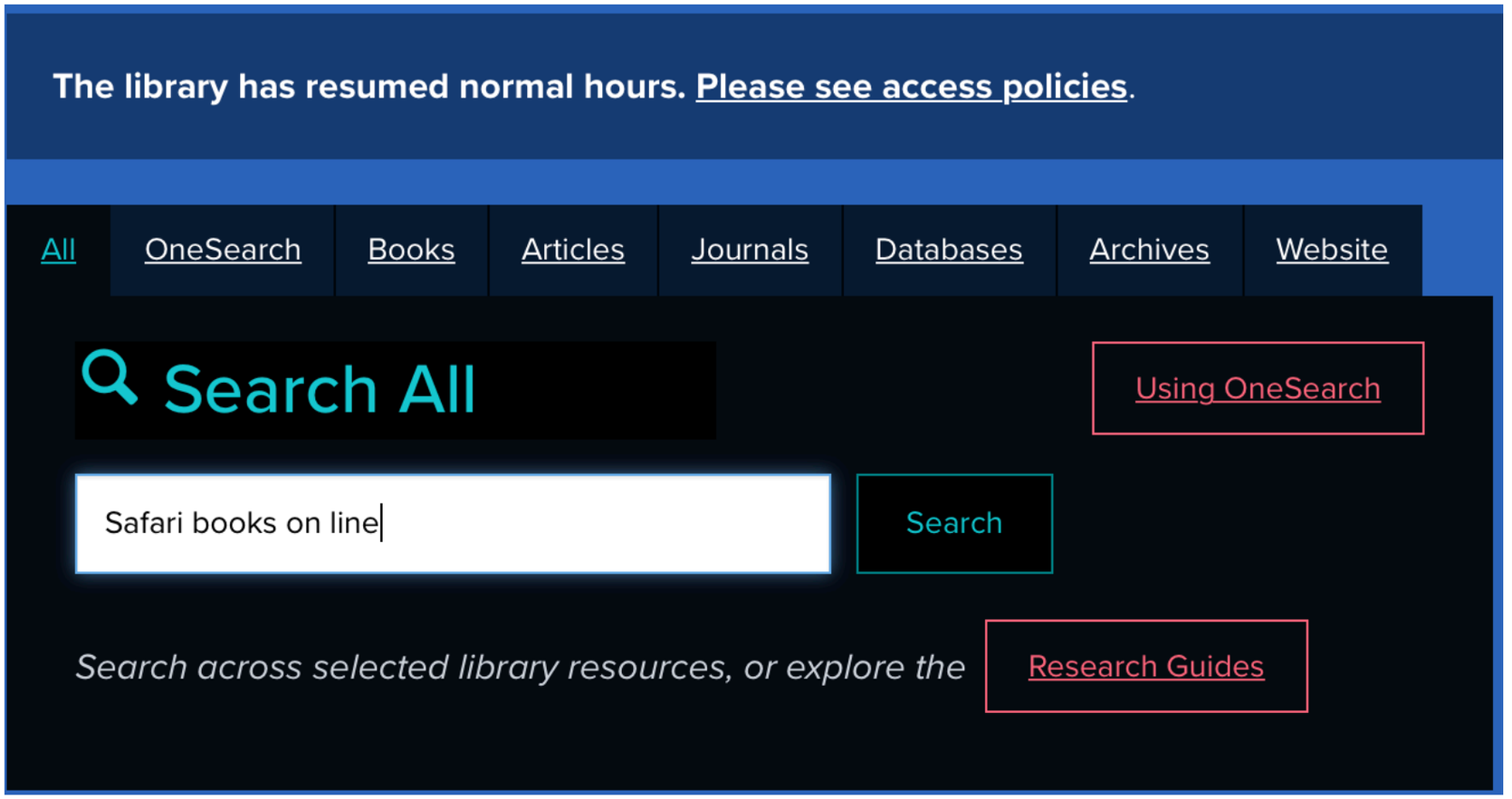
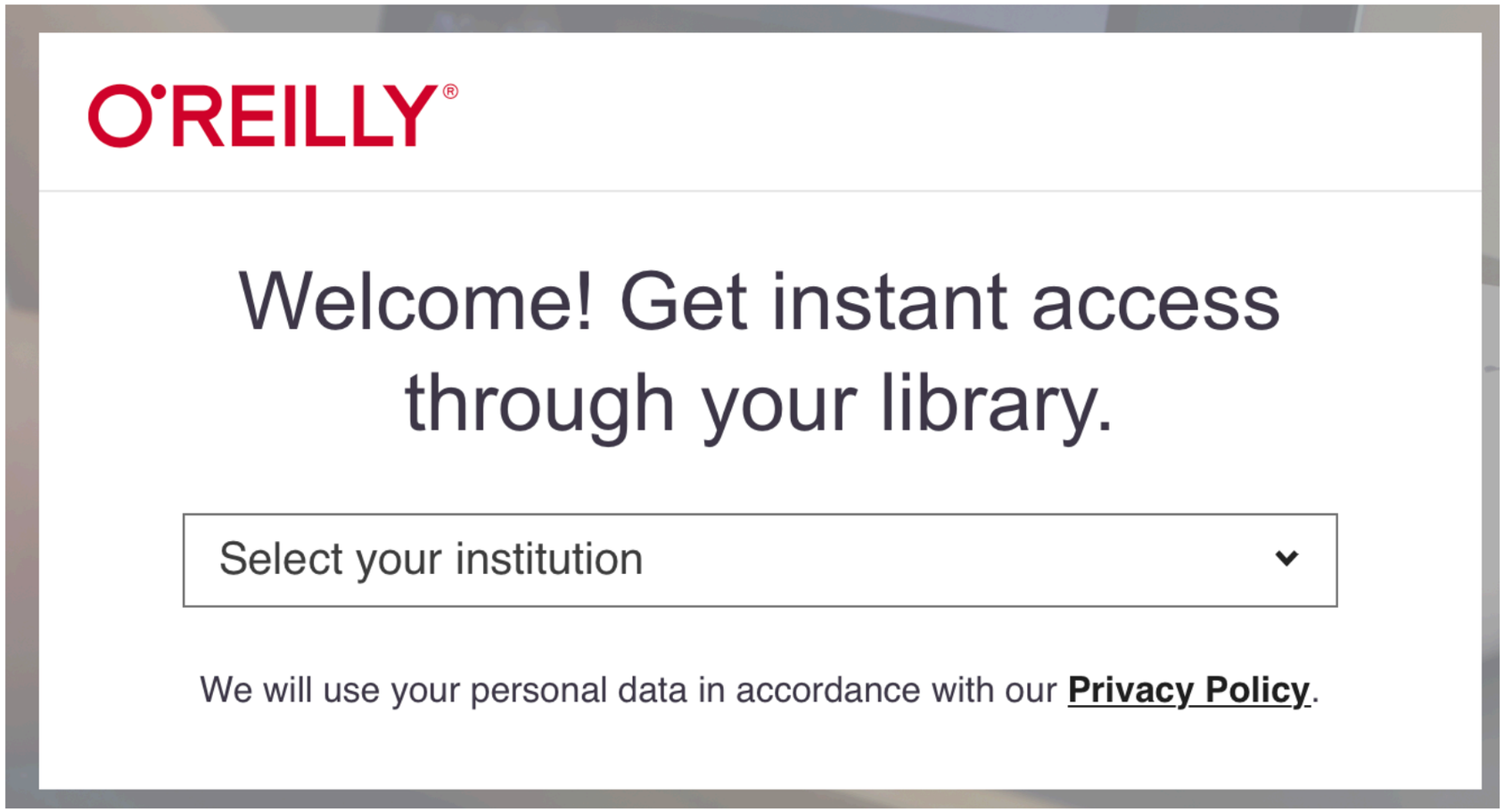
CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2021
Doc 09 Effective Java, Template, Decorator
Sep 28, 2021

