

ICDT 6002 – Object Oriented Programming

Week 3 & 4

Methods

Arrays

Files

# Modular Programming

Static methods other than main()

# Modular Programming

- Until now we have been writing all the codes in one main function
- Given a big problem we can split the problem into smaller sub-problems and then solve the sub-problems independently
- Using methods we can also run the same code several times without writing it again and again

# What are methods?

- At the most basic level, a **method** is a sequence of statements that has been collected together and given a name.
- The name makes it possible to execute the statements much more easily; instead of copying out the entire list of statements, you can just provide the method name.

# Some important terms

- The following terms are useful when learning about methods:
  - Invoking a method using its name is known as **calling** that method.
  - The caller can pass information to a method by using **arguments**.
  - When a method completes its operation, it **returns** to its caller.
  - A method can pass information to the caller by **returning a result**.

# Methods and Information Hiding

- One of the most important advantages of methods is that they make it possible for callers to ignore the inner workings of complex operations.
- When you use a method, it is more important to know what the method does than to understand exactly how it works.
- The underlying details are of interest only to the programmer who implements a method.
- Programmers who use a method as a tool can usually ignore the implementation altogether.

# Methods as Tools for Programmers

- Particularly when you are first learning about programming, it is important to keep in mind that methods are not the same as application programs, even though both provide a service that hides the underlying complexity involved in computation.
- The key difference is that an application program provides a service to a user, who is typically not a programmer but rather someone who happens to be sitting in front of the computer. By contrast, a method provides a service to a programmer, who is typically creating some kind of application.

# Method Calls as Expressions

- Syntactically, method calls in Java are part of the expression framework. Methods that return a value can be used as terms in an expression just like variables and constants.
- The **Math** class in the **java.lang** package defines several methods that are useful in writing mathematical expressions. Suppose, for example, that you need to compute the distance from the origin to the point (x, y), which is given by

$$\sqrt{x^2 + y^2}$$

You can apply the square root function by calling the **sqrt** method in the **Math** class like this:

```
double distance = Math.sqrt(x * x + y * y) ;
```

- Note that you need to include the name of the class along with the method name. Methods like **Math.sqrt** that belong to a class are called **static methods**.



# Useful Methods in the **Math** Class

<b>Math.abs</b> ( <i>x</i> )	Returns the absolute value of <i>x</i>
<b>Math.min</b> ( <i>x</i> , <i>y</i> )	Returns the smaller of <i>x</i> and <i>y</i>
<b>Math.max</b> ( <i>x</i> , <i>y</i> )	Returns the larger of <i>x</i> and <i>y</i>
<b>Math.sqrt</b> ( <i>x</i> )	Returns the square root of <i>x</i>
<b>Math.log</b> ( <i>x</i> )	Returns the natural logarithm of <i>x</i> ( $\log_e x$ )
<b>Math.exp</b> ( <i>x</i> )	Returns the inverse logarithm of <i>x</i> ( $e^x$ )
<b>Math.pow</b> ( <i>x</i> , <i>y</i> )	Returns the value of <i>x</i> raised to the <i>y</i> power ( $x^y$ )
<b>Math.sin</b> ( <i>theta</i> )	Returns the sine of <i>theta</i> , measured in radians
<b>Math.cos</b> ( <i>theta</i> )	Returns the cosine of <i>theta</i>
<b>Math.tan</b> ( <i>theta</i> )	Returns the tangent of <i>theta</i>
<b>Math.asin</b> ( <i>x</i> )	Returns the angle whose sine is <i>x</i>
<b>Math.acos</b> ( <i>x</i> )	Returns the angle whose cosine is <i>x</i>
<b>Math.atan</b> ( <i>x</i> )	Returns the angle whose tangent is <i>x</i>
<b>Math.toRadians</b> ( <i>degrees</i> )	Converts an angle from degrees to radians
<b>Math.toDegrees</b> ( <i>radians</i> )	Converts an angle from radians to degrees

# Writing Your Own Methods

- The general form of a method definition is

```
public static type name (argument list) {  
    statements in the method body  
}
```

where *type* indicates what type of value the method returns, *name* is the name of the method, and *argument list* is a list of declarations for the variables used to hold the values of each argument.

- If a method does not return a value, *type* should be **void**. Such methods are sometimes called **procedures**.

# Returning Values from a Method

- You can return a value from a method by including a **return** statement, which is usually written as

```
return expression ;
```

where *expression* is a Java expression that specifies the value you want to return.

- As an example, the method definition

```
private static double feetToInches(double feet) {  
    return 12 * feet;  
}
```

converts an argument indicating a distance in feet to the equivalent number of inches, relying on the fact that there are 12 inches in a foot.

# Calling a user-defined method

```
import java.util.Scanner;

public class Sample1 {

    public static double feetToInches(double feet) {
        return 12 * feet;
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner input=new Scanner(System.in);

        double ftValue, inchValue;

        System.out.print("Input value in feet: ");
        ftValue=input.nextDouble();
        inchValue=feetToInches(ftValue);
        System.out.println("Corresponding value in inches:
        "+inchValue);
    }
}
```

# Methods Involving Control Statements

- The body of a method can contain statements of any type, including control statements. As an example, the following method uses an **if** statement to find the larger of two values:

```
private static int max(int x, int y) {  
    if (x > y) {  
        return x;  
    } else {  
        return y;  
    }  
}
```

- As this example makes clear, **return** statements can be used at any point in the method and may appear more than once.

# Calling function max()

```
import java.util.Scanner;
```

```
public class Sample2 {
```

```
    private static int max(int x, int y) {
```

```
        if (x > y) {
```

```
            return x;
```

```
        } else {
```

```
            return y;
```

```
        }
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        Scanner input=new  
        Scanner(System.in);
```

```
        int a, b;
```

```
        System.out.print("Input 2 values: ");
```

```
        a=input.nextInt();
```

```
        b=input.nextInt();
```

```
        System.out.println("Maximum of the  
        two is: " + max(a,b));
```

```
    }
```

```
}
```

# The **factorial** Method

- The **factorial** of a number  $n$  (which is usually written as  $n!$  in mathematics) is defined to be the product of the integers from 1 up to  $n$ . Thus,  $5!$  is equal to 120, which is  $1 \times 2 \times 3 \times 4 \times 5$ .
- The following method definition uses a **for** loop to compute the factorial function:

```
public static int factorial(int n) {  
    int result = 1;  
    for (int i = 1; i <= n; i++) {  
        result *= i;  
    }  
    return result;  
}
```

# Calling function factorial()

```
import java.util.Scanner;
```

```
public class Sample3 {
```

```
    public static int factorial(int n) {
```

```
        int result = 1;
```

```
        for (int i = 1; i <= n; i++) {
```

```
            result *= i;
```

```
        }
```

```
        return result;
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        // TODO Auto-generated method stub
```

```
        Scanner input=new Scanner(System.in);
```

```
        int n;
```

```
        System.out.print( "Input n: ");
```

```
        n=input.nextInt();
```

```
        System.out.println(n+"!=" +factorial(n));
```

```
    }
```

```
}
```



