CS 5004

LECTURE 5

ADT, GENERICS, COLLECTIONS, HIGHER ORDER FUNCTIONS

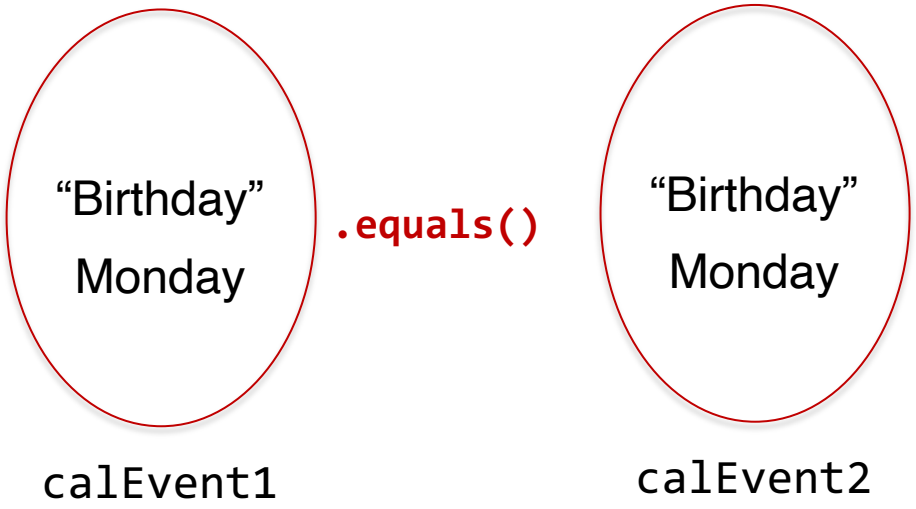
KEITH BAGLEY

SPRING 2022

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

REVIEW: EQUALITY AND COMPARISON: EQUALS()



== works for basic types like int, but probably NOT what you intend for objects

== essentially is asking: "is this the same object" (same identity), not
Are these equal

```
calEvent1 = new CalendarEvent(DAY.MONDAY, "Birthday");
calEvent2 = new CalendarEvent(DAY.MONDAY, "Birthday");
if (calEvent1.equals(calEvent2))  // true or false - depends on our override
```

FOR SEQUENCE ORDER: COMPARETO()

- The compareTo() method for comparing ordering sequence of objects. We get this method from Object (like toString()) and can override it if we want
  - Less than, equals, greater than
  - CompareTo method must return negative number if current object is less than other object, positive number if current object is greater than other object and zero if both objects are equal to each other.
  - CompareTo must be in consistent with equals method e.g. if two objects are equal via equals() , there compareTo() must return zero

COMPARETO()

- Example from Module 3 code from Amit & John
- Allows us to “order” objects sequentially based on the criteria WE set.

```
/**
 * Created by ashesh on 1/26/2017.
 */
public abstract class AbstractShape implements Shape {
  protected Point2D reference;

  public AbstractShape(Point2D reference) {
    this.reference = reference;
  }

  @Override
  public double distanceFromOrigin() {
    return reference.distToOrigin();
  }


  @Override
  public int compareTo(Shape s) {
    double areaThis = this.area();
    double areaOther = s.area();

    if (areaThis < areaOther) {
      return -1;
    } else if (areaOther < areaThis) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

COMPARETO()

- Before we write OUR implementation, we need to agree – conceptually – on what we're comparing for shapes
  - Is it the all of the instance variables?
    - Are some colors greater or less than others? Red > Blue?
  - Is it a derived value like area() or perimeter()?
  - Something else?

COMPARETO

- Let's add compareTo to our Shapes
  - We haven't talked about Generics yet (we'll get there this lecture) so for now, take the leap of faith with the strange syntax highlighted in yellow
- To do so, we'll need to add to the lists of interfaces our Shape hierarchy implements
  - We'll implement multiple protocols to add specific sets of behavior
  - Thus, classes can be subtypes of different, (even orthogonal) supertypes

```
public abstract class AbstractShape implements IShape, Comparable<AbstractShape>{
  private Point centerPoint;
```

COMPARETO() - THOUGHTS

- Notice we added compareTo to our abstract class so that all subclasses would be responsible for implementing this functionality.
- We changed the hierarchy and the change rippled through the subclass system.
  - What if we want to change how we sort? Instead of area, by absolute value of an attribute? Or by the perimeter?

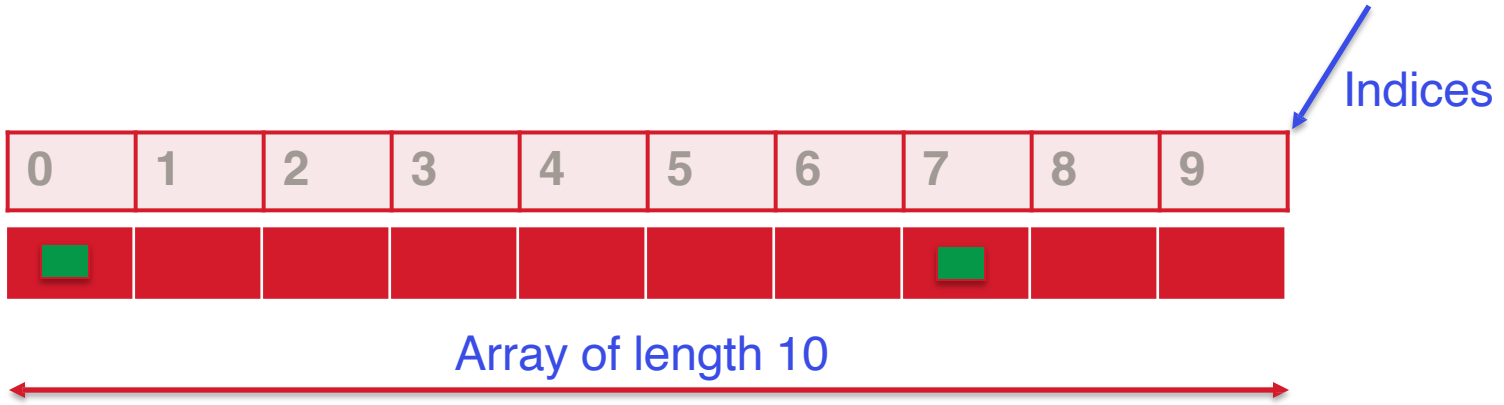
- In a previous lecture we reviewed the strength of composition vs. inheritance
- Let's consider the use of composition to do something similar, by relying on Comparators

SIMPLE COLLECTION: ARRAYS

Certain content in this section is based on content courtesy of:
Prof Tamara Bonaci (NU) and

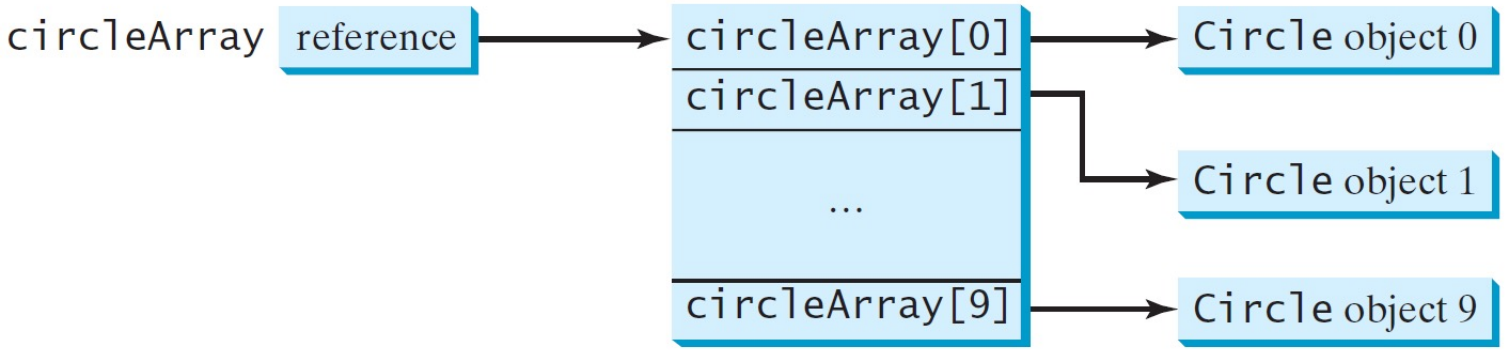
ARRAYS IN JAVA

- Array - container object that holds a fixed number of values of a single type
  - The length of an array is established when the array is created, and it is fixed after creation
  - Items in an array are called elements, and each element is accessed by its numerical index



ARRAY OF OBJECTS, CONT.

