COMP 2511
Object Oriented Design & Programming

Week 06

So far,

OO design principles:

- Encapsulate what varies,
- Program to an interface, not an implementation,
- Favour composition over inheritance

Design Patterns

- Strategy, State, Template Method Patterns

Design By Contract

This week,

More Exception Handling, Generics and Collections

OO design principles

- Single Responsibility Principle

Design Patterns

- Iterator, Composite Pattern (Encapsulating collection objects)

Exceptions

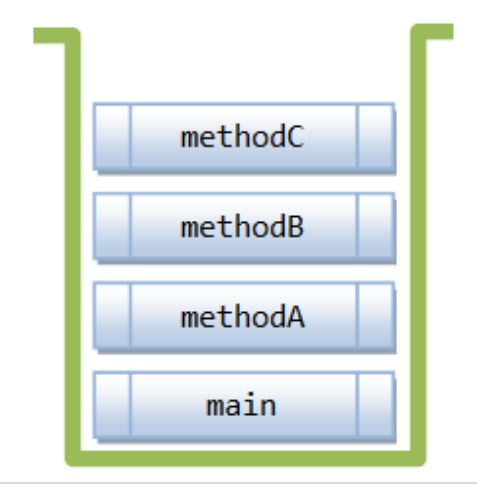
- Exceptions are abnormal events that occur during the execution of a program and disrupt the normal flow of the program e.g.,
  - attempting to open a file that does not exist
  - accessing before or after the valid range of an array
- Languages like C requires you to handle exceptions by intermingling main logic with several if-else statements
- Java provides a built-in exception handling mechanism to prevent programs from terminating abruptly when such event occur

Method Call Stack

A typical application involves many levels of method calls

```
public class CallStackDemo {

    public static void main(String[] args) {
       System.out.println("Enter main()");
       methodA();
       System.out.println("Exit main()");
    }

    public static void methodA() {
       System.out.println("Enter methodA()");
       methodB();
       System.out.println("Exit methodA()");
    }

    public static void methodB() {
       System.out.println("Enter methodB()");
       methodC();
       System.out.println("Exit methodB()");
    }

    public static void methodC() {
       System.out.println("Enter methodC()");
       System.out.println("Exit methodC()");
    }
}
```


A method call stack (Last-In-First-Out Queue) is used to manage these calls




```
<terminated> CallStackDemo [Java Application]
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit methodC()
Exit methodB()
Exit methodA()
Exit main()
```

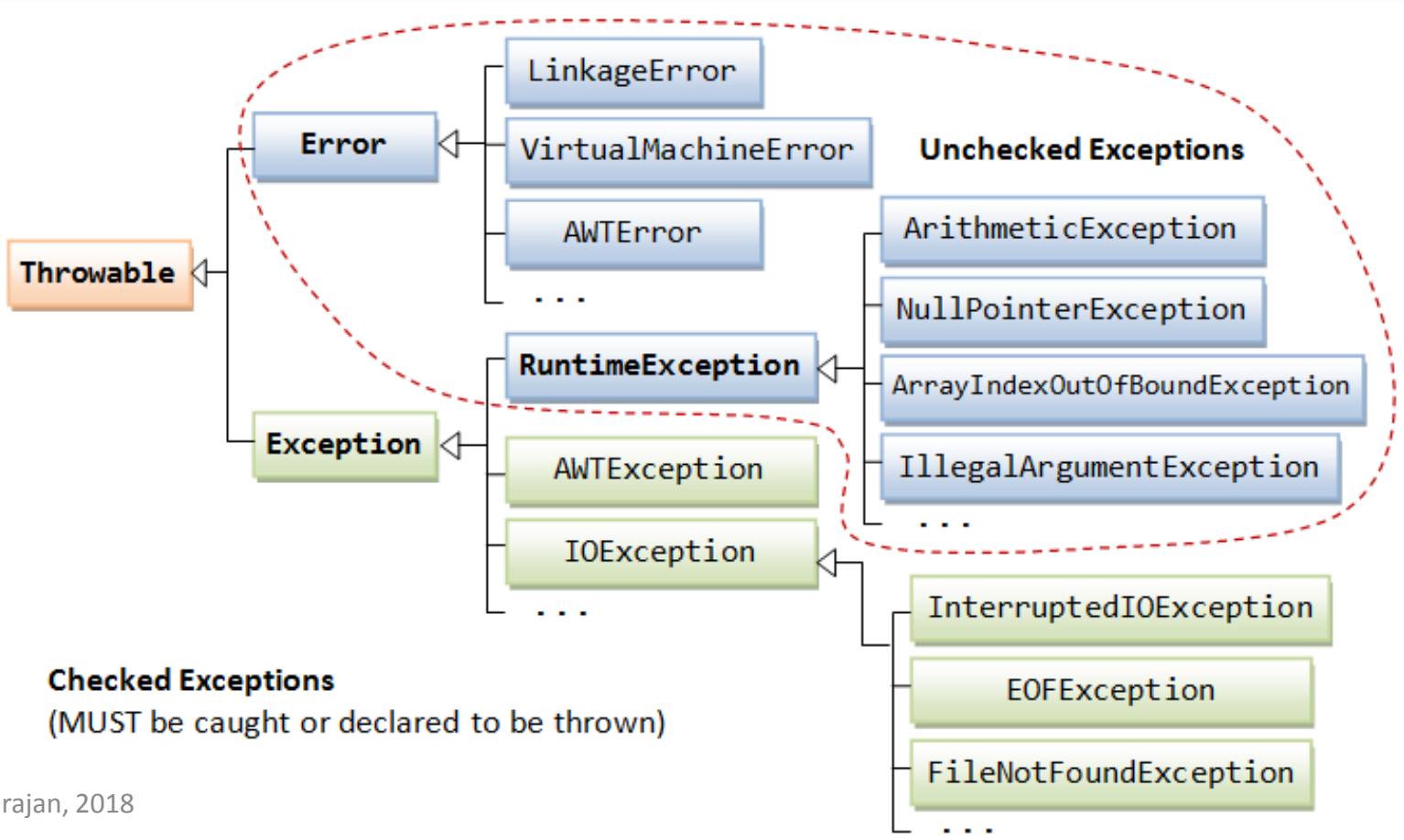
Exception and Call Stack

- `methodC()` performs a "divide by 0" operation, this will trigger an `ArithmeticException.`
- It does not handle this exception, so it is popped off from the call stack immediately.
- `methodB()` also does not handle this exception and popped off the call stack. So does `methodA()` and `main()` method.
- The `main()` method passes back to JVM, which abruptly terminates the program and print the call stack trace, as shown below

