COMP20010

Data Structures and Algorithms I

02 Data Abstraction

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Data Abstraction

Object Oriented Programming

- Programming in Java is largely based on building data types.
- This style of programming is known as *object-oriented programming*, as it revolves around the concept of an *object*, an entity that holds a data type value.
- With Java's primitive types we are largely confined to programs that operate on numbers, but with reference types we can write programs that operate on strings, pictures, sounds, or any of hundreds of other abstractions.
- Even more significant than libraries of predefined data types is that the range of data types available in Java programming is open-ended, because you can define your own data types.

Object Oriented Programming

Data types. A *data type* is a set of values and a set of operations on those values.

Abstract data types. An *abstract data type* is a data type whose internal representation is hidden from the client.

Objects. An *object* is an entity that can take on a data-type value. Objects are characterised by three essential properties:

The *state* of an object is a value from its data type;

the *identity* of an object distinguishes one object from another;

the *behaviour* of an object is the effect of data-type operations. In Java, a *reference* is a mechanism for accessing an object.

Object Oriented Programming

Applications programming interface (API). To specify the behaviour of an abstract data type, we use an *application programming interface* (API), which is a list of *constructors* and *instance methods* (operations), with an informal description of the effect of each:

`public class Point2D`	
`Point2D(float x, float y)`	*Create a 2D point object at position x,y*
`void move(float dx, float dy)`	*Move the position by dx, dy*
`void rotate(float angle)`	*Rotate by an angle*
`String toString`	*Convert to a string representation*

Object Oriented Programming

Client. A client is a program that uses a data type.

Implementation. An implementation is the code that implements the data type specified in an API.

Graphics

Spatial Data structures

Point2D

```
/**
 * Initializes a new point (x, y).
 *
 * @param x
 *          the x-coordinate
 * @param y
 *          the y-coordinate
 * @throws IllegalArgumentException
 *           if either {@code x} or {@code y} is {@code Double.NaN},
 *           {@code Double.POSITIVE_INFINITY} or
 *           {@code Double.NEGATIVE_INFINITY}
 */
public Point2D(double x, double y) {
        if (Double.isInfinite(x) || Double.isInfinite(y))
                throw new IllegalArgumentException("Coordinates must be finite");
        if (Double.isNaN(x) || Double.isNaN(y))
                throw new IllegalArgumentException("Coordinates cannot be NaN");
        if (x == 0.0)
                this.x = 0.0; // convert -0.0 to +0.0
        else
                this.x = x;

        if (y == 0.0)
                this.y = 0.0; // convert -0.0 to +0.0
        else
                this.y = y;
}
```

Clients **Interface** **Implementation**

Abstract Data Types

- mathematical models of a data structure
  - type of the data stored
  - operations supported on them
  - types of the parameters of the operations
- implementation details are hidden (encapsulation)
- A client does not need to know how a data type is implemented in order to be able to use it.

Abstract Data Types

- ADT abstracts from the organisation to the meaning of the data
- ADT abstracts from structure to use
- Representation does not matter
- The type is a set of operations
- Clients are forced to call operations to access the data

```
public class  Point2D
```

```
Point2D(float x, float y)
```

```
public class Point2D
```

```
Point2D(float r, float theta)
```

Abstract Data Types

- ADT abstracts from the organisation to the meaning of the data
- ADT abstracts from structure to use
- Representation does not matter
- The type is a set of operations
- Clients are forced to call operations to access the data

Classes are different

Both implement the Point2D concept

```
public class  Point2D
```

```
Point2D(float x, float y)
```

```
public class Point2D
```

```
Point2D(float r, float theta)
```

Abstract Data Types

We like to use ADT's because:

- Delay decisions
  - implementation details
- Fix bugs
  - can modify the underlying implementation without affecting clients
- Performance optimisations
  - can improve performance without requiring changes on the client side)

Example ADT

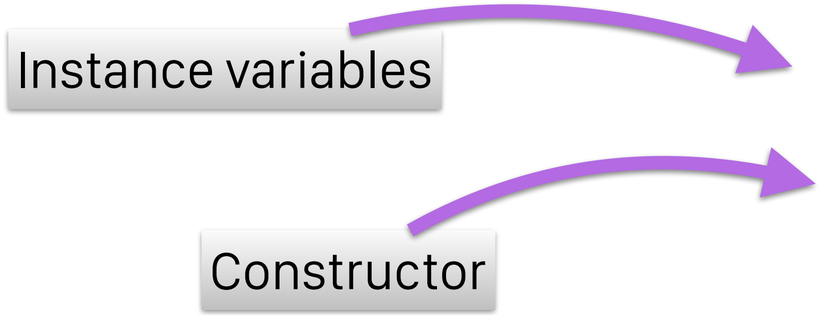
- The String type in Java is a useful ADT. String is an indexed sequence of char values, and has lots of useful methods:

public class	String
	String()
int	length()
int	charAt()
int	indexOf(String p)
String	concat(String p)
String	substring(int i, int j)
String[]	split(String delim)
int	compareTo(String p)
boolean	equals(String t)

Implementing ADT's

- We implement ADTs with a Java class
- The code in a file with the same name as the class, followed by the .java extension.
- The first statements in the file declare *instance variables* that define the data-type values.
- Following the instance variables are the *constructor* and the *instance methods* that implement operations on data-type values.

Implementing ADT's




```
public class Point2D {
    private float x;
    private float y;
    public Point2D(float x, float y) {
        this.x = x;
        this.y = y;
    }
    public void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
    String toString() {
        return "[" + x + ", " + y + "]";
    }

    public static void main(String [] args) {
        Point2D p = new Point2D(10.0, 20.0);
        System.out.println(p);
    }
}
```

Implementing ADT's

Instance variables.

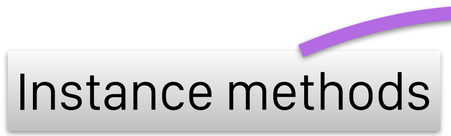
data-type values

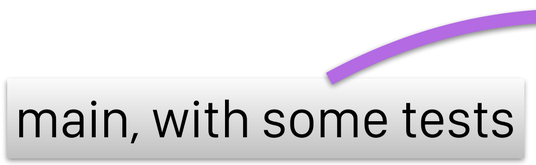
Each declaration is qualified by a *visibility modifier*.

