

From <http://www.onlineprogramminglessons.com>

These Java mini lessons will teach you all the Java Programming statements you need to know, so you can write 90% of any Java Program.

- Lesson 1 Input and Output**
- Lesson 2 Methods**
- Lesson 3 Classes and Inheritance**
- Lesson 4 Operators**
- Lesson 5 Programming Statements**
- Lesson 6 Arrays, ArrayList, Lists, Sets and Maps**
- Lesson 7 Overloading, Interfaces and Generics**
- Lesson 8 Enhanced Loops, Iterators, Comparators and Anonymous Functions**
- Lesson 9 File Access**
- Lesson10 Abstract Classes, Polymorphism and Java Objects**
- Lesson 11 Recursion**
- Lesson 12 Regular Expressions**
- Lesson 13 Java Project**

Conventions used in these lessons:

bold - headings, keywords, code

italics - code syntax

underline - important words

(open round bracket
) close round bracket

{ open curly bracket
} close curly bracket

[open square bracket
] close square bracket

Let's get started!

You first need a Java compiler/interpreter to run Java Programs and a Java IDE to edit, compile and run Java programs automatically. A Java IDE makes Java development much easier. There are many Java IDE's available, some are very complicated to use. In these lessons we will use a simple and powerful one called JCreator.

You first need to download the Java JDK or SE compiler/interpreter that is used to compile and run java programs manually.

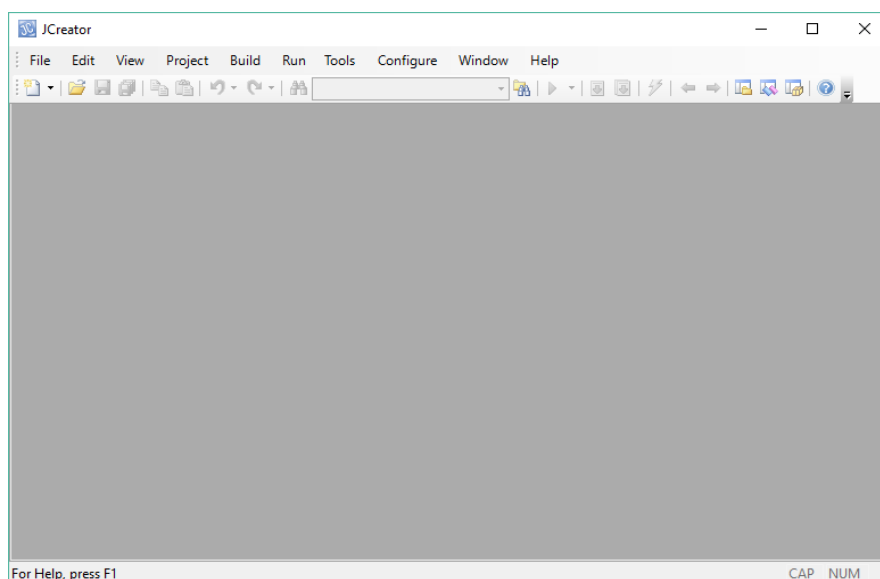
<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Next download the Java IDE JCreator (or Java IDE of your choice)

<http://www.jcreator.org/download.htm>

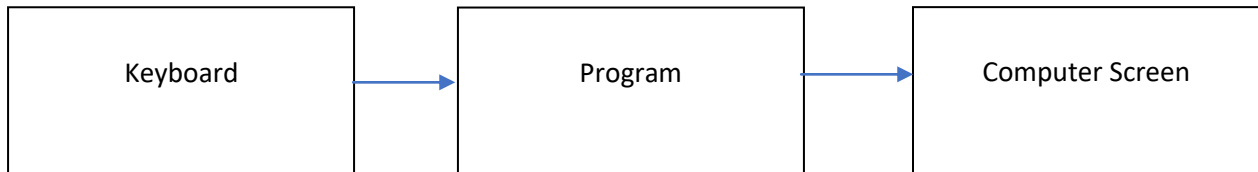
Unfortunately, you need to wait till they send the downloads details to you, but fortunately you can find third party downloads on the internet that let you download right away. Download the free light version.

Once you install Java and JCreator, run the JCreator, you will get this screen:

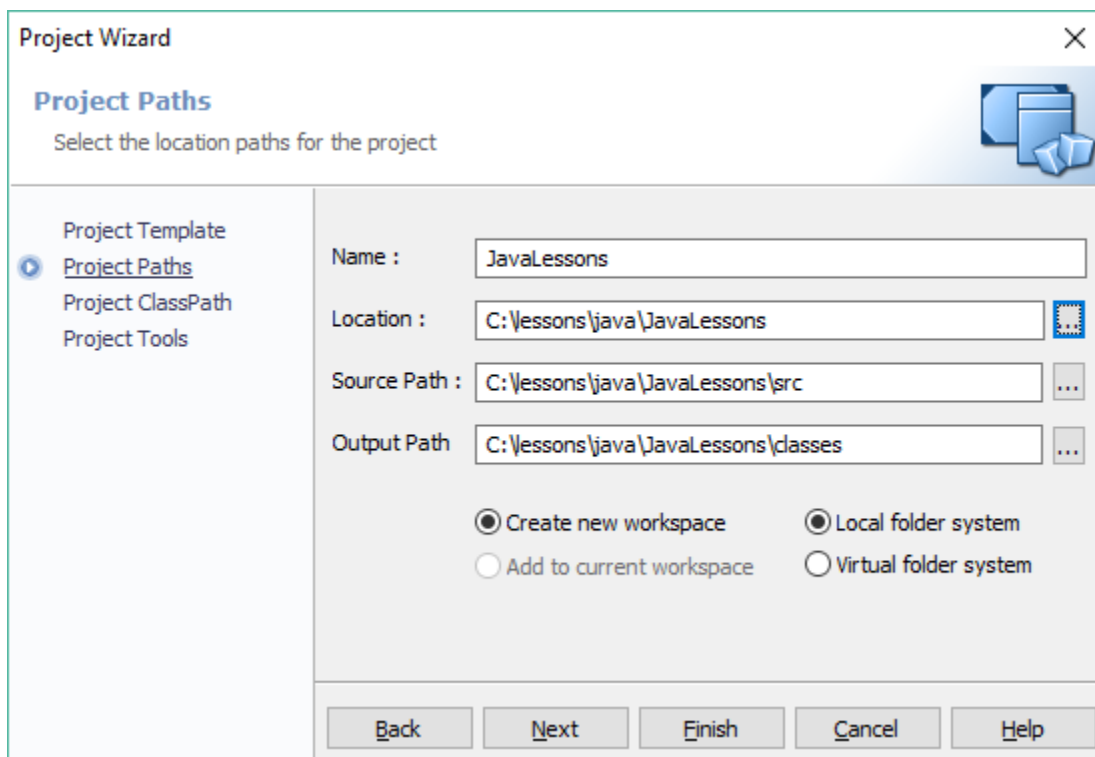


Lesson 1 Input and Output

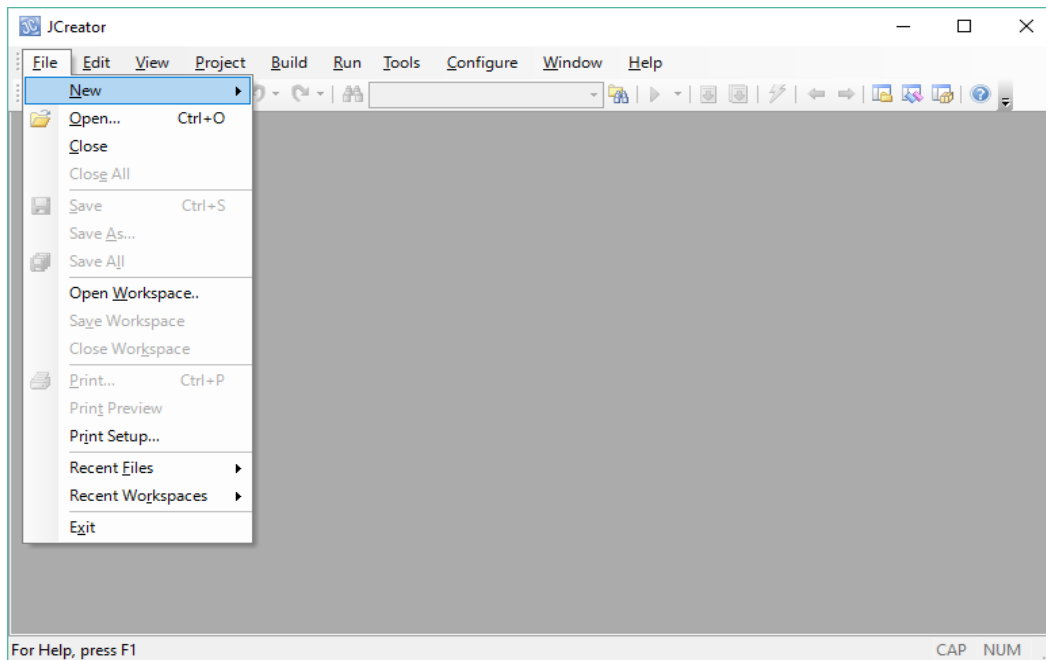
Input refers getting a value from the keyboard or a file and output refers writing a value on the computer screen or storing a value in a file.



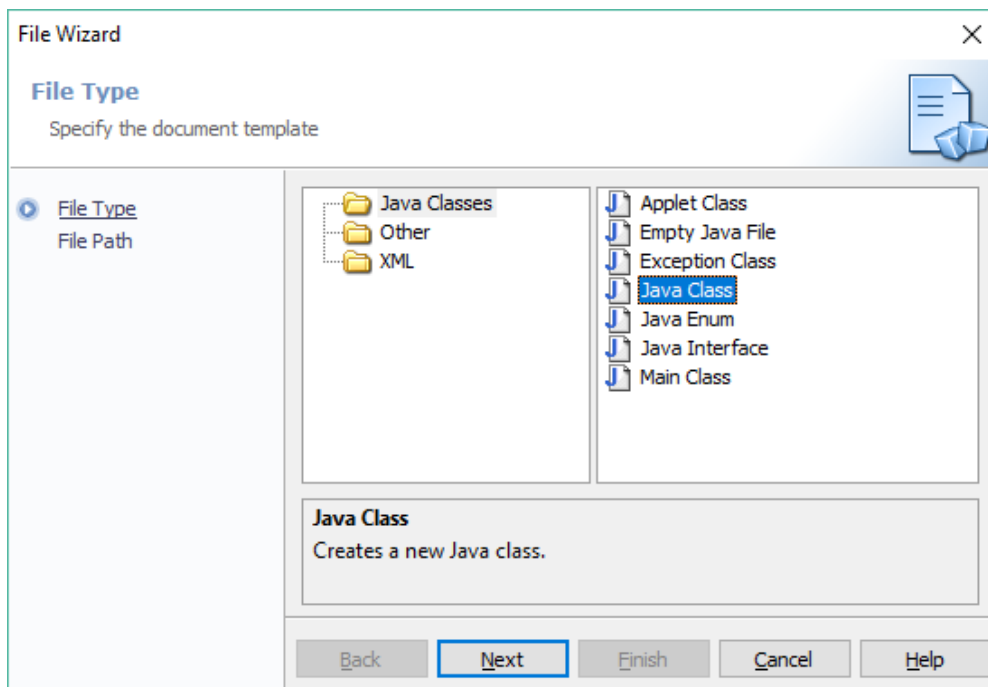
Before we start it is best to store all your Java lesson programs in a file. But you first need to make a Project to store all our Java lesson files in. We will use the project name JavaLessons. Make a folder in your computer called JavaLessons. Using JCreator or your IDE You need to change the location of the project to the location where your java lesson files will be stored on your computer. It is best to store all your java Lesson programs in a folder so you can easily find them.