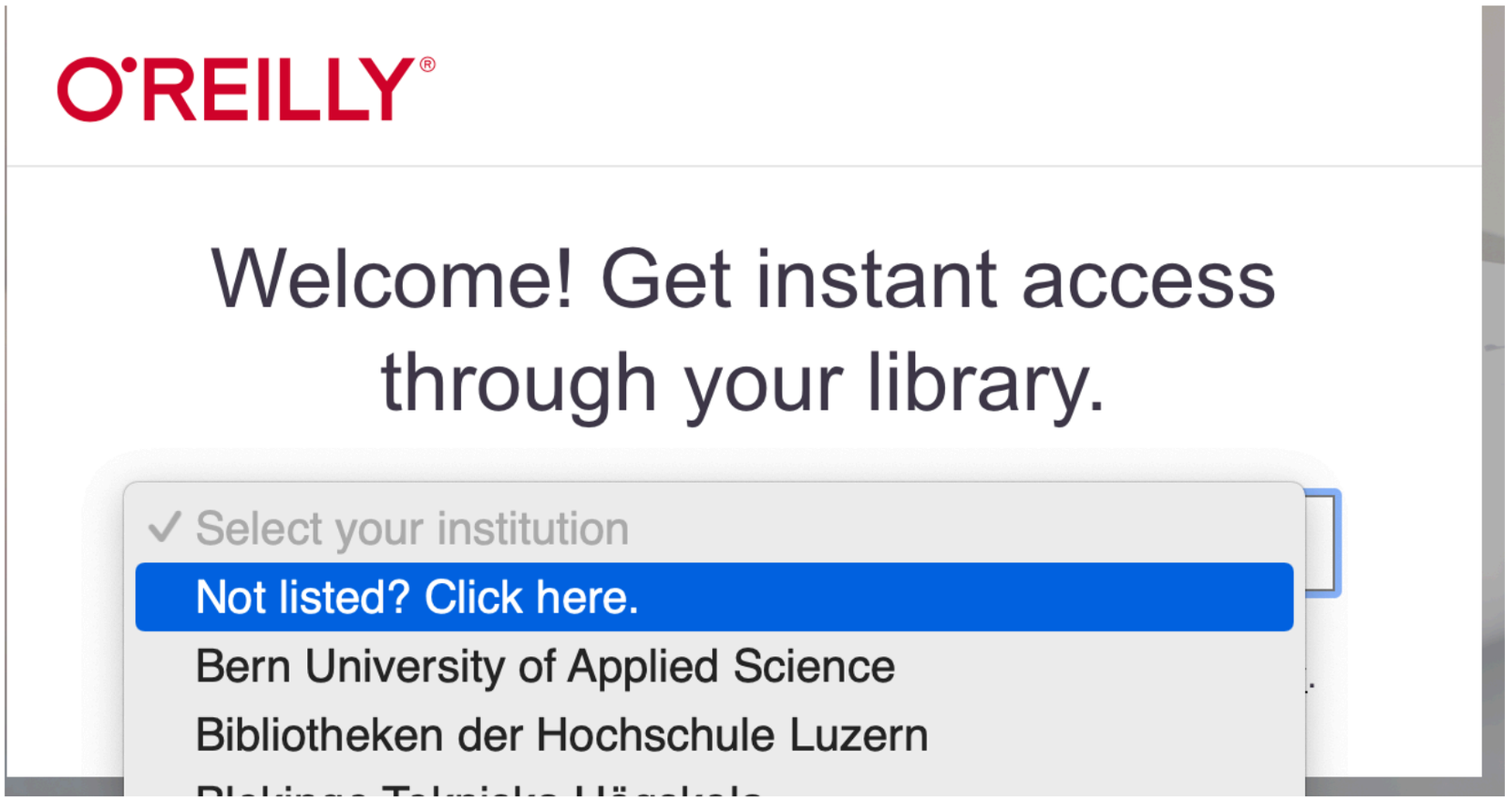

The library has resumed normal hours. Please see access policies.
All
OneSearch
Books
Articles
Journals
Databases
Archives
Website
Search All
Using OneSearch
Safari books on line
Search
Search across selected library resources, or explore the
Research Guides