# Non-numeric Methods

Methods in Java can return values of any type. The following method, for example, returns the English name of the day of the week, given a number between 0 (Sunday) and 6 (Saturday):

```
public static String weekdayName(int day) {  
    switch (day) {  
        case 0: return "Sunday";  
        case 1: return "Monday";  
        case 2: return "Tuesday";  
        case 3: return "Wednesday";  
        case 4: return "Thursday";  
        case 5: return "Friday";  
        case 6: return "Saturday";  
        default: return "Illegal weekday";  
    }  
}
```

# Calling method weekdayName()

```
import java.util.Scanner;

public class Sample4 {
    public static String weekdayName(int day)
    {
        switch (day) {
            case 0: return "Sunday";
            case 1: return "Monday";
            case 2: return "Tuesday";
            case 3: return "Wednesday";
            case 4: return "Thursday";
            case 5: return "Friday";
            case 6: return "Saturday";
            default: return "Illegal weekday";
        }
    }
}
```

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Scanner input=new Scanner(System.in);
    int n;

    System.out.print("Input a day of week in
    number: ");
    n=input.nextInt();
    System.out.println("It is
    "+weekdayName(n));
}
}
```

# Example Calculate Sum

```
import java.util.Scanner;
public class SumFunctionExample {

    public static void main(String[] args) {
        Scanner input=new Scanner(System.in);
        int x;
        double y;

        System.out.println("Enter integer value x: ");
        x=input.nextInt();
        System.out.println("Enter double value y: ");
        y=input.nextDouble();

        double z=sum(x,y);

        System.out.println("The sum is "+z);
    }
```

```
public static double sum(int a, double b){
    double c;
    c=a+b;
    return c;
}
}
```

# Example Calculate Area

```
class AreaFinder
{
    public static void main(String [] args)
    {
        int base=10;
        int height=12;
        double result;

        result = CalArea(base, height);
        System.out.println("Area is "+ result);
    }

    public static double CalArea(int b, int h)
    {
        double area;
        area = (b*h)/2;
        return area;
    }
}
```

# Example Calculate Power

```
public class PowerEx {  
    public static void main(String[] args) {  
        int x=2;  
        int y=3;  
  
        int z=power(x,y);  
        System.out.println("The result is "+z);  
    }  
  
    public static int power(int a, int b){  
        int result=1;  
        for (int i=0;i<b;i++){  
            result=result*a;  
        }  
        return result;  
    }  
}
```

# Arrays in JAVA

Manipulating multiple data items of  
the same type

# Introduction to Arrays

- The following variable declarations each allocate enough storage to hold one value of the specified data type.

```
int number;  
double income;  
char letter;
```

- An array is an object containing a list of elements of the same data type.

# Declaring an Array Variable

- Do not have to create an array while declaring array variable
  - *<type>* [] variable\_name;
  - *int* [] prime;
  - *int* prime[];
- Both syntaxes are equivalent
- No memory allocation at this point



# Defining an array

- We can define an array by:
  - Declaring an array reference variable to store the address of an array object.
  - Creating an array object using the **new** operator and assigning the address of the array to the array reference variable.

# Defining an Array

- Define an array as follows:
  - `variable_name=new <type>[N];`
  - `primes=new int[10];`
- Declaring and defining in the same statement:
  - `int[] primes=new int[10];`
- In JAVA, int is of 4 bytes, total space= $4*10=40$  bytes

# Defining an array (cont.)

- Here is a statement that declares an array reference variable named *dailySales*:

```
double[ ] dailySales;
```

- The brackets after the key word `double` indicate that the variable is an array reference variable. This variable can hold the address of an array of values of type `double`. We say the data type of *dailySales* is `double` array reference.

# Defining an array (cont.)

The second statement of the segment below creates an array object that can store seven values of type double and assigns the address of the array object to the reference variable named *dailySales*:

```
double[ ] dailySales;
```

```
dailySales = new double[7];
```

- The operand of the new operator is the data type of the individual array elements and a bracketed value that is the array size declarator. The **array size declarator** specifies the number of elements in the array.

# Defining an array (cont.)

- It is possible to declare an array reference variable and create the array object it references in a single statement.

Here is an example:

```
double[ ] dailySales = new double[7];
```

# Defining an array (cont.)

- Arrays can be created to store values of any data type.

The following are valid Java statements that create arrays that store values of various data types:

```
double[ ] measurements = new double[24];
```

```
char[ ] ratings = new char[100];
```

```
int[ ] points = new int[4];
```

# Graphical Representation

prime

Index

value

0	1	2	3	4	5	6	7	8	9
2	1	11	-9	2	1	11	90	101	2

# What happens if ...

- We define

```
int[] prime=new long[20];
```

- The right hand side defines an array, and thus the array variable should refer to the same type of array

Error: incompatible types



# Array size declarator

- The array size declarator must be an integer expression with a value greater than zero.

- Valid code:

```
int k=7;
```

```
long[] primes = new long[k];
```

- Invalid Code:

```
int k;
```

```
long[] primes =new long[k];
```

*Compilation Output:*

MorePrimes.java:6: variable k might not have been initialized

```
long[] primes = new long[k];
```

# Array size declarator

- It is common to use a named constant as the array size declarator and then use this named constant whenever you write statements that refer to the size of the array. This makes it easier to maintain and modify the code that works with an array.

The statements below define a named constant called *MAX\_STUDENTS* and an array with room for one hundred elements of type `int` that is referenced by a variable named *testScores*.

```
final int MAX_STUDENTS = 100;
```

```
int[ ] testScores = new int[MAX_STUDENTS];
```

# Array Size through Input

....

```
Scanner input=new Scanner(System.in));
```

```
int num;
```

```
System.out.print("Enter a Size for Array:");
```

```
num = input.nextInt();
```

```
long[] primes = new long[num];
```

```
System.out.println("Array Length="+primes.length);
```

....

## **SAMPLE RUN:**

```
Enter a Size for Array:4
```

```
Array Length=4
```

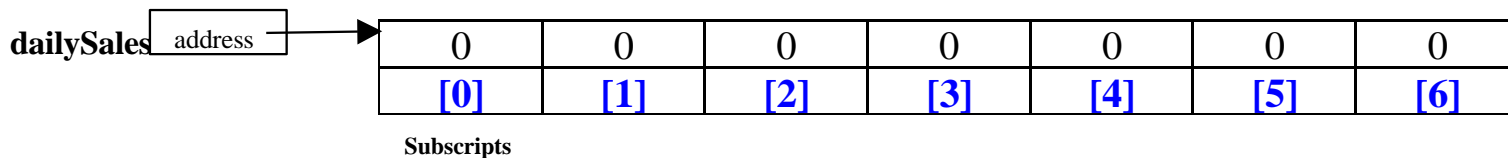
# Accessing Array Elements

- We can access the array elements and use them like individual variables.
- Each array element has a subscript. This subscript can be used to select/pinpoint a particular element in the array.
- **Array subscripts are offsets from the first array element.**
- The first array element is at offset/subscript 0, the second array element is at offset/subscript 1, and so on.
- The subscript of the last element in the array is one less than the number of elements in the array.

# Accessing Array Elements (cont.)