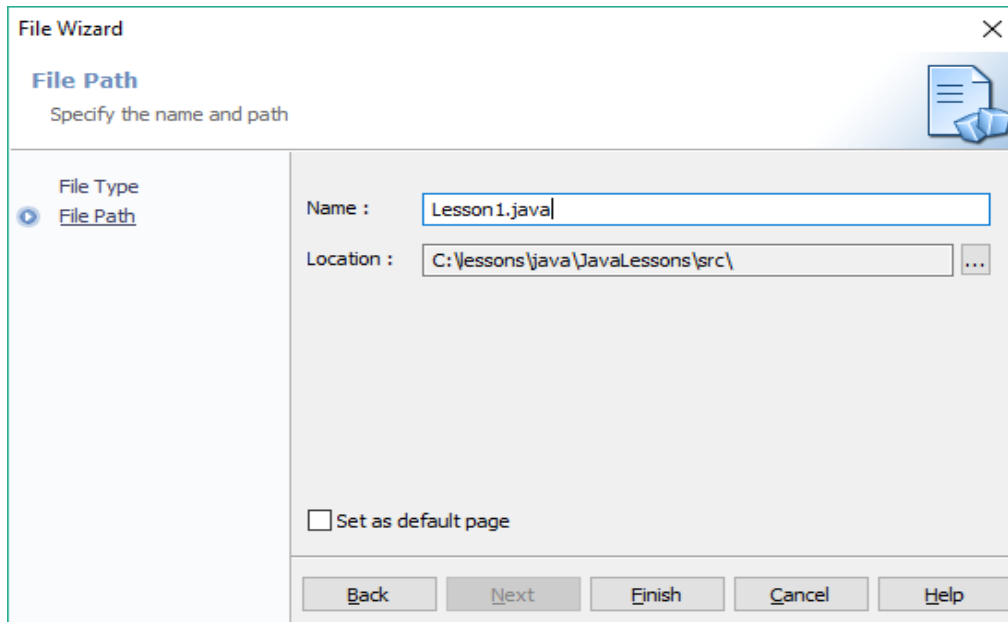
From the File Menu select File New.



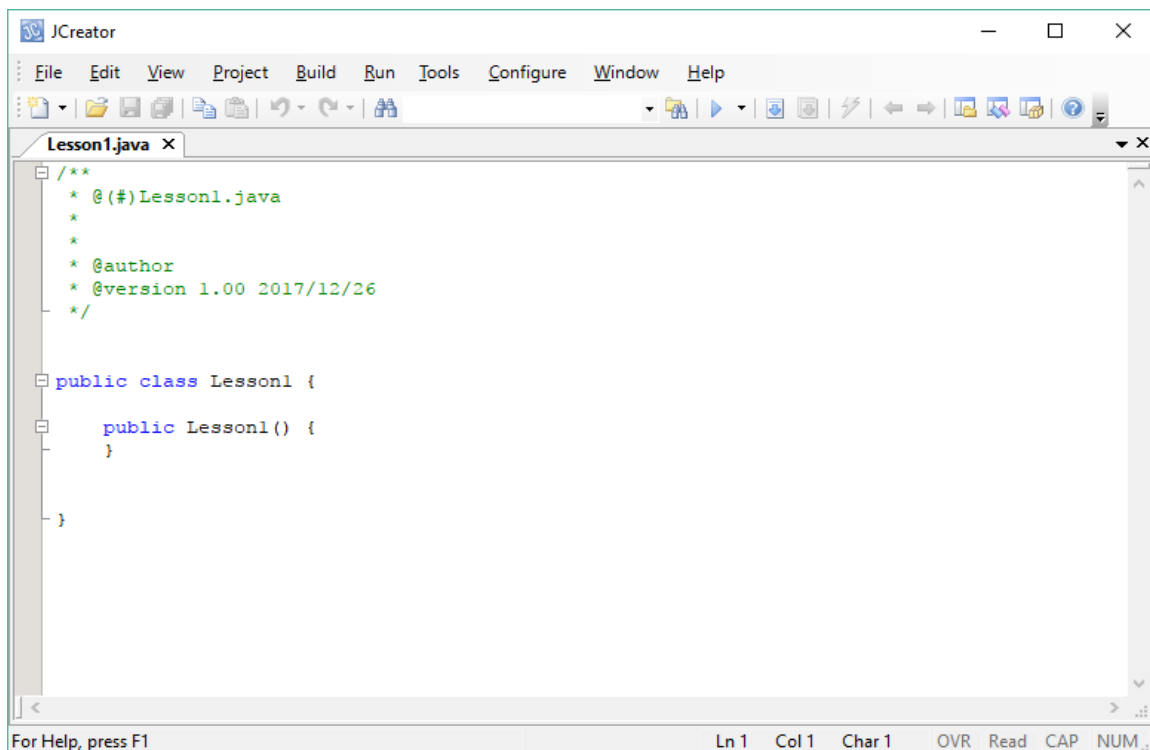
Then select Java Classes then Java Class



Press 'Next' then type in the java file name Lesson1.java



The editor window appears where you can type in Java programming statements. JCreator automatically writes the Lesson1 starter code for you.

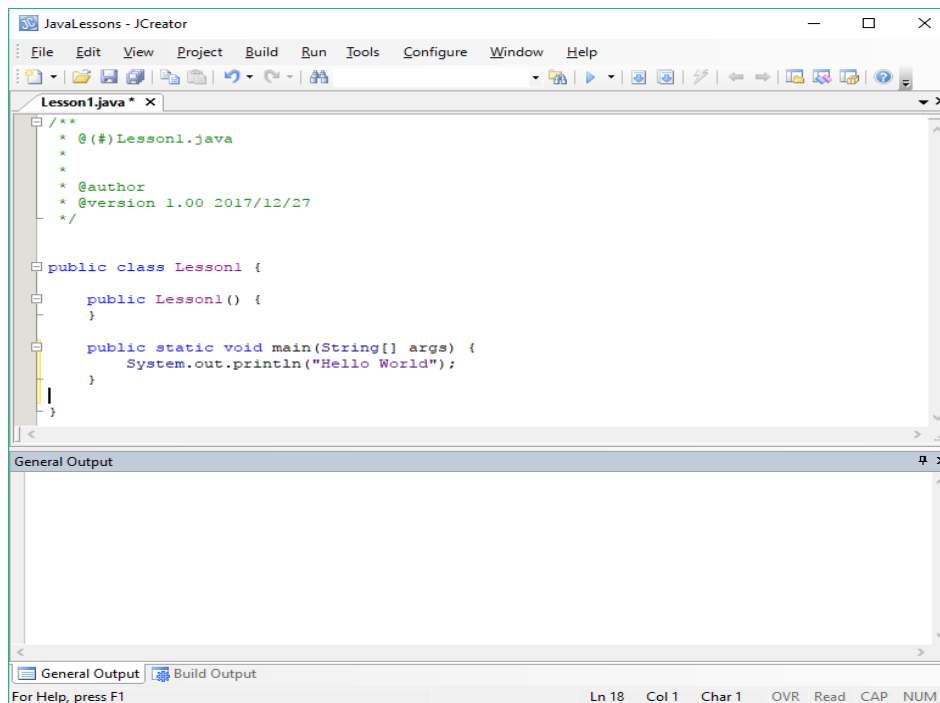


All Java programs are contained in a Java Class. A class contains variables that store information and methods that do operations (calculations) on these variables. Methods contain programming statements telling the computer what to do. A class is a big step in programming evolution as well as for someone just learning programming. It's now time to write your first Java Program. You will print "Hello World" on the screen. In the Java Editor near the bottom of the Lesson1.java type in:

```
public static void main(String[] args) {  
    System.out.println("Hello World");  
}
```

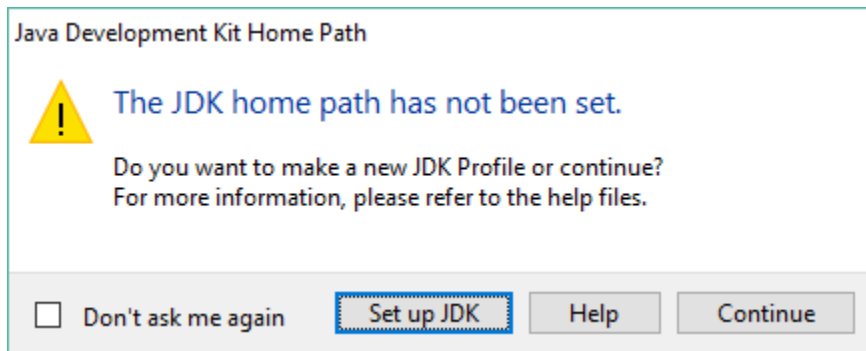
You should have this now in your editor window:

```
public class Lesson1 {  
    public static void main(String[] args) {  
        System.out.println("Welcome to my Program");  
    }  
}
```

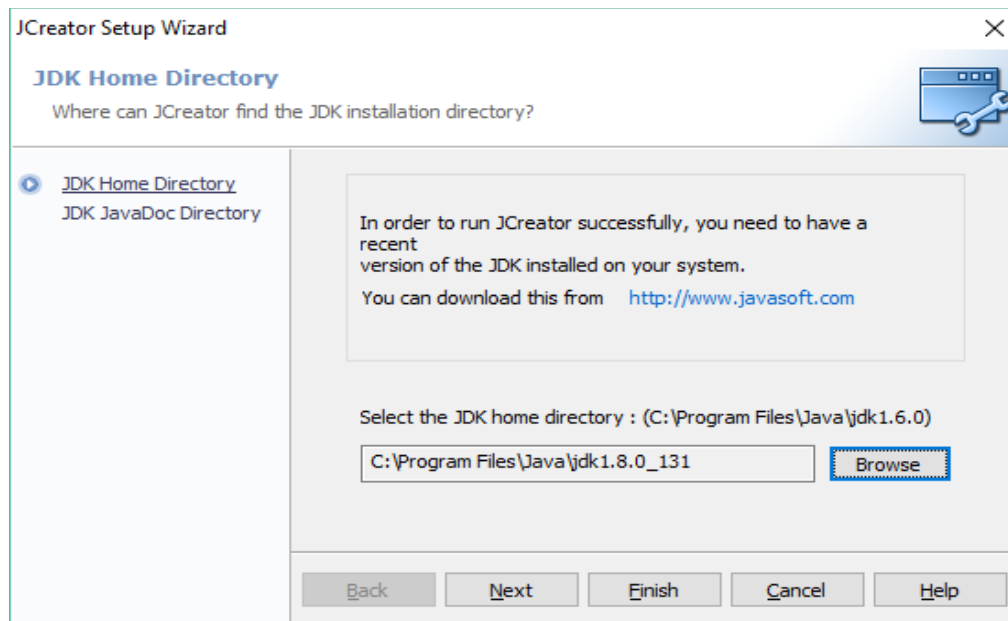


Next you need to build your Java Lesson1.java file, to compile it and check for any errors. You may first need to tell JCreator where the Java JDK is. Press the 'Set up

