

# Principles of Software Construction: Objects, Design, and Concurrency

## Design Patterns

**Christian Kästner**     Vincent Hellendoorn





# One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

# Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

# A third design scenario

- Suppose we need to sort a list in different orders...

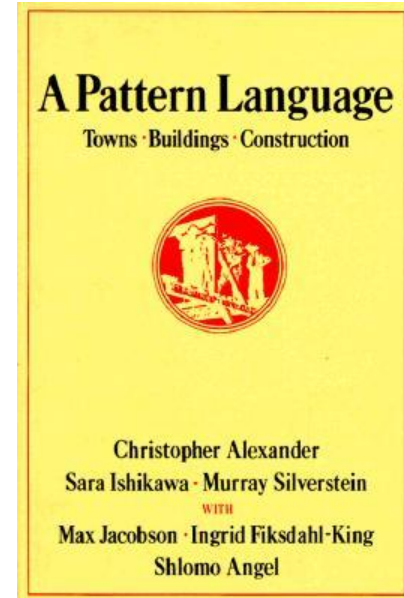
```
const ASC = function(i: number, j: number): boolean {  
    return i < j;  
}  
  
const DESC = function(i: number, j: number): boolean {  
    return i > j;  
}  
  
function sort(  
    list: number[],  
    order: (number, number) => boolean) {  
    // ...  
    boolean mustSwap = order(list[j], list[i]);  
    // ...  
}
```

# Design Patterns

# *Design patterns*

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

– Christopher Alexander,  
Architect (1977)



# How not to discuss design (from Shalloway and Trott)

- Carpentry:
  - How do you think we should build these drawers?
  - Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating...

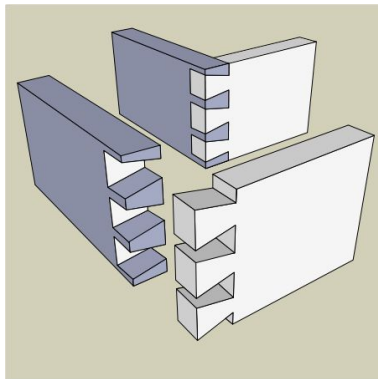


# How not to discuss design (from Shalloway and Trott)

- Software Engineering:
  - How do you think we should write this method?
  - I think we should write this if statement to handle ... followed by a while loop ... with a break statement so that...