```
final int DAYS = 7;
```

```
double[ ] dailySales = new double[DAYS];
```



`dailySales[0]`, pronounced *dailySales* sub zero, is the first element of the array.

`dailySales[1]`, pronounced *dailySales* sub one, is the second element of the array.

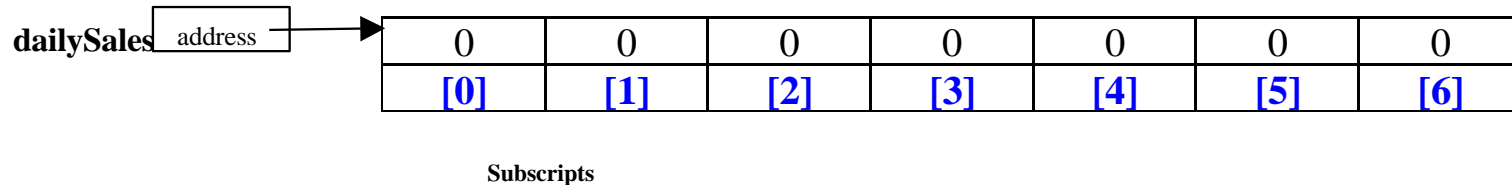
`dailySales[6]`, pronounced *dailySales* sub six, is the last element of the array.

# Accessing Array Elements (cont.)

- Array subscripts begin with **zero** and go up to  $n - 1$ , where  $n$  is the number of elements in the array.

```
final int DAYS = 7;
```

```
double[ ] dailySales = new double[DAYS];
```



# Accessing Array Elements (cont.)

- The value inside the brackets when the array is created is the array size declarator.
- The value inside the brackets of any other statement that works with the contents of an array is the array subscript.

# Accessing Array Elements (cont.)

- Each element of an array, when accessed by its subscript, can be used like an individual variable.

The individual elements of the *dailySales* array are variables of type `double`, we can write statements like the following:

```
final int DAYS = 7;
```

```
double[ ] dailySales = new double[DAYS];
```

```
dailySales[0] = 9250.56; // Assigns 9250.56 to dailySales sub zero
```

```
dailySales[6] = 11943.78; // Assigns 11943.78 to the last element of the array
```



# Accessing Array Elements (cont.)

```
final int DAYS = 7;
```

```
double[ ] dailySales = new double[DAYS];
```

```
dailySales[0] = 9250.56; // Assigns 9250.56 to dailySales sub zero
```

```
dailySales[6] = 11943.78; // Assigns 11943.78 to the last element of the array
```

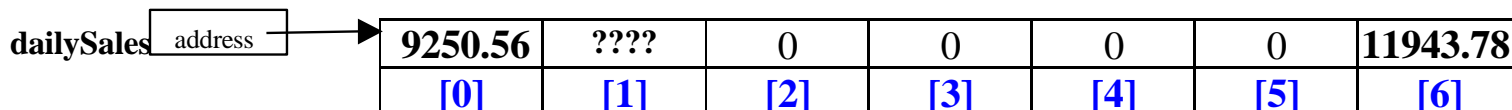
```
System.out.print("Enter the daily sales for Monday: ");
```

```
dailySales[1] = keyboard.nextDouble( ); // Stores the value entered in dailySales sub 1
```

```
double sum = dailySales[0] + dailySales[1]; // Adds the values of the first two elements
```

```
System.out.println("The sum of the daily sales for Sunday and Monday are " +  
sum + ".");
```

```
System.out.println("The sales for Monday were: " + dailySales[1]);
```



Subscripts

# Accessing Array Elements (cont.)

- Index of an array is defined as
  - Positive int, byte or short values
  - Expression that results into these types
- Any other types used for index will give error
  - long, double, etc.
  - In case Expression results in long, then type cast to int
- Indexing starts from 0 and ends at N-1
  - `primes[2]=0;`
  - `int k = primes[2];`
  - ...

# Inputting elements in an array

- A for loop can be used to input values into an array:

```
import java.util.Scanner;

public class Sample5 {

    public static void main(String[] args) {
        Scanner input=new Scanner(System.in);
        int[] numbers=new int[5];

        for(int i=0;i<5;i++){
            System.out.print("Input a value:");
            numbers[i]=input.nextInt();
        }
    }
}
```

## Output

```
Input a value:4
Input a value:3
Input a value:2
Input a value:1
Input a value:6
```

# Exercise 1

- Input of array of 10 marks and display of average.

# Displaying elements of an array

```
import java.util.Scanner;

public class Sample6 {

    public static void main(String[] args) {
        Scanner input=new Scanner(System.in);
        int[] numbers=new int[5];

        for(int i=0;i<5;i++){
            System.out.print("Input a value:");
            numbers[i]=input.nextInt();
        }

        System.out.print("Values inputted are: ");
        for(int i=0;i<5;i++)
            System.out.print(numbers[i]+" ");
    }
}
```

## Output

Input a value:4

Input a value:3

Input a value:2

Input a value:1

Input a value:6

Values inputted are: 4 3 2 1 6

# Validating Indexes

- JAVA checks whether the index values are valid at runtime
  - If index is negative or greater than the size of the array then an `IndexOutOfBoundsException` will be thrown
  - Program will normally be terminated unless handled in the `try { } catch { }`

# Java performs Bound Checking

- Java performs **bounds checking**, which means it does not allow a statement to use a subscript that is outside the range 0 through  $n - 1$ , where  $n$  is the value of the array size declarator.

```
long[] primes = new long[20];  
primes[25]=33;  
....
```

*Runtime Error:*

```
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 25  
at MorePrimes.main(MorePrimes.java:6)
```

# Java performs Bound Checking (cont.)

The valid subscripts in the array referenced by *dailySales* are 0 through 6. Java will not allow a statement that uses a subscript less than 0 or greater than 6. Notice the correct loop header in the segment below:

```
final int DAYS = 7;
```

```
int counter;
```

```
double[ ] dailySales = new double[DAYS];
```

```
for (counter = 0; counter < DAYS; counter++)
```

```
{
```

```
    System.out.print("Enter the sales for day " + (counter + 1) + ": ");
```

```
    dailySales[counter] = keyboard.nextDouble( );
```

```
}
```



# Java performs Bound Checking (cont.)

- Java does its bounds checking at runtime.
- The compiler **does not** display an error message when it processes a statement that uses an invalid subscript.
- Instead, Java throws an exception and terminates the program when a statement is executed that uses a subscript outside the array bounds. This is not something you want the user of your program to encounter, so be careful when constructing a loop that cycles through the subscripts of an array.

# Reusing Array Variables

- Array variable is separate from array itself
    - Like a variable can refer to different values at different points in the program
    - Use array variables to access different arrays
- ```
int[] primes=new int[10];  
  