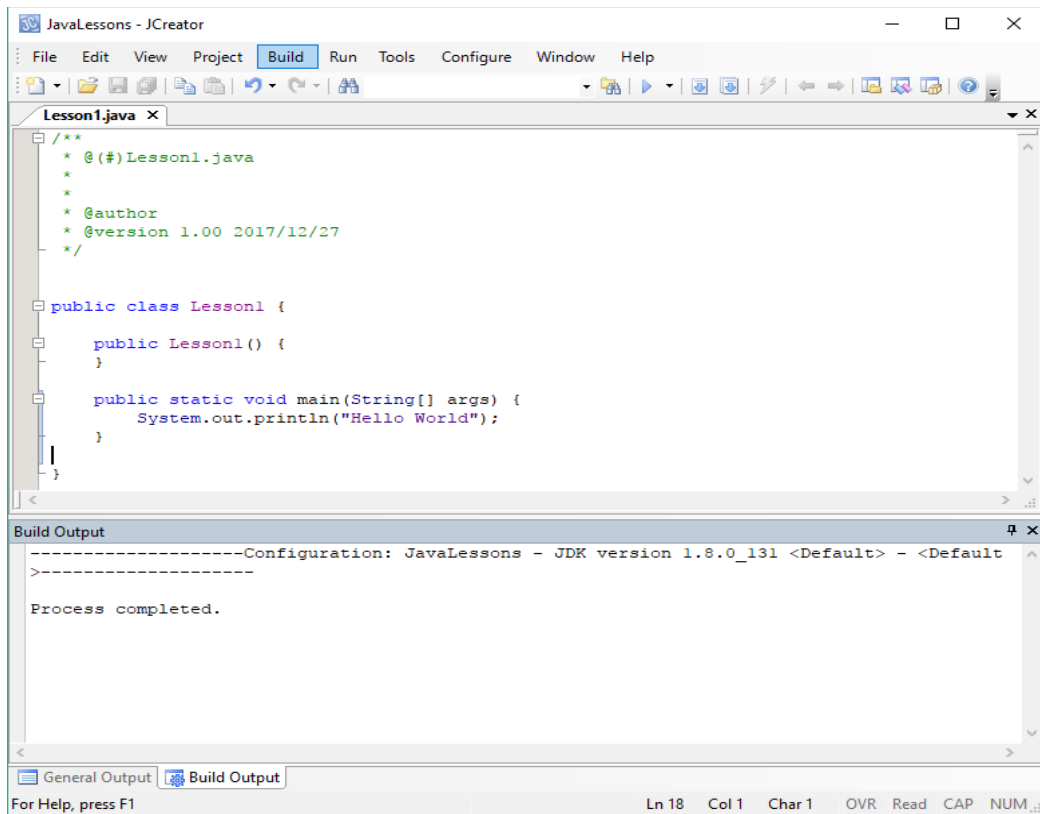
JDK' button. This screen will appear automatically just before you build it.



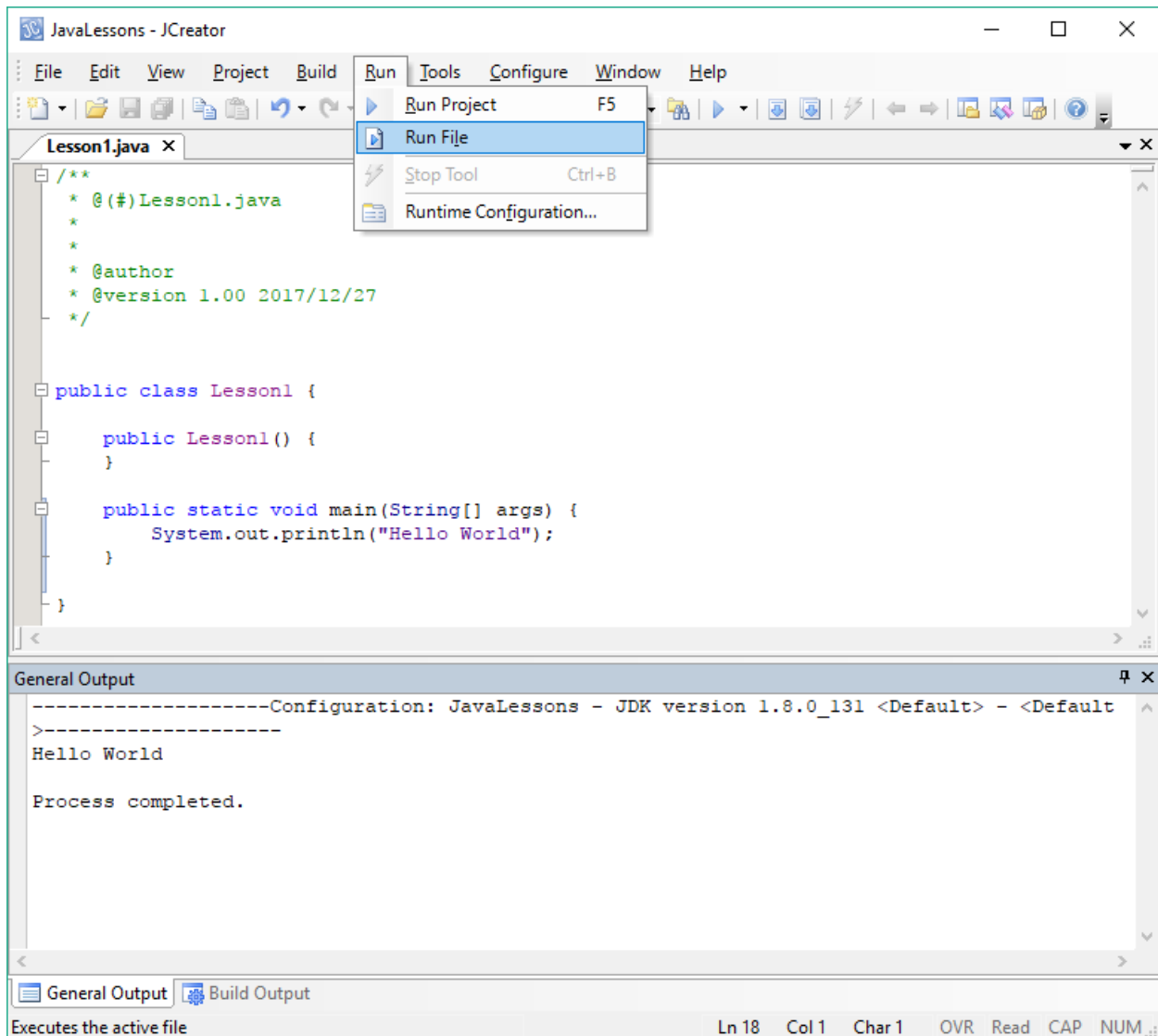
Browse where your Java is located on your computer.



From the Build menu select Build file.



If you get some errors, then you must correct them and then rebuild the file. Errors are typing mistakes that can be easily corrected but may be difficult to find. Errors are always the unexpected and the overlooked. If you do not have any errors, then you can run your file. From the Run menu select Run File. “Hello World” is now printed in the Build Output window. Always use Run File to run your opened file.



Recapping: In a Java program methods are enclosed in a class. A class definition starts with a **access modifier** like **public** the keyword **class** followed by the class name. The opening curly bracket '{' means start the class.

public class Lesson1 {

The methods of the class are written between the curly brackets{ } of the class definition. Methods contain the programming statements that tell the computer what to do. Inside the Lesson1 class we have a main method. The main method is executed when the program first runs.

```

public static void main(String[] args) {

    System.out.println("Hello World");
}

```

The main method contains the **System.out.println** statement that prints the message “Hello World” on the screen.

The closing curly bracket ‘}’ means end the class. Before we proceed it is important to understand the terminology: classes, methods, programming statements and objects and static.

data type	What type of data is to be represented
variable	Stores a string or numeric value. All variables have a data type
programming statement	is an instruction containing commands to perform a desired action, like printing a value on the screen, get a value from the key board or calculate a certain value?
method	contain programming statements that tell the computer what to do and performs operations on variables
class	Contains values and methods
object	Computer Memory allocated for variables defined in a class
static	Permanent code or value that can be used right away

The next thing we need to do is get values from the keyboard. We will ask the user to type in their name and then greet them. Type in the following statements in the Java editor right after the Hello World statement.

```

System.out.println("Welcome to my Program");
Scanner kybd = new Scanner(System.in);
System.out.print("Please type in your name: ");
String name = kybd.nextLine();
System.out.println("Nice to meet you " + name);

```

You will also need to put this statement on top of the **public class Lesson1** statement, it tells the java compiler where the Scanner is.

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

The Scanner is used to read values from the keyboard.

Your complete program should now be like this:

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

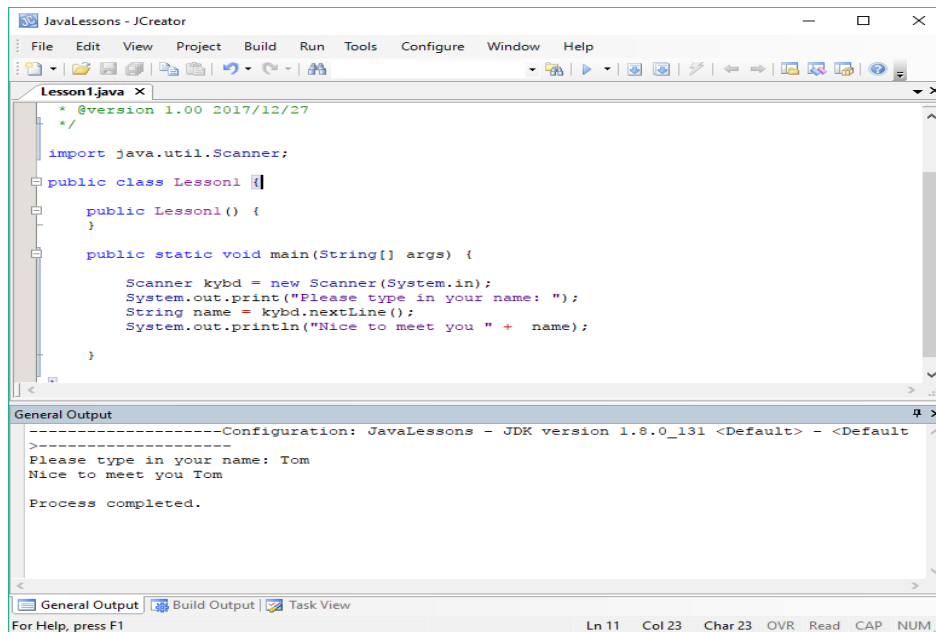
```
public class Lesson1 {
```

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

```
        System.out.println("Welcome to my Program");  
        Scanner kybd = new Scanner(System.in);  
        System.out.print("Please type in your name: ");  
        String name = kybd.nextLine();  
        System.out.println("Nice to meet you " + name);
```

```
    }  
}
```

Now run your program, and enter the name “Tom”. You will get something like this:



Recapping: We need to make our own keyboard reader from the Scanner class called kybd.

Scanner kybd = new Scanner(System.in);

We also needed to tell the Java Compiler where the keyboard Scanner code is to be found. We had to place the following statement near the top of the Java file.

import java.util.Scanner;

We first ask the user to type in their name using the **System.out.print** statement.