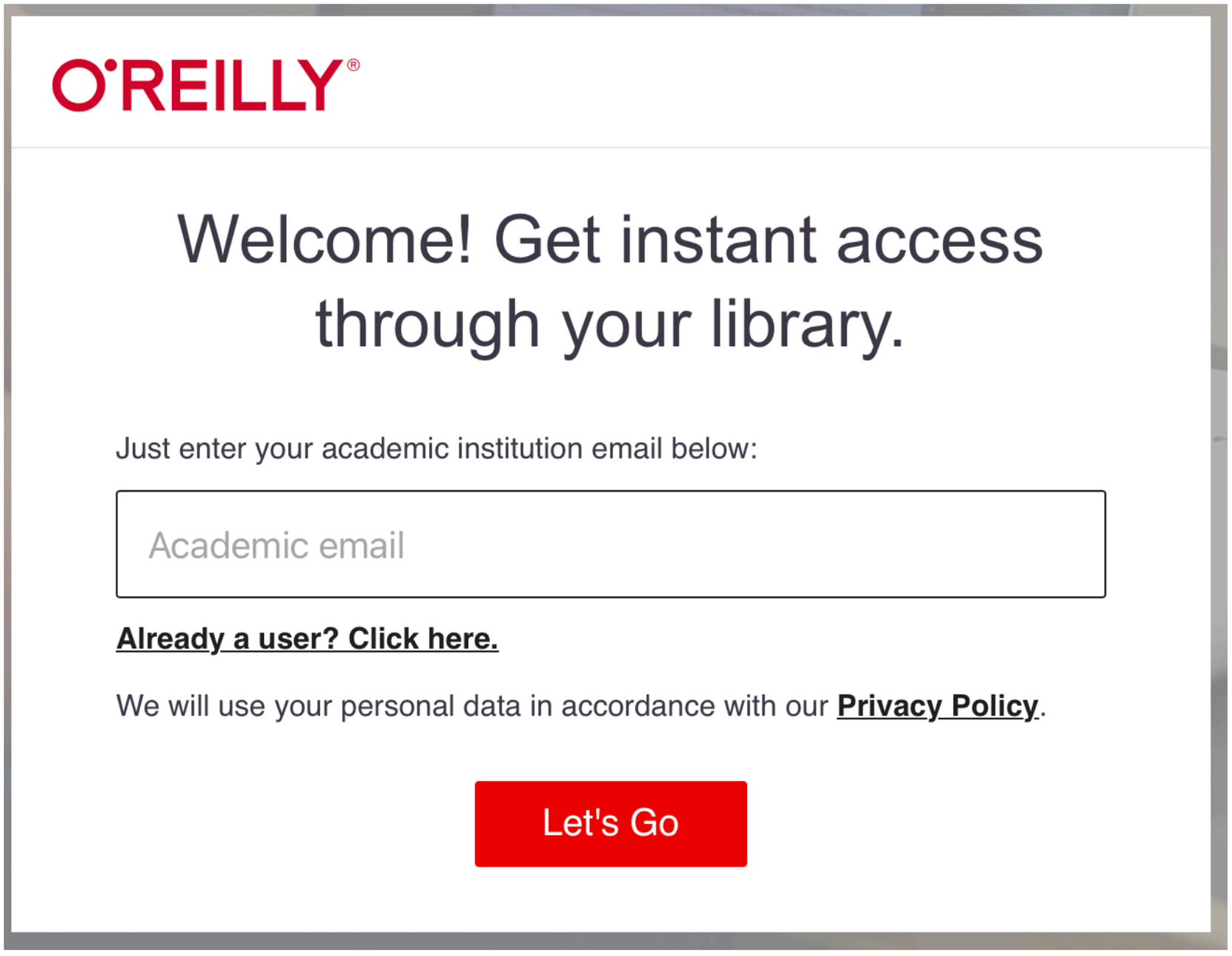
Databases

- Early English Books Online (EEBO)
- Safari eBooks Online
- Value Line

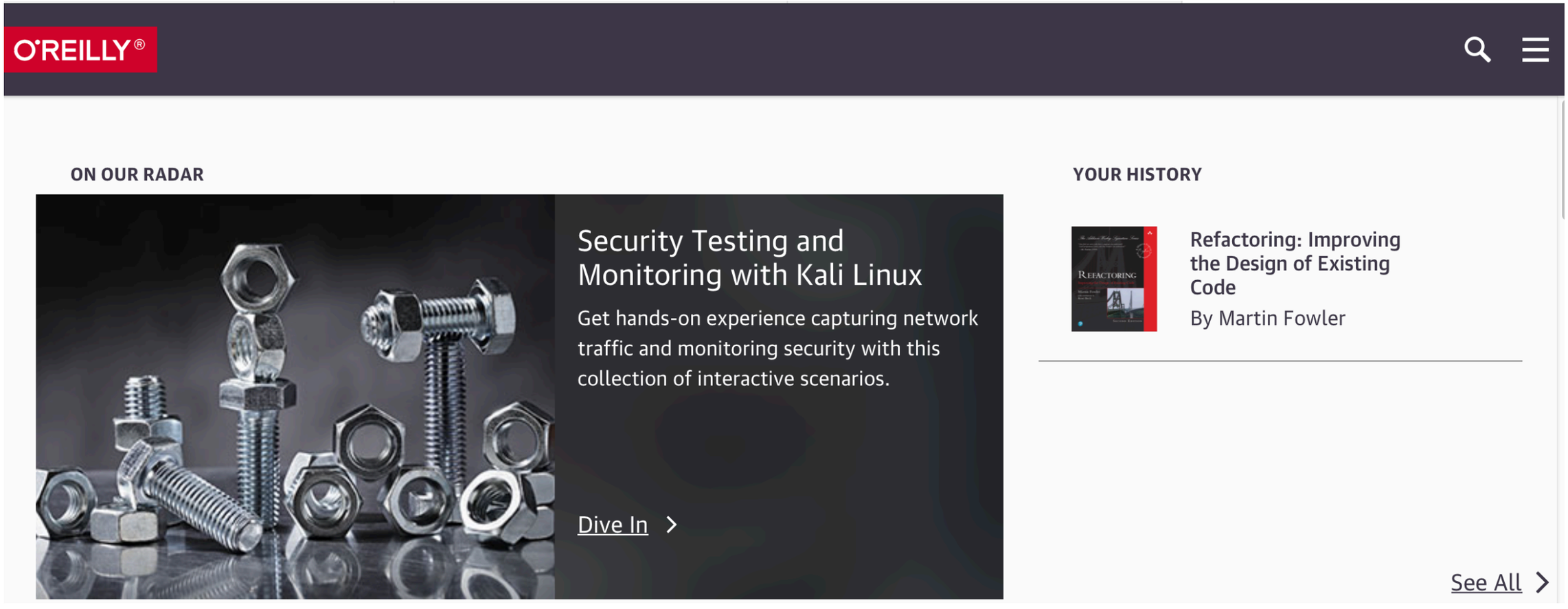
Browse Database A-Z List


O'REILLY®
Welcome! Get instant access through your library.
Select your institution
We will use your personal data in accordance with our Privacy Policy.


O'REILLY®
Welcome! Get instant access through your library.
Select your institution
Not listed? Click here.
Bern University of Applied Science
Bibliotheken der Hochschule Luzern


O'REILLY®
Welcome! Get instant access through your library.
Just enter your academic institution email below:
Academic email
Already a user? Click here.
We will use your personal data in accordance with our Privacy Policy.