```
Circle[] circleArray = new Circle[10];
```




```
Circle [] circles = new Circle[10];
circles[0] = new Circle( radius: 2);
System.out.println(circles[0]);
```

MULTIDIMENSIONAL ARRAYS

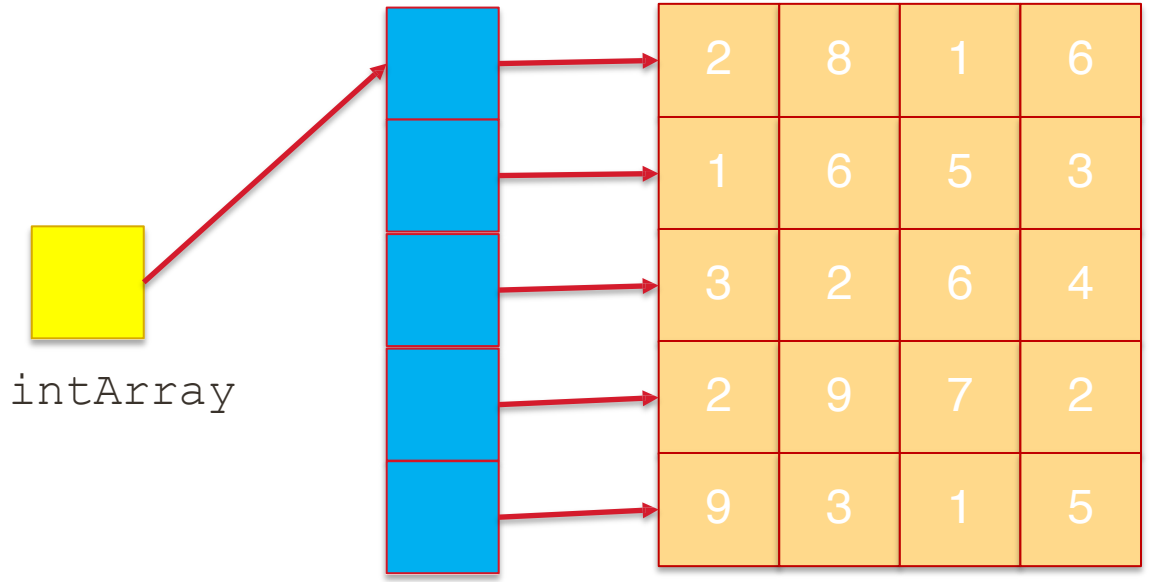
- Multidimensional arrays (jagged arrays) - arrays of arrays where each element of an array holds a reference to another array
- Created by appending one set of square brackets ([]) per dimension

```
int[][] intArray = new int[10][20]; //a 2D array
or matrix

int[][][] intArray = new int[10][20][10]; //a 3D
array
```

MULTIDIMENSIONAL ARRAYS

- Multidimensional arrays - arrays of arrays where each element of an array holds a reference to another array

```
int[][] intArray = new int[5][4]; //a 2D array or
matrix
```



MULTIDIMENSIONAL ARRAYS

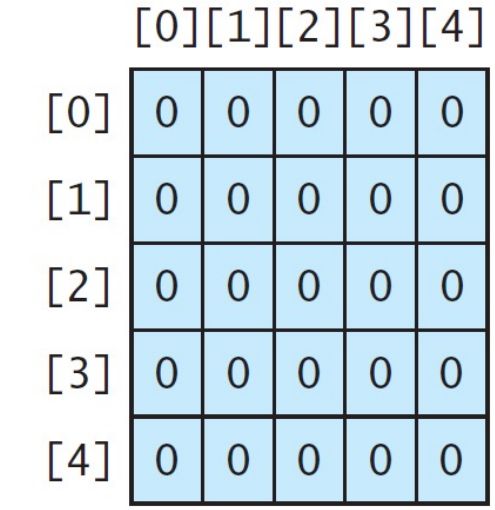
- Multidimensional arrays (jagged arrays) - arrays of arrays where each element of an array holds a reference to another array

```
int[][] intArray = new int[5][4]; //a 2D array or
matrix
```

The above is really equivalent to a 3-step process:

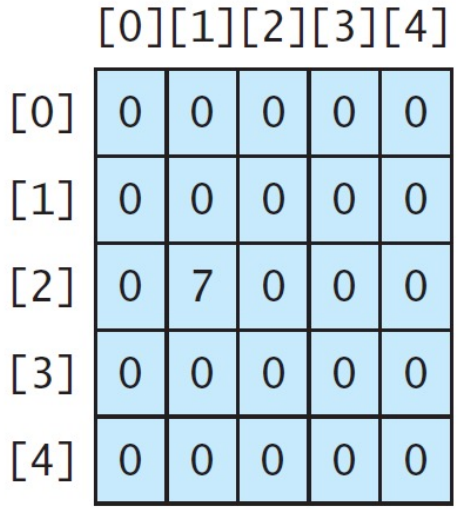
```
// create the single reference intArray (yellow square)
int [][] intArray;

// create the array of references (blue squares)
intArray = new int[5][];

// create the second level of arrays (red squares)
for (int i=0; i < 5 ; i++)
    intArray[i] = new int[4]; // create arrays of integers
```

TWO-DIMENSIONAL ARRAY ILLUSTRATION



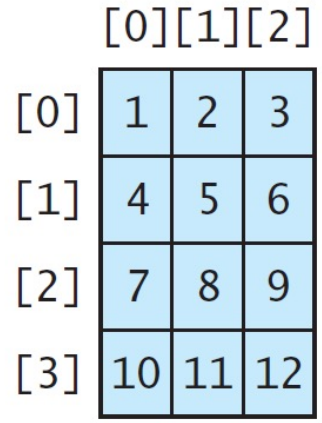

```
matrix = new int[5][5];
```

(a)




```
matrix[2][1] = 7;
```

(b)




```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

(c)

Primitive types are initialized for us

matrix.length? 5

matrix[0].length? 5

array.length? 4

array[0].length? 3

SHORTHAND FOR ARRAY INITIALIZATION

You can also use an array initializer to declare, create and initialize a two-dimensional array. For example,

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

Same as

```
int[][] array = new int[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

RAGGED ARRAYS

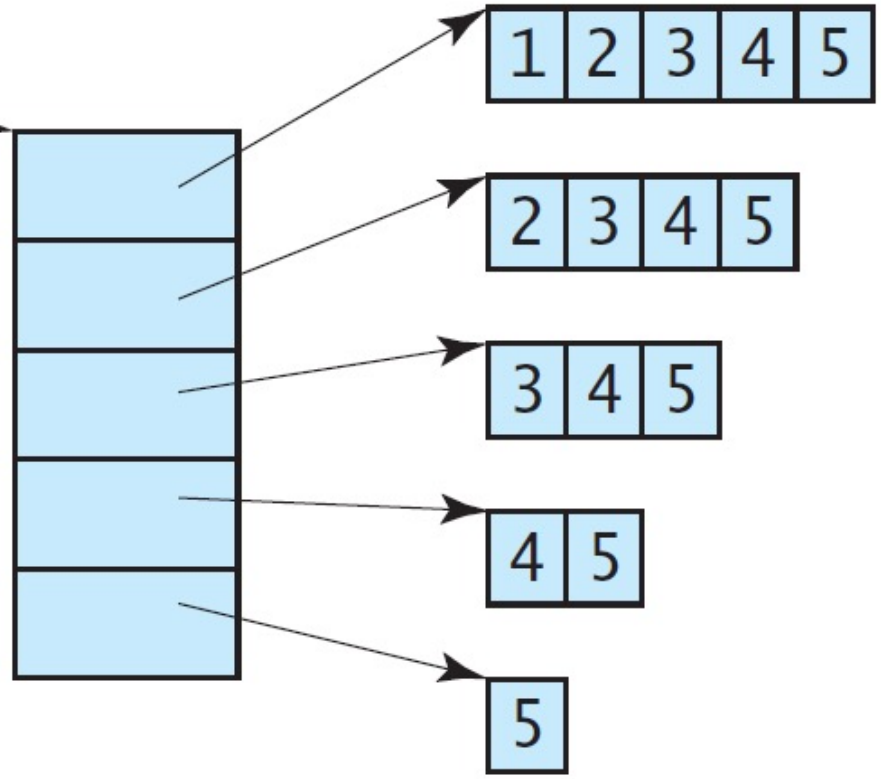
Each row in a two-dimensional array is itself an array. So, the rows can have different lengths. Such an array is known as *a ragged array*. For example,