System.out.print("Please type in your name: ");

Then we obtain the user name from the keyboard using the **nextLine** method from keyboard reader.

String name = kybd.nextLine();

The entered name is placed in the String variable name.

The **System.out.println** statement prints out the string message "Nice to meet you" and the name of the user stored in the variable name.

```
System.out.println("Nice to meet you " + name)
```

Note inside the **System.out.println** statement the string message and variable name are joined by a '+'.

Java has two types of values **String** values and **numeric** values. String values are messages enclosed in double quotes like "Hello World" where as numeric values are numbers like 5 and 10.5 Numeric values without decimal points like 5 are known as an **int** and numbers with decimal points like 10.5 are known as a **float** or **double**. Variable's store string or numeric values that can be used later in your program.

We now continue our program ask the user how old they are. Type in the following statements at the end of your program.

```
System.out.print("How old are you? ");  
int age = Integer.parseInt(kybd.nextLine());  
System.out.println("You are " + age + " years old");
```

Your complete program should be something like this:

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

```
public class Lesson1 {
```

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

```
        System.out.println("I like Java Programming");  
        Scanner kybd = new Scanner(System.in);  
        System.out.print("Please type in your name: ");  
        String name = kybd.nextLine();  
        System.out.println("Nice to meet you " + name);  
        System.out.print("How old are you? ");
```

```

    int age = Integer.parseInt(kybd.nextLine());
    System.out.println("You are " + age + " years old");
}
}

```

Run the program and enter Tom for name and 24 for age, you should get something like this.

The screenshot shows the JCreator IDE with a file named `Lesson1.java`. The code defines a `Lesson1` class with a `main` method that uses a `Scanner` to read input from the user. The output window shows the program's execution, where the user entered 'Tom' for the name and '24' for the age, resulting in the output 'You are 24 years old'.

```

public class Lesson1 {
    public Lesson1() {
    }

    public static void main(String[] args) {
        Scanner kybd = new Scanner(System.in);
        System.out.print("Please type in your name: ");
        String name = kybd.nextLine();
        System.out.println("Nice to meet you " + name);
        System.out.print("How old are you? ");
        int age = Integer.parseInt(kybd.nextLine());
        System.out.println("You are " + age + " years old");
    }
}

```

General Output

```

-----Configuration: JavaLessons - JDK version 1.8.0_131 <Default> - <Default>
>-----
Please type in your name: Tom
Nice to meet you Tom
How old are you? 24
You are 24 years old

Process completed.

```

Ln 3 Col 21 Char 21 OVR Read CAP NUM

Recapping: The **System.out.print("How old are you? ")** statement asks the user to enter their age. The **int age = Integer.parseInt(kybd.nextLine())** statement receives a string number from the keyboard using **kybd.nextLine()** and converts the string to a numeric int using **Integer.parseInt** and then assigns the numeric int to the variable **age**. The **System.out.println("You are " + age + " years old");** statement is used to print out the message, the persons name and age. Again the **+** operator is used to join the age numeric value to the string messages.

If you have got this far then you will be a great Java programmer soon.

Most people find Programming difficult to learn. The secret of learning program is to figure out what you need to do and then choose the right program statement to use. If you want to print messages and values to the screen you use a **System.out.println** statement. If you want to get values from the user, you use an **kybd.nextLine()** statement. If you need a numeric value, you use an **Integer.parseInt** statement to convert String numbers to int numbers or **Float.parseFloat** statement to convert String numbers to decimal float numbers or the **Double.parseDouble** statement to convert String numbers to decimal double. numbers. The difference between float decimal numbers and double decimal numbers is just accuracy.

You should concentrate on getting your programs running rather than understand how they work. Once you get your programs running and you execute them understanding will be come much easier. Understanding is much easier now because you can now make an association connection to the program statement that is running, that produces the desired input or output action.

Java Data Types

Programming is all about storing values in a computer memory location and doing operations on them like adding two numbers together. The values to store are classified into data types. Data types specify what kind of data a memory location will hold. Each memory location is represented by a variable name like **x**. When we declare a variable name we must also specify the data type so the Java compiler knows what kind of data the variable name represents.

```
int x;  
data_type variable_name;
```

When you declare a variable it is best to give it a default value, this is known as initialization.

```
int x = 0;  
data_type variable_name = initialized_value;
```

Java has many data types that can be used to represent many different kinds of numbers as follows: All data types in Java are signed meaning positive and negative numbers. Each different data type has a range of values it can represent. Smaller size data types can store smaller numbers, whereas larger size data types can store larger numbers. The size of the data type is determined by the number of **bits** the memory location holds. **Bits** is a memory storage unit having values 0 or 1. Bits are grouped together in a data type size to represent a number. The bit pattern 00000101 represents the value 5 and has a bit size of 8 representing a **Byte** data type.

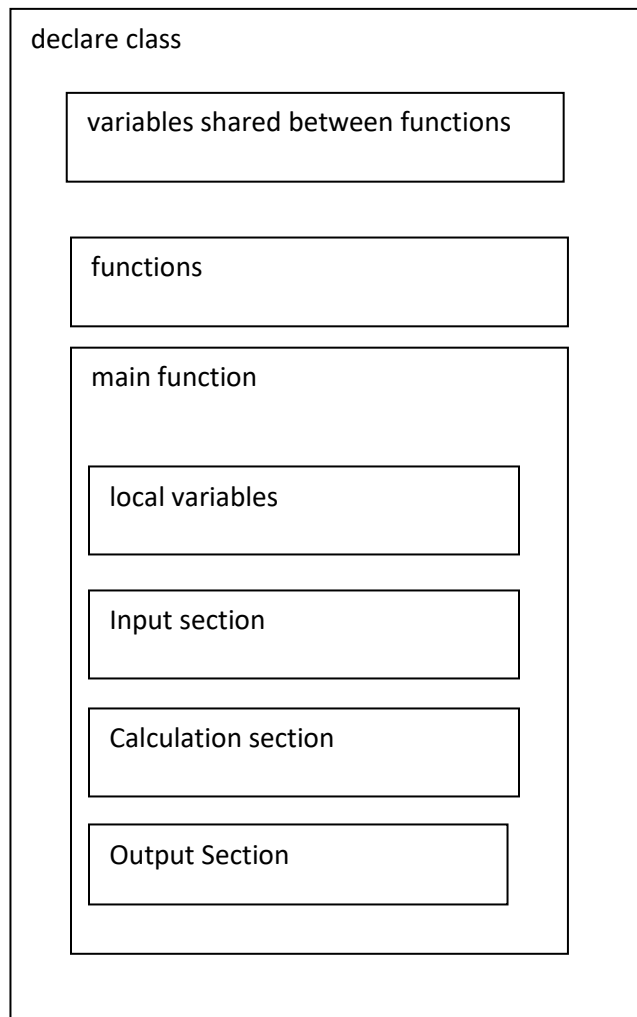
Data Type	Size (bits)	Min value	Max Value	Example
byte	8	-128	127	byte x = 100;
short	16	-32768	32767	short x = 1000;
int	32	-2 ³¹	2 ³¹ -1	int x = 10000;
long	64	-2 ⁶³	2 ⁶³ -1	long x = 10000;
float	32	1.4E-45	3.40282E38	float f = 10.5;
double	64	4.9E-324	4.9E-324	double d = 10.5;
boolean	1	false	True	boolean x = true;
char (unicode)	16	'\u0000' (0)	'\uffff' (65535)	char x = 'A';

Unicode can represent many different characters

Lesson 1 Homework

Write a Java program that asks someone what their profession **title** is and what their **salary** is. Their title could be a doctor, lawyer etc. Next print out the details to the screen, what their title is and how much money they make. For the profession **title** use a String variable. For the **salary** variable you can use **float** or **double** data type. You will need to use the **Float.parseFloat** statement to convert a String number to float number or the **Double.parseDouble** statement to convert String numbers to double number. Call your java program Homework1.java and class Homework1.

JAVA PROGRAM FORMAT



Lesson 2 Methods

Methods allow you to group many programming statements together so that you can reuse them repeatedly in your Java Program. Methods are analogous to functions in other programming languages. They are called methods because they are contained in a class and cannot be used independently by themselves. The most common method is the **main** method that starts a Java program, which we have used previously in Lesson 1. A class may have many methods. Each method has a dedicated purpose, some action to perform. Methods usually are defined at the top of the class in order as they are used. The main method is the last one because it will call all the proceeding methods. When a method is called in a programming statement, it means, it is executed. Java also has many built in methods that you can use, that make Java programming easier to do. You already used some of them in lesson 1: **print, println, nextLine**. As we proceed with these lessons you will learn and use many more methods. It is now time to add more methods to our previous lesson 1 program. We will make a **welcome, enterName, enterAge** and **displayInfo** methods. Make a new java file called Lesson2.java and then type in the following code.

```
import java.util.Scanner;

public class Lesson2 {

    private static Scanner kybd = new Scanner(System.in);

    public static void welcome(){
        System.out.println("Hello World")
    }

    public static String enterName(){
        System.out.print("Please type in your name: ")
        String name = kybd.nextLine();
        return name;
    }
}
```

```

public static int  enterAge(){
    System.out.print("How old are you? ");
    int age = Integer.parseInt(kybd.nextLine());
    return age;
}

public static void  displayInfo(String name, int age){
    System.out.println("Nice to meet you " + name);
    System.out.println (name + " You are " + age + " years old");
}

public static void  main(String[] args) {
    welcome();
    String name = getName();
    int age = getAge();
    displayInfo(name, age);
}
}

```

Methods make your program more organized and manageable to use. Methods have many different purposes. Methods can receive values, return values, receive and return values or receive or return nothing. A method definition is as follows:

access_modifier non-access_modifier return_datatype method_name (parameter_list)

parameter list = data_type parameter_name [,data_type parameter_name]

Access modifiers allow who can access the method. In this lesson we were introduced to the **public** and **private** access modifiers. Non-access modifiers indicate how the method can be used. In this lesson we used the **static** non-access modifier. Static is permanent code or value is placed directly in computer memory when the program is loaded and is readily available to be used. Methods return values using the **return** statement and receive values through the **parameter list**. The data type specifies what kind of data is returned or received. In this lesson we were introduced to the **int**, **float**, **double** and **String** data types. String is user data type and is actually a class.

A class allows you to make your own data types. In this situation the String class has been predefined for you so that you can use it right away. The welcome method just prints a statement and receives no values or returns no value. The **void** data type indicates no value is returned or received.

```
public static void welcome(){  
    System.out.println("Hello World");  
}
```

The getName() and getAge() methods both return a value using the return statement. The getName() method returns a **String** value whereas the enterAge method returns an **int** value.

```
public static String enterName(){  
    System.out.print("Please type in your name: ");  
    String name = kybd.nextLine();  
    return name;  
}
```

```
public static int enterAge(){  
    System.out.print("How old are you? ");  
    int age = Integer.parseInt(kybd.nextLine());  
    return age;  
}
```

The **displayInfo** function receives a name and age value to print out, but returns no value. The **displayInfo** method receives the name and age through the parameter list.

```
public static void displayInfo(String name, int age){  
    System.out.println("Nice to meet you " + name);  
    System.out.println(name + " You are " + age + " years old");  
}
```

The **name** and **age** inside the round brackets of the **displayInfo** method definition statement are known as **parameters** and contain values to be used by the method. The parameters just store values from the calling function and are not the same variables that are in the calling function. Although the parameter names and values may be same as in the calling function variable names, but they are different memory locations. The main purpose of the parameters is to receive values for the methods. The main method call's the preceding methods to run them and store the values in variables and pass the stored variable values to the methods. Calling a method means to execute the method. The values that are passed to the called method from the calling method is known as **arguments**.