.....  
primes=new int[50];
```
- Previous array will be discarded
  - Cannot alter the type of array

# Default Initialization

- When array is created, array elements are initialized
  - Numeric values (int, double, etc.) to 0
  - Boolean values to false
  - Char values to ‘\u0000’ (unicode for blank character)

# Initializing Arrays

- Like other variables, you may give array elements an initial value when creating the array.

Example:

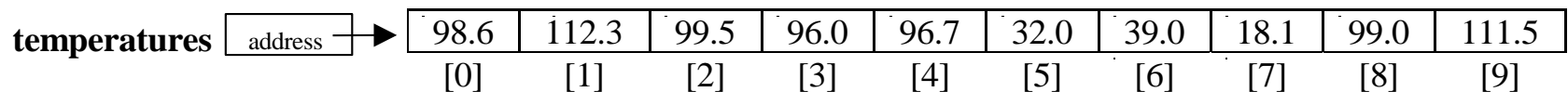
The statement below declares a reference variable named *temperatures*, creates an array object with room for exactly tens values of type `double`, and initializes the array to contain the values specified in the initialization list.

```
double[ ] temperatures = {98.6, 112.3, 99.5, 96, 96.7, 32, 39, 18.1, 99,  
                          111.5};
```

# Initializing Arrays

- A series comma-separated values inside braces is an **initialization list**.
- The values specified are stored in the array in the order in which they appear.
- Java determines the size of the array from the number of elements in the initialization list.

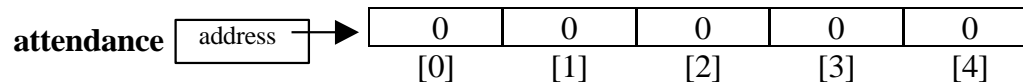
```
double[ ] temperatures = {98.6, 112.3, 99.5, 96, 96.7, 32, 39, 18.1, 99,  
                          111.5};
```



# Initializing Arrays

- By default, Java initializes the array elements of a numeric array with the value 0.

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



# Initializing Arrays

- You can initialize array with an existing array

```
int[] even={2,4,6,8,10};
```

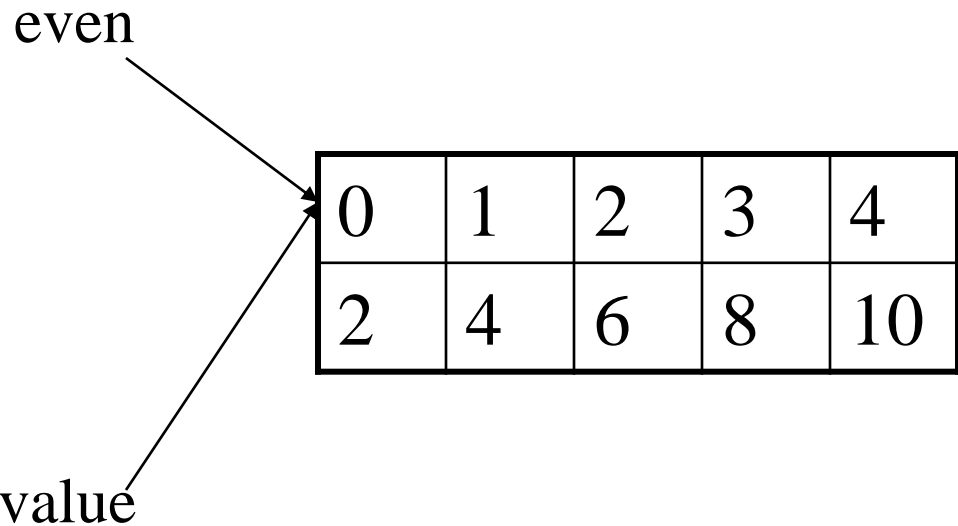
```
int[] value=even;
```

- One array but two array variables!
- Both array variables refer to the same array
- Array can be accessed through either variable name

# Graphical Representation

even

value



|   |   |   |   |    |
|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4  |
| 2 | 4 | 6 | 8 | 10 |



# Demonstration

```
long[] primes = new long[20];  
primes[0] = 2;  
primes[1] = 3;  
long[] primes2=primes;  
System.out.println(primes2[0]);  
primes2[0]=5;  
System.out.println(primes[0]);
```

# Output

2

5

# Array Length

- Refer to array length using *length*
  - A data member of array object
  - `array_variable_name.length`
  - `for(int k=0; k<primes.length;k++)`

....

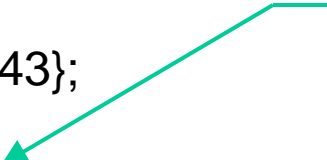
- Sample Code:

```
long[] primes = new long[20];
System.out.println(primes.length);
```
- Output: 20

# Array Length

To display the elements of the array referenced by *values*, we could write:

```
int count;  
int[ ] values = {13, 21, 201, 3, 43};
```



Notice, the valid subscripts are zero through *values.length - 1*.

```
for (count = 0; count < values.length; count++)  
{  
    System.out.println("Value #" + (count + 1) + " in the list of values is "  
        + values[count]);  
}
```

# Change in Array Length

- If number of elements in the array are changed, JAVA will automatically change the length attribute!

# Example

```
public class CalculateArrayAverageExample {  
  
    public static void main(String[] args) {  
  
        //define an array  
        int[] numbers = {10,20,15,25,16,60,100};  
  
        //calculate sum of all array elements  
        int sum = 0;  
  
        for(int i=0; i < numbers.length ; i++)  
            sum = sum + numbers[i];  
  
        //calculate average value  
        double average = sum / numbers.length;  
  
        System.out.println("Average value of array elements is : " + average);  
    }  
}
```

# Reassigning Array Reference Variables

- The assignment operator **does not** copy the contents of one array to another array.

# Reassigning Array Reference Variables

The third statement in the segment below copies the address stored in *oldValues* to the reference variable named *newValues*. It **does not** make a copy of the contents of the array referenced by *oldValues*.