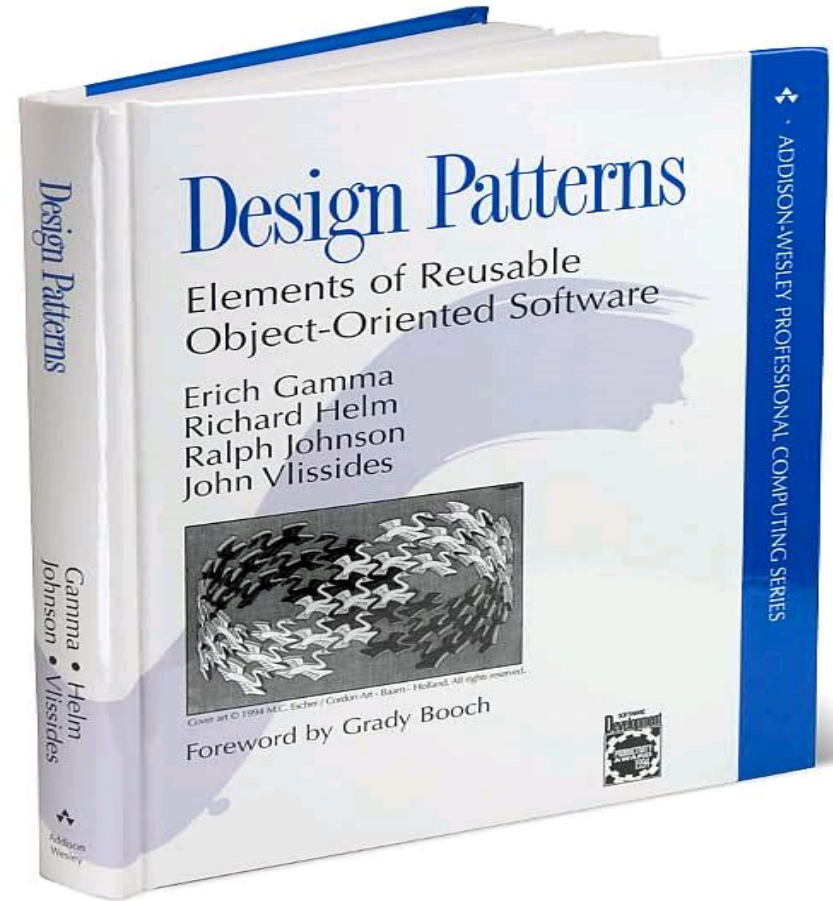
# Discussion with design patterns

- Carpentry:
  - "Is a dovetail joint or a miter joint better here?"
- Software Engineering:
  - "Is a strategy pattern or a template method better here?"