Variables inside a method are known as **local variables** and are known to that method only. Name and age are local variables in the main function but are also arguments to the **displayInfo** method.

```
public static void main(String[] args) {  
    welcome();  
    String name = getName();  
    int age = getAge();  
    printDetails(name, age);  
}
```

Our class conveniently store the variable to our keyboard reader.

```
private static Scanner kybd = new Scanner(System.in);
```

This is one of the features of a class to store variable that can be used by its own methods. If not for this feature each method will have to have its own keyboard which would be a waste of code. Duplication of code is to be avoided in programming. There are additional things you need to know about this sample program We have used the keywords **public** and **private**. If we have code to be used by others we use the access modifier **public**. For variables and methods to be used only by our class, we use the access modifier **private**. The other important concept in the key word **static**. Static means code that can be used right away with creating an object first. It is also said to be at the class level not at the object level. A static variable can be shared between many class objects.

The static variable is just created once and will have the same value in each object. There is lots on controversy how Java programs are to be written. The rule I use is a class is just used once or does not have any non-reusable variables, then all methods and variables should be **static** else it should be a object.

In our sample program the main purpose of our sample program is to gather and print out information therefore all variables and methods should be static. Static means the compiler will generate the memory for the variable's and code for our class and make it readily available to be used. This is what we want.

It's now time to comment you program All programs need to be commented so that the user knows what the program is about. Just by reading the comments in your program you will know exactly what the program is supposed to do. We have two types of comments in Java. Header comments that are at the start of a program or a method. They start with `/*` and end with a `*/` and can span multiple lines like this.

```
/*  
  Program to read a name and age from a user and  
  print the details to the screen  
*/
```

Other comments are for one line only and explain what the current or proceeding program statement it is to do. The one-line comment starts with a `//` like this:

```
// method to read a name from the key board are return the value
```

We now comment the program as follow. Please add all the following comments to your program.

```

/*
Program to read a name and age from a user and print
the details on the screen
*/
import java.util.Scanner;

public class Lesson2 {
    // make keyboard reader
    private static Scanner kybd = new Scanner(System.in);

    // method to print a welcome message
    public static void welcome(){
        System.out.println("Hello World");
    }

    // method to read a name from the key board are return the value
    public static String enterName(){
        System.out.print("Please type in your name: ");
        String name = kybd.nextLine();
        return name;
    }

    // method to read an age from the key board are return the value
    public static int enterAge(){
        System.out.print("How old are you? ");
        int age = Integer.parseInt(kybd.nextLine());
        return age;
    }

    // method to print out a user name and age
    public static void displayInfo(String name, int age){
        System.out.println("Nice to meet you " + name);
        System.out.println(name + " You are " + age + " years old");
    }

    // main method to run program
    public static void main(String[] args) {
        welcome();
        String name = getName();
        int age = getAge();
        displayInfo(name, age);
    }
}

```

Lesson 2 Homework

Take your homework program from Lesson 1 and add methods to it. Make methods welcome, enterTitle, enterSalary, displayInfo. Call all these methods from the main method. Call your java program Homework2.java and class Homework2.

LESSON 3 CLASSES

Classes represent another level in program organization. They represent programming units that contain variables to store values and methods to do operations on these variables. This concept is known as **Object Oriented Programming**, and is a very powerful concept. It allows these programming units to be used over again in other programs. The main benefit of a class is to store values and do operations on them transparent from the user of the class. It is very convenient for the programmers to use classes. They are like building blocks that are used to create many sophisticated programs with little effort.

A class starts with the keyword **class** and the class name. The class uses another keyword **this** that indicates which variables and functions belong to this class. We have already used a class in our previous lessons. We will now write a Person class that has variables to store a name and age and methods to do operations on them. Like initializing retrieval, assignment and output. Make a new Java class file called Person.java, and type the following code into it.

```
/*
  Person Class to store a person's name and age
  A main function to read a name and age from a user and print
  the details on the screen using the Person class
*/

// define a class Person
public class Person {
    private String name;
    private int age;

    // default constructor
    Public Person()
    {
        this.name = "";
        this.age = 0;
    }
}
```

```

// initialize Person
public Person(String name, int age){
    this.name = name;
    this.age = age;
}

// return name
public String getName(){
    return this.name;
}

// return age
public int getAge(){
    return this.age;
}

// assign name
public void setName(String name){
    this.name = name;
}

// assign age
public void setAge(int age){
    this.age = age;
}

// return person info as a string
public String toString() {
    String sout = "Nice to meet you " + this.name + "\n";
    sout += this.name + " You are " + this.age + " years old";
    return sout;
}
}

```

The **Person** class definition starts with the class key word and class name **Person**. We use the **public** access modifier because we want others to use our class.

```
public class Person{
```

Our Person class has 2 **private** variables to store person name and age.

```
private String name;  
private int age;
```

We make the variables **private** because we want them to be only access by our class methods, nobody else. A class contains a **Person** method that initializes the class. This **Person** methods are known as a **constructor**. The constructor has the same name as the class. There are many variations of a constructor. A constructor with no parameters is known as a default constructor. It is called a default constructor because the variables defined in the class will be assigned default values.

```
// default constructor  
Public Person ()  
{  
    this.name = "";  
    this.age = 0;  
}
```

Another constructor is known as an initializing constructor, because it assigns values to the variables defined in the class.

```
// initialize Person  
public Person(String name, int age){  
    this.name = name;  
    this.age = age;  
}
```

The programming statements inside the constructor assign values to the variables name and age from the parameters name and age.

```
this.name = name  
this.age = age
```

The keyword **this** specifies which variables belongs to the Person class. The parameter name and age just store values to be assigned to the Person class variables and are not the same ones in the Person class.

The get functions also known as **getters** and just return values of the variables stored in the Person class. Again, you notice the **this** keyword.

```
// return name  
public String getName(){  
    return this.name;  
}  
  
// return age  
public int getAge(){  
    return this.age  
}
```

We also have **set** functions known as **setters** or **mutators** that allow the user of the class to assign new values to the Person class variables.

```
// assign name  
public void setName(String name);  
    this.name = name;  
}  
  
// assign age  
public void setAge(int age);  
    this.age = age  
}
```

You will notice each method have a parameter to assign the name or age value. Again, the **this** keyword distinguishes the person variables from the parameters since they both have the same names.

All classes should have a **toString()** function so that it can easily return the class info as a string message.

```
// return person info as a string
public String toString() {
    String sout = "Nice to meet you " + this.name + "\n";
    sout += this.name + " You are " + this.age + " years old";
    return sout;
}
```

Notice we have no print statement in our **toString()** method. We assign information to the local variable **sout** and return's the sout value. A local variable is just known to the function it resides in. The sout variable uses the + operator to join values together as a message value. This class definition must not contain any input or output statements. A class must be a reusable program unit not dependent on any input or output print statements. The purpose of the class is to contain information that can be easily accessed. Therefore, our main function must provide all the input and output print statements. We will use the input and output method from our previous program.

Make a new file called Lesson3.java and type in the following code:

```
import java.util.Scanner;

public class Lesson3 {

    // make keyboard reader
    private static Scanner kybd = new Scanner(System.in);

    // method to print a welcome message
    public static void welcome(){
        System.out.println("Hello World");
    }
}
```

```

// method to read a name from the key board are return the value
public static String enterName(){
    System.out.print("Please type in your name: ");
    String name = kybd.nextLine();
    return name;
}

// method to read an age from the key board are return the value
public static int enterAge(){

    System.out.print("How old are you? ");
    int age = Integer.parseInt(kybd.nextLine());
    return age;
}

// main method to run program
public static void main(String[] args) {

    // print welcome message
    welcome();

    // get person info from keyboard
    String name = enterName();
    int age = enterAge();
    // make a Person object
    Person p = new Person(name, age);

    // print out person details
    System.out.println(p.toString());
}
}

```

Notice we create the Person object with the following statement:

```
Person p = new Person(name, age);
```

This calls the **Person** constructor of the person class to create the person object and initialized with the values name and age. The mechanism that allocates memory in the computer for the variables and method's defined in the class, is known as **instantiation**.

When a class is instantiated in computer memory it is known as an **object**. when a class is written in a program then it is still known as a class not an object.

Objects are made from class definitions, Just the same way many house objects are built from house drawing plans. The print statement calls **toString()** method to print out the Person information details.

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

You can automatically call the **toString()** method just by using the Person variable

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

Run the Lesson3.java program. You will get the same output as the previous Lesson2 program. You should get something like this:

```
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old
```

Lesson 3 Homework Question 1:

Make a Profession class that stores somebody's profession **title** and amount of **money** they make. Make default and initializing constructors, getters and setters and a toString methods. Call your class Profession and put your class in a file called Profession.java. Make a java program file that has a main method that instantiates a Profession class. The main method has additional methods to get the professions title and amount of money that they make from the keyboard. You can use some of the methods from Lesson2. Call your java main program file Homework3.java.

INHERITANCE

The beauty of classes is that they can be extended to increase their functionality. We can make a Student class that uses the public variables and methods from the Person class. This is known as **inheritance**.

The Student class can only access the non-private variables and methods from the Person class. We have additional access modifiers public, protected, package and private.

access modifier	Description	example
Public	Access by anybody	public int age;
Protected	Access by derived class	Protected int age;
Package	Access by classes in project	int age;
Private	Access by its own class only	private int age;

A Student class will have an additional variable called **idnum** that will represent a string student id number. Using inheritance, the student class will be able to use the public variables and functions of the Person class. The Person class is known as the **super** class and the Student class is known as the **derived** class. The Person class knows nothing about the Student class where as the Student class knows all about the Person class.

Create a new Java file called Student.java. Create a class called Student that inherits from the Person class using this statement.

```
// define a class Student that inherits the Person class  
public class Student extends Person {
```

The **extend** keyword is used to define the inheritance relationship. This means the Student class can use the public variables and methods from the Person class. We need to define a student id number variable for the Student class.

```
// student id number  
private String idnum;
```


Now make a default constructor that will initialize the variable in the classes to default values.

```
// initialize Student to default values  
public Student(){  
    super();  
    this.idnum = "";  
}
```

We call the `super()` method that calls the default constructor of the `Person` class. This way the variables defined in the `Person` class also gets initialized to default values.

Now make a constructor that will initialize the student name, age and idnum.

```
// initialize Student  
public Student(String name, int age, String idnum)  
    super(name, age);  
    this.idnum = idnum;  
}
```

The `super` method calls the constructor of the `Person` class to create a `Person` object and transfer the name and age values from the parameters name and age. The idnum is initialized in the `Student` constructor.