In ADT implementations, we use `private`,

representation of an ADT is to be hidden from the client

use `final`, if the value is not to be changed once it is initialised.






```java
public class Point2D {
    private float x;
    private float y;
    public Point2D(float x, float y) {
        this.x = x;
        this.y = y;
    }
    public void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
    String toString() {
        return "[" + x + ", " + y + "]";
    }

    public static void main(String [] args) {
        Point2D p = new Point2D(10.0, 20.0);
        System.out.println(p);
    }
}
```

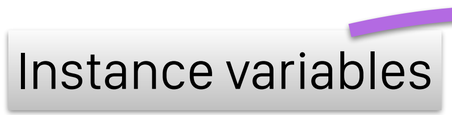
Implementing ADT's

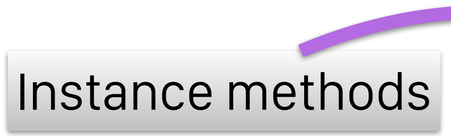
Constructors.

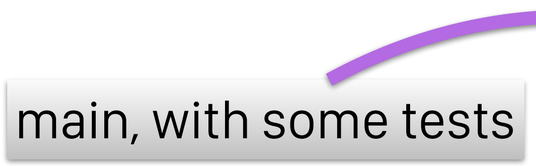
The constructor establishes an object's identity and initialises the instance variables.

We can overload the name and have multiple constructors with different signatures

If no other constructor is defined, a default no-argument constructor is implicit, has no arguments, and initialises instance values to default values.








```java
public class Point2D {
    private float x;
    private float y;
    public Point2D(float x, float y) {
        this.x = x;
        this.y = y;
    }
    public void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
    String toString() {
        return "[" + x + ", " + y + "]";
    }

    public static void main(String [] args) {
        Point2D p = new Point2D(10.0, 20.0);
        System.out.println(p);
    }
}
```

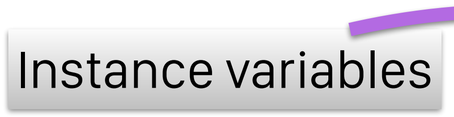
Implementing ADT's

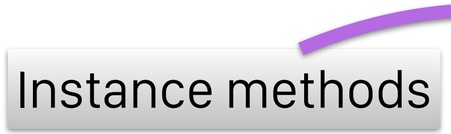
Instance methods.

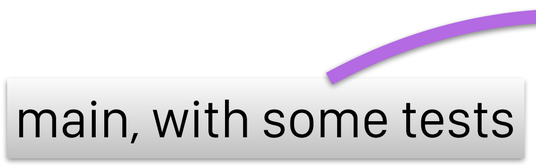
specify the data-type operations.

the *signature* specifies its name and the types and names of its parameter variables)

Instance methods may be *public* (specified in the API) or *private* (used to organise the computation and not available to clients).








```java
public class Point2D {
    private float x;
    private float y;
    public Point2D(float x, float y) {
        this.x = x;
        this.y = y;
    }
    public void move(float dx, float dy) {
        x += dx;
        y += dy;
    }
    String toString() {
        return "[" + x + ", " + y + "]";
    }

    public static void main(String [] args) {
        Point2D p = new Point2D(10.0, 20.0);
        System.out.println(p);
    }
}
```

Designing ADT's

Designing ADT's

- *Encapsulation.*
    - A hallmark of object-oriented programming is that it enables us to *encapsulate* data types within their implementations, to facilitate separate development of clients and data type implementations. Encapsulation enables modular programming.

Designing ADT's

Designing APIs.

Ideally, an API would clearly articulate behaviour for all possible inputs, including side effects, and then we would have software to check that implementations meet the specification, but this is usually impossible to achieve. There are numerous potential pitfalls when designing an API:

- Too hard to implement, making it difficult or impossible to develop.
- Too hard to use, leading to complicated client code.
- Too narrow, omitting methods that clients need.
- Too wide, including a large number of methods not needed by any client.
- Too general, providing no useful abstractions.
- Too specific, providing an abstraction so diffuse as to be useless.
- Too dependent on a particular representation, therefore not freeing client code from the details of the representation.

Designing ADT's

Designing APIs.

Ideally, an API would clearly articulate behaviour for all possible inputs, including side effects, and then we would have software to check that implementations meet the specification, but this is usually impossible to achieve. There are numerous potential pitfalls when designing an API:

- Too hard to implement, making it difficult or impossible to develop.
- Too hard to use, leading to complicated client code.
- Too narrow, omitting methods that clients need.
- Too wide, including a large number of methods not needed by any client.
- Too general, providing no useful abstractions.
- Too specific, providing an abstraction so diffuse as to be useless.
- Too dependent on a particular representation, therefore not freeing client code from the details of the representation.

provide to clients the methods they need and no others.

Design Patterns

- A standard solution to a common programming problem
  - a design or implementation structure that achieves a particular purpose
  - a high-level programming idiom
- A technique for making code more flexible
  - reduce coupling among program components
- Shorthand for describing program design
  - a description of connections among program components

Design Patterns - Encapsulation

- **Problem**: Exposed fields can be directly manipulated
  - Objects can be changed unexpectedly
  - Dependences prevent changing the implementation
- **Solution**: Hide some components
  - Permit only stylised access to the object
- **Disadvantages:**
  - Interface may not (efficiently) provide all desired operations – Indirection may reduce performance

Design Patterns - Encapsulation

- **Problem**: Exposed fields can be directly manipulated
  - Objects can be changed unexpectedly
  - Dependences prevent changing the implementation
- **Solution**: Hide some components
  - Permit only stylised access to the object
- **Disadvantages:**
  - Interface may not (efficiently) provide all desired operations – Indirection may reduce performance

