ARRAYS

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.
Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes 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 $a[k]$. 
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:

\[
\text{elementType[]} \text{arrayName} = \{\text{initialValue}_0, \text{initialValue}_1, \ldots, \text{initialValue}_{N-1}\};
\]

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.
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:

```java
new elementType[length]
```

`length` 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 of Characters or Object References

- An array can store primitive elements, such as characters.

- An array can also store references to objects.
A game entry stores the name of a player and her best score so far in a game

```java
public class GameEntry {
    private String name; // name of the person earning this score
    private int score; // the score value

    /** Constructs a game entry with given parameters. */
    public GameEntry(String n, int s) {
        name = n;
        score = s;
    }

    /** Returns the name field. */
    public String getName() { return name; }

    /** Returns the score field. */
    public int getScore() { return score; }

    /** Returns a string representation of this entry. */
    public String toString() {
        return "(" + name + ", " + score + ")";
    }
}
```
JAVA EXAMPLE: SCOREBOARD

- Keep track of players and their best scores in an array, board
  - The elements of board are objects of class GameEntry

```java
/** Class for storing high scores in an array in nondecreasing order. */
public class Scoreboard {
    private int numEntries = 0; // number of actual entries
    private GameEntry[] board; // array of game entries (names & scores)
    /** Constructs an empty scoreboard with the given capacity for storing entries. */
    public Scoreboard(int capacity) {
        board = new GameEntry[capacity];
    }
    ... // more methods will go here
}
```
To add an entry $e$ into array $board$ at index $i$, we need to make room for it by shifting forward the $n - i$ entries $board[i], \ldots, board[n-1]$.
** Attempt to add a new score to the collection (if it is high enough) */

```java
public void add(GameEntry e) {
    int newScore = e.getScore();
    // is the new entry e really a high score?
    if (numEntries < board.length || newScore > board[numEntries-1].getScore()) {
        if (numEntries < board.length) // no score drops from the board
            numEntries++;
        // so overall number increases
        // shift any lower scores rightward to make room for the new entry
        int j = numEntries - 1;
        while (j > 0 && board[j-1].getScore() < newScore) {
            board[j] = board[j-1]; // shift entry from j-1 to j
            j--;
        // and decrement j
        }
        board[j] = e; // when done, add new entry
    }
}
```
To remove the entry \( e \) at index \( i \), we need to fill the hole left by \( e \) by shifting backward the \( n - i - 1 \) elements \( board[i + 1], \ldots, board[n - 1] \).
/** Remove and return the high score at index i. */

public GameEntry remove(int i) throws IndexOutOfBoundsException {
    if (i < 0 || i >= numEntries)
        throw new IndexOutOfBoundsException("Invalid index: " + i);

    GameEntry temp = board[i];
    // save the object to be removed

    for (int j = i; j < numEntries - 1; j++)
        // count up from i (not down)
        board[j] = board[j+1];

    board[numEntries - 1] = null;
    // null out the old last score

    numEntries--;  // return the removed object

    return temp;
}
INSERTION-SORT ALGORITHM

Algorithm InsertionSort(A):

*Input:* An array A of n comparable elements

*Output:* The array A with elements rearranged in nondecreasing order

for k from 1 to n − 1 do

Insert A[k] at its proper location within A[0], A[1], ..., A[k].

*Code Fragment 3.5:* High-level description of the insertion-sort algorithm.

The algorithm proceeds by considering one element at a time, placing the element in the correct order relative to those before it.
Insertion-Sort Java Example:

```java
public class InsertionSort {

    /** Insertion-sort of an array of characters into nondecreasing order */
    public static void insertionSort(char[] data) {
        int n = data.length;
        for (int k = 1; k < n; k++) {
            char cur = data[k];
            int j = k;
            while (j > 0 && data[j-1] > cur) {
                data[j] = data[j-1];
                j--;
            }
            data[j] = cur;
        }
    }

    public static void main(String[] args) {
        char[] a = {'B', 'C', 'D', 'A', 'E', 'H', 'G', 'F'};
        System.out.println(java.util.Arrays.toString(a));
        insertionSort(a);
        System.out.println(java.util.Arrays.toString(a));
    }
}
```
Array-of-arrays: we can define a two-dimensional array to be an array with each of its cells being another array.

```java
int[ ][ ] data = new int[8][10];
```
**EXAMPLES WITH ARRAYS**

```java
int [] A;

int [] A = new int [4];

int [][] A = new int[4][3];

int [][] A = new int[4][];
A[1] = new int[4];
```