The `getID` and `setID` getters and setters would be like this:.

```
// return idnum  
public String getID(){  
    return this.idnum;  
}  
  
// assign idnum  
public void setID(String idnum){  
    this.idnum = idnum;  
}
```

The last thing you need to make the **toString()** method. By using the **super()** method you can call functions directly from the super Person class inside the Student derived class. Here is the Student **toString()** method

```
// return student info as a string
public String toString() {
    String sout = super.toString() + "\n";
    sout += " Your student id number is " + this.idnum;
    return sout;
}
```

Once you got the Student class made then add programming statements to the Lessons3.java file to obtain a student name, age and idnum. You will have to make an additional getID() function to obtain a student id number from the key board. Make a student object and use the obtained name, age and idnum values from the methods of the Lesson3 class, then print out the student details. You should get something like this:

```
Hello World
Please type in your name: Tom
How old are you? 24
Nice to meet you Tom
Tom You are 24 years old
Please type in your name: Bill
How old are you? 18
What is your student ID? S1234
Nice to meet you Bill
Bill You are 18 years old Your student id number is S1234
```

Next use the getter methods to obtain the name and age from Person object p and assign the name and age to the Student object s using the setter methods. Give the student object a new id using the id setter method. Print out the student object.

Lesson 3 Homework Question 2:

Make a Payroll class that inherits your previous homework Profession class that stores a boolean or a String value to indicate if a worker works full time or part time. Use your java main program file Homework3.java from Question 2.

private boolean fullTime;

You will need default and initializing constructors, getters, setters and a toString methods. The initializing constructor would receive a profession title, salary and a boolean indicating full time or part time, A true value would indicate full time employment, where false value would indicate part time employment.

For the part time variable getter use:

public boolean isFullTime();

For the part time variable setter use:

public void setFullTime(boolean partTime);

Store the Payroll class in a java file called Payroll.java.

In the main method of your Homework3.java program make an additional **public static boolean enterFullTime()** method asking if a worker is full time or part time, that returns a Boolean true or false value. If they enter 'Y' return true, if they enter 'N' return false.

boolean fullTime = enterFullTime();

Then instantiate a Payroll class object and pass this **fullTime** value to your Payroll class constructor as well as the profession and salary of the worker.

Use the Payroll class toString method to print out the workers profession, pay amount and indicate if it is a full time or part time worker.

You can still use file Homework3.java.

Lesson 4 Operators

Operators

Operators do operations on variables like addition + , subtraction – and comparisons > etc. We now present all the Java operators with examples. Make a java class with a main method called Lesson4.java and try each operator example one by one. Type in the examples then run the program to see what it does.

Unary Operators

Unary operator act upon a single variable, adds or subtracts a number from 0.

```
int x = -5; // x = 0 - 5
int y = +5; // x = 0 + 5
```

```
System.out.println("-5 =" + x);
System.out.println("+5 =" + y);
```

Operator	Description	Example	Result
+	Adds a number to 0	int x = +5;	5
-	Subtracts a number from 0	int y = -5	-5

Arithmetic Operators

Arithmetic operators are used to do operations on numbers like addition and subtraction.

You can type in the operation right inside the System.out.println() statement just like this System.out.println(3+2) or System.out.println(3 > 2). Alternatively, you can use variables instead.

```
//arithmetic operators
int x = 5;
int y = 2
System.out.println(x + "+" + y + "=" + (x + y));
```

Operator	Description	Example	Result
+	Add two operands	5 + 2	7
-	Subtract right operand from the left	5 - 2	3
*	Multiply two operands	5 * 2	6
/	Divide left operand by the right one	5 / 2	2
%	Modulus - remainder of the division of left operand by the right	5 % 2	3

Note: * and / have precedence over + and -

Comparison Operators

Comparison operators are used to compare values. Compound operators and values are known as conditions. It either returns true or false according to the condition. True and false variables and values are known as Boolean values.

Operator	Description	Example	Result
>	Greater than - true if left operand is greater than the right	5 > 3	true
<	Less than - true if left operand is less than the right	5 < 3	false
==	Equal to - true if both operands are equal	5 == 5	true
!=	Not equal to - true if operands are not equal	5 != 5	true
>=	Greater than or equal to - true if left operand is greater than or equal to the right	5 >= 3	true
<=	Less than or equal to - true if left operand is less than or equal to the right	5 <= 3	true

```
x = 5;
```

```
y = 3;
```

```
System.out.println(x + ">" + y + "=" + (x > y)); // 5 > 3 = true
```

Logical Operators

Logical operators are the **and**, **or**, **not** boolean operators.

Operator	Description	Example	Result
&&	true if both the operands are true	true && true	true
	true if either of the operands is true	true false	true
!	true if operand is false (complements the operand)	! false	true

Logical operators only work on true and false values.

System.out.println(true && true); // prints true

```
boolean bx = false;  
boolean by = true;
```

```
System.out.println(x + "&&" + y + "=" + (x && y)); // false && true = false  
System.out.println(x + "||" + y + "=" + (x || y)); // false || true = true
```

```
System.out.println("! " + y + "=" + (! y)); // ! true = false
```

&& means both must be true to be true

|| means either can be true to be true

! means opposite

Compound Conditions

Logical operators are combined with Conditional operators to give true or false result values.

condition logical operator condition

where as a condition is:

value conditional operator value

Here are some examples you can try out:

```
int x = 5;
int y = 3
System.out.println(x > y && x < y); // prints out true
System.out.println(x > y || x < y); // prints out true
```

Note: || has precedence over && precedence means what is done first.

```
System.out.println((x > y && x < y) || (x < y && x != y));
```

Use brackets to force precedence, the operations inside the round brackets are performed first.

Todo:

Make your own compound conditions

Bitwise Operators

Bitwise operators act on operands as if they were binary digits. It operates bit by bit. Binary numbers are base 2 and contain only 0 and 1's. Every decimal number has a binary equivalent. Every binary number has a decimal equivalent. For example, decimal 2 is 0010 in binary and decimal 7 is binary 0111.

In the table below: 10 = (0000 1010 in binary) 4 = (0000 0100 in binary)

Operator	Description	Example	Result
&	Bitwise AND	10 & 4	0 (0000 0000 in binary)
	Bitwise OR	10 4	14 (0000 1110 in binary)
^	Bitwise XOR	10 ^ 4	14 (0000 1110 in binary)
~	Bitwise NOT (flip bits)	~10	-11 (1111 0101 in binary)

```
System.out.println(10 | 4); // prints out 14
```

```
System.out.println(Integer.toString(10 | 4, 2) // prints binary 1110
```

You may want to use variables values like 0 and 1 instead like this:

```
x = 0
```

```
y = 1
```

```
System.out.println(x & y); // 0
```

Using 0 and 1's rather than numbers make the bitwise operations easier to understand:

AND &	OR	XOR ^	Complement ~
0 & 0 = 0	0 0 = 0	0 ^ 0 = 0	~0 = 1
0 & 1 = 0	0 1 = 1	0 ^ 1 = 1	~1 = 0
1 & 0 = 0	1 0 = 1	1 ^ 0 = 1	
1 & 1 = 1	1 1 = 1	1 ^ 1 = 0	

For complement:

(you need to use &1 just to get last bit. example: `x = ~x & 1`)

Shift Operators

Multiply and divide by powers of 2

<<	left shift	10 << 2	40 (0010 1000 in binary)
>>	signed right shift	10 >> 2	2 (0000 0010 in binary)
>>>	unsigned right shift	10 >>> 2	2 (0000 0010 in binary)

```
int x = 5;
```

```
System.out.println("x = " + x); // 5
```

```
// left shift operator
```

```
x = x << 3; // multiplies by powers of 2
```

```
System.out.println("x = " + x); // 40 because 5 * 2 * 2 * 2 = 40 (5 * 8) = 40
```


// right shift operator

x = x >> 5; // divides by powers of 2

System.out.println("x = " + x); // 5 because $40 / 2 / 2 / 2 = 5$ ($40/8$) = 5

Increment/Decrement Operators ++ --

Increment operators **++** increment a variable value by 1 and decrement operators **--** decrement a value by 1.

They come in two versions, **prefix** increment/decrement value **before** or **postfix** increment/decrement value **after**.

Prefix Increment before ++x

x is incremented then value of y is assigned the value of x

// pre increment operator

int x = 5;

System.out.println("x = " + x);

int y = ++x;

System.out.println("x = " + x + " y = " + y); // x = 5 y = 5

	x	y
	-----	-----
int x = 5	5	?
y = ++x	6	6

x increments first then y gets the value of x

post increment after x++

The value of y is assigned the value of x and then x is incremented

```
// post increment operator
int x = 5;
System.out.println("x = " + x);
int y = ++x;
System.out.println("x = " + x + " y = " + y); // x = 6 y = 5
```

	x	y
	-----	-----
int x = 5	5	?
y = x++	6	5

y get's the value of x first then x increments

prefix Decrement before --x

x is decremented then value of y is assigned the value of x

```
// pre decrement operator
int x = 5;
System.out.println("x = " + x);
int y = --x;
System.out.println("x = " + x + " y = " + y); // x = 4 y = 4
```

	x	y
	-----	-----
int x = 5	5	?
y = --x	4	4

x decrements first then y get's the value of x

postfix decrement after y = x--

The value of y is assigned the value of x and then x is decremented

```
// post decrement operator
int x = 5;
System.out.println("x = " + x);
int y = x--;
System.out.println("x = " + x + " y = " + y); // x = 4 y = 5
```

	x	y
	-----	-----
int x = 5	5	?
y = x--	4	5

y get's the value of x first then x decrements

Increment decrement operators are usually used stand alone to increment or decrement a variable value by 1 using:

```
x++;
```

```
x--;
```

Assignment Operators

Assignment operators are used to assign values to variables.

x = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left. There are various compound operators in Java like x += 5 that adds to the variable and later assigns the same. It is equivalent to x = x + 5.

```
x=5;
```

```
System.out.println("x = " + x );
```

```
x += 5; // 10
```

```
System.out.println("x += 5 = " + x );
```

Operator	Assignment	Equivalent	Result
=	x = 5	x = 5	5
+=	x += 5	x = x + 5	10
-=	x -= 5	x = x - 5	5
*=	x *= 5	x = x * 5	25
/=	x /= 5	x = x / 5	5
%=	x %= 5	x = x % 5	0

&=	x &= 5	x = x & 5	0
 =	x = 5	x = x 5	5
^=	x ^= 1	x = x ^ 5	4
<<=	x <<= 5	x = x << 2	16
>>=	x >>= 5	x = x >> 2	4

String Operators

String operators operate on String objects, String objects contain string data and are **immutable** meaning they cannot be changed internally.

There are many string operation's, most of them are method calls. Here are just a few of them:

```
// declare and assign String
String s1 = "hello";
String s2 = "goodbye";

// join two strings together
String s3 = s1 + s2;
System.out.println(s3); // hellogoodbye

// print out length of string
int len = s3.length();
System.out.println(len); // 10

// get a character from string
char c = s3.charAt(0);
System.out.println(c); // h

//get a first 5 characters sub string index 0 to 5-1
String s4 = s3.substring(0,5);
System.out.println (s4); // hello

// print out characters from index 5
String s4 = s3.substring(5);
System.out.println (s4); // there
```

```

// add a character to a string
String s5 = s3.substring(0,5)+ 'X' + s3.substring(5);
System.out.println (s5); // helloXthere

// make string lower case
String s6 = s5.toLowerCase();
System.out.println (s6); // helloxthere

// make string upper case
String s7 = s5.toUpperCase();
System.out.println (s7); // HELLOX THERE

// test if 2 strings are equal
System.out.println(s1.equals(s2)); // false

// test if 2 strings are equal but ignore case
System.out.println(s1.equalsIgnoreCase(s2)); // false

// test if 2 strings have the same memory location
System.out.println(s1 == s2)); // true

S2 = new String("there"); // make new memory location
System.out.println(s1 == s2)); // false

// test if 2 strings are less greater or equal
// -1 = less 0 = equal 1 = greater
System.out.println(s1.compareTo( s2)); // -1

```

Primitive data types

Primitive data types in java are **char**, **int**, **long**, **float**, **double** and **boolean**. Primitive data types just store values in memory but have no methods to do operations and just rely on operators like +, -, *, /, &. | etc. To do operations on values using methods every primitive data type in Java has a corresponding class, there are called **Wrapper** classes because they wrap a primitive data type inside it.

