  = students.stream()
                                .filter( student -> student.gpa() >= 4.0)
                                .collect(Collectors.toList());
```

Stream methods

count()
distinct
filter
findAny
findFirst
flatMap
forEach
forEachOrdered
limit
map
max
min
nonMatch
reduce
sorted

For More Information

State of the Lambda: Libraries Edition

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html

http://tinyurl.com/mshjfkj

State of the Lambda

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html

http://tinyurl.com/kg5m9zu

Ruby Iterator Examples

a = [1, 2, 3, 4]

a.each {|x| puts x}	1 2 3 4
result = a.collect {|x| x + 10} puts result	11 12 13 14
result = a.find_all {|x| x > 2 } puts result	3 4
puts a.any? {|x| x > 2}	true
puts a.detect {|x| x > 2 }	3

Python

```
a = ['house', 'car', 'bike']

def is_even(n) :
      return n % 2 == 0

result = map(len, a)
b = list(result)
print(b)                                # [5, 3, 4]

even = filter(is_even, map(len, a))
print(list(even))                       # [4]
print(list(even))                       # [ ]
```

Some Higher Order Functions

reduce

Processes a collection to a single value (which could be a collection)

filter

Select elements of a collection

map

Transforms elements of a collection

reduce

Common pattern

loop through a collection to compute some result

```
let data = [1,1,2,3,5,8]

var sum = 0
for n in data {
    sum += n
}

var product = 1
for n in data {
    product *= n
}
```

```
let easySum = data.reduce(0, +)

let easyProduct = data.reduce(1, *)
```

More Reduce Examples

```
let words = ["The", "cat", "in", "the", "hat"]

let title = words.reduce("", {$0 + " " + $1})                    // “ The cat in the hat”


let data = [1,8,1,2,3,5]
let maxElement = data.reduce(data[0], {max($0, $1)})
```

Filter

```
let data = [1,8,1,2,3,5]

let foo = data.filter( {$0 > 3})          // [ 8, 5] Swift 2.3 & 3.0

let foo = data.filter {$0 > 3}            // don't need the ( )


let largeSum = data.filter {$0 > 3}
                   .reduce(0, +)
```

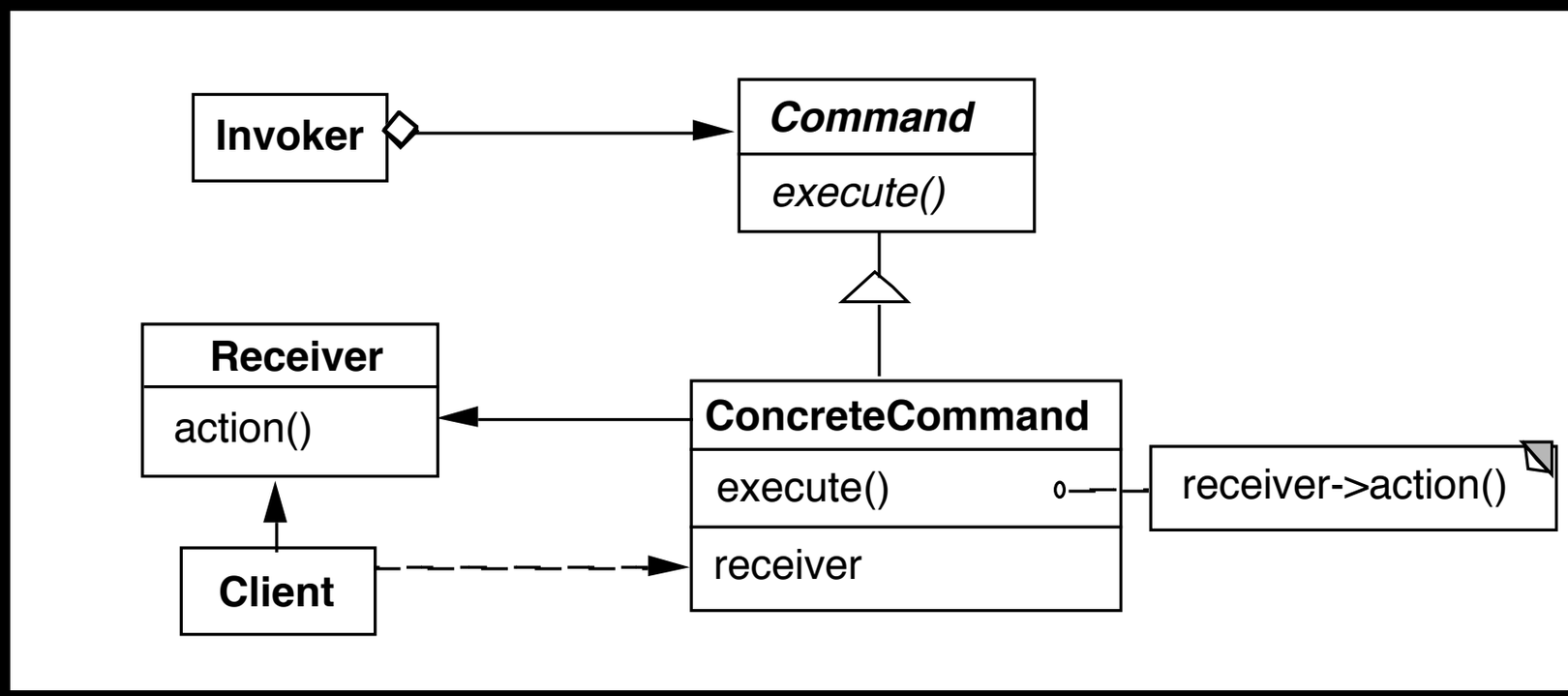
map