```
short[ ] oldValues = {10, 100, 200, 300};  
short[ ] newValues = new short[4];
```

```
newValues = oldValues; // Does not make a copy of the contents of the array ref.  
                       // by oldValues
```

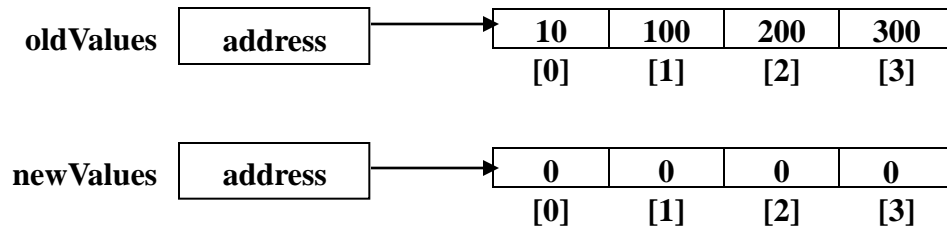


# Reassigning Array Reference Variables

After the following statements execute, we have the situation illustrated below:

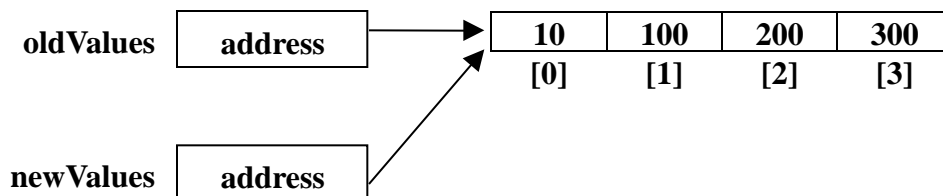
```
short[ ] oldValues = {10, 100, 200, 300};
```

```
short[ ] newValues = new short[4];
```



When the assignment statement below executes we will have:

```
newValues = oldValues; // Copies the address in oldValues into newValues
```



# Copying The Contents of One Array to Another Array

- To copy the contents of one array to another you must copy the individual array elements.

# Copying The Contents of One Array to Another Array

To copy the contents of the array referenced by *oldValues* to the array referenced by *newValues* we could write:

```
int count;
```

```
short[ ] oldValues = {10, 100, 200, 300};  
short[ ] newValues = new short[4];
```

**// If newValues is large enough to hold the values in oldValues**

```
if (newValues.length >= oldValues.length)  
{  
    for (count = 0; count < oldValues.length; count++)  
    {  
        newValues[count] = oldValues[count];  
    }  
}
```

## Class Exercise 2

- Write a program to allow a user to input the value of an integer  $n$  that would represent the number of students in a class and then create two arrays, IDs and marks to store the ID and marks of  $n$  students in a class.

# Class Exercise2

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    Scanner input=new Scanner(System.in);  
    int n;  
  
    System.out.print("How many students?");  
    n=input.nextInt();  
  
    int[] IDs=new int[n];  
    double[] marks=new double[n];  
}
```

# Class Exercise 3

- Modify Exercise 1 to input the ID and the marks of the  $n$  students and then display the:
  - Average of the class
  - ID and marks of the students who got the lowest marks
  - ID and marks of the students who got the highest marks

# Solution

```
public static void main(String[] args) {  
    Scanner input=new Scanner(System.in);  
    int n;  
  
    int[] IDs;  
    float[] marks;  
    float sum, average;  
  
    float minMarks, maxMarks;  
    int minID,maxID;  
  
    System.out.print("How many students?");  
    n=input.nextInt();  
  
    IDs=new int[n];  
    marks=new float[n];
```

# Solution (cont.)

```
for(int i=0;i<n;i++){  
    System.out.println("Student# "+(i+1));  
    System.out.print("\tInput ID: ");  
    IDs[i]=input.nextInt();  
    System.out.print("\tInput marks: ");  
    marks[i]=input.nextFloat();  
}
```



# Solution (cont.)

```
minMarks=101;
maxMarks=-1;
minID=maxID=0;
sum=0;
for(int i=0;i<n;i++){
    if(marks[i]<minMarks){
        minMarks=marks[i];
        minID=IDs[i];
    }

    if(marks[i]>maxMarks){
        maxMarks=marks[i];
        maxID=IDs[i];
    }
    sum=sum+marks[i];
}
```

# Solution (cont.)

```
        average=sum/n;
        System.out.println("Class Average="+average);
        System.out.println("Details of student with lowest
marks: ID="+minID+", marks="+minMarks);
        System.out.println("Details of student with highest
marks: ID="+maxID+", marks="+maxMarks);
    }

}
```

# Class Exercise 3

- Write a program to input the sales made over a week and then display the average sales for the week. Use an array to store the sales values.

# Class Exercise 3

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    Scanner input=new Scanner(System.in);  
  
    double[] sales=new double[7];  
    double sum=0, average;  
  
    System.out.print("Input sales for week: ");  
    for(int i=0;i<7;i++){  
        sales[i]=input.nextDouble();  
        sum=sum+sales[i];  
    }  
    average=sum/7;  
  
    System.out.println("Average sales="+average);  
}
```

# PASSING AN ARRAY AS AN ARGUMENT TO A METHOD

- An array can be passed as an argument to a method.
- To pass an array as an argument you include the name of the variable that references the array in the method call.
- The parameter variable that receives the array must be declared as an array reference variable.

# PASSING AN ARRAY AS AN ARGUMENT TO A METHOD

- When an array is passed as an argument, the memory address stored in the array reference variable is passed into the method.
- The parameter variable that stores this address references the array that was the argument.
- The method **does not** get a copy of the array.
- **The method has access to the actual array that was the argument and can modify the contents of the array.**

# SOME USEFUL ARRAY OPERATIONS

## Comparing Arrays

The decision in the segment below **does not** correctly determine if the contents of the two arrays are the same.

```
char[ ] array1 = {'A', 'B', 'C', 'D', 'A'};  
char[ ] array2 = {'A', 'B', 'C', 'D', 'A'};  
boolean equal = false;
```

```
if (array1 == array2) // This is a logical error  
{  
    equal = true;  
}
```

# SOME USEFUL ARRAY OPERATIONS

## Comparing Arrays

We are comparing the addresses stored in the reference variables *array1* and *array2*. The two arrays are not stored in the same memory location so the conditional expression is `false` and the value of *equal* stays at `false`.

```
char[ ] array1 = {'A', 'B', 'C', 'D', 'A'};  
char[ ] array2 = {'A', 'B', 'C', 'D', 'A'};  
boolean equal = false;
```

```
if (array1 == array2) // This is false - the addresses are not equal  
{  
    equal = true;  
}
```



# SOME USEFUL ARRAY OPERATIONS

## **Comparing Arrays**

- To compare the contents of two arrays, you must compare the individual elements of the arrays.
- Write a loop that goes through the subscripts of the arrays comparing the elements at the same subscript/offset in the two arrays.

# SOME USEFUL ARRAY OPERATIONS

## Comparing Arrays

```
public static boolean equals(char[ ] firstArray, char[ ] secArray)
{
    int subscript;
    boolean sameSoFar = true;

    if (firstArray.length != secArray.length)
    {
        sameSoFar = false;
    }

    subscript = 0;

    while (sameSoFar && subscript < firstArray.length)
    {
        if (firstArray[subscript] != secArray[subscript])
        {
            sameSoFar = false;
        }

        subscript++; // Incr. the counter used to move through the subscripts
    }
    return sameSoFar;
}
```

## **SOME USEFUL ARRAY OPERATIONS**

### **Finding the Sum of the Values in a Numeric Array**

- To find the sum of the values in a numeric array, create a loop to cycle through all the elements of the array adding the value in each array element to an accumulator variable that was initialized to zero before the loop.

# SOME USEFUL ARRAY OPERATIONS

## Finding the Sum of the Values in a Numeric Array