Example **int** has the corresponding wrapper class **Integer**. These corresponding class are mainly used to do conversions from a numeric value to String and a String to a numeric value.

To convert from a string to a int

```
Int x = Integer.parseInt("1234")
```

To convert from a int to a string we use the **valueOf** method from the String class.

```
String s = String.valueOf(x)
```

Here are all the primitive data types and corresponding wrapper classes.

Primitive data type	Corresponding class	Conversion from string
char	Character	"A".charAt(0);
short	Short	Short.parseShort("12")
int	Integer	Integer.parseInt("1234")
float	Float	Float.parseFloat("10.5")
double	Double	Double.parseDouble("10.765")
boolean	Boolean	Boolean.parseBoolean(true)

In your Lesson4.java file try out all the primitive data types and see how they work. Convert strings to primitive data types and the primitive data types back to strings, like this:

```
int i = Integer.parseInt("1234");  
System.out.println(String.valueOf(i));
```

These wrapper classes have their own constructors to take in primitive values and an **Equals** method to compare values for equality

```
Integer i1 = new Integer(5)  
Integer i2 = new Integer(10)  
System.out.println(i1.equals(i2)) // false
```

There is also a **compareTo** method to compare values. If value is less then returns a negative number, if greater return a positive number and if equals returns a 0.

```
System.out.println(i.compareTo(i2)); // -1 less than  
System.out.println(i2.compareTo(i)); // 1 greater than  
System.out.println(i.compareTo(i)); // 0 equal
```

These classes also do automatic conversions for you, this is called boxing. You can convert a primitive to its class and a class back to its primitive.

```
Integer i3 = 5;  
System.out.println(i3); // 5  
int x3 = i3;  
System.out.println(x3); // 5
```

LESSON 4 HOMEWORK

1. Print out if a number is even, using just a print statement and a arithmetic operator
2. Print out if a number is odd, using just a print statement and a arithmetic operator
3. Swap 2 number using a temporary variable
4. Swap 2 numbers using operators + and -
5. Multiply a number by 8 using a shift operator
6. Divide a number by 8 using a shift operator
7. Using a print statement, add 2 numbers together and check if they are less than multiplying them together

8. Using a print statement, add 2 numbers together and check if they are less than multiplying them together and greater then multiplying them together.
9. Using a print statement, add 2 numbers together and check if they are less than multiplying them together or greater then multiplying them together
10. Make a string of your favourite word and replace the first letter with another letter, hint use substring.
Example : change "hello" to "jello"
11. Make a string of your favourite word and replace the last letter with another letter, hint use substring.
Example : change "jello" to "jelly"
12. Make a string of your favourite word and remove the middle letter, hint use substring.
Example : change "jelly" to "jely"
13. Make a string of your favourite word, Split it in the middle, make the first part lower case and the second part upper case, hint use substring.
Example : change "jely" to "jeLY"
14. Using substring replace the last letter with the first letter in a word
Example : change "jeLY" to "YeLj"
15. Split the word in the middle put the last part at the first and the first part at the last
Example : change "YeLj" to "LjYe"

Call your Java file homework4.java

LESSON 5 PROGRAMMING STATEMENTS

Programming statements allow you to write complete Java Program. We have already looked at simple input, print and assignment statements. We now present you with branch and loop programming statements. Start a new Java class Lesson5.java to test all these programming statements.

Branch Control Statements

Branch control statements allow certain program statements to execute and other not.

if statement

The **if** branch **control** statements use conditional operators from the previous lesson to direct program flow.

If (condition)
Statement(s)

When the condition is evaluated to be true the statements belonging to the if statement execute. An if statement is a one-way branch operation.

```
// if statement
x = 5;
if (x == 5)
{
    System.out.println("x is 5");
}
```

x is 5
I like Java Programming

else-if statement

We now add an else statement to our if statement. An if-else control construct is a two-way branch operation.

```
If (condition)  
    statements  
else  
    statements  
// if – else statement  
x = 2;  
if (x == 5)  
{  
    System.out.println ("x is 5");  
}  
else  
{  
    System.out.println ("x is not 5");  
}
```

x is not 5
I like Java Programming

We can also have extra else if statements to make a multi-branch.

```
// multi if else  
x = 10;  
if (x == 5)  
    System.out.println ("x is 5");  
else if (x < 5)  
    System.out.println ("x less than 5");  
else if (x > 5)  
    System.out.println ("x greater than 5");  
    System.out.println ("I like Java Programming")
```

x greater than 5
I like Java Programming

Our multi branch **if-else** can also end with an **else** statement.

```
// multi if-else else
x = 5;
if (x < 5)
    System.out.println ("x less than 5");
else if (x > 5)
    System.out.println("x greater than 5");

else
    System.out.println("x is 5");
System.out.println("I like Java Programming");
```

x is 5
I like Java Programming

nested if statements

if statements can also be nested to make complicated conditions simpler.

```
// nested if statement
x = 5;
if (x >= 0)
{
    if (x > 5)
        System.out.println ("x greater than 5");
    else
        System.out.println ("x less than equal 5");
}
```

x less than equal 5
I like Java Programming

To do:

Add an else if statement to the inner if statement

Add an else statement to the first if statement.

switch statement

A switch statement is an organized if-else statement. It is a little limited since it can only handle equal values. Switch statements work on **char**, **int**, **float** and **String**.

```
// switch statement
x = 2;
switch(x)
{
    case 1:
        System.out.println("x is 1");
        break;
    case 2:
        System.out.println("x is 2");
        Break;
    case 3:
        System.out.println("x is 3");
        break;
    default:
        System.out.println("x is " + x);
        break;
}
```

x is 2

The **break** statements are optional, their purpose is to stop execution to the next **case** statement. Take them out and see what happens.

Loop Control Statements

Loop control statements allow program statements to repeat themselves.

while loop

The while loop allows you to repeat programming statements repeatedly until some condition is satisfied. While loops are used when you do not know how many items you have.

The while loop requires an initialized counter, a condition, program statements and then increment or decrement a counter.

```
Initialize counter  
while condition:  
    statement(s)  
    increment/decrement counter
```

Here is a while loop you can try out that prints out the number 0 to 4

```
// while loop  
x = 0;  
while (x <5)  
{  
    System.out.println(x);  
    x++;  
}
```

```
0  
1  
2  
3  
4
```

To do

Change the above while loop to print out the numbers 1 to 5.
Make a while loop that prints out numbers 5 to 1 backwards.

do loop

The do loop also known as a **do-while** loop allows you to repeat programming statements repeatedly until some condition is satisfied. The condition is at the end of the loop, so the programming statements execute at least once.

The do loop requires an initialized counter, program statements, increment or decrement a counter and finally a condition.

```
Initialize counter  
do{  
    statement(s)  
    increment/decrement counter  
}  
    while condition;
```

When the condition is false the loop execution exits. **do loops** are used when you want to execute a loop statement at least once and also when you do not know how many items you have.

Here is a do loop you can try out that prints out the number 0 to 4

```
// do loop  
x = 0;  
do  
{  
    System.out.println( x );  
    x++;  
} while (x <5);
```



```
0  
1  
2  
3  
4
```

To do

Change the above do loop to print out the numbers 1 to 5.
Make a do loop that prints out numbers 5 to 1 backwards.

for Loop

A for loop, is much more automatic then the while loop but more difficult to use. All loops must have counter mechanism. The for loop needs a start value, end condition, step value. For loops are used when you know how many items you have.

*for (start_value,end_condition, increment):
statement(s)*

Here we have a for loop to print out values 0 to 4

Note: We declare an index variable int i=0 inside the for loop for convenience. The index variable i is only known inside the for loop.

```
// for loop
for (int i=0;i<5;i++)
{
    System.out.println (i);
}
```

```
0
1
2
3
4
```

To do

Change the above for loop to print out the numbers 1 to 5.

Make a for loop that prints out numbers 5 to 1 backwards

Skill Testing Questions:

What loop do you use when you do not know how many items you have?

while loop

What loop do you use when you know how many items you have?

for loop

What loop do you use that executes the loop statements at least once?

do loop

Nested for loops

Nested for loops are used to print out values by rows and columns.

```
// nested for loop
for (int r=0;r<5;r++)
{
    System.out.println(r + " : ");
    for (int c =0; c < 5; c++)
        System.out.print( c + " ")
    System.out.println ("")
}
```

```
0 : 0 1 2 3 4
1 : 0 1 2 3 4
2 : 0 1 2 3 4
3 : 0 1 2 3 4
4 : 0 1 2 3 4
```

Loops can also be used to print out characters in a string variable. We use the **length** method of the String class to get the length of the string. We use the **charAt** method from the String class to get each individual character from the string accessed by index.

```
// print out characters in a string
s = "Hello";
for (int i=0;i<s.length();i++)
    System.out.println(s.charAt(i));
System.out.println("");
```

```
H
e
l
l
o
```

To do:

Print string backwards using the **for** loop

LESSON5 HOMEWORK

Exam Grader

Ask someone to enter an exam mark between 0 and 100. If they enter 90 or above printout an "A", 80 or above print out a "B", 70 or above print out a "C", 60 or above print out a "D" and "F" if below 60. Hint: use if else statements.

You can visualize a grade chart like this:

Mark Range	Exam Grade
90 to 100	A
80 to 89	B
70 to 79	C
60 to 69	D
0 to 59	F

Mini Calculator

Make a mini calculator that takes two numbers and a operation like -, +, * and /.
Prompt to enter two number's and a operation like this:

Enter first number: 3
Enter second number: 4
Enter (+, -, *, /) operation: +

Then print out the answer like this:
3 + 4 = 7

Hint: use a switch statement.

Use a while or do while loop so that they can repeatedly enter many calculations.
Terminate the program when they enter a letter like 'X' for the first number.

Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

```
      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * *
```

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Enhanced Triangle Generator:

Use nested for loops to print out a triangle using '*' like this:

```
  *
 * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
```

Ask the user how many rows they want.

Hint: use 2 nested for loops, start with a square of stars

Reverse a String

Reverse a String using a **while** loop or a **for** loop. You will have to put the reversed characters in a second string, since you cannot in Java directly change the characters in the original String.

Test if a number is prime

Make a function called **isPrime(x)** that tests if a number is prime. In a loop divide the number between 2 to number-1 (or 2 to square root of number+1. For square root use:

```
x = (int)Math.sqrt(n);
```

If the number can be divided by any of the divisors then the number is not prime, else it is prime. Print out the first 100 prime numbers.

The first 10 prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29

Print out all factors of a number

Make a function call **factors(x)** that will print out all the factors of a number. The factors of a number is all the divisors divided by the number evenly.

Example:

The Factors of 50 are:

1
2
5
10
25
50

Print out all prime factors of a number

Make a function call **prime_factors(x)** that will print out all the prime factors of a number. The prime factors of a number is all the prime number divisors divided by the number evenly.