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at CallStackDemo.methodC(CallStackDemo.java:25)
        at CallStackDemo.methodB(CallStackDemo.java:18)
        at CallStackDemo.methodA(CallStackDemo.java:12)
        at CallStackDemo.main(CallStackDemo.java:6)
```

Java Exception Hierarchy

- `java.lang.Throwable` is the base class for all the exception classes
- `Error` class represents internal system errors (e.g., `VirtualMachineError` that rarely occur.



Exception Handling in Java

- Subclasses of `Error` and `RuntimeException` (e.g, `ArithmeticException`) are known as *unchecked* exceptions and are not checked by compiler, hence not typically "handled" or "declared"
- All other exceptions are called as *checked* exceptions and are checked by compiler
- Java's handling of checked exceptions consists of three operations:
  - Declaring exceptions
  - Throwing an exception
  - Catching an exception

Exception Handling in Java (1)

1. All checked exceptions must be declared

A Java method must declare in its signature the types of checked exception it may "throw" from its body, via the keyword "throws“ e.g.,

```
public void methodA() throws ExceptionX {
// An abnormal condition inside this method body may
// trigger ExceptionX
}

public Scanner(File source) throws
FileNotFoundException;
```

By declaring the exceptions in the method's signature, programmers are made to aware of the exceptional conditions in using the method.

Exception Handling in Java (2)

2. All checked exceptions must be handled

If a method declares an exception in its signature, you cannot use this method without handling the exception – the program will not compile e.g.,

```
public static void main(String[] args)  {
  File file = new File("E://file.txt");
  FileReader fr = new FileReader(file);
}
```

```
Compilation Error: Unhandled exception type
FileNotFoundException
```

The program above does not handle the exception declared and results in a compilation error

Exception Handling in Java (3)

3. To use a method that declares an exception in its signature, you MUST either:

- provide exception handling code in a `try-catch-finally` construct

```
public void doSomething(File file) {
try {
    FileReader fr = new FileReader(file);
} catch (IOException ie) {
    System.out.println("File not found");
}
```

- OR not handle the exception in the current method, but declare the exception to be thrown up the call stack for the next higher-level method to handle e.g.,

```
public void doSomething(File file) throws
IOException { ... }
```

Exception Handling in Java (4)

You can optionally catch a run-time exception, although the compiler does not force you to do so:

```
// Assume an array of Employee objects, where //
each index of the array is the employee id
public Employee findEmployee(int index) {
try {
   return employees[index];
} catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println(“Invalid employee id");
}
```

Exception Handling (5)
Custom Exceptions

You can create custom exception classes by extending `Exception` or one of its sub-classes

```
public class MaximumRetriesExceeded extends Exception {
      public MaximumRetriesExceeded() {
          super();
      }

      public MaximumRetriesExceeded(String message) {
          super(message);
      }
```

You must throw a custom exception yourself

```
throw new MaximumRetriesExceeded(" You have
exceeded the maximum allowable retry attempts");
```

Assertions in Java

- Syntax of assertion in Java:

  `assert <boolean_expression>;`

  `assert <boolean_expression>: “String Expression”`

  If `<boolean_expression>` evaluates to ***false***, then throw `AssertionError`

  The second argument is converted to a string and used as text in the `AssertionError` message

- Assertions are **disabled** by default in Java in production
- Enable assertions with:

```
java –enableassertions MyProgram
Java –ea MyProgram
```

Assertions in Java

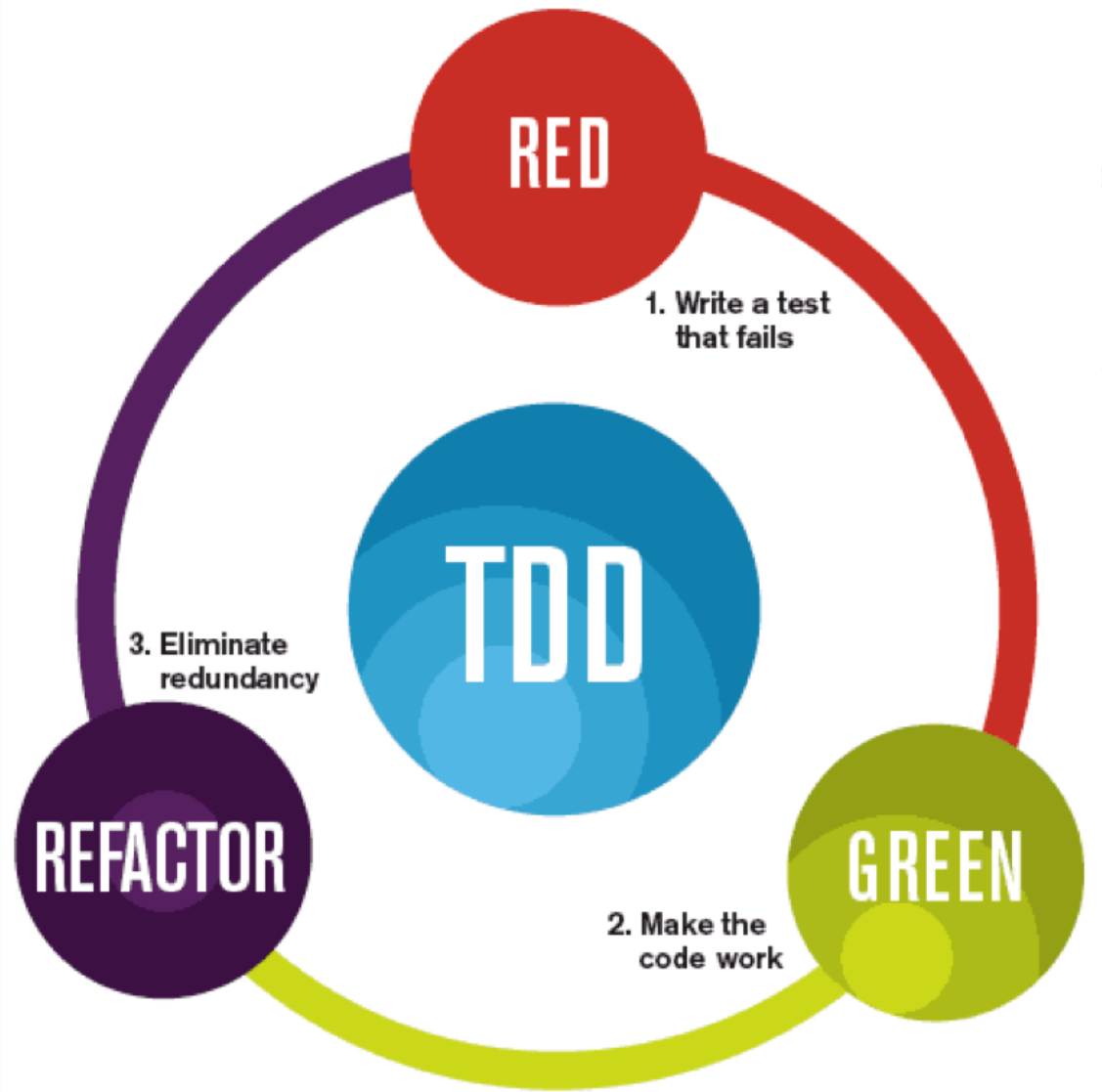
- Use assertions **only** to test that code behaves according to design contract
  - pre- and post-conditions
  - class invariants
- Inappropriate use of assertions

  Assertions are **disabled** by default in Java in production. Therefore do not use assertions for:
  - To check the parameters of a method
  - Do not use methods that can cause side effects in the assertion check

Test Driven Development

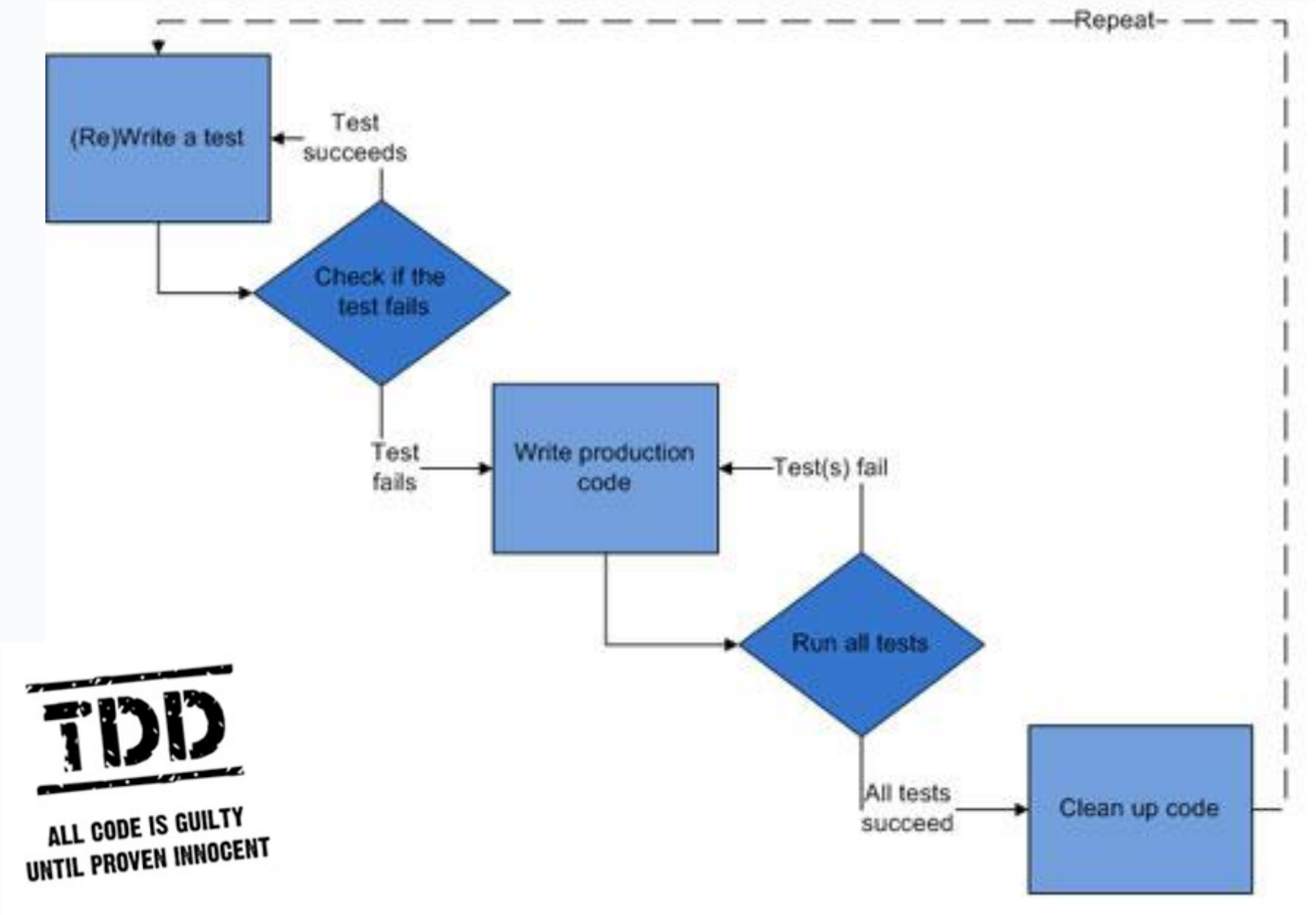
Test Driven Development

- Every step in the development process must start with a plan of how to verify that the result meets a goal
- Developer should not create a software artifact (a UML diagram, or source code) unless they know how it will be tested
- An important principle in XP, Scrum



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

© https://businessanalystlearnings.com/technology-matters/2014/8/13/test-driven-vs-behaviour-driven-development



Lecture demo…writing tests in JUnit

So, it continues…

- Many cases to consider
- Unit tests help to validate the complex logic and check for regression errors
- Brings the emphasis that every unit of code is tested
- So, the general rule is:
  - Start by writing a test (that fails)
  - Write just enough code to pass the test - You aren't gonna need it! (YAGNI)
  - Test again, and make corrections until test passes
  - Once passed, refactor code to remove redundancies
  - Move on to next piece of code

Generic Types, Iterator, Collections

Generics

```
// Prior to Java 5
public class BoxOld {

private Object t;
public Object get() {
        return t;
}
public void set(Object t) {
        this.t = t;
}

public static void main(String args[])
{
  BoxOld type = new BoxOld();
  type.set("apple");
  String s = (String) type.get();
  //type casting, - ClassCastException
  // return type is Object
...
```

```
// With Generics
public class Box<T> {

  private T t;
  public T get() {
        return t;
  }

  public void set(T t) {
        this.t = t;
  }

public static void main(String args[]){
  Box<String> type = new Box<>();
  type.set("apple");
  //type.set(10); // compiler-error

  // Compiler warning:
  // Box is a raw type. References to
generic
  // type Box<T> should be parameterized
  Box type1 = new Box();
  type1.set("apple");
  type1.set(10);

  }
}
```

Generics

- Generics added to Java 5 to provide **compile-time type checking** and avoid risk of **ClassCastException**
- A generic type is a class or interface that is parameterised over types. Use angle brackets <> to specify the **type parameter**

```
// Prior to Java 5

List list = new
ArrayList();
list.add(“apple”);
list.add(10);

for (Object obj: list) {
  String s = (String) obj;
}
```

```
// With Generics

List<String> list = new
ArrayList<>();
list.add(“apple”);
// Compiler-error
// list.add(10);

for (String val: list) {
    // no type casting
    String s = val;
}
```

Iterator Pattern

Diner and Pancake House Merger

```
public class MenuItem {
        private String name;
        private String description;
        private boolean vegetarian;
        private double price;

  public MenuItem(String name,
                  String description,
                  boolean vegetarian,
                  double price)
  {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
  }

  // all getter and setter methods ...

  public String toString() {
        return (name + ", $" + price +
        "\n   " + description);
  }
}
```


Diner Menu has lunch-items, while Pancake house has breakfast-items. Every menu-item has a name, description and price

Pancake House Menu