- **Encapsulation**
- **Inheritance**
- **Encapsulation**
- **Iteration**
- **...**

Encapsulation

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

```
public final class Point2D {
    private double x, y;

    public Point2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double x() {
        return x;
    }

    public double y() {
        return y;
    }

    public double r() {
        return Math.sqrt(x*x + y*y);
    }

    public double theta() {
        return Math.atan2(y, x);
    }

    public static void main(String[] args) {
        int x0 = 1;
        int y0 = 2;
        Point2D p = new Point2D(x0, y0);
        System.out.println(p.x(), p.y(), p.r(), p.theta());
    }
}
```

Design Patterns - Inheritance (subclassing)

- **Problem**:
  - Repetition in implementations
  - Similar abstractions have similar members (fields, methods)
- **Solution**:
  - Inherit default members from a superclass
  - Implementation is determined at runtime
- **Disadvantages:**
  - Code for a class can be spread over multiple files
  - Runtime performance overhead of dispatching

```java
public interface List<E> {
    int size();
    boolean isEmpty();
    boolean add(E e);
    boolean remove(E e);
}
```

```java
public class LinkedList<E> implements List<E> {
    Node<E> start, end;

    public LinkedList<E>() {
    }

    public boolean add(E e) {
        // implementation
    }

    public boolean remove(E e) {
        // implementation
    }
}
```

Design Patterns - Inheritance (subclassing)

- **Problem**:
  - Repetition in implementations
  - Similar abstractions have similar members (fields, methods)
- **Solution**:
  - Inherit default members from a superclass
  - Implementation is determined at runtime
- **Disadvantages:**
  - Code for a class can be spread over multiple files
  - Runtime performance overhead of dispatching

base class

```
public interface List<E> {
    int size();
    boolean isEmpty();
    boolean add(E e);
    boolean remove(E e);
}
```

```
public class LinkedList<E> implements List<E> {
    Node<E> start, end;

    public LinkedList<E>() {
    }

    public boolean add(E e) {
        // implementation
    }

    public boolean remove(E e) {
        // implementation
    }
}
```

Design Patterns - Inheritance (subclassing)

- **Problem**:
  - Repetition in implementations
  - Similar abstractions have similar members (fields, methods)
- **Solution**:
  - Inherit default members from a superclass
  - Implementation is determined at runtime
- **Disadvantages:**
  - Code for a class can be spread over multiple files
  - Runtime performance overhead of dispatching

base class

```
public interface List<E> {
    int size();
    boolean isEmpty();
    boolean add(E e);
    boolean remove(E e);
}
```

sub-class

```
public class LinkedList<E> implements List<E> {
    Node<E> start, end;

    public LinkedList<E>() {
    }

    public boolean add(E e) {
        // implementation
    }

    public boolean remove(E e) {
        // implementation
    }
}
```

Design Patterns - Iteration

Problem:

- To access all members of a collection, we must perform a specialised traversal for each data structure
- Undesirable dependencies
- Does not generalise well to other collections

Solution:

- Allow the implementation to perform the traversal, and do the bookkeeping
- Communicate the result to the clients

Disadvantages:

- Iteration order is implementation dependent and not controlled by client

```
Iterator it = coll.iterator();

while(it.hasNext()) {
    System.out.print(itr.next() + " ");
}
```

```
// Returns true if there are more elements.
// Otherwise, returns false.
boolean hasNext();

// Returns the next element.
Object next();

// Removes the current element.
void remove();
```

Design Patterns - Iteration

to get this behaviour...

Problem:

- To access all members of a collection, we must perform a specialised traversal for each data structure
- Undesirable dependencies
- Does not generalise well to other collections

Solution:

- Allow the implementation to perform the traversal, and do the bookkeeping
- Communicate the result to the clients

Disadvantages:

- Iteration order is implementation dependent and not controlled by client

```
Iterator it = coll.iterator();

while(it.hasNext()) {
    System.out.print(itr.next() + " ");
}
```

```
// Returns true if there are more elements.
// Otherwise, returns false.
boolean hasNext();

// Returns the next element.
Object next();

// Removes the current element.
void remove();
```

Design Patterns - Iteration

Problem:

- To access all members of a collection, we must perform a specialised traversal for each data structure
- Undesirable dependencies
- Does not generalise well to other collections

Solution:

- Allow the implementation to perform the traversal, and do the bookkeeping
- Communicate the result to the clients

Disadvantages:

- Iteration order is implementation dependent and not controlled by client

to get this behaviour...

```
Iterator it = coll.iterator();

while(it.hasNext()) {
    System.out.print(itr.next() + " ");
}
```

implement the Iterator pattern in our object

```
// Returns true if there are more elements.
// Otherwise, returns false.
boolean hasNext();

// Returns the next element.
Object next();

// Removes the current element.
void remove();
```

Design Patterns - Exceptions

Problem:

- Errors in one part of the code should be handled in another part
- Code should not be cluttered with error-handling routines

Solution:

- Use language facilities for throwing, catching exceptions

```
/**
 * Returns the first element in this list.
 *
 * @return the first element in this list
 * @throws NoSuchElementException if this list is empty
 */
public E getFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return f.item;
}
```

Disadvantages:

- Code may still be cluttered
- It can be hard to decide where the exception should be handled
- Using exceptions for normal control flow can be confusing and inefficient

Design Patterns - Exceptions

Problem:

- Errors in one part of the code should be handled in another part
- Code should not be cluttered with error-handling routines

Solution:

- Use language facilities for throwing, catching exceptions

```
/**
 * Returns the first element in this list.
 *
 * @return the first element in this list
 * @throws NoSuchElementException if this list is empty
 */
public E getFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return f.item;
}
```

the exception is thrown but the caller is responsible for doing something about it...