Let's Go

https://library.sdsu.edu



Effective Java

Effective Java

Book by Joshua Bloch

First Edition	2001
Second Edition	2008
Third Edition	2017

Item 1. Consider Static Factory methods

Consider using static Factory methods instead of constructors

Java String class

```
public static String valueOf(boolean b) {
        return b ? "true" : "false";
    }


public static String valueOf(char c) {
    char data[] = {c};
    return new String(data, true);
}
```

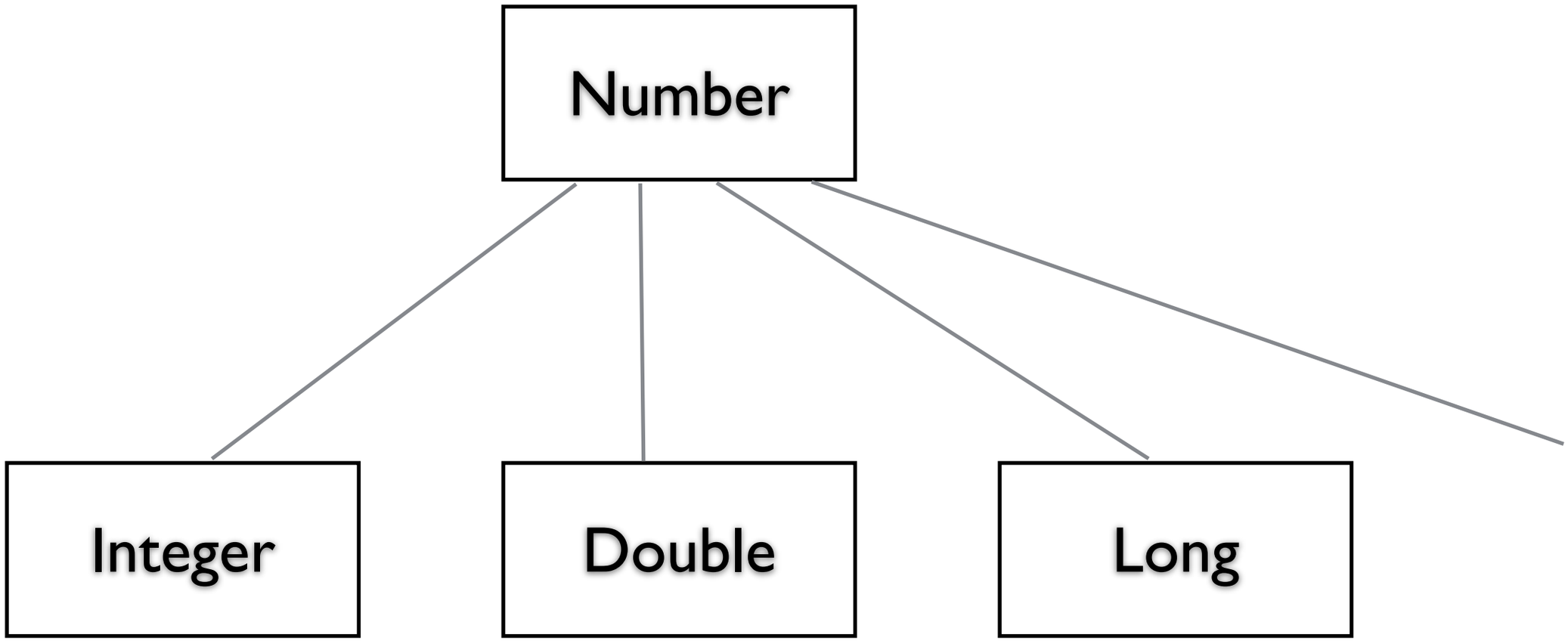
Advantages of Static Factory methods

They have names

Don’t need to create a new object each time

They can return an Object of any type

Java Boxing of Primitives




```
Integer x = new Integer( 5);
Boolean y = new Boolean( true);
```