```
public class PancakeHouseMenu {
   private ArrayList<MenuItem> menuItems;

   public PancakeHouseMenu() {
     menuItems = new ArrayList<MenuItem>();

     addItem("K&B's Pancake Breakfast", "Pancakes
     with scrambled eggs, and toast", true, 2.99);

     addItem("Regular Pancake Breakfast", "Pancakes
     with fried eggs, sausage", false, 2.99);

     // ... Add other items
}

   public void addItem(String name, String
     description,boolean vegetarian, double price)
   {
     MenuItem menuItem = new MenuItem(name,
       description, vegetarian, price);
     menuItems.add(menuItem);
    }

public ArrayList<MenuItem> getMenuItems() {
return menuItems;
}
// other menu methods here
}
```

Each menu-item is added to the ArrayList, in the constructor

Other code that depends on the ArrayList implementation

Diner Menu

Diner Menu uses an Array so that they can control the max size of the menu

```
public class DinerMenu {
        public static final int MAX_ITEMS = 6;
        private int numberOfItems = 0;
        private MenuItem[] menuItems;

 public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];

        addItem("Vegetarian BLT",
        "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);

        // add other items...
 }
public void addItem(String name,String description, boolean vegetarian, double price)
{
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full!  Can't add item to menu");
        } else {
           menuItems[numberOfItems] = menuItem;
           numberOfItems = numberOfItems + 1;
        }
  }
public MenuItem[] getMenuItems() {
        return menuItems;
}
// other menu methods here
}
```

Problem with two different menu representations?

Java-Enabled Waitress: code-name "Alice"

```
printMenu()
   - prints every item on the menu

printBreakfastMenu()
   - prints just breakfast items

printLunchMenu()
   - prints just lunch items

printVegetarianMenu()
   - prints all vegetarian menu items

isItemVegetarian(name)
   - given the name of an item, returns true
     if the items is vegetarian, otherwise,
     returns false
```

The spec for the Waitress

Problem with two different menu representations?

```
public void printMenu() {
    // To print all the items in the menu, you need to call getMenuItems() on each menu
    // Each call to getMenuItems() returns a different type
    // Then implement two different loops to step through menu items of each menu
    // What if a third cafe needs to be merged?  Three loops?

    ArrayList<MenuItem> breakfast_items = pancakeHouseMenu.getMenuItems();

    System.out.println("MENU\n----\nBREAKFAST");
    for (MenuItem menuItem : breakfast_items) {
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
    }

    MenuItem[] lunchItem = dinerMenu.getMenuItems();
    for (int i=0; i< lunchItem.length; i++) {
        System.out.print(lunchItem[i].getName() + ", ");
        System.out.print(lunchItem[i].getPrice() + " -- ");
        System.out.println(lunchItem[i].getDescription());
    }
}
```

Implementation of every other method in the class `Waitress` `printBreakfastMenu()`, `printVegetarianMenu()` etc are all going to be a variation of the above theme.

More mergers, more implementation types, more loops...

Can we encapsulate the iteration?

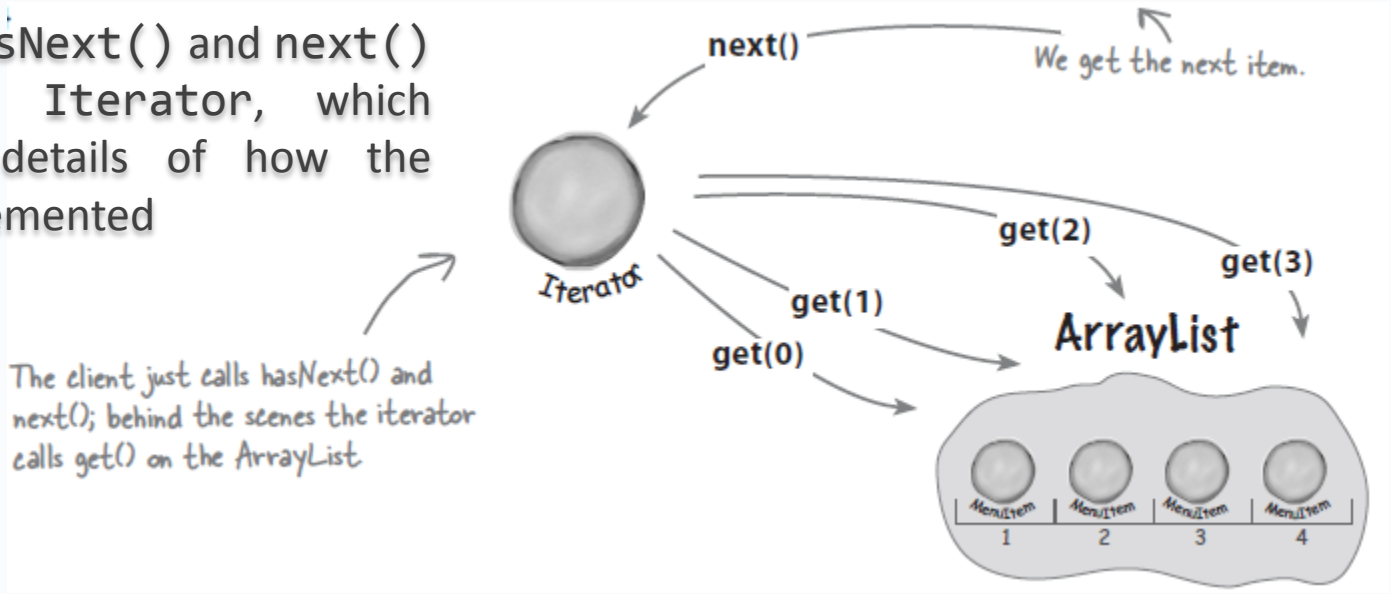
```
public void printMenu() {
    // To print all the items in the menu, you need to call getMenuItems() on each menu
    // Each call to getMenuItems() returns a different type
    // Then implement two different loops to step through menu items of each menu
    // What if a third cafe needs to be merged?  Three loops?

    ArrayList<MenuItem> breakfastItems = pancakeHouseMenu.getMenuItems();

    System.out.println("MENU\n----\nBREAKFAST");
    for (int i=0; i<=breakfastItems.size(); i++) {
        MenuItem menuItem = breakfastItems.get(i);
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
    }