Disadvantages:

- Code may still be cluttered
- It can be hard to decide where the exception should be handled
- Using exceptions for normal control flow can be confusing and inefficient

Generics

Generics

- When using a collection (e.g., LinkedList, HashSet, HashMap), we generally have a single type *T* of elements that we store in it (e.g., Integer, String)
- Before Java 5, when extracting an element, had to cast it to *T* before we could invoke T's methods
- Compiler could not check that the cast was correct at compile-time, since it didn't know what *T* was
- Inconvenient and unsafe, could fail at runtime
- Generics in Java provide a way to communicate the type of elements in a collection, to the compiler
- Compiler can check that you have used the collection consistently
- Result: safer and more-efficient code

Generics

- Without generics, the collections hold objects and are not type safe
- The compiler can add any type of *object*
- Boxing/unboxing are necessary

```
Vector vector = new Vector(10);
for(int i = 0; i < 10; ++i) {
    vector.add(new String("xyz"));
}

String s = (String) vector.elementAt(0);
```

unboxing to convert from Object to String

Generics

- Without generics, the collections hold objects and are not type safe
- The compiler can add any type of *object*
- Boxing/unboxing are necessary

```
Vector vector = new Vector(10);
for(int i = 0; i < 10; ++i) {
    vector.add(new String("xyz"));
}

String s = (String) vector.elementAt(0);
```

unboxing to convert from Object to String

- With generics we specify the type of object to store in the collection
- boxing/unboxing is done automatically by the compiler

```
Vector<String> vector = new Vector<String>(10);
for(int i = 0; i < 10; ++i) {
    vector.add(new String("xyz"));
}

String s = vector.elementAt(0);
```

Generics: Autoboxing

- Instead of:

```
Vector<Integer> vector = new Vector<Integer>(10);
vector.put(new Integer(10));
sum += vector.elementAt(0).intValue();
```

Generics: Autoboxing

- Instead of:

```
Vector<Integer> vector = new Vector<Integer>(10);
vector.put(new Integer(10));
sum += vector.elementAt(0).intValue();
```

- autoboxing/ unboxing converts from “int” to “Integer”, float, double, byte, etc

```
Vector<Integer> vector = new Vector<Integer>(10);
vector.put(10);
sum += vector.elementAt(0);
```

Generics

- <T> is read: “of T”
- Vector<Integer> : “vector of Integer”
- The type annotation informs the compiler that all extractions from this collection should be automatically cast to T
- Specify the type in the declaration, checked at compile time

Generics

- Declaring Collection<String> c tells us something about the variable c (i.e., c holds only Strings)
- This is true wherever c is used
- The compiler checks this and won't compile code that violates this
- Without use of generic types, explicit casting must be used
- A cast tells us something the programmer thinks is true at a single point in the code
- The Java virtual machine checks this only at runtime

Generics

- All occurrences of formal type parameter T are replaced with the actual type (eg. “Integer”)

```
public interface List<T> { // T is a type variable
    void add(T x);
    Iterator<T> iterator();
}

public interface Iterator<T> {
    T next();
    boolean hasNext();
    void remove();
}

List<Integer> ll = new List<Integer>();
```

Generics - Wildcards

Without generics

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}
```

With generics

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Wildcards

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Generics

- Usually wildcards are bounded
- Specify exactly what types are required
- For sort, we need to able to compare the elements, so the type must implement the Comparable interface

```
static void sort(List<? extends Comparable> c) {
    // TODO
}
```

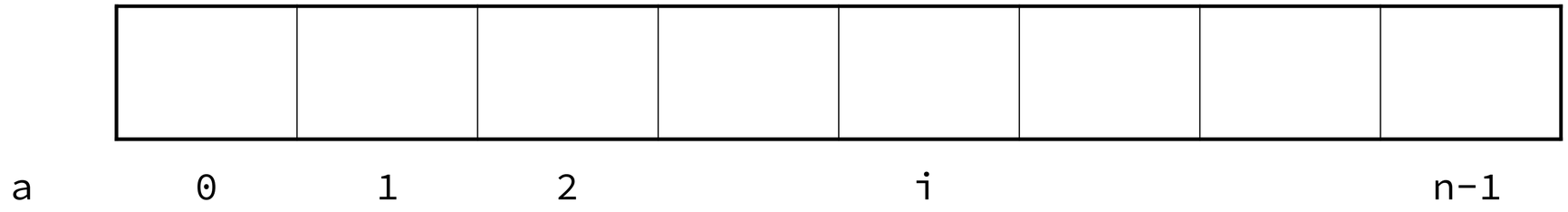
Generics

- java.lang.Comparable<T>
- public int compareTo(T x);
  - Returns a value (< 0), (= 0), or (> 0)
    - (< 0) implies this is before x
    - (= 0) implies this.equals(x) is true
    - (> 0) implies this is after x
- Many classes implement Comparable
- String, Double, Integer, Char, java.util.Date,…
- If a class implements Comparable then that is considered to be the class's natural ordering

Arrays

Array Definition

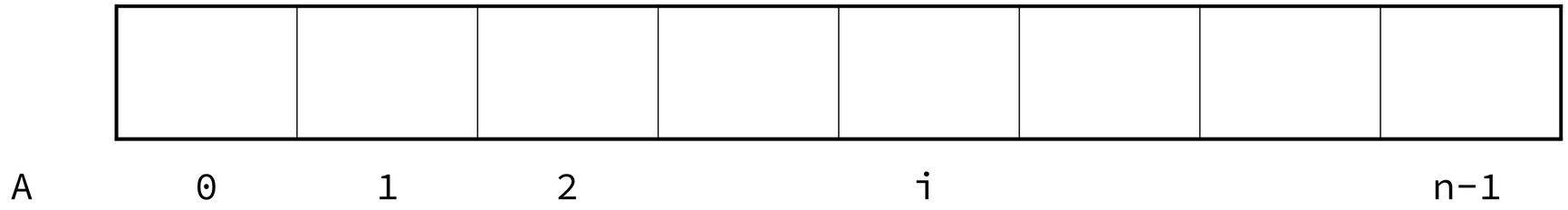
- An **array** is a sequenced collection of variables all of the same type.
- Each variable, or **cell**, in an array has an **index**, which uniquely refers to the value stored in that cell.
- The cells of an array, **a**, are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an **element** of that array.



Array Length and Capacity

Since the length of an array determines the maximum number of things that can be stored in the array, we will refer to the length of an array as its **capacity**.

In Java, the length of an array named **a** can be accessed using the syntax **a**.length. Thus, the cells of an array, **a**, are numbered 0, 1, 2, and so on, up through a.length–1, and the cell with index k can be accessed with syntax a[k].



Declaring Arrays

The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