# History:

## *Design Patterns* (1994)



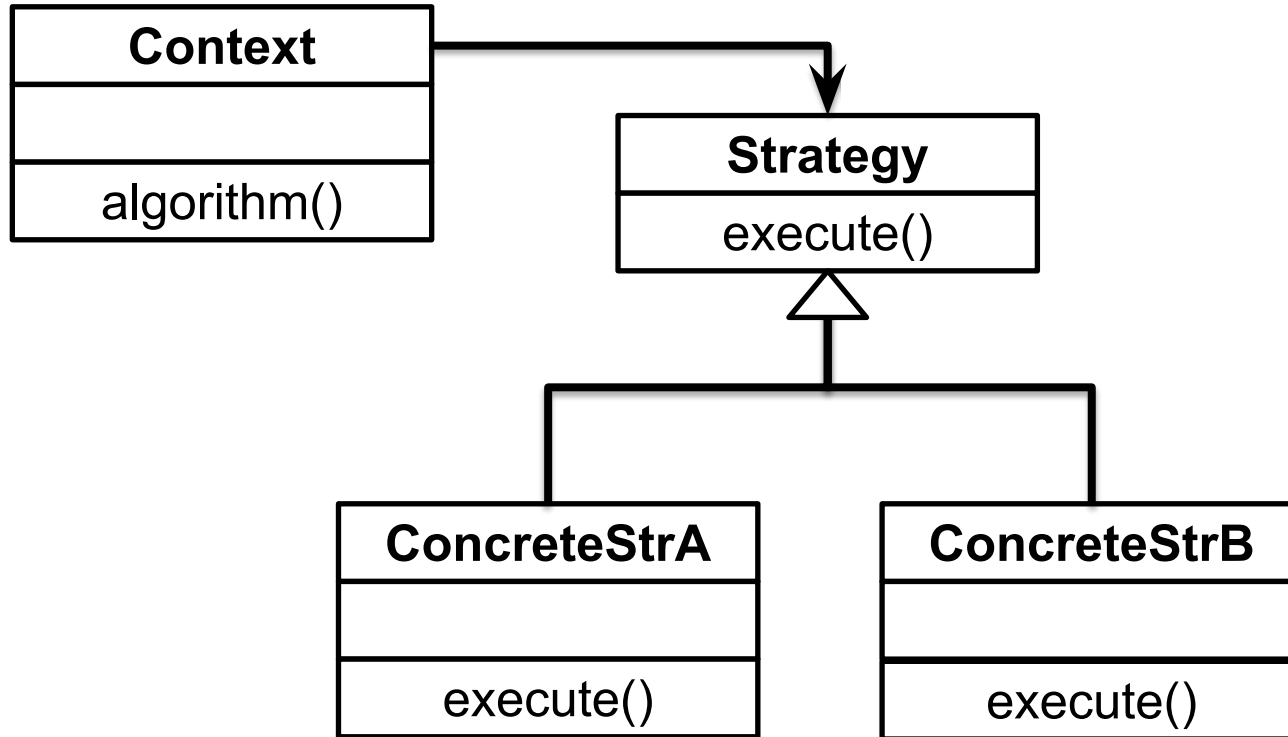
# Elements of a design pattern

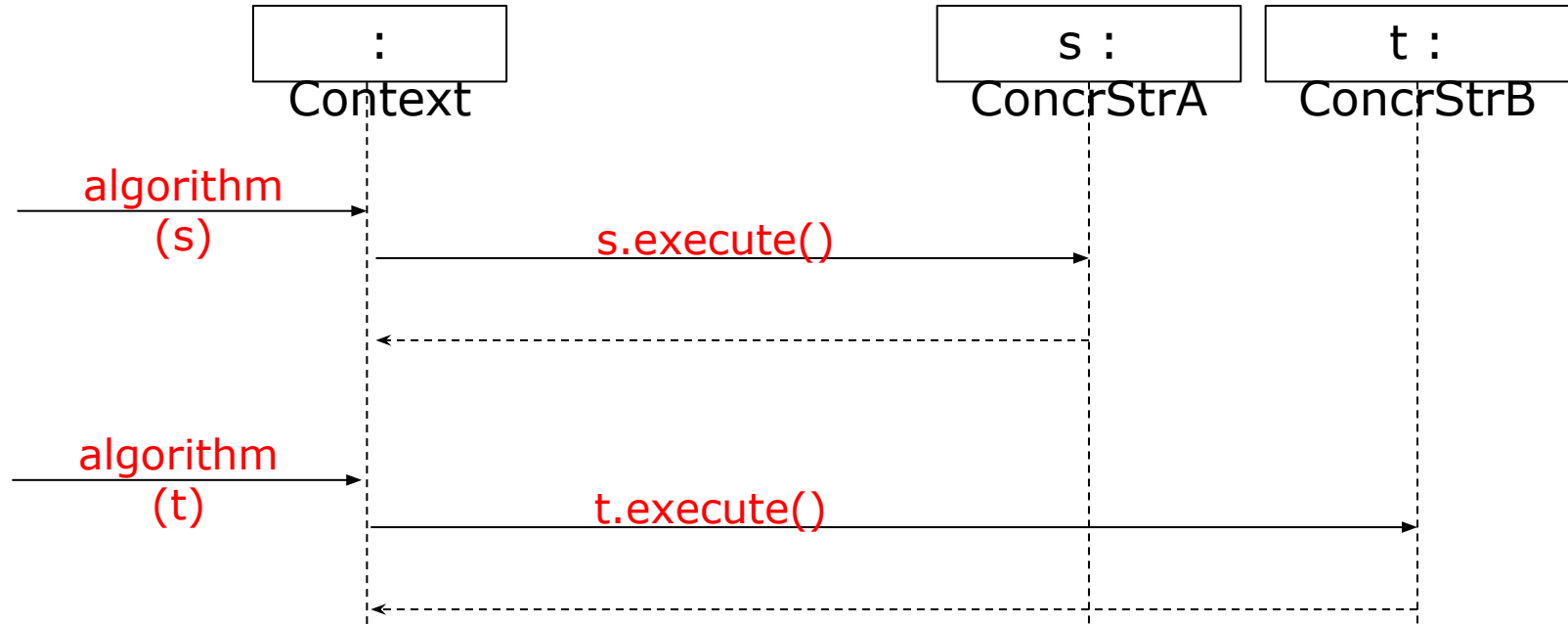
- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences

# Strategy Pattern

# Strategy pattern

- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes: (1) Code can be harder to understand, (2) Lots of overhead if the strategies are simple





*Strategy can be provided in method call or in any other way to context*



# One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

# Another design scenario

- A vision processing system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

# Design Patterns and Programming Languages

Design patterns address general design challenges

Some patterns address problems with built-in solutions

Example: Strategy pattern vs higher-order functions

```
const ASC = function(i: number, j: number): boolean {  
    return i < j;  
}  
const DESC = function(i: number, j: number): boolean {  
    return i > j;  
}
```

# Strategy Pattern vs Higher-Order Function

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
  
class AscendingOrder implements Order {  
    public boolean lessThan(int i, int j) {  
        return i < j; }  
}  
  
class DescendingOrder implements Order {  
    public boolean lessThan(int i, int j) {  
        return i > j; }  
}  
  
void sort(int[] list, Order order) ;
```

```
const ASC =  
    function(i: number, j: number): boolean {  
        return i < j;  
    }  
  
const DESC =  
    function(i: number, j: number): boolean {  
        return i > j;  
    }  
  
function sort(  
    list: number[],  
    order: (number, number) => boolean) ...;
```

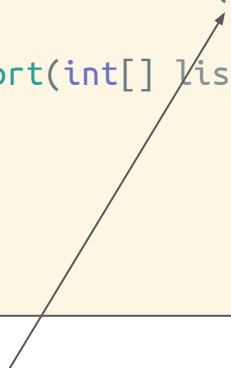
# Strategy Pattern vs Higher-Order Function

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
  
class AscendingOrder implements Order {  
    public boolean lessThan(int i, int j) {  
        return i < j; }  
}  
  
class DescendingOrder implements Order {  
    public boolean lessThan(int i, int j) {  
        return i > j; }  
}  
  
void sort(int[] list, Order order) ;
```

```
const ASC = function(i, j) { return i < j; }  
const DESC = function(i, j) { return i > j; }  
  
function sort(list, order) ...;
```