    MenuItem[] lunchItems = dinerMenu.getMenuItems();
    for (int i=0; i< lunchItems.length; i++) {
        MenuItem menuItem = lunchItems[i];
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
    }
}
```

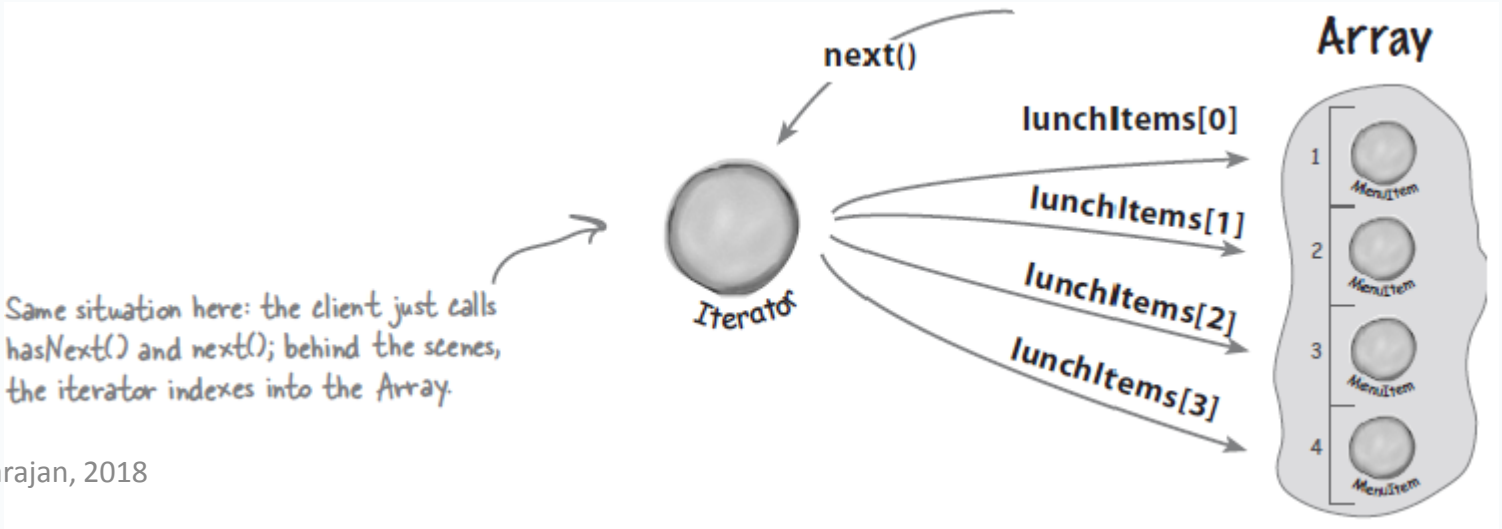
Remember our golden rule “Encapsulate what varies”

The Iterator Class

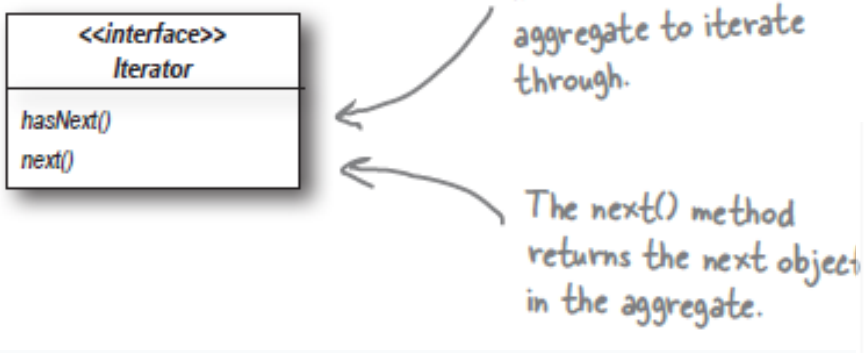
What if we create a class called `Iterator` that encapsulates the way we iterate through a collection of objects?

A client calls `hasNext()` and `next()` on the class `Iterator`, which abstracts the details of how the `next()` is implemented

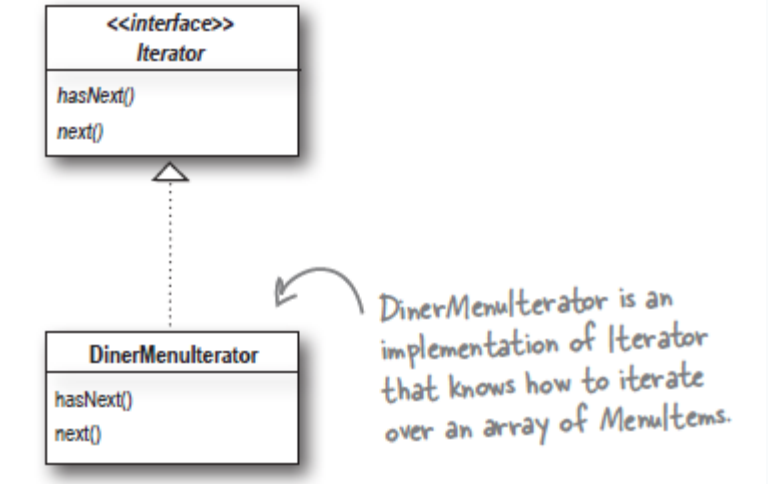




Meet the Iterator Pattern




```
public interface Iterator<MenuItem> {

    public MenuItem next();
    public boolean hasNext();
}
```




```
public class DinerMenuIterator implements Iterator<MenuItem> {
    private MenuItem[] list;
    private int position = 0;

    public DinerMenuIterator(MenuItem[] list) {
        this.list = list;
    }

    public MenuItem next() {
        MenuItem menuItem = list[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= list.length || list[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

position maintains the current position of the iteration over the array.

The constructor takes the array of menu items we are going to iterate over.

Integrate the iterator with the DinerMenu

```
public class DinerMenu {
    private static final int MAX_ITEMS = 6;
    private int numberOfItems = 0;
    private MenuItem[] menuItems;

    // constructor here

    // add Item


/*
 *   getMenuItems() which exposes our internal implementation
 *   will be replaced by a new metho
 */

//  public MenuItem[] getMenuItems()
//      return this.menuItems;
//  }

/*  New method createIterator() which creates a DinerMenuIterator
 * from the menuitems array and returs it to the client
 */
    public Iterator createIterator() {
        return new DinerMenuIterator(menuItems);
    }
    //other methods
}
```