```
elementType[] arrayName = {initialValue0, initialValue1, …, initialValuen-1};
```

The elementType can be any Java base type or class name, and arrayName can be any valid Java identifier. The initial values must be of the same type as the array.

Declaring Arrays

- The second way to create an array is to use the **new** operator.
  - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:

```
elementType[] arrayName = new elementType[n];
```

n is a positive integer denoting the length of the new array.

The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable.

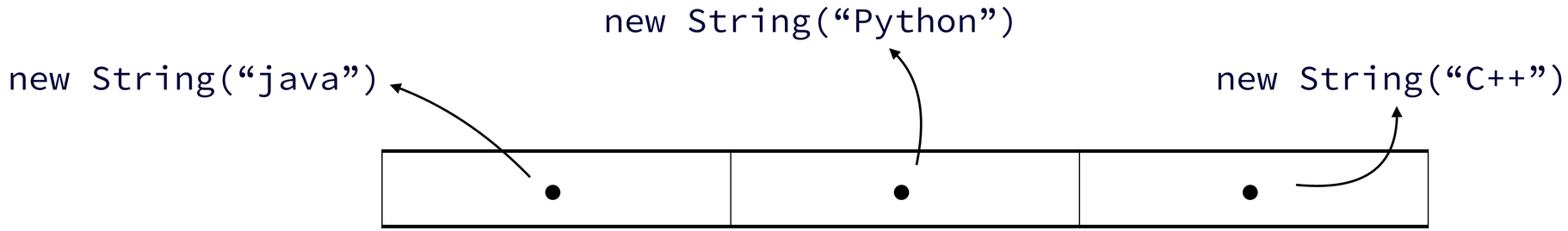
Arrays

- Arrays can be used to store primitives
  - boolean, byte, char, short, int, long, float, double

1	0	0	1	0

Or references to objects:

```
String[] s = new String{“java”, “C++”, “Python”};
```



Multidimensional Arrays

- In Java, the syntax for two-dimensional arrays is similar to the syntax for one-dimensional arrays, except that an extra index is involved:

```
int[][] a = new int[10][10];

int[][] b = { { 1, 2, 3 }, { 4, 5, 6, 9 }, { 7 }, };
```

Direct initialisation

- Java does not actually have two-dimensional arrays
- The elements in a 2D array of type `int[][]` are variables of type `int[]`.
- A variable of type `int[]` can only hold a pointer to an array of `int`. So, a 2D array is really an array of pointers, where each pointer can refer to a one-dimensional array.

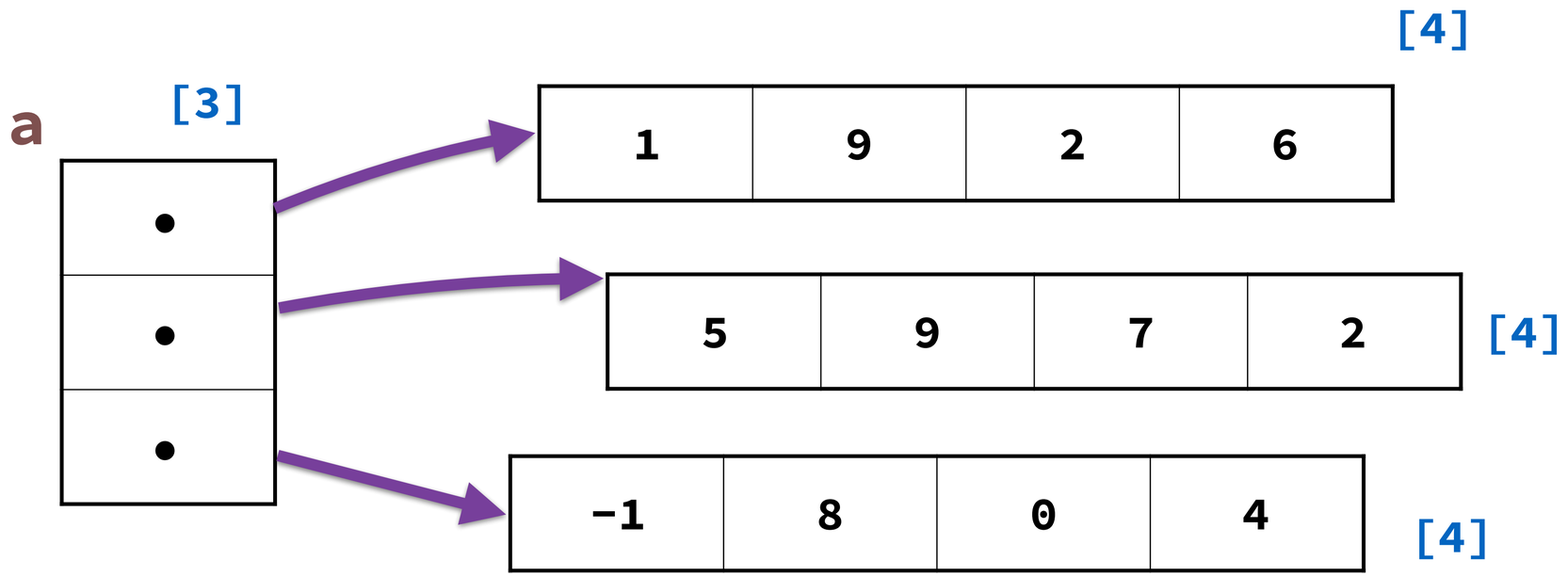
Multidimensional Arrays

rows

columns

```
int[][] a = new int[3][4];
```

1	9	2	6
5	9	7	2
-1	8	0	4



Multidimensional Arrays

- Higher dimensional arrays:

```
int[][][] a = new int[10][10][10];
```

- Straightforward extention of 2D syntax

java.util.Arrays

- The Arrays class of the java.util package contains several static methods that we can use to fill, sort, search, etc in arrays. This class is a member of the Java Collections Framework and is present in java.util.arrays. Sample of methods:

public static String toString(int[] a)	The string representation consists of a list of the array's elements,
public static void sort(int[] a)	Sorts the specified array into ascending numerical order.
public static int[] copyOf(int[] original, int newLength)	Copies the specified array and length. It truncates the array if provided length is smaller and pads if provided
public static void fill(int[] a, int val)	Fills all elements of the specified array with the specified value.

Array Examples

- Creating arrays
- Different object types
- Iterating over arrays
- Passing arrays as arguments to functions

List ADT

List ADT

- The List interface has the following methods:

size()	Returns the number of elements in the list
isEmpty()	Returns a boolean indicating whether the list is empty
get(i)	Returns the element of the list with index I, error condition occurs if i is outside the range [0, size()-1]
set(i, e)	Replaces the element at index i with e, and returns the old element that was replaced; an error occurs if i is not in range [0, size()-1]
add(i, e)	Inserts a new element e into the list at index i, moving all subsequent elements one index later in the list; an error occurs if i is not in range [0, size()-1]
remove(i)	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error occurs if i is not in range [0, size()-1]

List ADT

- Example sequence of List operations:

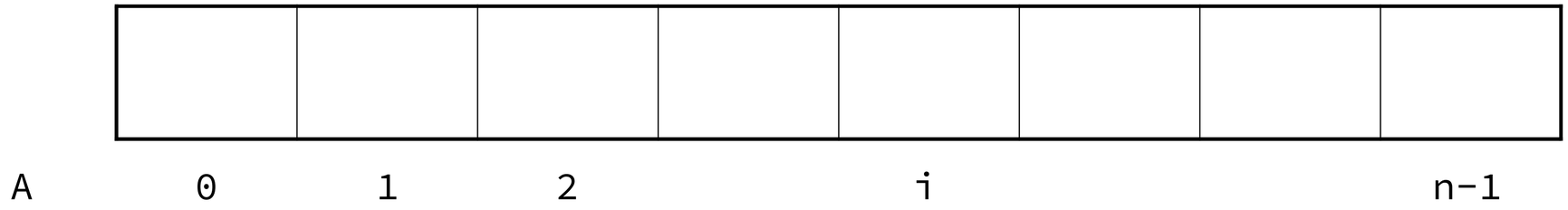
Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	“error”	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	“error”	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	“error”	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

List ADT

- How would we implement the List ADT?

Array List

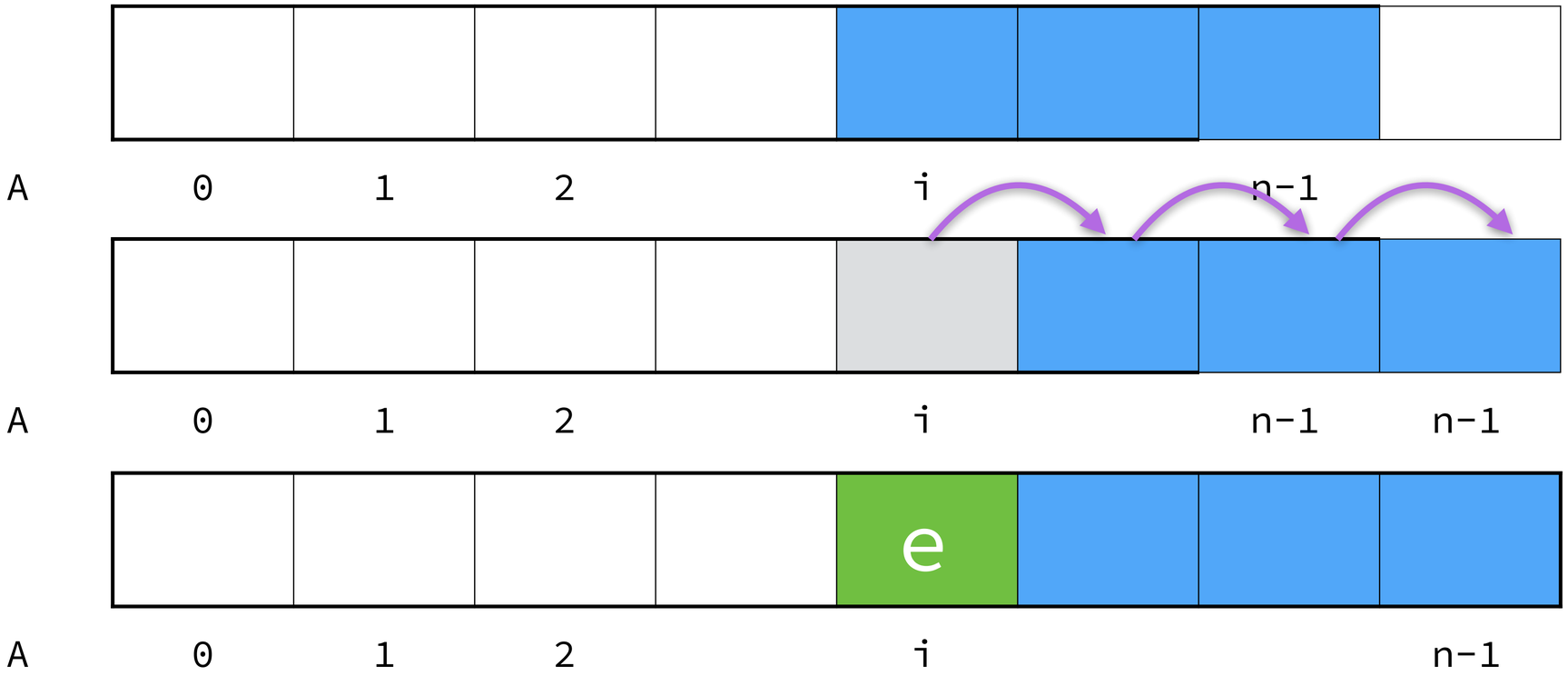
- An obvious choice for implementing the list ADT is to use an array, **A**, where **A[i]** stores (a reference to) the element with index **i**.
- With a representation based on an array **A**, the get(**i**) and set(**i**, **e**) methods are easy to implement by accessing **A[i]** (assuming **i** is a legitimate index).



Array List - Insertion

In an operation add(i, e), we need to make room for the new element by shifting forward the n - i elements A[i], …, A[n - 1]

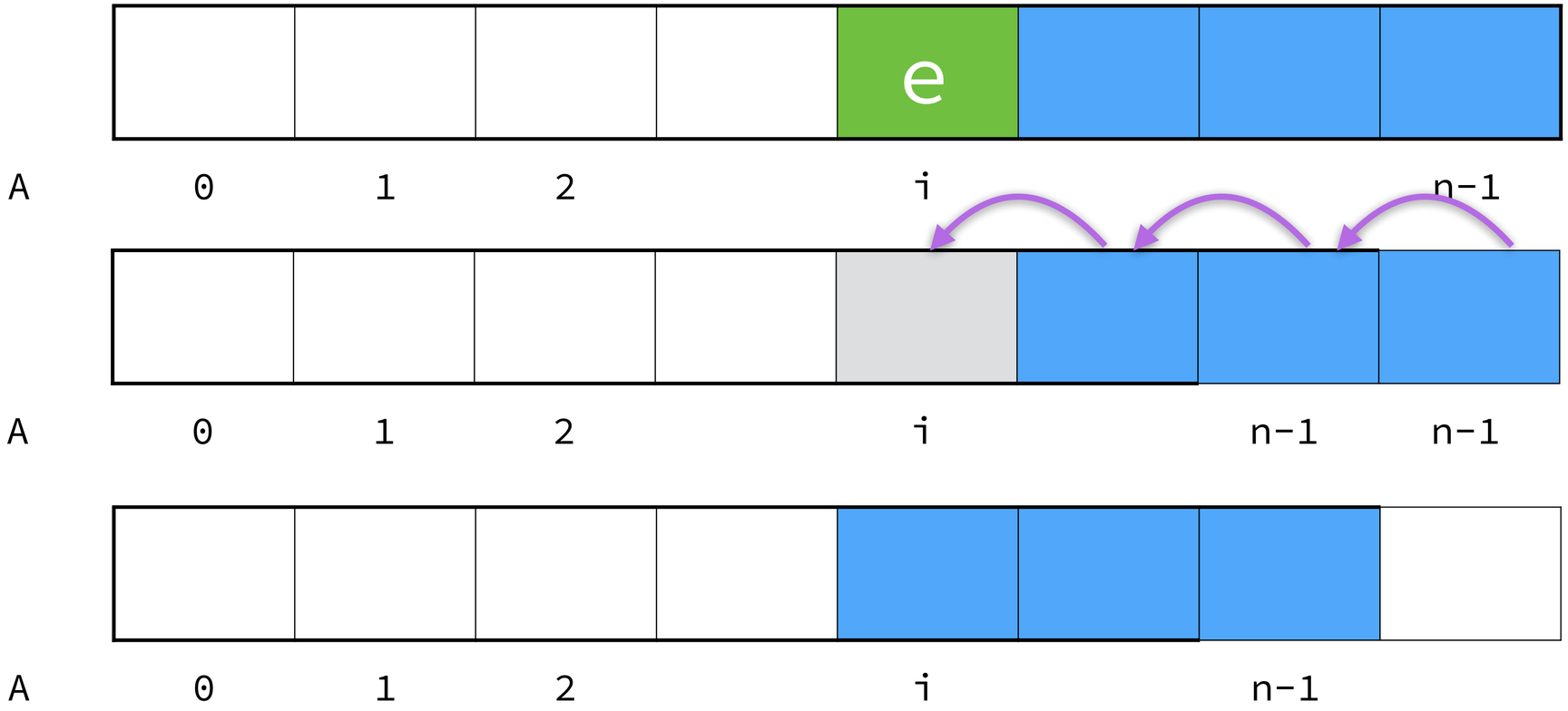
In the worst case (i = 0), this takes O(n) time



Array List - Removal

In an operation remove(i), we need to fill the space left by the removed element by shifting backwards the n - i -1 elements A[i+1], …, A[n - 1]

In the worst case (i = 0), this takes O(n) time



ArrayList Implementation

Java ArrayList

- Let’s take quick browse through the source code of ArrayList in the JDK
- This is likely some of the most executed code in the JDK and is behind some of the key data structures.
- Look for how the class uses encapsulation, exceptions, iteration
- Pay attention to the style, how the error conditions are handled, cases are carefully checked and care taken not to allow the data structure get into an inconsistent state

ArrayList.java

check out the ArrayList.java source code