Objective C Boxing of Primitives

Uses static factory methods in Number

```
Number x = Number.value(5);
Number y = Number.value(true);
```

Programmers only need to know Number class

Class Cluster

Smalltalk

No constructors

Just static factory methods

Item 15 Minimize accessibility

Rule of thumb

Make each class or member as inaccessible as possible

Item 17 Favor Immutability - Minimize Mutability

Immutable objects are simple

Immutable objects are thread-safe

Immutable objects can be shared freely

Immutable objects are good building blocks for other objects

Item 17 Favor Immutability - Minimize Mutability

Don’t provide any methods that modify the object (setters)

Ensure that no methods may be overridden

Make all fields final

Make all fields private

Ensure exclusive use to any mutable components

Make defensive copies of data provided/given to client

Item 50 Make Defensive Copies when Needed

```
public final class Period {
   private final Date start;
   private final Date end;

   public Period(Date start, Date End) {
      if (start.compareTo(end) > 0 )
         throw new IllegalArgumentException(start + " is after " + end);
      this.start = start;
      this.end = end;
   }

   public Date start() {
      return start;
   }
```

Item 50 Make Defensive Copies when Needed

```
public final class Period {
   private final Date start;
   private final Date end;

   public Period(Date start, Date End) {
      this.start = new Date(start.getTime());
      this.end = new Date(end.getTime());
      if (this.start.compareTo(this.end) > 0 )
         throw new IllegalArgumentException(start + " is after " + end);
   }

   public Date start() {
      return start.clone();
   }
```

Item 18 Favor Composition over Inheritance

Inheritance breaks encapsulation

Safe to use inheritance when

Superclass and subclass in same package

When superclass is designed for inheritance

Item 20 Prefer Interfaces to Abstract Classes

Existing classes can be modified to implement a new interface

Interfaces are ideal for defining mixins

Interfaces allow construction of nonhierarchical frameworks

Provide skeletal implementation class to go with nontrivial interface

Item 59 Know and use the Libraries

Item 62 Avoid strings if other types are better

String compoundKey = name + “#” + i.next();

What happens if “#” is in name?

Create CompoundKey class

Item 64 Refer to objects by their Interfaces

Your code will be more flexible

List subscribers = new ArrayList();

ArrayList subscribers = new ArrayList();

If no interface exists then ok to refer to object via class

Item 54 Return empty collections not nulls

```
private final List<Cheese> cheesesInStock = ...;

/**
 * @return a list containing all of the cheeses in the shop,
 *     or null if no cheeses are available for purchase.
 */

public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? null : new ArrayList<>(cheesesInStock);
}
```

```
List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

Item 54 Return empty collections not nulls

```
private final List<Cheese> cheesesInStock = ...;

public List<Cheese> getCheeses() {
    return cheesesInStock.isEmpty() ? null : new ArrayList<>(cheesesInStock);
}

List<Cheese> cheeses = shop.getCheeses();
if (cheeses != null && cheeses.contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

```
public List<Cheese> getCheeses() {
    return new ArrayList<>(cheesesInStock);
}
```

Item 68 Use accepted naming conventions

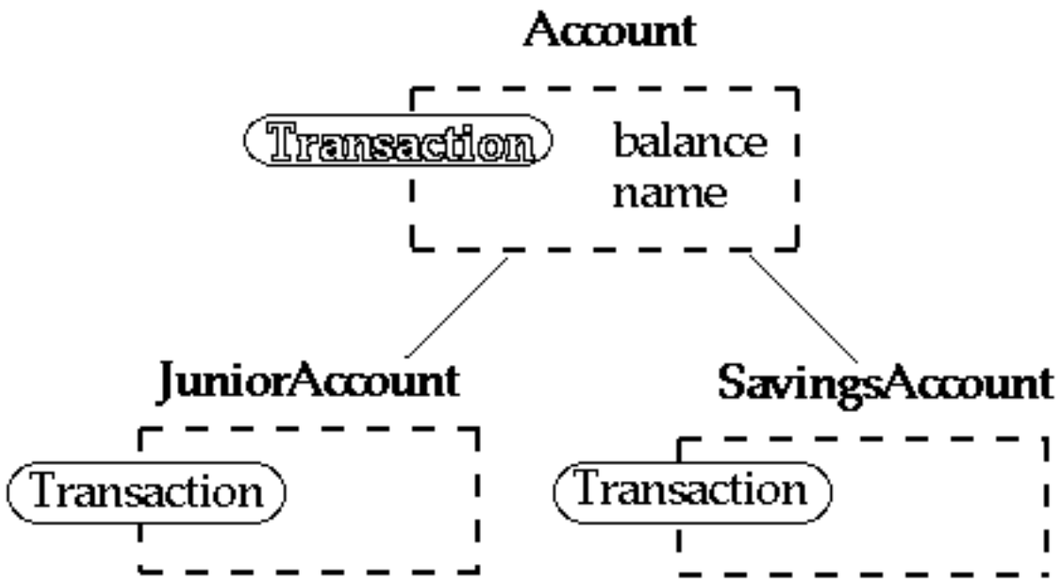
Template Method

Polymorphism

```
class Account {
    public:
        void virtual Transaction(float amount)
                { balance += amount;}
        Account(char* customerName, float InitialDeposit = 0);
    protected:
        char* name;
        float balance;
}

class JuniorAccount : public Account {
    public:   void Transaction(float amount) {//code here}
}

class SavingsAccount : public Account {
    public:   void Transaction(float amount) {//code here}
}

Account* createNewAccount(){
    // code to query customer and determine what type of
    // account to create
};
```

```
main() {
    Account*  customer;
    customer = createNewAccount();
    customer->Transaction(amount);
}
```