```
int[][] matrix = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```

matrix.length is 5
matrix[0].length is 5
matrix[1].length is 4
matrix[2].length is 3
matrix[3].length is 2
matrix[4].length is 1

RAGGED ARRAYS, CONT.

```
int[][] triangleArray = {
   {1, 2, 3, 4, 5},
   {2, 3, 4, 5},
   {3, 4, 5},
   {4, 5},
   {5}
};
```



SORTING WITH ARRAYS.SORT()

- The Java Arrays class gives us an easy way to sort elements in an array
  - Elements must respond to compareTo or comparable interface, or be primitive types that understand < > ==
  - Import java.util.Arrays
- Let's sort a collection of Circles

COMPARATORS

- In a similar approach as compareTo, we can create general-purpose comparators to help us sort elements in a Collection
- Comparators realize the functional interface
  - The functional interface allows us to create classes that have a single method and use them as a “function object”
  - In this case, comparators implement one method: compare()
- A comparator is a separate class that decouples comparison algorithms from the class(es) they are used on.
  - Composition and delegation to allow for flexibility. Allows us to specify different comparison strategies based on need.

EXAMPLE: COMPARATORS

```
1  import java.util.Comparator;
2  class ShapeSizeComparator implements Comparator<IShape> {
3    public int compare(IShape shape1, IShape shape2) {
4      if (shape1.getArea() == shape2.getArea())
5        return 0;
6      else if (shape1.getArea() > shape2.getArea())
7        return 1;
8      else
9        return -1;
10   }
11 }
```

What if we want to use the Perimeter rather than Area for the sort?
Here is the original based on Area

YOU TRY!

- Take 5 minutes and see if you can write a new comparator that uses the perimeter to compare our shapes

EXAMPLE: COMPARATORS

```
3   import java.util.Comparator;
4   import shapes.IShape;
5
6   public class ShapePerimeterComparator implements Comparator<IShape> {
7   @   public int compare(IShape shape1, IShape shape2) {
8           // alternate approach
9           return Double.compare(shape1.getPerimeter(), shape2.getPerimeter());
10      }
11  }
```

Here is a second one based on Perimeter. Depending on the situation, we can select either one.

PREDICATES

- A predicate is a function that returns true or false based on a 1-parameter input value
- We can use predicates (and comparators) in expressions for higher-order functions to help process data more flexibly and efficiently
  - More on HOF's a bit later today
- The Functional Interface PREDICATE is defined in the *java.util.function* package. The main method we need to implement is the test() method

PREDICATE FOR OUR SHAPES

- Let's write a predicate to help us filter certain shapes from our array

```
3   import java.util.function.Predicate;
4   import othershapes.IShape;
5
6   public class LessArea implements Predicate<IShape>{
7     private final double area;
8
9     public LessArea(double area) {
10      this.area = area;
11    }
12
13    @Override
14  @ public boolean test(IShape shape) {
15      return shape.getArea() < area;
16    }
17  }
```

PREDICATES & LAMBDA

- Since Predicates are short test functions, we can often write them as lambda expressions rather than creating entire separate classes
  - Lambda expressions are “anonymous functions”
  - Used extensively in functional programming languages
  - If you know Python, you may have encountered lambda
  - Added to Java for convenience, not fully part of the Java DNA

PREDICATES & LAMBDA

- Lambda functions allow us to:
  - write a method that takes a (predicate) object as an argument and uses it.
  - In Java, functions cannot be passed as arguments directly, because Java treats *data* and *methods* as two different things. Therefore we enclose the function inside a (Predicate) object, and then pass it around.
- Most functional programming languages do not distinguish between data and functions. The approach in those languages is more concise than in Java.

- Don’t worry…we’ll work through some code this week for this stuff

USING A LAMBDA FOR OUR PREDICATE

- Short-hand lambda for the predicate we wrote previously
  - Don’t worry, we’ll cover the collection classes like Lists more extensively – just roll with it for now!

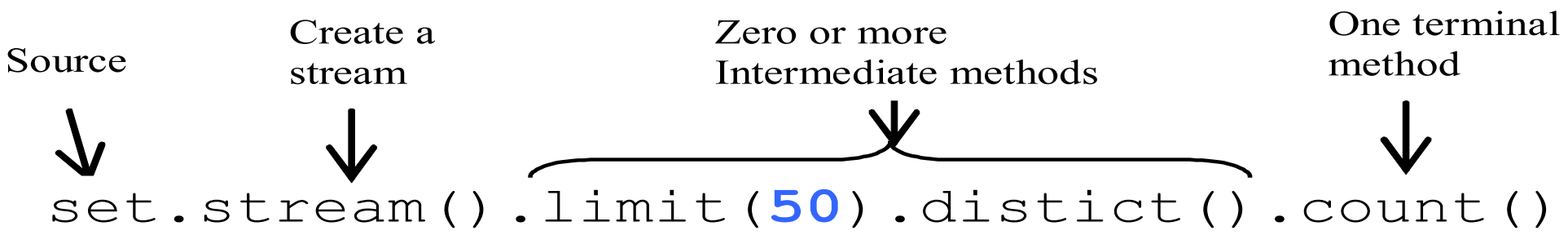
```
List<IShape> result = shapeList.stream().filter(iShape -> iShape.getArea() < 20).collect(Collectors.toList());
```

STREAMS

- A *stream* is a sequence of elements. *filter* and *count* are example operations that you can apply on a stream.
- Streams are used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- A stream is not a data structure: it takes input from Collections, Arrays or I/O channels.
- Streams don’t change the original data structure; they return a result per the pipeline.
- Terminal operations mark the end of the stream and return the result

EXAMPLE STREAM PIPELINE

A stream pipeline consists of a stream created from a data source, zero or more intermediate methods, and a final terminal method.