```
public class ArrayList<E> implements List<E> {
    // instance variables
    /** Default array capacity. */
    public static final int CAPACITY=16;      // default array capacity

    /** Generic array used for storage of list elements. */
    private E[] data;                         // generic array used for storage

    /** Current number of elements in the list. */
    private int size = 0;                     // current number of elements

    // constructors
    /** Creates an array list with default initial capacity. */
    public ArrayList() { this(CAPACITY); }    // constructs list with default capacity

    /** Creates an array list with given initial capacity. */
    @SuppressWarnings({"unchecked"})
    public ArrayList(int capacity) {          // constructs list with given capacity
        data = (E[]) new Object[capacity];      // safe cast; compiler may give warning
    }
}
```

ArrayList.java

```java
// public methods
/**
 * Returns the number of elements in the list.
 * @return number of elements in the list
 */
public int size() { return size; }

/**
 * Tests whether the array list is empty.
 * @return true if the array list is empty, false otherwise
 */
public boolean isEmpty() { return size == 0; }
```

ArrayList.java

```java
/**
 * Returns (but does not remove) the element at index i.
 * @param  i   the index of the element to return
 * @return the element at the specified index
 * @throws IndexOutOfBoundsException if the index is negative or greater than size()-1
 */
public E get(int i) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    return data[i];
}