**RESOURCE:**
int [] A = new int[4];
int [] B = {0,1,2,3,4,5,6,7,8,9};
System.arraycopy(B,2,A,0,4);

int [] A = new int[4];
int [] B = {2,3,4};
System.arraycopy(B,0,A,0,4);

int [] A = {1,1,1,1};
int [] B = {2,2,2};
System.arraycopy(A,0,B,1,2);
System.arraycopy(B,0,A,0,3);

RESOURCE:
int [] A = new int[4];
int [] B = {0,1,2,3,4,5,6,7,8,9};
System.arraycopy(B,0,A[0],0,3);
System.arraycopy(B,1,A[1],0,3);
System.arraycopy(B,2,A[2],0,3);
System.arraycopy(B,3,A[3],0,3);

int [][] A = new int[4][3];
int [] B = {1,2,3,4,5,6,7,8,9,10};
System.arraycopy(B,0,A[0][0],0,3);
System.arraycopy(B,1,A[1][0],0,3);
System.arraycopy(B,2,A[2][0],0,3);
System.arraycopy(B,3,A[3][0],0,3);
Choose to make the cells in the board array be integers, with a 0 indicating an empty cell, a 1 indicating an X, and a –1 indicating an O.

Given board configuration, X or O wins, if the values of a row, column, or diagonal add up to 3 or –3, respectively.
public class TicTacToe {
    public static final int X = 1, O = -1;         // players
    public static final int EMPTY = 0;            // empty cell
    private int board[][] = new int[3][3];        // game board
    private int player;                           // current player

    /** Constructor */
    public TicTacToe() { clearBoard(); }

    /** Clears the board */
    public void clearBoard() {
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                board[i][j] = EMPTY;                         // every cell should be empty
        player = X;                                    // the first player is 'X'
    }

    /** Puts an X or O mark at position i,j. */
    public void putMark(int i, int j) throws IllegalArgumentException {
        if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
            throw new IllegalArgumentException("Invalid board position");
        if (board[i][j] != EMPTY)
            throw new IllegalArgumentException("Board position occupied");
        board[i][j] = player;                           // place the mark for the current player
        player = - player;                             // switch players (uses fact that O = - X)
    }

    /** Checks whether the board configuration is a win for the given player. */
    public boolean isWin(int mark) {
        return ((board[0][0] + board[0][1] + board[0][2] == mark*3)     // row 0
            || (board[1][0] + board[1][1] + board[1][2] == mark*3)     // row 1
    }
}
/** Returns the winning player's code, or 0 to indicate a tie (or unfinished game). */
public int winner() {
    if (isWin(X))
        return(X);
    else if (isWin(O))
        return(O);
    else
        return(0);
}

/** Returns a simple character string showing the current board. */
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++) {
            switch (board[i][j]) {
                case X: sb.append("X"); break;
                case O: sb.append("O"); break;
                case EMPTY: sb.append(" "); break;
            }
            if (j < 2) sb.append("|"); // column boundary
        }
        if (i < 2) sb.append("\n-----\n"); // row boundary
    }
    return sb.toString();
}

/** Test run of a simple game */
public static void main(String[ ] args) {
    TicTacToe game = new TicTacToe();
    /* X moves: */ /* O moves: */
    game.putMark(1,1); game.putMark(0,2);
    game.putMark(2,2); game.putMark(0,0);
    game.putMark(0,1); game.putMark(2,1);
    game.putMark(1,2); game.putMark(1,0);
    game.putMark(2,0);
    System.out.println(game);
    int winningPlayer = game.winner();
    String[ ] outcome = {"O wins", "Tie", "X wins"}; // rely on ordering
    System.out.println(outcome[1 + winningPlayer]);
}