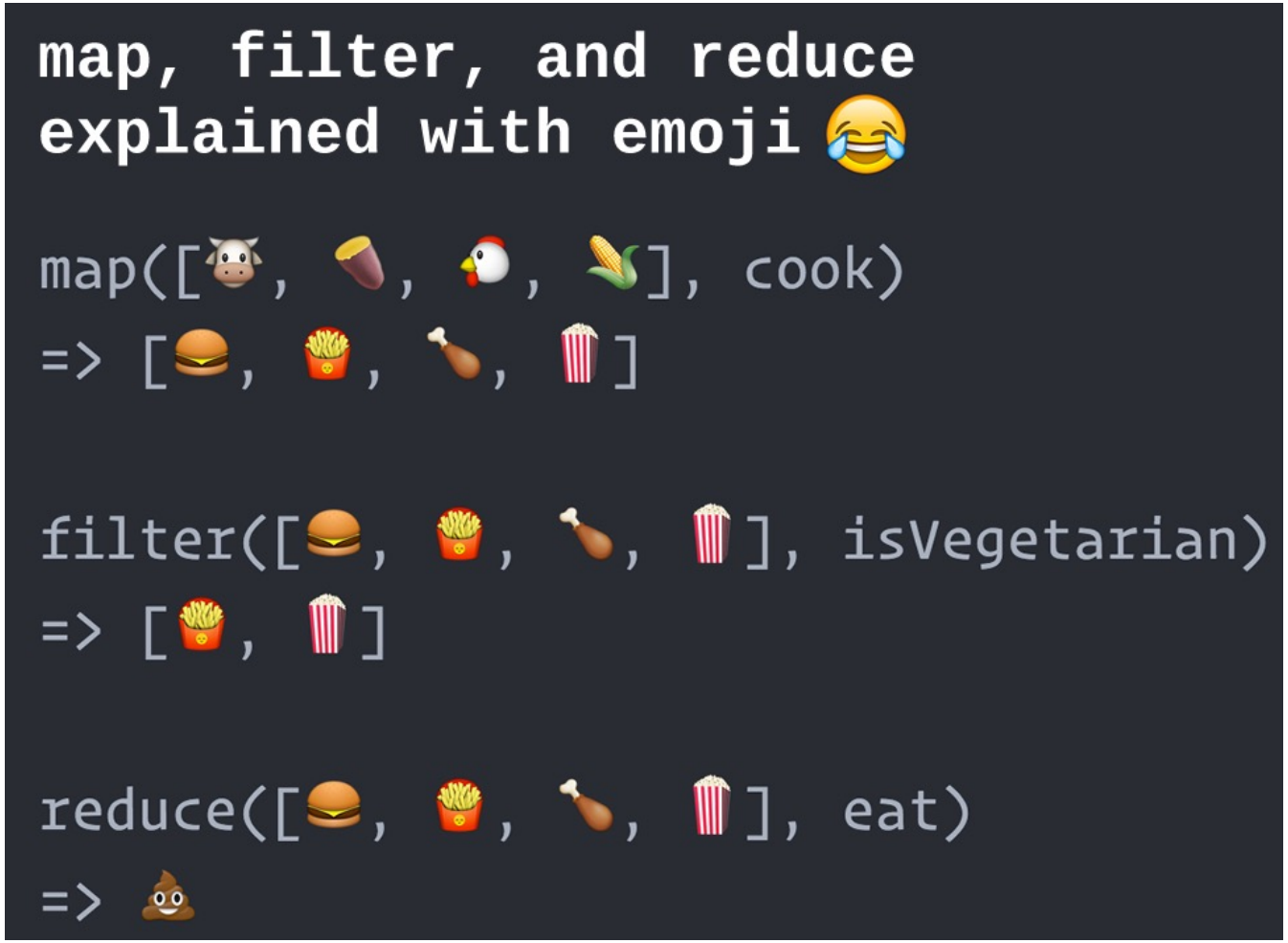


HIGHER ORDER FUNCTIONS

PUTTING IT ALL TOGETHER: DATA IN COLLECTIONS

- Often, we want to do more than just hold information. We actually want to process it
- Ask questions, get answers, build models and other representations
  - Filter: create a sublist that satisfies a condition
  - Map: 1-1 “mapping”, returning a list the same size containing relevant elements
  - Fold/Reduce: combine initial value with list elements to get to some combined representation

PREVIOUS SLIDE FOR THE QUIET NERD ALL OF US



Content from: http://www.globalnerdy.com/wordpress/wp-content/uploads/2016/06/map-filter-reduce-in-emoji-1.png

Courtesy of Joey Devilla

HIGHER ORDER FUNCTIONS

- The previous slide outlines higher order functions
- We can use the concept of predicates (and lambda), streams, and collections to pass our requirements to the HOFs and have the pipeline do the heavy lifting for us
- A higher-order function is a function that takes another function as an argument, and possibly returns a function as a value. Designing an operation as a higher-order function offers a powerful means of abstraction. Higher-order functions are supported to some degree in many programming languages, often supported best by languages that support functional programming.

HIGHER ORDER FUNCTIONS

- ***Filter***: *Given a list, create a sublist of things in the original list that satisfy a given condition*. This *List => List* operation is called a **filter**. One may think of a filter as a generic operation on a list, and returns a sublist of the same type. It takes the general form
  - List filter(List input, Predicate p)
- *Fold* is of the form *List => value*. A *fold* operation starts with an initial value, an algorithm to combine this initial value with elements in the list and return the combination. Examples of fold include counting the number of books in the list, calculating the total price of all books, determining if the list contains a book by Rowling, etc.

HIGHER ORDER FUNCTIONS

- ***Map***: A *map* operation takes in a list and returns a list of the same size, but possibly of a different type. The function that a *map* applies to each element of the list can be thought of as mapping the element to another element

HOF EXAMPLE: TOTAL THE AREAS OF OUR SHAPES

- We can map from Shapes to double values (Double)
  - Then use reduce (fold) to calculate a single value
  - We'll write this code later today after we cover lists!

```
System.out.println("---- Using List Map and Reduce to get a single value ----");

// map and reduce. First map the shapes 1-1 to their areas
List<Double> areas = shapeList.stream()
        .map(s->s.getArea())
        .collect(Collectors.toList());
// Now we have a list of areas (Double). Add them together to get a single value
Double total = areas.stream().reduce( identity: 0.0, (one, two) -> one + two);
System.out.println("Total =  " + total);
}
```

ADT REVIEW

Certain content in this section is based on content courtesy of:
Prof Tamara Bonaci (NU)

REMEMBER: AN ADT HAS A PROTOCOL

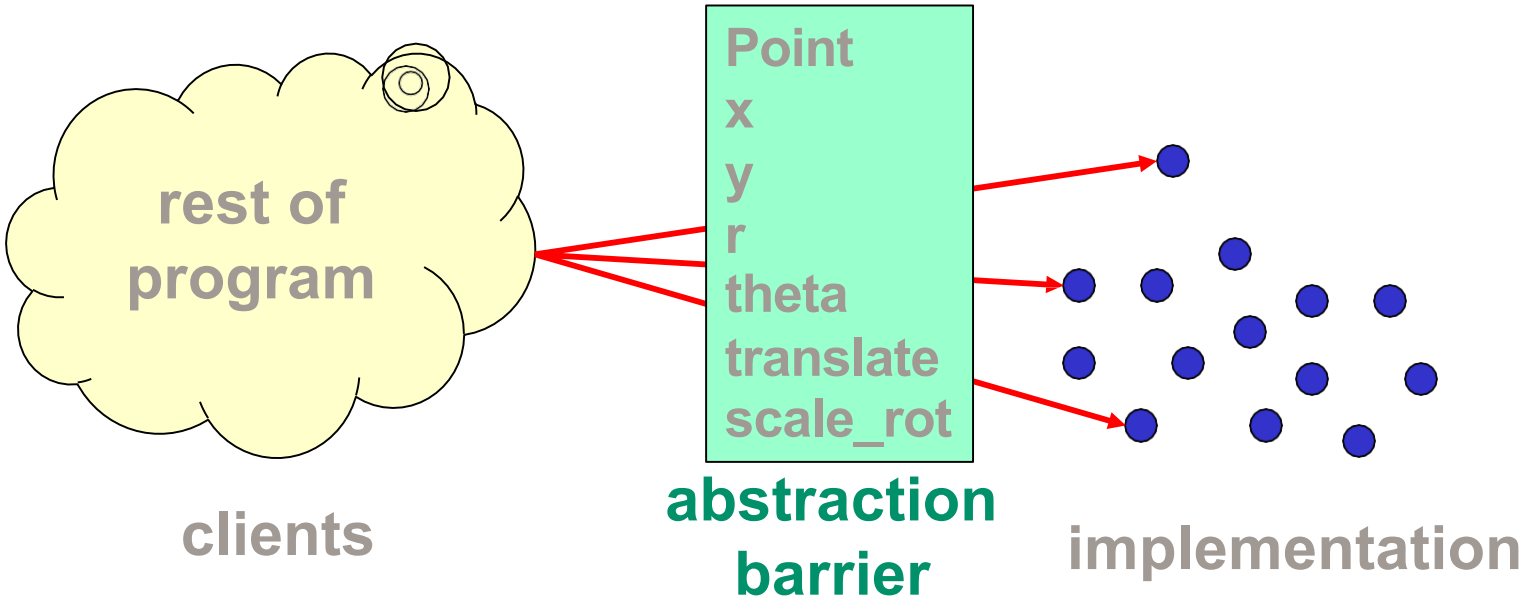
- ADT abstract from the organization to the meaning of data
  - protocol
- ADT abstracts from structure to use
- Idea: representation should not matter to the client ➔ so hide it from the client
- Example:

```
Class RightTriangle1 {
   Float base;
   Float altitude;
}
```

```
Class RightTriangle2 {
   Float base;
   Float hypothenuse;
   Float angle;
}
```