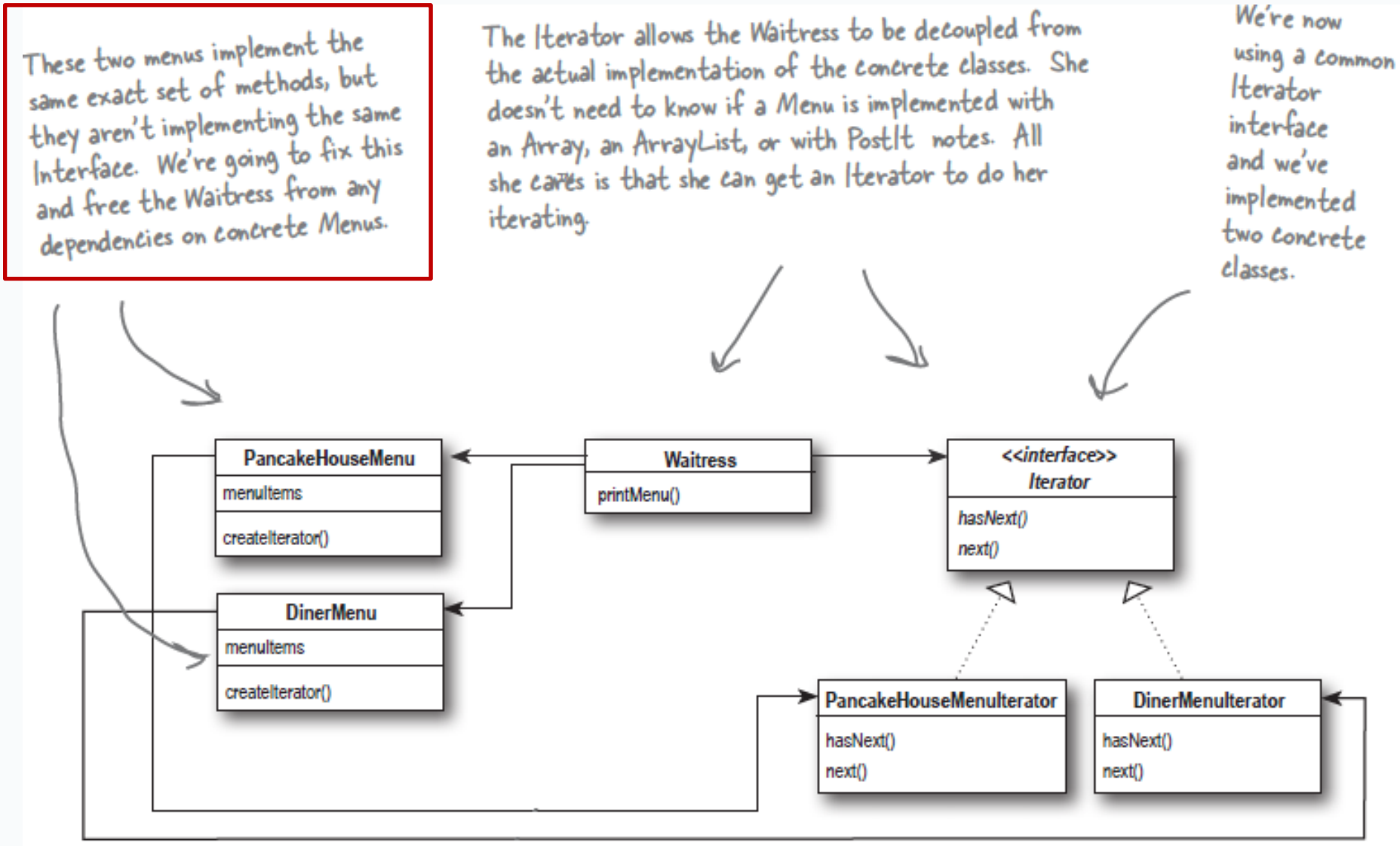
We're not going to need the getMenuItems() method anymore and in fact, we don't want it because it exposes our internal implementation!

We're returning the Iterator interface. The client doesn't need to know how the menuItems are maintained in the DinerMenu, nor does it need to know how the DinerMenuIterator is implemented. It just needs to use the iterators to step through the items in the menu.

And finally our Waitress class…

```
public class Waitress {
    private PancakeHouseMenu pancakeHouseMenu;
    private DinerMenu dinerMenu;

    public Waitress() {
        this.pancakeHouseMenu = new PancakeHouseMenu();
        this.dinerMenu = new DinerMenu();
    }

    public void printMenu() {
        // Integrate the iterator code into the waitress class

        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);

        System.out.println();
        System.out.println("----\nLUNCH");
        Iterator dinerIterator = dinerMenu.createIterator();
        printMenu(dinerIterator);

    }

    private void printMenu(Iterator dinerIterator) {
        while (dinerIterator.hasNext()) {
            MenuItem menuItem = dinerIterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
```