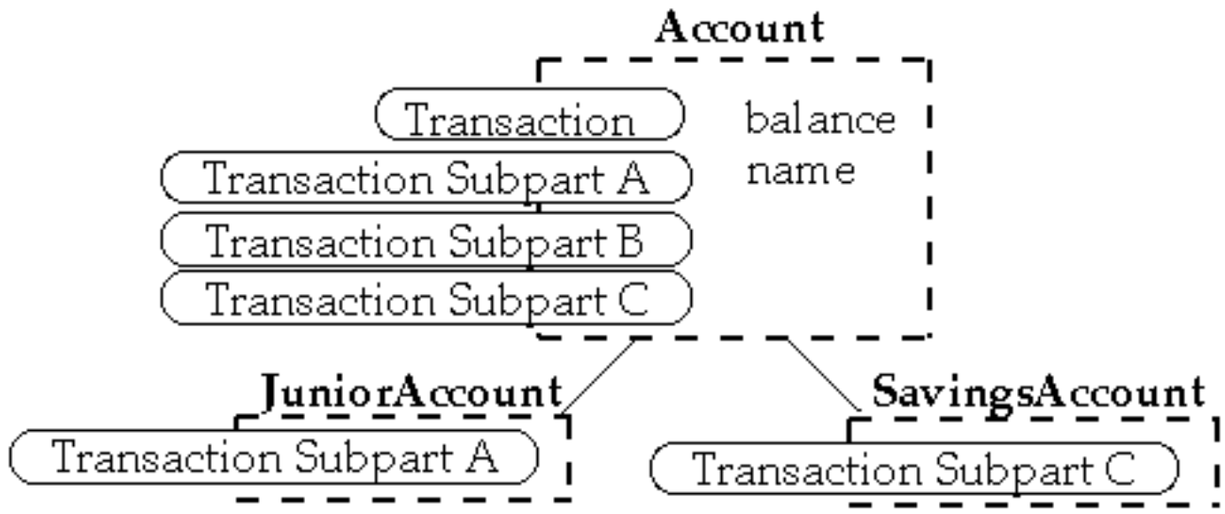
Deferred Methods




```
class Account {
   public:
      void virtual Transaction() = 0;
}

class JuniorAccount : public Account {
   public
      void Transaction() { put code here}
}
```

Template Method

```
class Account {
    public:
        void Transaction(float amount);
    protected:
        void virtual TransactionSubpartA();
        void virtual TransactionSubpartB();
        void virtual TransactionSubpartC();
}

void Account::Transaction(float amount)  {
    TransactionSubpartA();        TransactionSubpartB();
    TransactionSubpartC();        // EvenMoreCode;
}

class JuniorAccount : public Account {
    protected:            void virtual TransactionSubpartA(); }

class SavingsAccount : public Account {
    protected:            void virtual TransactionSubpartC(); }

Account*  customer;
customer = createNewAccount();
customer->Transaction(amount);
```



Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure

Java Example

```
import  java.awt.*;
class  HelloApplication  extends  Frame
   {
   public  void  paint(  Graphics  display  )
      {
      int startX  =  30;
      int startY  =  40;
      display.drawString(  "Hello World", startX, startY  );
      }
}
```

Ruby LinkedList Example

```
class LinkedList
  include Enumerable

  def [](index)
    Code not shown
  end

  def size
    Code not shown
  end

  def each
     Code not shown
  end

  def push(object)
   Code note shown
  end

end
```

```
def testSelect
  list = LinkedList.new
  list.push(3)
  list.push(2)
  list.push(1)

  a = list.select {|x| x.even?}
  assert(a == [2])
end
```

Where does list.select come from?

Methods defined in Enumerable

all?	any?	collect	detect
each_cons	each_slice	each_with_index	entries
enum_cons	enum_slice	enum_with_index	find
find_all	grep	include?	inject
map	max	member?	min
partition	reject	select	sort
sort_by	to_a	to_set	zip

All use "each"

Implement "each" and the above will work

java.util.AbstractCollection

Subclass AbstractCollection

Implement
- iterator
- size
- add

Get
- addAll
- clear
- contains
- containsAll
- isEmpty
- remove
- removeAll
- retainAll
- size
- toArray
- toString

Consequences

This is the most commonly used of the 23 GoF patterns

Important in class libraries

Inverted control structure

- Parent class calls subclass methods

Consequences

Inverted control structure

Java's paint method is a primitive operation called by a parent method

Beginning Java programs don't understand how the following works:

```
import  java.awt.*;
class  HelloApplication  extends  Frame
  {
  public  void  paint(  Graphics  display  )
     {
     int startX  =  30;
     int startY  =  40;
     display.drawString(  "Hello World", startX, startY  );
     }
  }
```

Consequences

Template methods tend to call:

- Concrete operations
- Primitive (abstract) operations
- Factory methods
- Hook operations

Provide default behavior that subclasses can extend

It is important to denote which methods

- Must overridden
- Can be overridden
- Can not be overridden

Refactoring to Template Method

Simple implementation

- Implement all of the code in one method
- The large method you get will become the template method