- Think of a data type as a set of operation:
  - `create, getBase, getAltitude, getBottomAngle`
- This way of thinking forces clients to use operations to access data

REMEMBER: ADT = OBJECT + OPERATIONS



- Implementation is hidden
- The only operations on objects of the type are those provided by the abstraction

SPECIFYING A DATA ABSTRACTION

- A collection of procedural abstractions
  - Not a collection of procedures
- An abstract state
  - Not the (concrete) representation in terms of fields, objects, …
  - "Does not exists", but used to specify operations
  - Concrete state, not part of the specification, implements the abstract state
- Each operation described in terms of "creating", "observing", "producing" or "mutating"
  - No operations other than those in the specification

SPECIFYING AN ADT

Immutable

1. overview
2. abstract state
3. creators
4. observers
5. producers
6. ~~mutators~~

Mutable

1. overview
2. abstract state
3. creators
4. observers
5. producers (rare)
6. mutators

- **Creators:** return new ADT values (e.g., Java constructors)
- **Producers:** ADT operations that return new ADT values
- **Mutators:** modify a value of an ADT (setters)
- **Observers:** return information about an ADT (getters)

CONCEPT OF A 2D POINT AS AN ADT

```
public class Point {
    private Float x;
    private Float y;

    //Can be created
    public Point(Float x, Float y){…}

    //Can be produced
    public Point centroid(Set<Point> points){..}

    //Can be observed
    public Float getX() {return this.x;}
    public Float getY() {return this.x;}

    //Can be moved (mutated)
    public void translate(Float deltaX, Float deltaY){…}
    public void scaleAndRotate(Float deltaX, Float deltaY){…}
}
```

Creator

Producer

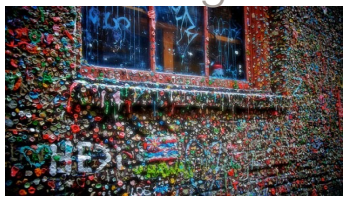
Observers

Mutators

JAVA COLLECTIONS

DATA COLLECTIONS

Collection of chewed gums



Collection of pens

Collection of cassette tapes

Collection of old radios

What is a data collection?

Shoes collection

Star wars collection

Cars collection

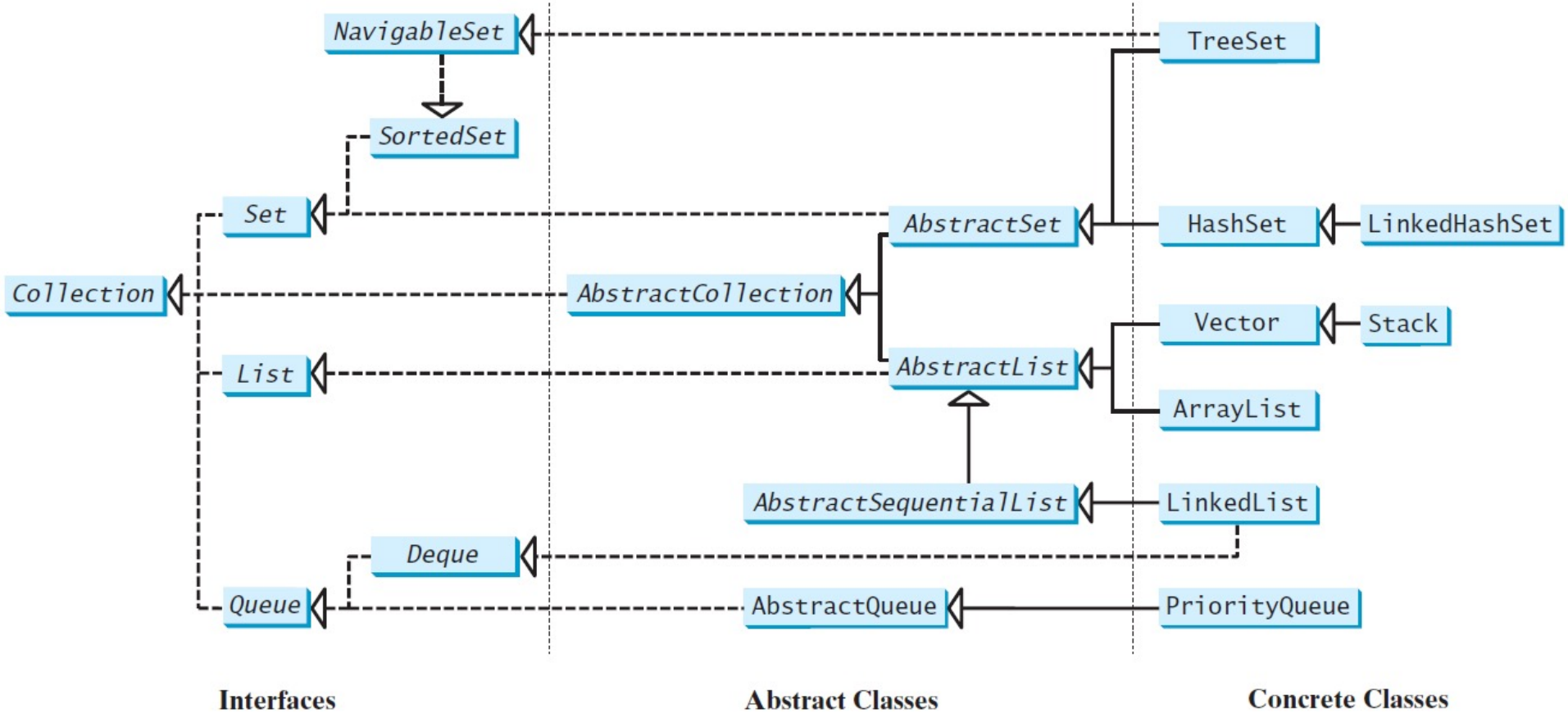
[Pictures credit: http://www.smosh.com/smosh-pit/articles/19-epic-collections-strange-things]

COLLECTIONS

- Data collection - an object used to store data (think data structures)
  - Stored objects called elements
  - Some typically operations:
    - `add()`
    - `remove()`
    - `clear()`
    - `size()`
    - `contains()`
- Some examples: ArrayList, LinkedList, Stack, Queue, Maps, Sets, Trees

Why do we need different data collections?