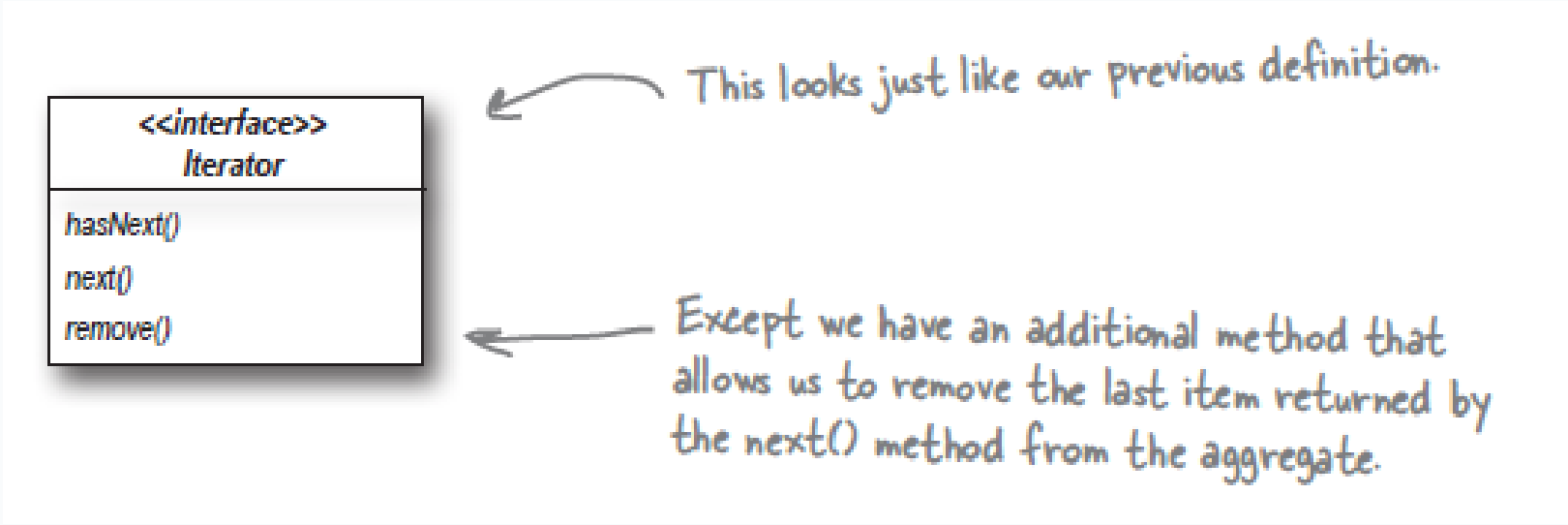
Our design so far…



We still need to fix the two menus, not implementing the same interface

Improving our design further…

- Use `java.util.Iterator` interface, instead of implementing our own



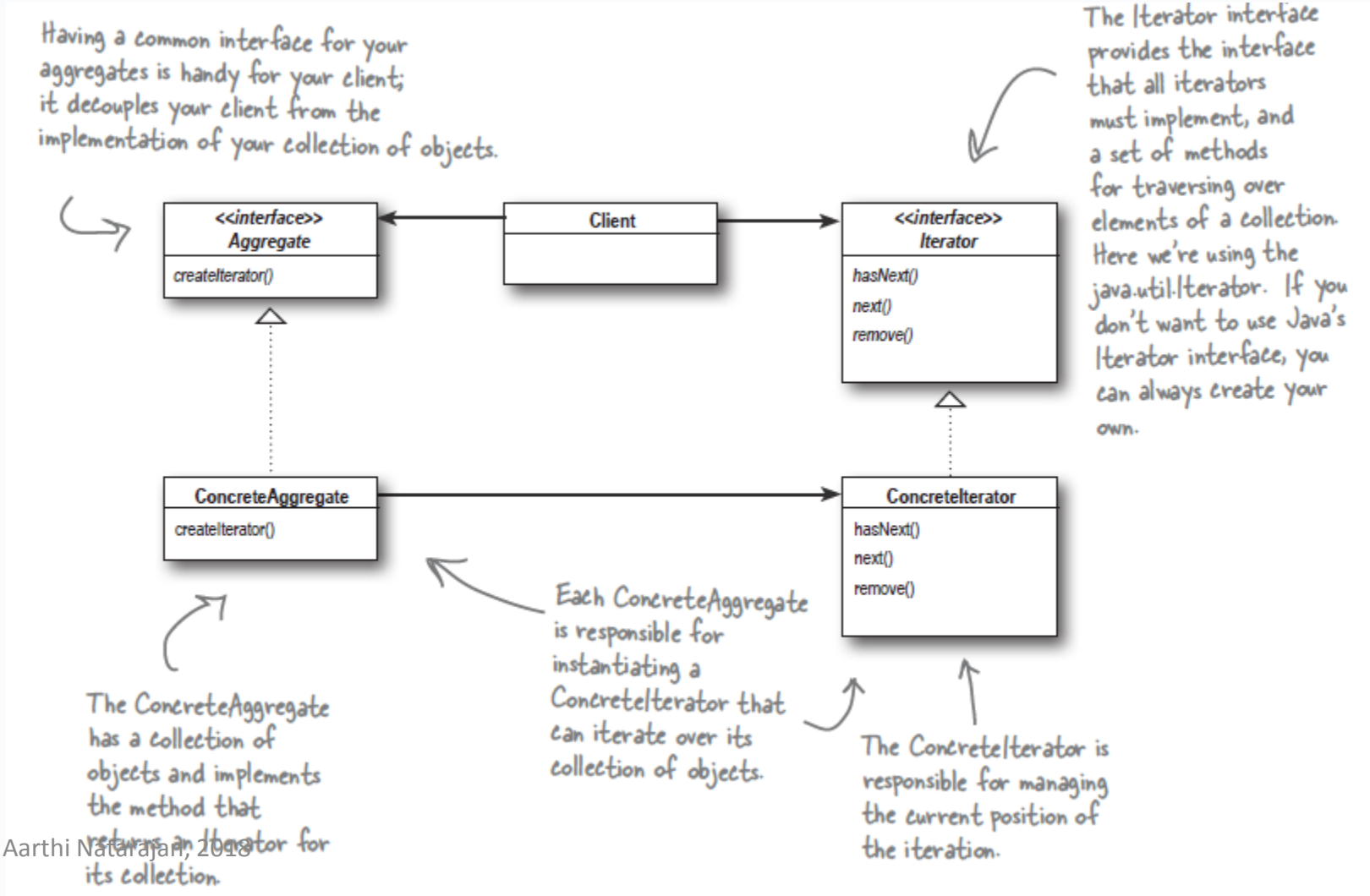
- Use the `iterator()` method on the `ArrayList`
- Still need an iterator for the `DinerMenu`, as this uses an `Array`, which doesn't support the `iterator()` method

Pattern #4: Iterator Pattern

- Motivation
  - Need to traverse items of diverse sorts of aggregate objects (lists, arrays) in uniform manner without exposing the internal structure of the aggregate object
- Intent
  - Create an iterator object that maintains a reference to one item in the aggregate object, and methods for advancing through the collection

The Iterator Pattern

Design Pattern #4: The `Iterator Pattern` provides a way to access the elements of an *aggregate object* sequentially without exposing its underlying implementation.



Single Responsibility Principle

- What if we allowed our aggregates (e.g. `DinerMenu` to implement their internal collections and related operations AND the iteration methods?
  - We are giving this aggregate class two responsibilities, managing the aggregate and iteration
  - And this means, it gives the class two reasons to change; (1) if the collection changes in some way or (2) if the way we iterate changes

Cohesion is a term you'll hear used as a measure of how closely a class or a module supports a single purpose or responsibility.

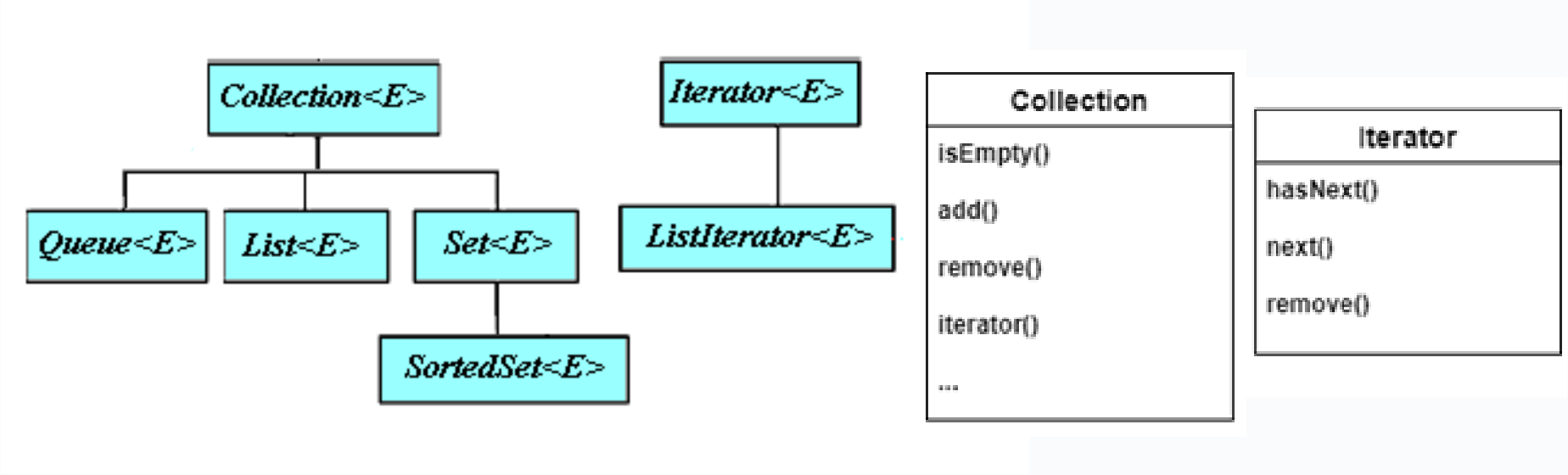
We say that a module or class has *high cohesion* when it is designed around a set of related functions, and we say it has *low cohesion* when it is designed around a set of unrelated functions.