```
/** A method that finds and returns the sum of the values in the array of ints that is  
    passed into the method.  
    @param array The array of ints  
    @return The double value that is the sum of the values in the array.  
*/
```

```
public static double getSum(int[ ] array)  
{  
    int offset; // Loop counter used to cycle through all the subscripts in the array  
    double sum; // The accumulator variable - it is initialized in the header below  
  
    for (sum = 0, offset = 0; offset < array.length; offset++)  
    {  
        sum += array[offset];  
    }  
  
    return sum;  
}
```

## SOME USEFUL ARRAY OPERATIONS

### **Finding the Average of the Values in a Numeric Array**

- To find the average of the values in a numeric array, first find the sum of the values in the array. Then divide this sum by the number of elements in the array.

# SOME USEFUL ARRAY OPERATIONS

## Finding the Average of the Values in a Numeric Array

```
/** A method that finds and returns the average of the values in the array of ints that  
    is passed into the method.  
    @param array The array of ints  
    @return The double value that is the average of the values in the array.  
*/
```

```
public static double getAverage(int[ ] array)  
{  
    int offset; // Loop counter used to cycle through all the subscripts in the array  
    double sum; // The accumulator variable - it is initialized in the header below  
    double average;  
  
    for (sum = 0, offset = 0; offset < array.length; offset++)  
    {  
        sum += array[offset];  
    }  
    average = sum / array.length;  
  
    return average;  
}
```

## SOME USEFUL ARRAY OPERATIONS

### Finding the Highest Value in a Numeric Array

- To find the highest value in a numeric array, copy the value of the first element of the array into a variable called *highest*. This variable holds a copy of the highest value encountered so far in the array.
- Loop through the subscripts of each of the other elements of the array. Compare each element to the value currently in *highest*. If the value of the element is higher than the value in *highest*, replace the value in *highest* with the value of the element.
- When the loop is finished, *highest* contains a copy of the highest value in the array.

# SOME USEFUL ARRAY OPERATIONS

## Finding the Highest Value in a Numeric Array

```
/**A method that finds and returns the highest value in an array of doubles.  
    @param array An array of doubles  
    @return The double value that was the highest value in the array.  
*/
```

```
public static double getHighest(double[ ] array)  
{  
    int subscript;  
  
    double highest = array[0];  
  
    for (subscript = 1; subscript < array.length; subscript++)  
    {  
        if (array[subscript] > highest)  
        {  
            highest = array[subscript];  
        }  
    }  
    return highest;  
}
```



# SOME USEFUL ARRAY OPERATIONS

## Finding the Lowest Value in a Numeric Array

- The lowest value in an array can be found using a method that is very similar to the one for finding the highest value.
- Keep a copy of the lowest value encountered so far in the array in a variable, say *lowest*.
- Compare the value of each element of the array with the current value of *lowest*. If the value of the element is less than the value in *lowest*, replace the value in *lowest* with the value of the element.
- When the loop is finished, *lowest* contains a copy of the lowest value in the array.

## SOME USEFUL ARRAY OPERATIONS

### **Finding the Lowest Value in a Numeric Array**

```
public static double getHighest(double[ ] array)
{
    double min=array[0]; // initialize the current
        minimum
    for ( int index=0; index < array.length; index++ )
        if ( array[ index ] < min )
            min = array[ index ] ;
    return min;
}
```

# WORKING WITH PARTIALLY FILLED ARRAYS

- Quite often we do not know the exact number of items that we will need to store when we write a program. When this is the case, we can create an array that is large enough to hold the largest possible number of items and keep track of the actual number of items stored in the array.
- If the actual number of items stored in the array is less than the number of elements, we say that the array is **partially filled**.
- When you process the items in a partially filled array you only want to process array elements that contain valid data items.
- Keep a count of the actual number of items in the array in an integer variable as the array is filled and use this value as the upper bound on the array subscripts when processing the contents of the array.

# WORKING WITH PARTIALLY FILLED ARRAYS

Suppose we wanted to get up to one hundred whole numbers from the user and store them in an array and then do some processing on the contents of the array.

```

public class PartiallyFilledArray {

    public static void main(String[] args) {
        Scanner input=new Scanner(System.in);
        int[] array=new int[100];
        int numElements=0;
        int num, sum;
        double average;

        System.out.print("Input a set of numbers, 0 to stop: ");
        do{
            num=input.nextInt();
            if(num!=0)
            {
                //array[numElements]=num;
                //numElements++;
                array[numElements++]=num;
            }
        }while(num!=0);

        sum=0;
        for(int i=0;i<numElements;i++)
            sum=sum+array[i];
        average=(double)sum/numElements;

        System.out.println("Size of array="+array.length);
        System.out.println("Number of elements input="+numElements);
        System.out.println("Average="+average);
    }
}

```

## WORKING WITH PARTIALLY FILLED ARRAYS

Notice that it is not necessary, actually, it is wasteful to use an array to get the functionality produced by the program *PartiallyFilledArray.java*. We could instead add each number to an accumulator as we read it and keep a count of the valid numbers read and added to the accumulator. After all the numbers are read we can calculate and display the average.

# RETURNING AN ARRAY FROM A METHOD

- A method can return an array to the statement that made the method call. Actually, it is a reference to the array (the address of the array) that is returned.
- When a method returns an array reference, the return type of the method must be specified as an array reference.
- Return the address of an array from a method by putting the name of the variable that references the array after the key word `return` at the end of the method.

# RETURNING AN ARRAY FROM A METHOD

/\*\*A method that gets ten whole numbers from the user and stores them in an array and returns the reference to the array to the calling method.

@return The array containing the integers entered by the user

\*/

```
public static int[ ] getArrayOfInts( )
{
    Scanner keyboard = new Scanner(System.in);

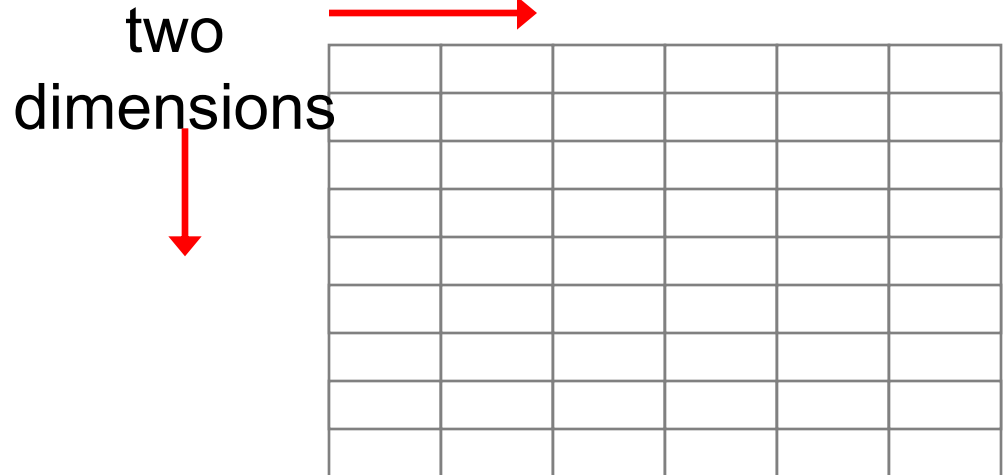
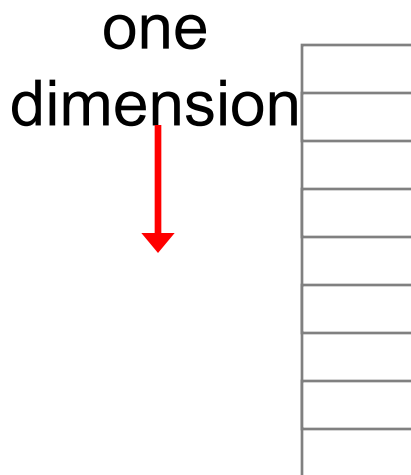
    final int MAX_NUMBERS = 10;
    int[ ] numbers = new int[MAX_NUMBERS];
    int subscript;