JAVA COLLECTIONS API



[Picture credit: Liang 2020

JAVA COLLECTIONS FRAMEWORK

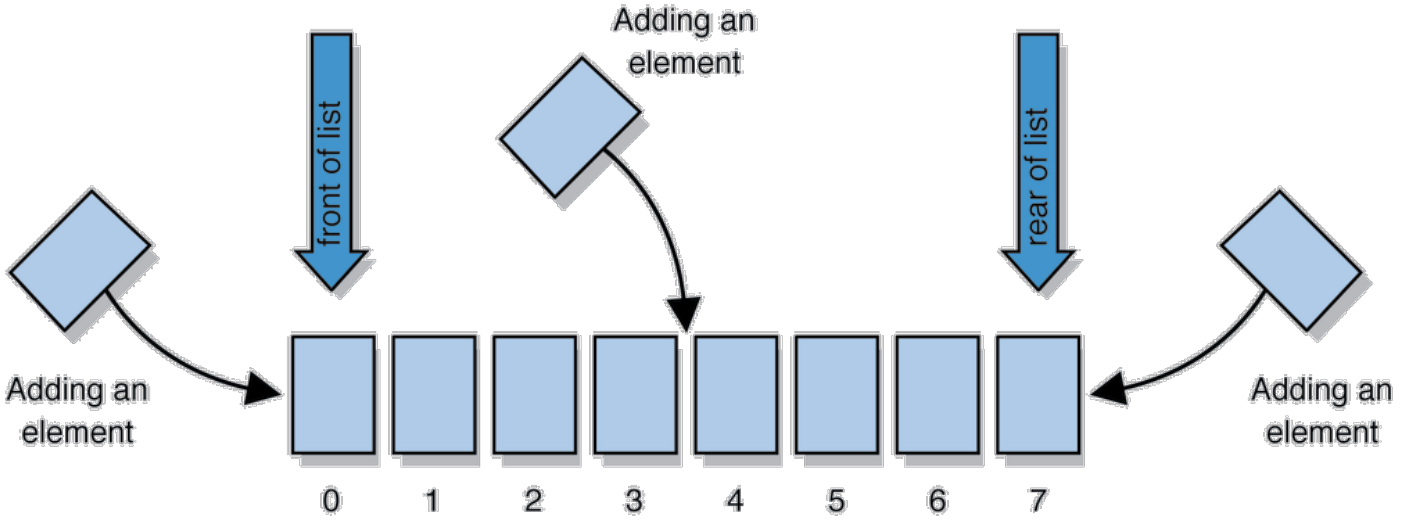
- Part of the `java.util` package

- Interface `Collection<E>`:
  - Root interface in the collection hierarchy
  - Extended by four interfaces:
    - `List<E>`
    - `Set<E>`
    - `Queue<E>`
    - `Map<K,V>`
  - Extends interface `Iterable<T>`

LISTS

LISTS

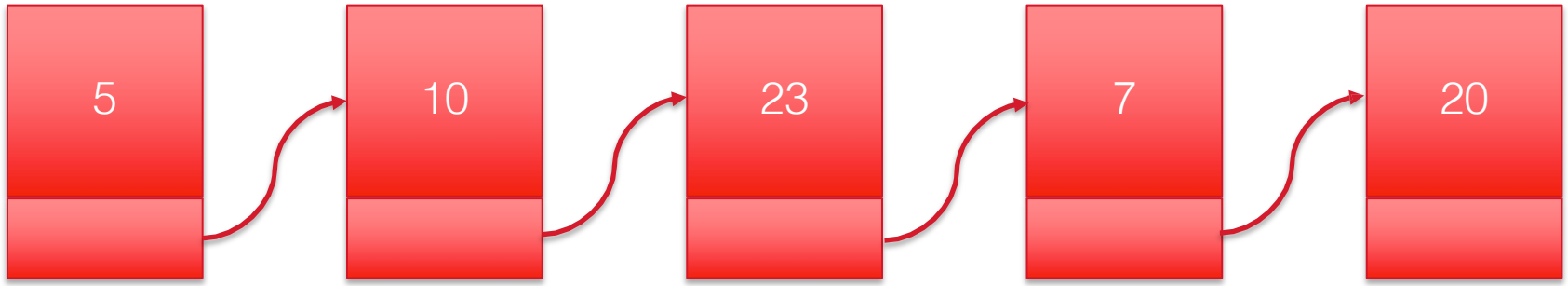
- List – an ordered collection (also known as a sequence)
  - A user controls where in the list each element is inserted
  - A user can access elements by their integer index (position in the list), and search for elements in the list
- Size - the number of elements in the list
- List allows for elements to be added to the front, to the back, or arbitrary



LISTS AND LINKED DATA STRUCTURES

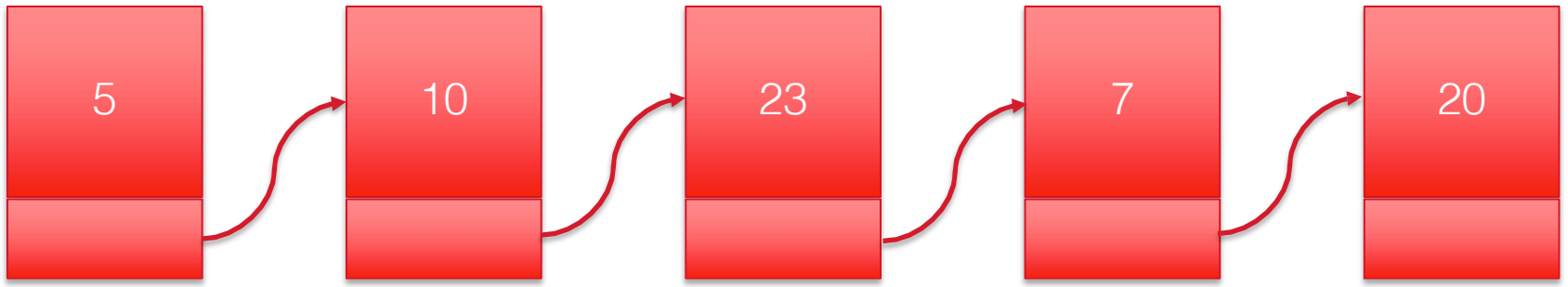
- Lists use one of the following underlying structures:
  - An array of elements

  - A collection of linked objects, each storing one element, and one or more references to other elements



LINKED LIST

- A collection of linked objects, each storing one element, and one or more references to other elements

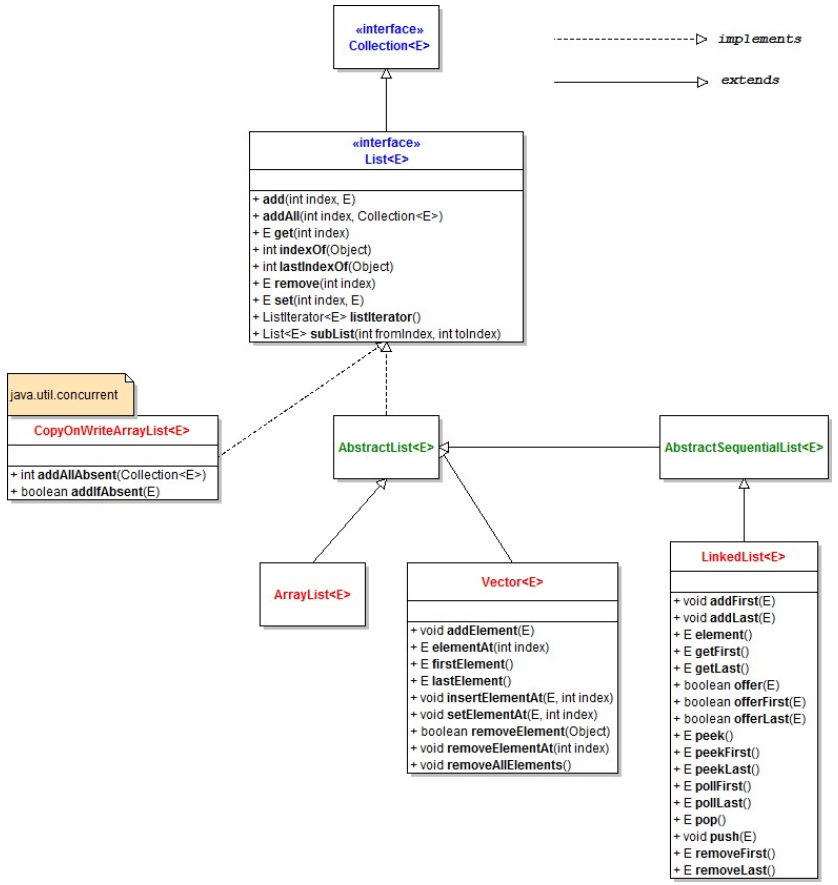


ARRAYLIST VS. LINKED LIST

- List - a collection of elements with 0-based indexes
  - Elements can be added to the front, back, or in the middle
  - Can be implemented as an ArrayList or as a LinkedList

  - What is the complexity of adding an element to the front of an:
    - ArrayList?
    - LinkedList?