Design Principle #4: A class should have only one reason to change (Single Responsibility Principle)

Cohesion and SRP are related. Classes that adhere to SRP tend to have high cohesion

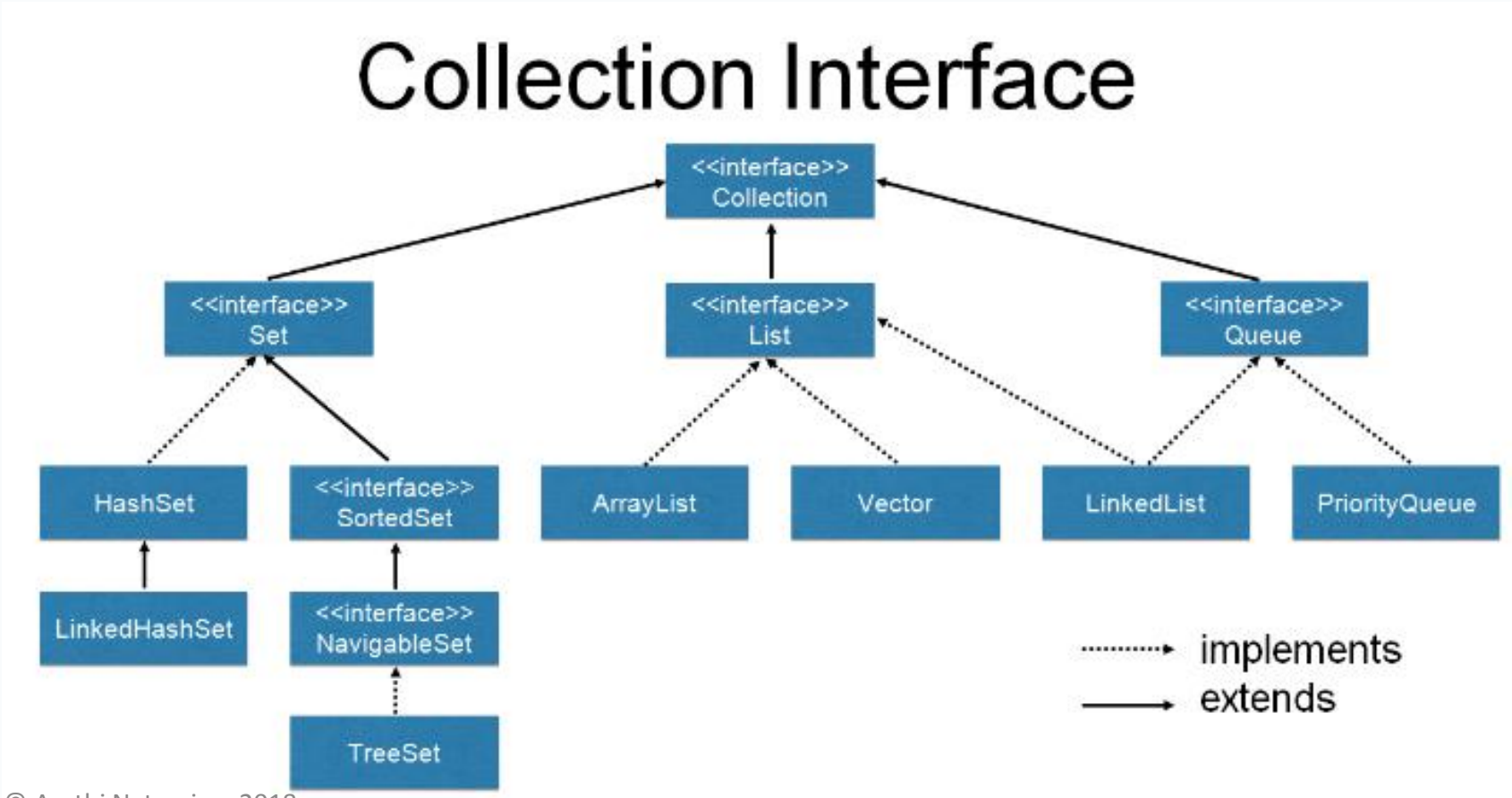
Collection Framework

- The collection framework provides two fundamental interface types: `Collection` and `Iterator`
  - A collection is a data-structure containing objects or elements
  - An iterator is a mechanism for visiting the elements of the collection
  - Makes it easier to handle collections of objects
  - Work like arrays, but size is dynamic and provide more advanced functionality



Collection Implementations

- Each collection interface has concrete classes that implement the interface
- e.g. ***`List`*** interface has the following concrete implementations: ***`ArrayList, LinkedList`***



Collections and Iterators

The nice thing about Collections and Iterator is that each Collection object knows how to create its own Iterator. Calling iterator() on an ArrayList returns a concrete Iterator made for ArrayLists, but you never need to see or worry about the concrete class it uses; you just use the Iterator interface.

Accessing a Collection: Three Ways

```
public class ListExample {


    public static void main(String[] args) {
        List<Country> countries = new ArrayList<>();

        List<String> fruits = new ArrayList<>();
        fruits.add("apple");
        fruits.add("orange");

        // Three methods of access

        //access via index
        String fruit1 = fruits.get(0);
        String fruit2 = fruits.get(1);

        //access via new for-loop
        for(String s : fruits) {
            System.out.println(s);
        }

        //access via iterator
        Iterator<String> it = fruits.iterator();
        while (it.hasNext()) {
            String fruitName = it.next();
            System.out.println(fruitName);
        }

    }
}
```

Inner classes

- A class declared within the body of another class
- There are three types of inner classes
  - Member classes
  - Local classes
  - Anonymous classes
- Commonly used in applications with GUI elements
- Serves as a “helper class” whose utilization is limited to the enclosing upper class

Inner class: An example

```
public class Car {
    private boolean running = false;
    private Engine engine = new Engine();

    // An inner class
    private class Engine {
        public void start() {
            running = true;
        }
    }

    public void start() {
        engine.start();
    }
}
```

Anonymous Inner class

A class defined without a name

```
public static void main(String[] args) {
    Age a = new Age() {

        @Override
        public int getAge() {
            // TODO Auto-generated method stub
            return 0;
        }

    };
    a.getAge();
}
```

Lecture Demo: Using anonymous inner classes in Collections.sort()

Design Toolbox

OO Basics

- *Abstraction*
- *Encapsulation*
- *Inheritance*
- *Polymorphism*

Refactoring Techniques

- *Extract Method*
- *Move Method*
- *Replace Temp With Query*
- *Replace Type Code with State/Strategy*
- *Replace conditional code with polymorphism*

OO Design Principles

- *Principle of least knowledge – talk only to your friends*
- *Encapsulate what varies*
- *Favour composition over inheritance*
- *Program to an interface, not an implementation*
- *Classes should be open for extension and closed for modification*
- *Don't call us, we'll call you*
- ***A class should have only one reason to change***

OO Patterns

- *Strategy*
- *State*
- *Template Method*
- ***Iterator***

Next week…
Composite Pattern