# New Java Syntax for “Functions”

```
interface Order {  
    boolean lessThan(int i, int j);  
}  
  
final Order ASCENDING = (i, j) -> i < j;  
final Order DESCENDING = (i, j) -> i > j;  
  
static void sort(int[] list, Order order);
```



```
const ASC =  
    function(i: number, j: number): boolean {  
        return i < j;  
    }  
const DESC =  
    function(i: number, j: number): boolean {  
        return i > j;  
    }  
  
function sort(  
    list: number[],  
    order: (number, number) => boolean) ...;
```

Convenient syntax (introduced for lambdas) to create objects of interface with single method.

# Module Pattern

# Module pattern: Hide internals in closure

```
(function () {  
    // ... all vars and functions are in this scope only  
    // still maintains access to all globals  
})();
```

Function provides local scope, internals not accessible

Function directly invoked to execute it once

Wrapped in parentheses to make it expression

Discovered around 2007, became very popular, part of Node



# Using closures to hide methods and fields

```
function createPolarPoint(len, angle) {  
  let xcache = -1;  
  let internalLen=len;  
  function computeX() {...}  
  return {  
    getX: function() {  
      computeX(); return xcache; },  
    getY: function() {  
      return len * sin(angle); }  
  };  
}
```

# Module pattern: Decide what to export

```
var MODULE = (function () {  
    var my = {},  
        privateVariable = 1;  
  
    function privateMethod() {  
        // ...  
    }  
  
    my.moduleProperty = 1;  
    my.moduleMethod = function () {  
        // ...  
    };  
  
    return my;  
})();
```

# Java: Module Pattern?

Public/private built in, problem does not exist

Fully qualified names (“edu.cmu.cs17214.FlashCard”) as convention/pattern to solve naming clashes

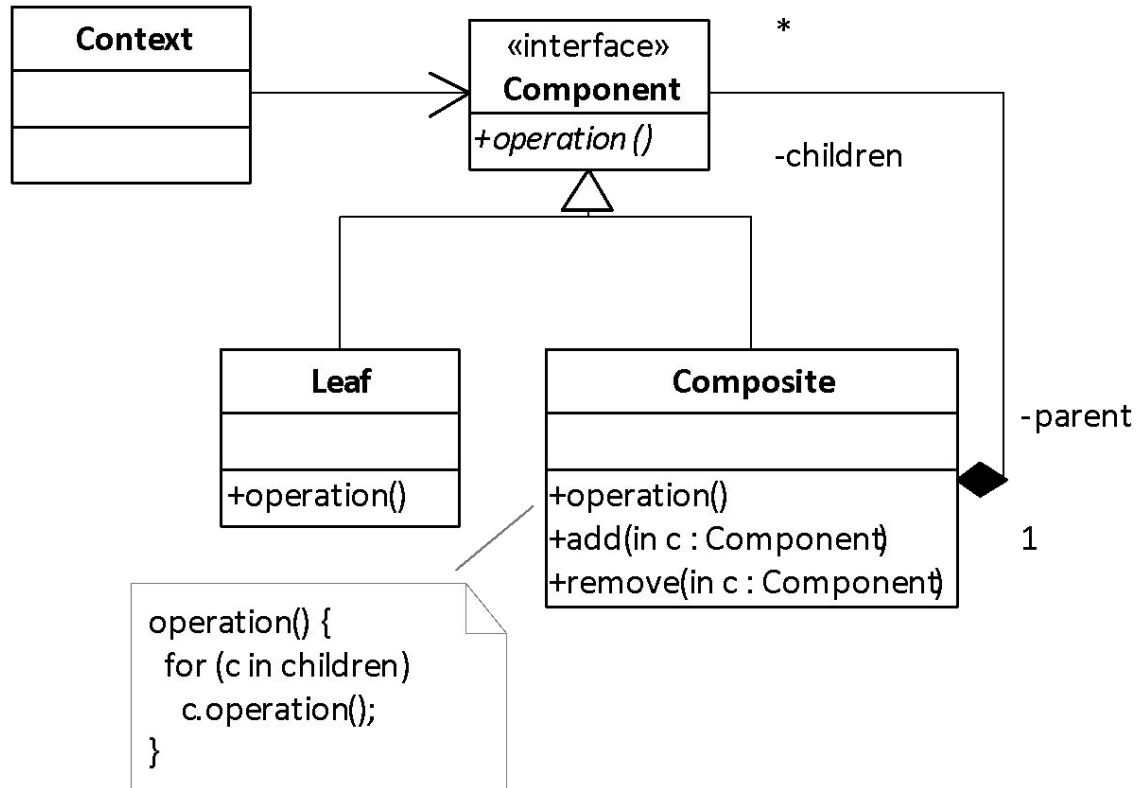
Never JavaScript/TypeScript features make it less important (ES6 modules, classes, public/private)