```
let data = [1,1,2,3,5,8]

let fiveAdded = data.map{$0 + 5}                    // [6, 6, 7, 8, 10, 13] Swift 2.3 & 3.0


let students = ["Sam": 3.2, "Pete": 3.9, "Jill": 3.7]

let scores = students.map({$0.1})                   // [3.2, 3.9, 3.7]

let sumOfSquares = data.map {$0 * $0}.reduce(0, +)  // 104
```

Command

Command

Encapsulates a request as an object



Example

Invoker be a menu

Client be a word processing program

Receiver a document

Action be save

Sample Command

```
public abstract class Command {
public abstract void execute();
public abstract void undo();
}

public class IncreaseCommand extends Command {
private Counter subject;

public IncreaseCommand(Counter toIncrease) {
subject = toIncrease;

public abstract void execute() { subject.increase() };

public abstract void undo() { subject.decrease() };
}
```

Sample Command - Text Editing

Requires more details

- Text that is being edited
- Location in text to changed
- Replacement text

Undo requires

- Text that is being edited
- Location in text that was changed
- Text that was replaced

When to Use the Command Pattern

Need action as a parameter (replaces callback functions)
Lambda's replace this use

Specify, queue, and execute requests at different times

Undo

Logging changes

High-level operations built on primitive operations

- A transaction encapsulates a set of changes to data

- Systems that use transaction often can use the command pattern

Macro language

Consequences

Command decouples the object that invokes the operation from the one that knows how to perform it

It is easy to add new commands, because you do not have to change existing classes

You can assemble commands into a composite object

Pluggable Commands

Can create one general Command using reflection

Don’t hard code the method called in the command

Pass the method to call an argument

Java Example of Pluggable Command

```
import java.util.*;
import java.lang.reflect.*;

public class Command
    {
    private Object receiver;
    private Method command;
    private Object[] arguments;

    public Command(Object receiver, Method command,
                   Object[] arguments )
    {
    this.receiver = receiver;
    this.command = command;
    this.arguments = arguments;
    }

public void execute() throws InvocationTargetException,
                             IllegalAccessException
    {
    command.invoke( receiver, arguments );
    }
}
```

Using the Pluggable Command

```
public class Test {
   public static void main(String[] args) throws Exception
      {
      Vector sample = new Vector();
      Class[] argumentTypes = { Object.class };
      Method add =
         Vector.class.getMethod( "addElement", argumentTypes);
      Object[] arguments = { "cat" };
      Command test = new Command(sample, add, arguments );
      test.execute();
      System.out.println( sample.elementAt( 0));
      }
}
```

Output

cat

Pluggable Commands using Lambdas

```
public interface Command {
void execute();
}

public class PluggableCommand {
Command do;
Command undo;

public PluggableCommand(Command do, Command undo) {
this.do = do;
this.undo = undo;
}

public void execute() { do.execute(); }

public void undo() { undo.execute(); }
```

Pluggable Commands using Lambdas

```
final Counter example = new Counter();
PluggableCommand increase;

increase = new PluggableCommand(
() -> example.increase(),
() -> example.decrease());

increase.execute();
```

Note
Java's lambdas put restrictions on the variable example

Command Pattern & Lambda

Lambda's can replace command objects for
Callbacks
Batch processing
Logging
Macro language

Functional Programming & Command

Simple cases - can just use function

But what if function needs
State
Receiver

Closures

```
function counter()
  n = 0
  return () -> n += 1
end
```

```
counter_a = counter()
counter_b = counter()
counter_a()                    # 1
counter_a()                    # 2
counter_a()                    # 3
counter_b()                    # 1
```

So functions can maintain state

With Multiple Functions

```
function counter(start = 0)
  n = start
  return () -> n += 1, () -> n = start
end
```

```
(plus_a, reset_a) = counter(10)
(plus_b, reset_b) = counter()

plus_a()                    # 11
plus_a()                    # 12
reset_a()                   # 10
plus_b()                    # 1
```

General Command

```
type Command
  execute::Function
  undo::Function
end

function execute(command::Command)
  command.execute()
end

function undo(command::Command)
  command.undo()
end
```

```
function counter(start)
  n = start
  return Command(()-> n += 1, ()-> n -= 1)
end


  count = counter(5)
execute(count)          # 6
undo(count)             # 5
```