JAVA LIST API



[Pictures credit: http://www.codejava.net]

- `List<E>` - the base interface
- Abstract subclasses:
  - `AbstractList<E>`
  - `AbstractSequentialList<E>`
- Concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`
  - `Vector<E>` (legacy collection)
  - `CopyOnWriteArrayList<E>` (class under java.util.concurrent package)
- Main methods:
  - `E get(int index);`
  - `E set(int index, E newValue);`
  - `Void add(int index, E x);`
  - `Void remove(int index);`
  - `ListIterator<E> listIterator();`

JAVA LIST API

- `List<E>` - the base interface
- Concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`
- Main methods:
  - `E get(int index);`
  - `E set(int index, E newValue);`
  - `Void add(int index, E x);`
  - `Void remove(int index);`
  - `ListIterator<E> listIterator();`

LET'S USE A LIST TO COLLECT MULTIPLE SHAPES

- Rather than an array, let's use a list to collect our shapes and then sort them.

FILTERING & FOLDING OUR LIST

- How would we filter our list to collect shapes of a certain area? Or of a certain color?
- Let's write some code to do that, then we'll also fold the list into a single value using the template code from one of the previous slides.

JAVA CLASS STACK

`Stack <E> ()`	Object constructor – constructs a new stack with elements of type E
`push(value)`	Places given value on top of the stack
`pop()`	Removes top value from the stack, and returns it. Throws `EmptyStackException` if the stack is empty.
`peek()`	Returns top value from the stack without removing it. Throws EmptyStackException if the stack is empty.
`size()`	Returns the number of elements on the stack.
`isEmpty()`	Returns true if the stack is empty.

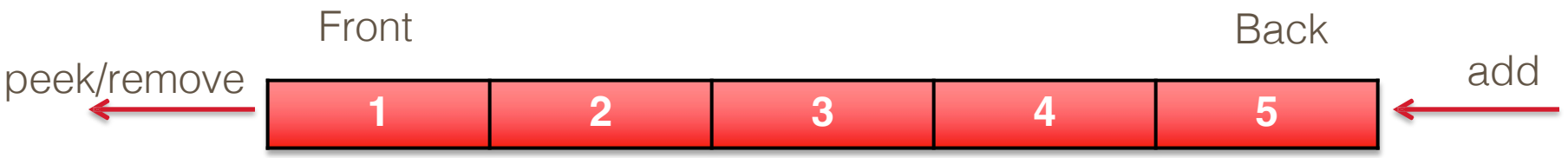
- Example:

```
Stack<String> s = new Stack<String>();
s.push("Hello");
s.push("PDP");
S.push("Fall 2017"); //bottom ["Hello", "CS 5004", "Spring 2020"] top
System.out.println(s.pop()); //Spring 2020
```

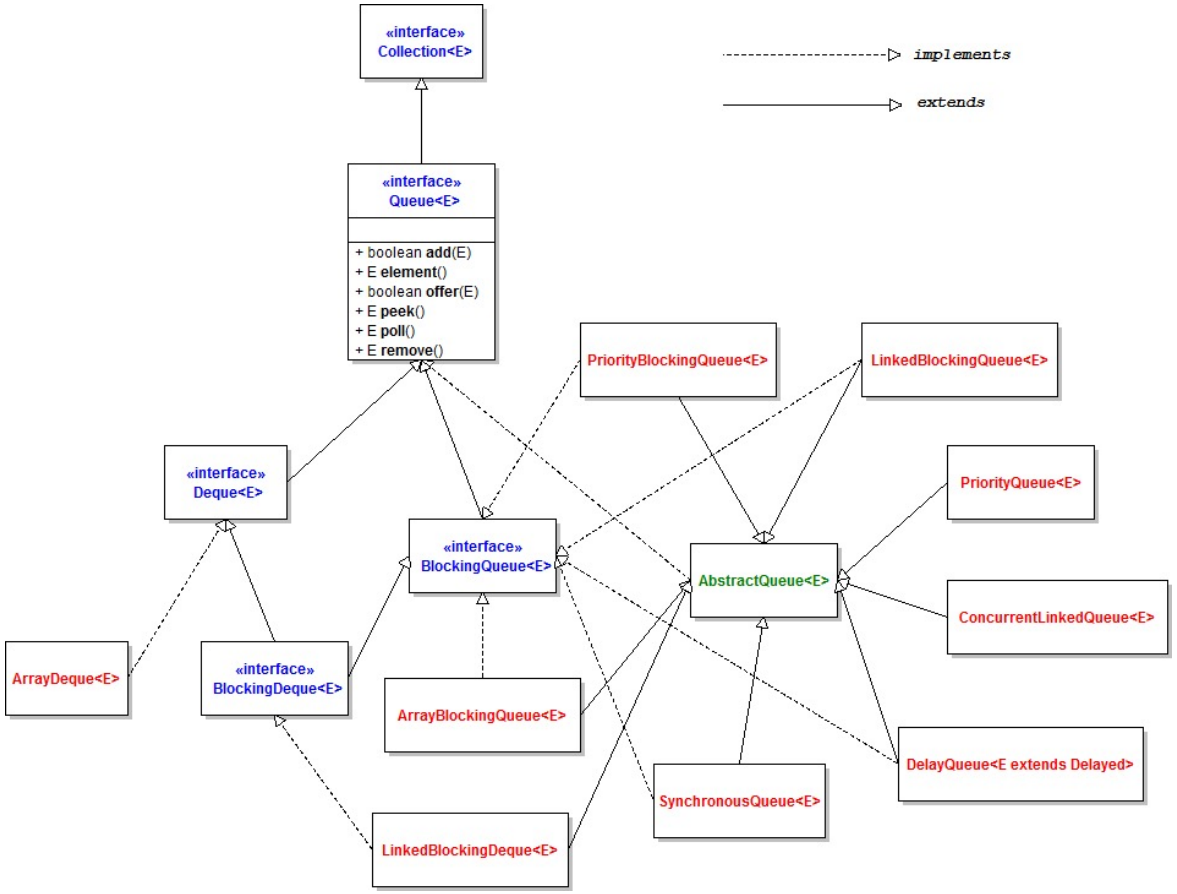
QUEUE

- Queue – a data collection that retrieves elements in the FIFO order (first in, first out)
  - Elements are stored in order of insertion, but don't have indexes
    - Client can only:
      - Add to the end of the queue,
      - Examine/remove the front of the queue

- Basic queue operations:
  - `Add (enqueue)` - add an element to the back of the queue
  - `Peek` - examine the front element
  - `Remove (dequeue)` - remove the front element



CLASS DIAGRAM OF THE QUEUE API



[Pictures credit: http://www.codejava.net/java-core/collections/class-diagram-of-queue-api]

JAVA INTERFACE QUEUE

`add(value)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; r eturns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

- Example:

```
Queue<Integer> myQueue = new LinkedList<Integer>();
myQueue.add(10);
myQueue.add(18);
myQueue.add(2017); // front [10, 18, 2017] back
System.out.println(myQueue.remove()); // 10
```

DEQUE

[Pictures credit: http://www.java2novice.com/data-structures-in-java/queue/double-ended-queue/]

DEQUE

	First Element (Head)	
	Throws exception	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`
Remove	`removeFirst()`	`pollFirst()`
Examine	`getFirst()`	`peekFirst()`

	Last Element (Tail)	
	Throws exception	*Special value*
Insert	`addLast(e)`	`offerLast(e)`
Remove	`removeLast()`	`pollLast()`
Examine	`getLast()`	`peekLast()`