    for (subscript = 0; subscript < numbers.length; subscript++)
    {
        System.out.print("Enter number " + (subscript + 1) + ": ");
        numbers[subscript] = keyboard.nextInt( );
    }
    return numbers;
} // End of the method getArrayOfInts
```



# Two-Dimensional Arrays

- A *one-dimensional array* stores a list of elements
- A *two-dimensional array* can be thought of as a table of elements, with rows and columns



# Two-Dimensional Arrays

- To be precise, in Java a two-dimensional array is an array of arrays
- A two-dimensional array is declared by specifying the size of each dimension separately:

```
int[][] scores = new int[12][50];
```

- A array element is referenced using two index values:

```
value = scores[3][6]
```

- The array stored in one row can be specified using one index

| Expression                | Type                 | Description                                             |
|---------------------------|----------------------|---------------------------------------------------------|
| <code>table</code>        | <code>int[][]</code> | <b>2D array of integers, or array of integer arrays</b> |
| <code>table[5]</code>     | <code>int[]</code>   | <b>array of integers</b>                                |
| <code>table[5][12]</code> | <code>int</code>     | <b>integer</b>                                          |

# Two-Dimensional Arrays

- Two-Dimensional arrays
  - `float[][] temperature=new float[10][365];`
  - 10 arrays each having 365 elements
  - First index: specifies array (row)
  - Second Index: specifies element in that array (column)
  - In JAVA float is 4 bytes, total  
Size= $4*10*365=14,600$  bytes

# Graphical Representation

[illegible][illegible][illegible]

# Initializing Array of Arrays

```
int[][] array2D = { {99, 42, 74, 83, 100},  
    {90, 91, 72, 88, 95}, {88, 61, 74, 89, 96},  
    {61, 89, 82, 98, 93}, {93, 73, 75, 78, 99},  
    {50, 65, 92, 87, 94}, {43, 98, 78, 56, 99}  
};
```

//7 arrays with 5 elements each

# Arrays of Arrays of Varying Length

- All arrays do not have to be of the same length

```
float[][] samples;
```

```
samples=new float[6][]; //defines # of arrays
```

```
samples[2]=new float[6];
```

```
samples[5]=new float[101];
```

- Not required to define all arrays

# Initializing Varying Size Arrays

```
int[][] uneven = { { 1, 9, 4 }, { 0, 2}, { 0, 1, 2, 3, 4 } };
```

```
//Three arrays
```

```
//First array has 3 elements
```

```
//Second array has 2 elements
```

```
//Third array has 5 elements
```

# Array of Arrays Length

```
long[][] primes = new long[20][];  
primes[2] = new long[30];  
System.out.println(primes.length); //Number of arrays  
System.out.println(primes[2].length); //Number of elements in the second array
```

OUTPUT:

20

30



# Sample Program

```
class unevenExample3
{
    public static void main( String[] arg )
    { // declare and construct a 2D array
        int[][] uneven = { { 1, 9, 4 }, { 0, 2}, { 0, 1, 2, 3, 4 } };
        // print out the array
        for ( int row=0; row < uneven.length; row++ ) //changes row
        {
            System.out.print("Row " + row + ": ");
            for ( int col=0; col < uneven[row].length; col++ ) //changes
column
                System.out.print( uneven[row][col] + " ");
                System.out.println();
        }
    }
}
```

# Output

Row 0: 1 9 4

Row 1: 0 2

Row 2: 0 1 2 3 4

# Example

```
class Testarray3{
public static void main(String args[]){
    //declaring and initializing 2D array
    int arr[][]={{ 1,2,3},{2,4,5},{4,4,5}};

    //printing 2D array
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            System.out.print(arr[i][j]+" ");
        }
        System.out.println();
    }
}}
```

# Class Exercise on 2D-array

- Write a program that will define a two-dimensional array marks. The program will then request the user to input the number of students and create the array for maintaining the marks of the requested number of students.
- Next the program will request the number of modules taken by each student and create the appropriate number of elements to represent the same.
- The program must allow user to input marks for each student in the modules s/he has taken and calculate and display the average mark of each student

# Class Exercise on 2D-array

```
public class Students {  
    public static void main(String[] args) {  
        Scanner input=new Scanner(System.in);  
        int numStudents;  
        System.out.print("How many students? ");  
        numStudents=input.nextInt();  
  
        double[][] marks=new double[numStudents][];  
        for(int i=0;i<numStudents;i++){  
            int numModules;  
            System.out.print("Input no. of modules for Student# "+(i+1)+"  
:");  
  
            numModules=input.nextInt();  
            marks[i]=new double[numModules];  
  
            System.out.print("Input "+numModules+" marks: ");  
            for(int j=0;j<marks[i].length;j++)  
                marks[i][j]=input.nextDouble();  
        }  
    }  
}
```

# Class Exercise on 2D-array (cont.)

```
System.out.println();
```

```
//If you have a method findAverage()
```

```
//findAverage(marks);
```

```
double sum,avg;
```

```
for(int i=0;i<numStudents;i++){
```

```
    sum=0;
```

```
    for(int j=0;j<marks[i].length;j++)
```

```
        sum=sum+marks[i][j];
```

```
    avg=sum/marks[i].length;
```

```
    System.out.println("average for Student# "+(i+1)+":
```

```
 "+avg);
```

```
    }
```

```
}
```

```
}
```

# Passing a multi-dimensional array as parameter to a method

**public static <type>name (<type>[][] marks)**

```
public static double[] getAvgMarks(double[][] marks){  
    int rows=marks.length; // Get the size of the first dimension i.e. no. of rows  
    double[] results=new double[rows];  
  
    for(int x=0;x<rows;x++){  
        double sum=0;  
        int cols=marks[x].length; //Get the length of each row  
        for(int y=0;y<cols;y++){  
            sum+=marks[x][y];  
        }  
        results[x]=sum/cols;  
    }  
    return results;  
}
```

# Calling the method `getAvgMarks()`