# Composite Pattern

# Design Exercise (on paper)

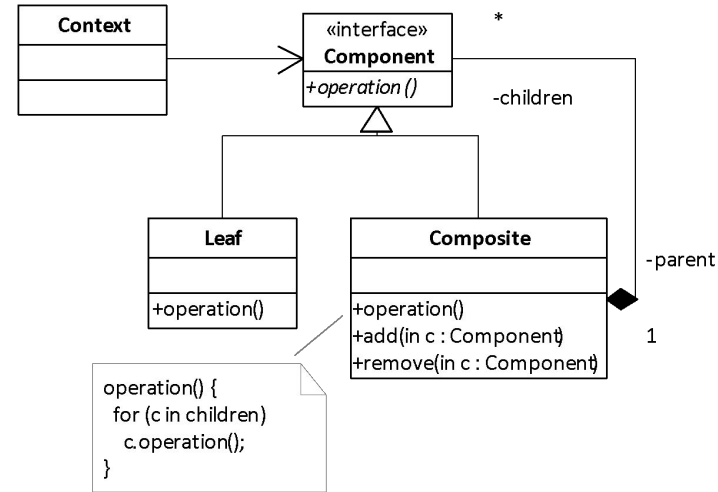
- You are designing software for a shipping company.
- There are several different kinds of items that can be shipped: letters, books, packages, fragile items, etc.
- Two important considerations are the **weight** of an item and its **insurance cost**.
  - Fragile items cost more to insure.
  - All letters are assumed to weigh an ounce
  - We must keep track of the weight of other packages.
- The company sells **boxes** and customers can put several items into them.
  - The software needs to track the contents of a box (e.g. to add up its weight, or compute the total insurance value).
  - However, most of the software should treat a box holding several items just like a single item.
- Think about how to represent packages; what are possible interfaces, classes, and methods? (letter, book, box only)

# The Composite Design Pattern



# The Composite Design Pattern

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



# We have seen this before

```
interface Point {  
    int getX();  
    int getY();  
}  
  
class MiddlePoint implements Point {  
    Point a, b;  
    MiddlePoint(Point a, Point b) {this.a = a; this.b = b; }  
    int getX() { return (this.a.getX() + this.b.getX()) / 2;}  
    int getY() { return (this.a.getY() + this.b.getY()) / 2; }  
}
```



# Iterator and Flash Cards?

# We have seen this before

```
function newCombinedCardOrganizer (cardOrganizers: CardOrganizer[]): CardOrganizer {  
  return {  
    reorganize: function (cards: CardStatus[]): CardStatus[] {  
      let status = cards.slice()  
      for (const cardOrganizer of cardOrganizers) {  
        status = cardOrganizer.reorganize(status)  
      }  
      return status  
    }  
  }  
}
```

# Fluent APIs / Cascade Pattern

# Setting up Complex Objects

Long constructors, lots of optional parameters, long lists of statements

```
Option find = OptionBuilder
    .withArgName("file")
    .hasArg()
    .withDescription("search..." )
    .create("find");
```

```
client.getItem('user-table')
    .setHashKey('userId', 'userA')
    .setRangeKey('column', '@')
    .execute()
    .then(function(data) {
        ...
    })
```