Break into steps

- Use comments to break the method into logical steps
- One comment per step

Make step methods

- Implement separate methods for each of the steps

Call the step methods

- Rewrite the template method to call the step methods

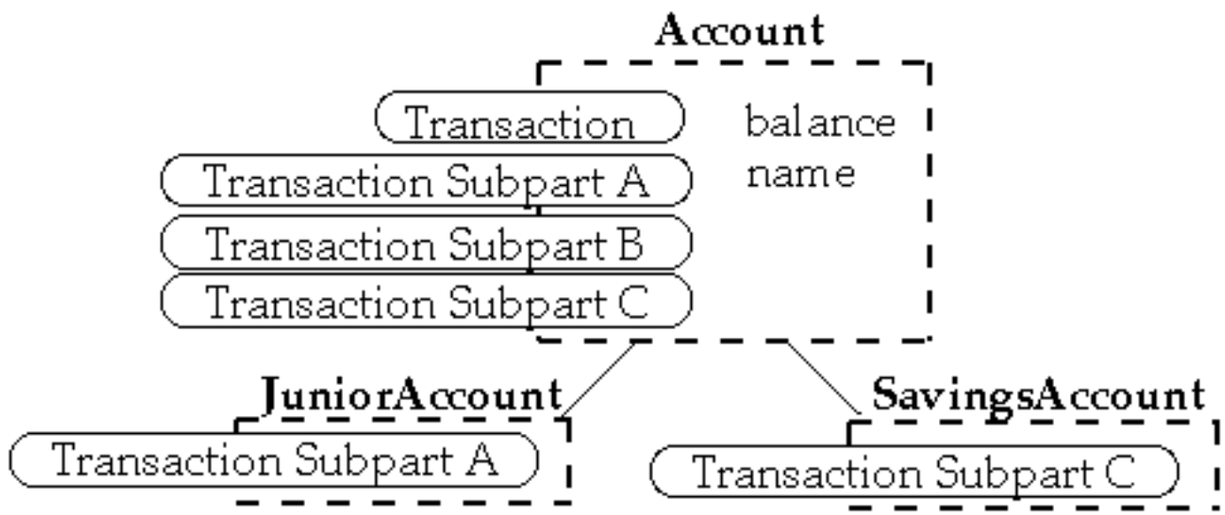
Repeat above steps

- Repeat the above steps on each of the step methods
- Continue until:
  - All steps in each method are at the same level of generality
  - All constants are factored into their own methods

Design Patterns Smalltalk Companion pp. 363-364.

Template Method & Functional Programming

```
class Account {
    public:
        void Transaction(float amount);
    protected:
        void virtual TransactionSubpartA();
        void virtual TransactionSubpartB();
        void virtual TransactionSubpartC();
}

void Account::Transaction(float amount)  {
    TransactionSubpartA();        TransactionSubpartB();
    TransactionSubpartC();        // EvenMoreCode;
}

class JuniorAccount : public Account {
    protected:            void virtual TransactionSubpartA(); }

class SavingsAccount : public Account {
    protected:            void virtual TransactionSubpartC(); }

Account*  customer;
customer = createNewAccount();
customer->Transaction(amount);
```



Template Method & Functional Programming

Pass in functions

```
def transaction(defaultPartA, defaultPartB, defaultPartC, amount, account) {
   defaultPartA();
   defaultPartB();
   defaultPartC();
   code code;
}
```

But this adds a lot of arguments

Requires knowing internal workings of transaction

Currying & Partial Evaluation

Pass in functions

```
def defaultTransaction = transaction(defaultPartA, defaultPartB, defaultPartC);
def juniorTransaction = transaction(juniorPartA, defaultPartB, defaultPartC);

defaultTransaction(amount, account);
juniorTransaction(amount, account);
```

But this requires knowing the account type

Multi-methods

```
defmulti transaction(amount, account) (getAccountType)

defmethod transaction(amount, account) (:default) {
   return defaultTransaction(amount, account);
}

defmethod transaction(amount, account) (:junior) {
   return juniorTransaction(amount, account);
}

transaction(amount, account);
```