[Pictures credit:https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

DEQUE

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

[Pictures credit:https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

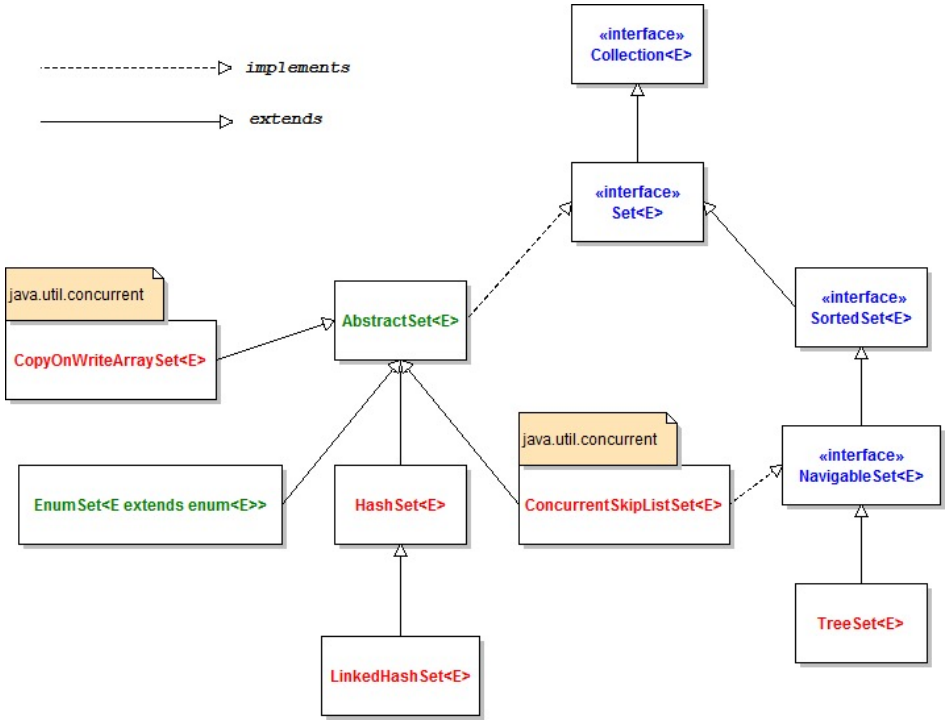
DEQUE

Stack Method	Equivalent Deque Method
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`

SET

- Set - a collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - `add`,
  - `remove`,
  - `search (contains)`
- We don't think of a set as having indexes; we just add things to the set in general and don't worry about order

SET API CLASS DIAGRAM



[Pictures credit:
http://www.codejava.net/images/articles/javacore/collections/Set%20API%20class%20diagram.png]

GENERICS

GENERICS

- Java is strongly typed and generally “type safe”, so unlike Python (which uses dynamic typing) Java uses the concept of generics to be able to reuse source (not .class/.obj) implementation across types
- Allows us to have compile-time safety and type replacement for similar algorithms
  - Parametric polymorphism rather than runtime polymorphism
- By using generics (called templates in other languages) we can have a List (or Set or whatever) of anything, by using a placeholder for an actual type used at compile-time
- We’ll cover this more next time!

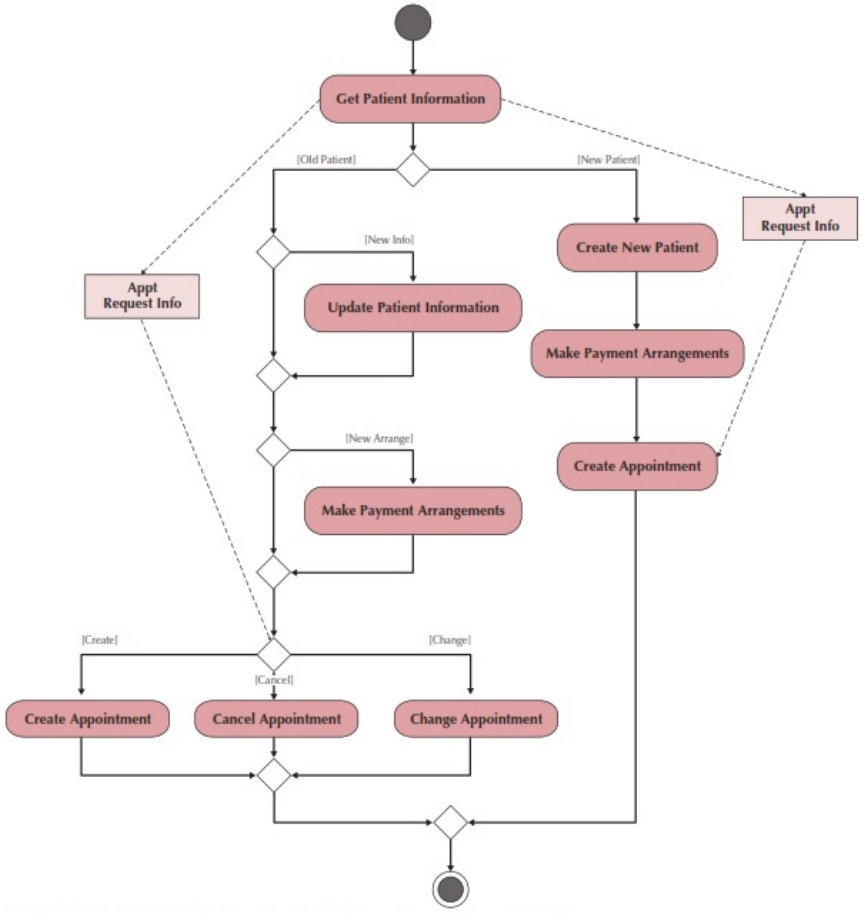
GENERICS

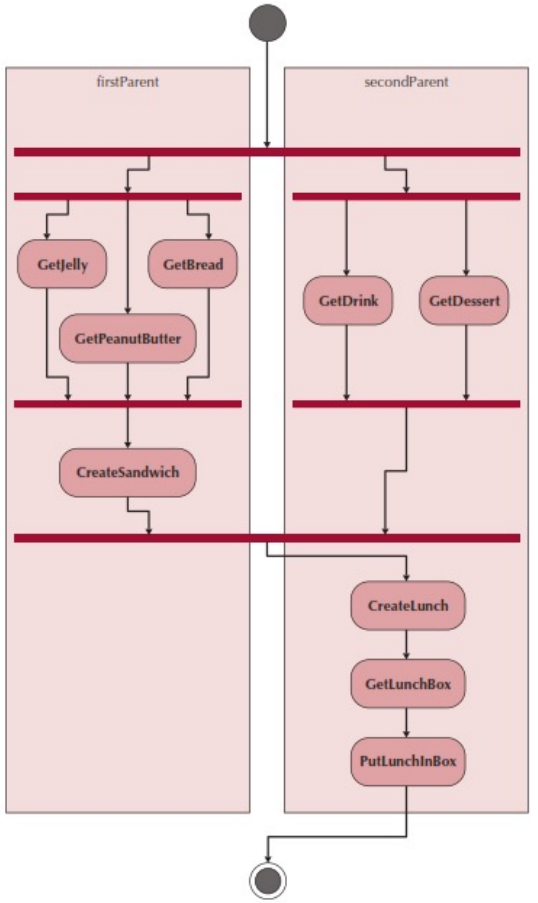
```java
// Example of Java Generics
// Placeholder T represents any type. Allows for compile time type-safe code
public class GenericsExample<T> {

  private T data;

  public T getData(){
    return this.data;
  }

  public void setData(T data){
    this.data=data;
  }

  public static void main(String [] args) {
    GenericsExample<Dog> d = new GenericsExample<Dog>();
    d.setData(new Dog( name: "Fifi",  isMale: false,  age: 12));
    System.out.println(d.getData().getName() + " " + d.getData().getAge());

    /* Two different types of containers even though it shares same implementat
       Cannot put a Person in a Dog container
    // d.setData(new Person("Fannie", 22)); // <-- Error!
    */

    GenericsExample<Person> p = new GenericsExample<Person>();
    p.setData(new Person( name: "Fannie", age: 22));
    System.out.println(p.getData().getName() + " " + p.getData().getAge());
```

Typically

- T is used as a placeholder for Type
- E as a placeholder for Element
- Etc.

A TOUCH OF UML: ACTIVITY DIAGRAMS





Activity Diagrams are high-level, business-oriented diagrams that outline large-grained process flow. Similar to other business modeling approaches, they can have swim lanes to partition responsibilities.

Activity diagrams are similar to old-school flowcharts, so some practitioners opt not to use them. They are, however, part of the UML, and some practitioners find them useful, particularly when interacting with non-technical subject matter experts.

From Dennis, Wixom, & Tegarden: Systems Analysis & Design with UML, 5th Ed.