```
public static void main(String[] args) {  
    //Marks of three students in different subjects  
    double[][]marks={ {20,40,50,44},  
                       {45,54,66,33,67},  
                       {55,44,45} };  
  
    double[] avg=getAvgMarks(marks);  
  
    for(int i=0;i<avg.length;i++){  
        System.out.println("Student"+(i+1)+" : Average="+(float)avg[i]); }  
}
```



# Output

No. of students= 3

Student1: Average=38.5

Student2: Average=53.0

Student3: Average=48.0

# Multidimensional Arrays

- An array can have many dimensions – if it has more than one dimension, it is called a *multidimensional array*
- Each dimension subdivides the previous one into the specified number of elements
- Each dimension has its own length constant

# Multidimensional Arrays

- A farmer has 10 farms of beans each in 5 countries, and each farm has 30 fields!

- Three-dimensional array

```
long[][][] beans=new long[5][10][30];  
//beans[country][farm][fields]
```

Files

# File Manipulation

- **Why files are important?**
- Computers use files because they are stored on secondary storage devices, namely hard disks, optical disks and magnetic tapes. Compared to primary memory, a secondary memory device
- represents a storage medium of high-capacity
- stores data in a persistent manner
- provides a means of several programs sharing data via the use of files

# Hierarchy of data

- Bit
- Byte or ASCII character (Digits, letters and symbols): A group of 8 bits
- Unicode Characters: Characters in java, Consist of two bytes
- Field: A group of characters that conveys meaning
- Record: A group of related fields
- File: A group of related records
- Database: A groups of related files
- DBMS: A group of programs to create and manage databases

# Types of File

- Binary File
- Input and output of data in terms of bytes
- May include a structure where the first few bits indicate the header (important information that is required for the application reading the file), the remaining bits indicate the file actual data.
- Examples: JPEG, MPEG etc
- Read by programs that converts the data into human-readable form (mpeg player)
- Java performs file processing using classes from java.io
  - FileInputStream
  - FileOutputStream
  - ObjectInputStream
  - ObjectOutputStream

# Text File

- Input and output of data in terms of characters
- Seen as grouping of characters. Characters may include digits, letters, symbols and special characters like end-of-line or end-of file.
- Examples: files with extention .txt or .dat.
- Is read by a text editor
- Java performs file processing using classes from java.io
  - FileWriter
  - FileReader



# Text File

- In this lecture, you will perform character-based input and output with existing classes
- Scanner
  - Class scanner was extensively used to input data from keyboard. Can also be used to input data from a file object
- Formatter
  - Class Formatter can be used to write formatted output into a file, with the same formatting capabilities as method `System.out.printf`

# Formatter Class

- The Formatter class has a constructor which enables you to open an existing file for writing or, if the file does not exist, to create one for writing. Note the Formatter constructor may throw two exceptions, namely, *SecurityException* (occurs if user does not have permission to write data into the file) and *FileNotFoundException* (occurs if the file does not exist and a new file cannot be created).

```
try{
    Formatter outfile = new Formatter ("Client.txt");
}
catch (FileNotFoundException fnfe){
    System.out.println(.....);
}
catch (SecurityException se){
    System.out.println (.....)
}
```

# Format Specifiers

```
Formatter outfile = new Formatter ("Client.txt");  
outfile.format("%s  tax  %6.2f", emp_id, salary);
```

- The *format()* method of *Formatter* class is used to write formatted output into a file. The *format* method takes as the first argument a format specifier string that contains all the formatting information for each argument in the list of arguments that follows next.
- In the example above, the format specifier string is  
    "%s tax %6.2f"
- It contains two format specifiers, each beginning with a % sign and followed by a character that represents the data type.

# Format Specifiers

%s string

%d integer

%ld long

%f float

%lf double (long float)

%45s string to fit in a 45-char space right-justified (45 is the field width)

%.2f float to 2 decimal places

%6.2f float to 2 decimal places, to fit in a 6-char space,

- In the above example, the two arguments to write into the file Client.txt are *emp\_id* (a string) and *salary* (a float).

43899 tax 56.22

# Formatter Class

- In general  
`outfile.format("format specifier string", arg1, arg2, ...argn)`
- Note that the number of arguments to dump into a file should correspond with the number of format specifiers and their types, as laid out in the format specifier string.
- Finally, the Formatter class also contains a method *close()*, which is normally used to close a file before our program exits.  
`outfile.close();`
- If the method `close` is not called explicitly, the OS will normally close the file when the program terminates.

# Scanner Class

- To open a file for reading, we would instantiate a Scanner object. However, a File object needs to be passed to Scanner constructor, which specifies a text file that the scanner object will read from. Also note that the constructor may an exception, *FileNotFoundException* (occurs if the file does not exist). Thus,

```
try{
Scanner infile = new Scanner(new File("Clients.txt"));
}
catch (FileNotFoundException fnfe){
System.out.println(.....);
}
```

# Scanner Class

- Once a file is open for reading, we can use the methods of Scanner class for reading data from the file, namely
  - `next():string`
  - `nextInt():int`
  - `nextFloat():float`
  - `nextDouble():double`
  - `nextLine():string`
  - `hasNext():boolean`
  - `hasNextInt():boolean`
  - etc
- For example:  
`String str = infile.next()`
- Will read into a string object a set of consecutive characters (until a blank or tab) from the file associated with *infile*.

# Example

- The file student.txt stores information about HSC students, namely their gender and their marks in Physics, Chemistry and Maths, and their phone number. A sample of the file is given below:

|            |   |    |    |    |          |
|------------|---|----|----|----|----------|
| Madvi Pen  | f | 45 | 56 | 34 | 752-2222 |
| Paul Gavin | m | 78 | 88 | 85 | 696-2315 |
| .....      |   |    |    |    |          |

- Write a program that reads information from the above file and create two files, one file that stores the name, gender, phone number, and the average mark of all students who have passed all modules, and one file that the name, gender, phone number, and the number of passed modules for all students who have failed at least one module. Assume the pass mark is 40%.



# Example

## Writing to a file:

```
import java.io.*;
import java.util.*;
public class InputFileEx {
    public static void main (String args[]){
        try{
            String empid = "B001";
            float salary = 10000;

            Formatter outfile = new Formatter("Client.txt");
            outfile.format("%s tax %6.2f",empid, salary);
            outfile.close();
        }
        catch(FileNotFoundException fnfe){
            System.out.println("File Not Found");
        }
        catch(SecurityException se){
            System.out.println("No Permission");
        }
    }
}
```

# Example

## Reading from a file:

```
import java.io.*;
import java.util.*;

public class ReadFileEx {
    public static void main(String args[]){
        try {
            Scanner infile = new Scanner(new File("Client.txt"));
            String empid = infile.next();
            infile.next();
            float salary = infile.nextFloat();
            System.out.println(empid);
            System.out.println(salary);
        }
        catch (FileNotFoundException fnfe){
            System.out.println("File Not Found");
        }
    }
}
```