Now have dynamic dispatch on the type like Java

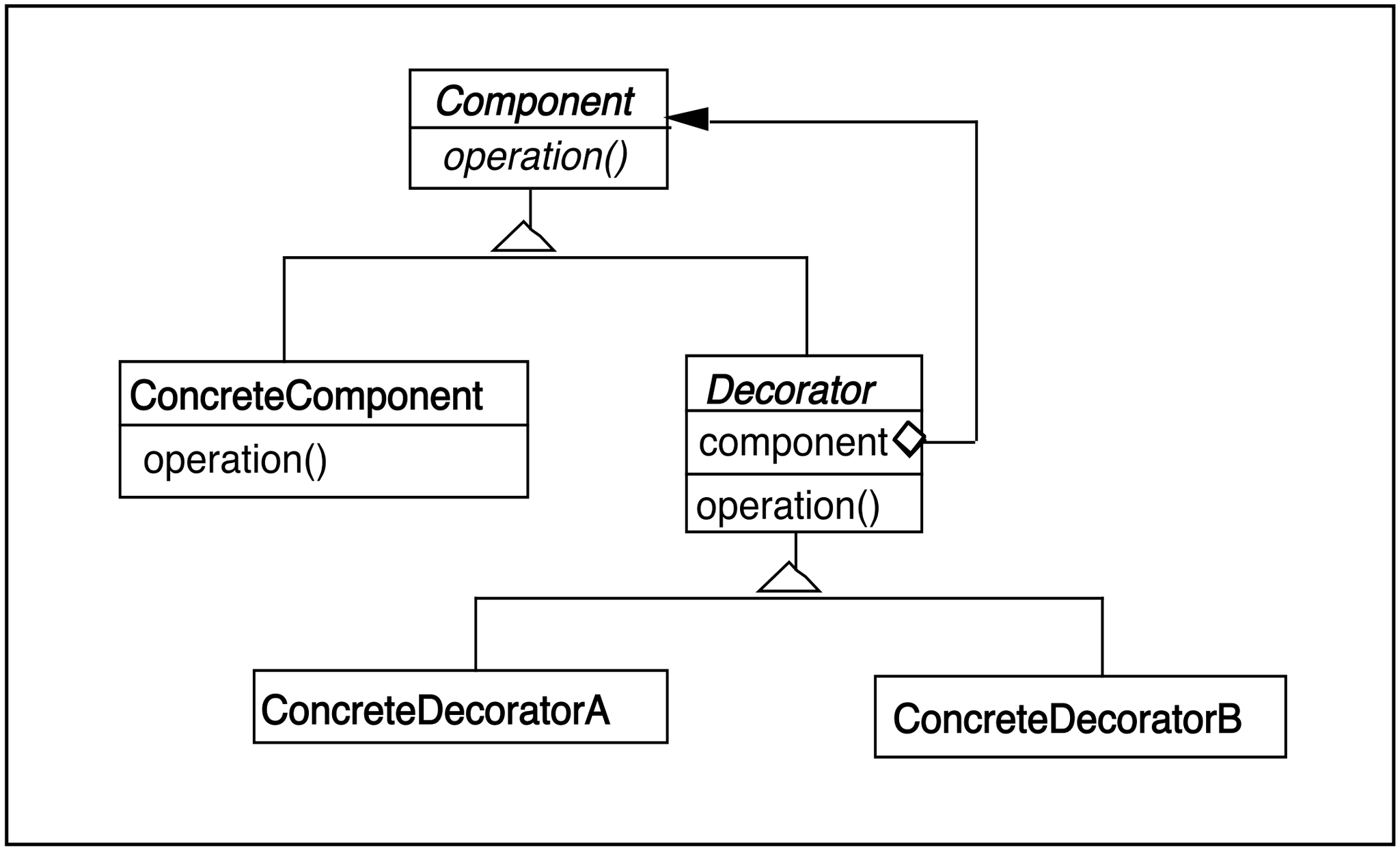
Template Method vs Functional Solution

	Template Method	Functional
Method Variation	In multiple classes/files	In one place
Add new Variation	Add class/file + method	Add function
Type Logic	In class & parent class	

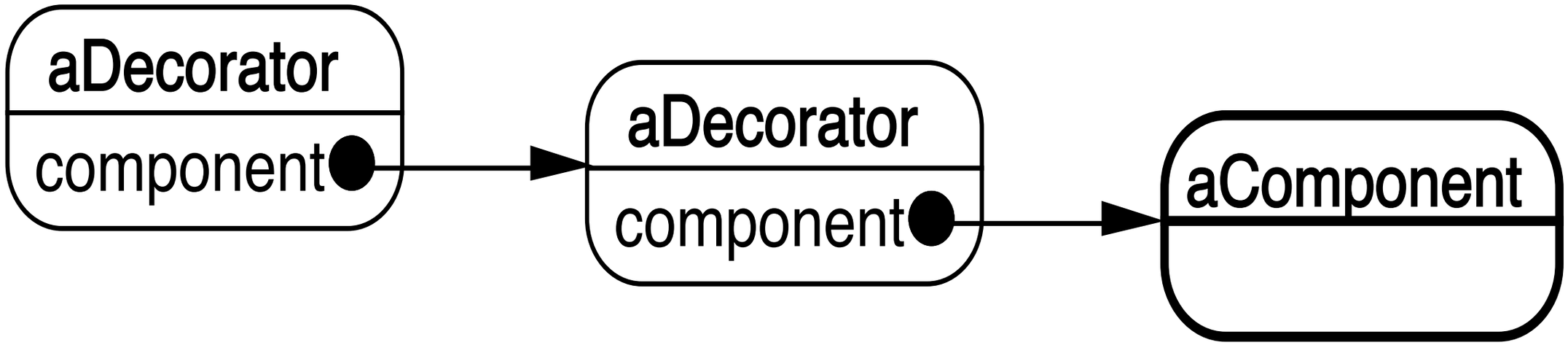
Decorator Pattern

Adds responsibilities to individual objects

Dynamically
Transparently


Component
operation()
ConcreteComponent
operation()
Decorator
component
operation()
ConcreteDecoratorA
ConcreteDecoratorB

Decorator forwards all component operations



Coffee Example from Wikipedia

Compute the cost of coffee

Base Price $1

Cost of Milk $0.50

Cost of Sprinkles $0.20

```
public interface Coffee {
    public double getCost();
    public String getIngredients();
}

public class SimpleCoffee implements Coffee {
    public double getCost() {  return 1;  }

    public String getIngredients() { return "Coffee"; }
}
```

Abstract Decorator

```
public abstract class CoffeeDecorator implements Coffee {
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    public double getCost() { // Implementing methods of the interface
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}
```

Milk

```
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    public double getCost() {
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
}
```

```
public class Main {
    public static void printInfo(Coffee c) {
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```

Simple Examples

Good for explaining basic concept

But

- Prices change
- New extras
- Seasonal extras

System driven by data

Download information to cash register

Simple Example & Functional Programming

Just compose functions

```
with-sprinkles(with-milk(base-price(amount)))

(with-sprinkles(with-milk(base-price amount)))


(def sprinkles-milk (comp with-sprinkles with-milk base-price))

(sprinkles-milk amount)
```

Functional Programming

```
(defn sprinkles-milk2
    [amount]
    (-> amount
        base-price
        with-milk
        with-sprinkles
        compute-tax))
```

Simple Example & Functional Programming

Use maps

prices

```
{:large 4.0
 :medium 3.5
 :small 3.0
 :milk 0.5
 :sprinkles 0.2 }
```

order

```
{:size :medium
 :extras [:milk :sprinkles]}
```

```
(compute-cost prices order)
```

Favor Composition over Inheritance

Benefits & Liabilities

Benefits

Simplifies a class

Distinguishes a classes core responsibilities from embellishments

Liabilities

Changes the object identity of a decorated object

Code harder to understand and debug

Combinations of decorators may not work correctly together