# Liquid APIs

Each method changes  
state,  
then returns **this**

(Immutable version:  
Return modified copy)

```
class OptBuilder {  
    private String argName = "";  
    private boolean hasArg = false;  
    ...  
    OptBuilder withArgName(String n) {  
        this.argName = n;  
        return this;  
    }  
    OptBuilder hasArg() {  
        this.hasArg = true;  
        return this;  
    }  
    ...  
    Option create() {  
        return new Option(argName,  
                           hasArgs, ...)  
    }  
}
```

# Python: Named parameters

```
parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')
```

# JavaScript: JSON Objects

```
var argv = require('yargs/yargs')(process.argv.slice(2))
  .option('size', {
    alias: 's',
    describe: 'choose a size',
    choices: ['xs', 's', 'm', 'l', 'xl']
  })
  .argv
```

Notice the combination of cascading and complex JSON parameters

# Liquid APIs: Discussion and Tradeoffs

Problem: Complex initialization and configuration

Advantages:

- Fairly readable code
- Can check individual arguments
- Avoid untyped complex arguments

Disadvantages:

- Runtime error checking of constraints and mandatory arguments
- Extra complexity in implementation
- Not always obvious how to terminate
- Possibly harder to debug



# Iterator Pattern & Streams

# Traversing a collection

- Since Java 1.0:

```
Vector arguments = ...;  
for (int i = 0; i < arguments.size(); ++i) {  
    System.out.println(arguments.get(i));  
}
```

- Java 1.5: enhanced for loop

```
List<String> arguments = ...;  
for (String s : arguments) {  
    System.out.println(s);  
}
```

- Works for every implementation of `Iterable`

```
public interface Iterable<E> {  
    public Iterator<E> iterator();  
}  
  
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

- In JavaScript (ES6)

```
let arguments = ...  
for (const s of arguments) {  
    console.log(s)  
}
```

- Works for every implementation with a “magic” function `[Symbol.iterator]` providing an iterator

```
interface Iterator<T> {  
    next(value?: any): IteratorResult<T>;  
    return?(value?: any): IteratorResult<T>;  
    throw?(e?: any): IteratorResult<T>;  
}  
  
interface IteratorReturnResult<TReturn> {  
    done: true;  
    value: TReturn;  
}
```

# The Iterator Idea

Iterate over elements in arbitrary data structures (lists, sets, trees) without having to know internals

Typical interface:

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
}
```

(in Java also remove)

# Using an iterator

Can be used explicitly

```
List<String> arguments = ...;  
for (Iterator<String> it = arguments.iterator(); it.hasNext(); ) {  
    String s = it.next();  
    System.out.println(s);  
}
```

Often used with magic syntax:

```
for (String s : arguments)  
for (const s of arguments)
```

# Java: Getting an Iterator

```
public interface Collection<E> extends Iterable<E> {  
    boolean    add(E e);  
    boolean    addAll(Collection<? extends E> c);  
    boolean    remove(Object e);  
    boolean    removeAll(Collection<?> c);  
    boolean    retainAll(Collection<?> c);  
    boolean    contains(Object e);  
    boolean    containsAll(Collection<?> c);  
    void       clear();  
    int        size();  
    boolean    isEmpty();  
    Iterator<E> iterator();  
    Object[]   toArray()  
    <T> T[]    toArray(T[] a);  
    ...  
}
```

*Defines an interface for creating an Iterator,  
but allows Collection  
implementation to decide  
which Iterator to create.*

# Iterators for everything

```
public class Pair<E> {  
    private final E first, second;  
    public Pair(E f, E s) { first = f; second = s; }  
  
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");  
for (String s : pair) { ... }
```

# An Iterator implementation for Pairs

```
public class Pair<E> implements Iterable<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
    public Iterator<E> iterator() {
        return new PairIterator();
    }
    private class PairIterator implements Iterator<E> {
        private boolean seenFirst = false, seenSecond = false;
        public boolean hasNext() { return !seenSecond; }
        public E next() {
            if (!seenFirst) { seenFirst = true; return first; }
            if (!seenSecond) { seenSecond = true; return second; }
            throw new NoSuchElementException();
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { ... }
```

# Iterator design pattern

- Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
  - Order is unspecified, but access every element once
- Solution: A strategy pattern for iteration
- Consequences:
  - Hides internal implementation of underlying container
  - Easy to change container type
  - Facilitates communication between parts of the program



# Iterator and FlashCards?

# Design pattern conclusions

- Provide shared language
- Convey shared experience
- Can be system and language specific