/**
 * Replaces the element at the specified index, and returns the element previously stored.
 * @param  i   the index of the element to replace
 * @param  e   the new element to be stored
 * @return the previously stored element
 * @throws IndexOutOfBoundsException if the index is negative or greater than size()-1
 */
public E set(int i, E e) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    E temp = data[i];
    data[i] = e;
    return temp;
}
```

ArrayList.java

```
/**
 * Inserts the given element at the specified index of the list, shifting all
 * subsequent elements in the list one position further to make room.
 * @param  i   the index at which the new element should be stored
 * @param  e   the new element to be stored
 * @throws IndexOutOfBoundsException if the index is negative or greater than size()
 */
public void add(int i, E e) throws IndexOutOfBoundsException {
    checkIndex(i, size + 1);
    if (size == data.length)                // not enough capacity
        resize(2 * data.length);              // so double the current capacity
    for (int k=size-1; k >= i; k--)         // start by shifting rightmost
        data[k+1] = data[k];
    data[i] = e;                            // ready to place the new element
    size++;
}

/** Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
    E[] temp = (E[]) new Object[capacity];     // safe cast; compiler may give warning
    for (int k=0; k < size; k++)
        temp[k] = data[k];
    data = temp;                               // start using the new array
}
```

ArrayList.java

```
/**
 * Removes and returns the element at the given index, shifting all subsequent
 * elements in the list one position closer to the front.
 * @param  i   the index of the element to be removed
 * @return the element that had be stored at the given index
 * @throws IndexOutOfBoundsException if the index is negative or greater than size()
 */
public E remove(int i) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    E temp = data[i];
    for (int k=i; k < size-1; k++)          // shift elements to fill hole
        data[k] = data[k+1];
    data[size-1] = null;                    // help garbage collection
    size--;
    return temp;
}
```

Practical 1

School of Computer Science

COMP20010: Data Structures and Algorithms I

Semester I

Practical 1
On: 23/10/2018
Due: Not For Assessment

Q1: _____ **(5points)**

What does the following code do?

```
int[] a = new int[5];
System.out.println(a);
```

Q2: _____ **(5points)**

Write your own `toString(int [] a)` method which returns a more useful string representation of an array, and test it works.

```
public static String toString(int [] a) {
    // TODO
}
int[] a = new int[5];
System.out.println(toString(a));
```

Q3: _____ **(10points)**

Create a class `ArrayEquals` which checks two 1 dimensional arrays for equality. The class should have the following methods:

```
public class ArrayEquals {

    // return true if two integer arrays have same length and all
    // corresponding pairs of integers are equal
    public static boolean eq(int[] a, int[] b) {
        // TODO
    }


    // test client
    public static void main(String[] args) {
        int[] a = { 3, 1, 4, 1, 5 };
        int[] b = { 3, 1, 4, 1 };
        int[] c = { 3, 1, 4, 1, 5 };
        int[] d = { 2, 7, 1, 8, 2 };

        System.out.println(eq(a, a));
        System.out.println(eq(a, b));
        System.out.println(eq(a, c));
        System.out.println(eq(a, d));
    }
}
```

Q4: _____ **(5points)**

Write a function which copies an array by iteration and returns the copy:

```
public class Exercise {
 public static int[] copyArray(int [] a) {
    // TODO
 }

 public static void main(String[] args) {
   int[] a = {56, 14, -46, 15, 36, 99, 77, 18, 29, 49};

   int[] b = copyArray(a);
   // check its not a clone
   a[0] = -1;
   System.out.println(toString(b)); // print a string representation of the array
  }
 }
```

Q5: _____ **(5points)**

Write a Java statement which creates an array with 3 rows and 5 columns?

Q6: _____ **(5points)**

Write a Java function which prints a representation of a 2 dimensional array?

```
public class Exercise {
    public static String toString(int [][] a) {
        // TODO
    }
    public static void main(String [] args) {
        int [][] a = new int[5][5];
        System.out.println(toString(a));
    }
}
```

Q7: _____ **(5points)**

Is Java pass-by-reference or pass-by-value? What does the following code do? Is it what you expect?

```java
public class ExerciseArrayIncrement {
    public static void increment(int [] a) {
        for (int i = 0; i < a.length ; i++) {
            a[i] += 1;
        }

    }
    public static void main(String[] args) {
        int N = 10;
        int[] a = new int[N];
        Random random = new Random();
        for(int i = 0; i < N; i++) {
            a[i] = random.nextInt(100);
        }
        System.out.println("Original array : " + Arrays.toString(a));
        increment(a);
        System.out.println("Incremented array : " + Arrays.toString(a)↩
            );
    }
}
```

Q8: _____ **(5points)**

Write a function which takes an array as an argument and reverses the order of the elements of the array.

```java
public class ExerciseArrayIncrement {
    public static void reverse(int [] a) {
        // TODO
    }
    public static void main(String[] args) {
        int N = 10;
        int[] a = new int[N];
        Random random = new Random();
        for(int i = 0; i < N; i++) {
            a[i] = random.nextInt(100);
        }
        System.out.println("Original array : " + Arrays.toString(a));
        reverse(a);
        System.out.println("Reversed array : " + Arrays.toString(a));
    }
}
```

Q9: _____ **(10points)**

Write a Java program to move all 0's to the end of an array. Maintain the relative order of the other (non-zero) array elements:

```java

public class Exercise {
   public static void moveZeros(int [] a) {
       // TODO
   }

    public static void main(String[] args) throws Exception {
       int[] a = {0,0,12,0,2,0,0,0,5,0,8};
        int i = 0;
       System.out.print("Original array: " + Arrays.toString(a));

       moveZeros(a);

       System.out.print("After moving 0: " + Arrays.toString(a));
   }
}
```

```
Original array: [0, 0, 12, 0, 2, 0, 0, 0, 5, 0, 8]
After moving 0: [12, 2, 5, 8, 0, 0, 0, 0, 0, 0, 0]
```