Example: $12 = 2 \times 2 \times 3$

Following are the steps to find all prime factors.

- 0) Enter a number n
- 1) While n is divisible by 2, print 2 and integer divide n by 2
- 2) In a **for** loop from i = 3 to square root of n + 1 increment by 2
 in a **while** loop while n is divisible by i
 print i
 integer divide n by integer i
- 3) print n if it is greater than 2.

For square root use:

x = Math.sqrt(n);

Make a Guessing game

Ask the user to guess a number between 1 and 100. If they guess too high tell them "Too High". If they guess too low tell them they guess "Too Low". If they guess correct tell them "Congratulations you are Correct! ". Keep track how many tries they took to guess. At the end of each game ask the user if they want to play another game. After all games have been played print out the average number of tries for all games played. Store the sum of the tries per game and divide by the number of games played. Play up to a maximum of 10 games.

Make a java class called **GuessingGame** with a main function.

Use a constant for the maximum number to guess 100.

```
public static final int MAX_NUMBER = 100;
```

Place the constant right after you declare your class.

You will also need to generate random numbers to guess.
You can use this code to generate a random number:

```
Random rgen = new Random();  
int number = rgen.nextInt(MAX_NUMBER )+ 1;
```

Where **MAX_NUMBER** is a constant to represent the maximum number to guess 100

You will also need to includes this at the top of your program:

```
import java.util.Random;
```

to use the random number generator.

Also use a constant for the maximum number of tries to play 10.

```
public static final int MAX_TRIES = 10;
```

You should have functions to print a welcome message, explaining how to play the game, generate a random number, get a guess from the keyboard, check if a guess is correct and print out the number of tries per game and average score for all games. The main function should just call your functions in a loop. Call your java file GuessingGame.java.

Object Oriented Guessing Game

Make a **GuessGame** class that store the random number to guess, and the number of tries to guess the random number. The constructor should generate the random number, and you should have a method to check if a guess is correct, too low or too high. Return a 0 if they are correct a negative number if too low and a positive number if too high. Also make a method to return the number of tries. The main function should just handle inputs from the keyboard and printing output to the console. The GuessGame class should not handle any input and output, and is used, mainly to store data.

The main function would instantiate a new **GuessingGame** object per game. The guessing game class should also have a static variable to store the sum of the tries per game and the number of games.

```
public static int TotalTries = 0;  
public static int NumGames = 0;
```

Call your java file with the main function GuessingGame2.java

LESSON6 ARRAY, ARRAYLIST, LISTS, SETS and MAPS

ARRAYS

Arrays store many sequential values together. We have one dimensional arrays and multi dimensional arrays. One dimensional arrays are considered a single row of values having multiple column elements. You can visualize a one-dimensional array as follows.

Value1	Value2	Value3	Value4	Value5
--------	--------	--------	--------	--------

We declare and initialize 1 dimensional array as follows

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

1	2	3	4	5
---	---	---	---	---

To get the length of an array you use the **length** public variable of the array.

```
int length = a.length;  
System.out.length(length);
```

5

Arrays locations are retrieved by an index

```
int x = a[0];  
System.out.println(x);
```

1

We use for loops to print out values in array. Java does not print arrays automatically for you. We use the **length** variable from the array to get the length of the array and the square brackets and index [index] to retrieve a value at a specified index.

```
// print out array  
for (int i=0;i<a.length;i++)  
    System.out.print (a[i] + " ");  
System.out.println("");
```

1 2 3 4 5

We also use the square brackets [index] and index to assign a new value at an array index

```
a[0] = 8;
```



Specify which
element index
to access

We again print out array to see new value:

```
// print out array  
for (int i=0;i<a.length;i++)  
    System.out.print (a[i] + " ");  
System.out.println("");
```

8 2 3 4 5

Allocating memory for a 1 dimensional array

We use **new** keyword, the data type and a specified size to allocate memory for a one dimensional array . Here we allocate a one dimensional array with 5 elements.

```
int[] a2 = new int[5];
```



Specify size
of array

In Java all allocated array elements values are initialized to value 0. In this situation you need to assign values separately.

Arrays locations are assigned by an index. All indexes start at 0. Here we assign the numbers 1 to 5 to the array using indexes 0 to 4.

```
a2[0] = 1;
```

```

a2[1] = 2;
a2[2] = 3;
a2[3] = 4;
a2[4] = 5;

```

1	2	3	4	5
---	---	---	---	---

Print out array again

```

// print out array
for (int i=0;i<a.length;i++)
    System.out.print (a[i] + " ");
System.out.println("");

```

1 2 3 4 5

To do

Allocate and initialize a 1 dimensional array of size 5. Use a for loop to initialize and print out the values of the 1 dimensional array.
Use a formula like $a[i] = i*2 + 4$ to assign values to the array.

Two-dimensional arrays

Two-dimensional arrays are a grid of rows and columns. A 3 by 3 two-dimensional array is visualized as follows. The rows are horizontal and the columns are vertical.

1	2	3
4	5	6
7	8	9

Here we declare and initialize a two-dimensional array with the that were displayed in the grid.

```
int [][] b = {{1,2,3},{4,5,6},{7,8,9}};
```

Each row has columns values enclosed in curly {} brackets like {1,2,3}.

We use the length method of the two-dimensional array to get the number of rows.

```
int rows = n.length;  
System.out.println(rows); # 3
```

We use the length method of a specified row to get how many columns are in that row

```
int columns = b[i].length  
System.out.println(columns); # 3
```

The row index and column index of a two-dimensional array can be visualized as follows. The row index is first and the column index second. All indexes start at 0.

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

We retrieve values from the two-dimensional array by the row index and column index. The row index is specified first and the column index is specified second.

value = array_name [row index][column index];

```
x = b2[1][2];  
System.out.println(x);
```



We print the values of a two-dimensional array using nested for loops. The outer for loop prints out the rows the inner for loop prints out the columns.

Here we print out the two dimensional array using two nested for loops.

```
for (int r=0;r < b.length; r++)
{
    for (int c=0;c < b[r].length; c++)
    {
        System.out.println(b[r][c] + " ");
    }
    System.out.println("");
}
```

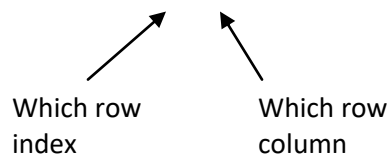
1 2 3
4 5 6
7 8 9

We assign values to the two-dimensional array by using row and column indexes. The row index is specified first and the column index is specified second.

array_name [row index][column index] = value;

Here we assign the number 11 to the array using row index 1 and column index 2.

b2[1][2] = 11;



Which row
index

Which row
column

Here we again print out the two dimensional array using two nested for loops.

```
for (int r=0;r < b.length; r++)
{
    for (int c=0;c < b[r].length; c++)
    {
        System.out.println(b[r][c] + " ");
    }
    System.out.println("");
}
```

1 2 3
4 5 11
7 8 9

There may be instances when the number of columns are different for each row.

```
// different column sizes  
int[][] b = {{1,2,3},{4,5,6,7},{8,9}};
```

to do:

Print out this two dimensional in nested **for** loops to see the different row lengths.

Allocating memory for a 2 dimensional array

We use **new** keyword the data type and the specified rows and columns to allocate memory for a two-dimensional array. Here we allocate a two dimensional array with 3 rows and 3 columns with default values of 0.

```
int[][] b2 = new int[3][3];
```

Number of rows Number of columns

To do

Allocate and initialize a 2 dimensional array of size 3 row and 3 columns. Use a for loop to initialize and print out the values of the 2 dimensional array. Use a formula like $a[i][j] = i*2 + 4 * j$;

LESSON 6 HOMEWORK Part1

Question 1

Make an array to store 5 numbers 1 to 5.

Swap two values in an array. Print array before and after using a for loop.

Question 2

Make an array of 10 numbers 1 to 10, print out the numbers in the array, then add up all the numbers and print out the sum.

Question 3

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Ask the user of your program to enter a number in the array. Search for the number in the array and report if it is found or not found.

Question 4

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Ask the user of your program to enter a number in the array. Search for the number in the array and report the array index where the number was found otherwise print -1 meaning no index found.

Question 5

Make an array of 10 numbers 1 to 10, print out the numbers in the array. Reverse all the numbers in the array in-place using a loop. Hint: use swap and 2 indexes i and j. Index i starts at the beginning of the array and index j starts at the end of the array. The i's increment and the j's decrement. Print out the reversed array.

Question 6

Make a 2 dimensional array of 3 rows and 3 columns. Fill the 2 dimensional array with numbers 1 to 9. Add up the sum of all rows, and print the sum at the end of each row. Add up the sums of all columns, and print the sums at the end of each column. Your output should look like this.

```
1 2 3 : 6
4 5 6 : 15
7 8 9 : 24
---
11 15 18
```

Question 7

Make an array to hold 10 numbers 1 to 10.
Generate 1000 random numbers between 1 and 10.
Keep track of the random numbers generated in your array.
Print out all the numbers and their counts from the array.
Print out the numbers with the smallest and largest count.
Print out the number of even and odd number counts.
You can make a random number like this:

```
int x = (int)(Math.random()*10) + 1;
```

Put all answers in a java file called Homework6.java

ARRAYLISTS

ArrayLists are expandable arrays. They are a little awkward to use but convenient. An ArrayList needs to know what type of data type it will use by specify the data type to be used inside **diamond brackets** <Integer>. The data type must be a object. We use the Integer class to represent the primitive **int** data type, You need to put the following import statements at the top of your Lesson4.java file so that the Java compiler knows about ArrayList.

```
import java.util.ArrayList;

// To create an empty array list
ArrayList<Integer> list1 = new ArrayList<Integer>();
```

```
// Add a value to a ArrayList
list1.add(1);
list1.add(2);
list1.add(3);
list1.add(4);
list1.add(5);
```

```
// Print out array list
System.out.println (list1) // [1,2,3,4,5];
```

When you use the ArrayList variable inside the System.out.println statement it is actually calling the ArrayList **toString()** method.

```
System.out.println (list1.toString())
```

```
[1, 2, 3, 4, 5]
```

Get the number of elements in an ArrayList

```
Int x = list1.size();
```

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

```
5
```

Get a value from the ArrayList at a specified location

```
x = list1.get(0);
System.out.println (x) ;
```

```
1
```

Remove a ArrayList from a list

```
list1.remove (new Integer(3));  
System.out.println (list1) ;
```

[1, 2, 4, 5]

We have to use a Integer object to remove the value because the ArrayList stores Integer objects not primitive data type int.

Remove an item by index

```
list1.remove(0);  
System.out.println (list1);
```

[2, 4, 5]

This remove method remove's a value at a specified index, not by value.

Note: It is a little confusing to have the same name method **remove** to remove by value or remove by Index.

Test if a value is in a ArrayList using the **contains** method. The **contains** method returns true or false

```
System.out.println (list1.contains(7)) // false
```

Set a value at a specified index

```
list1.set(2,8);  
System.out.println (list1) ;
```

Index 2 (4) gets the new value 8

[8, 4, 5]

Check if a value is in an ArrayList

```
boolean bin = list1.contains(4);  
System.out.println (list1) ;
```

true

HASHSETS

HashSets are like ArrayLists but store values internally by a calculated index. They only **store unique values**, no duplicates are allowed. To keep track of values in a HashSet, each value is stored in a calculated location. A HashSet need to know what type of data type it will use by specify the data type to be used inside triangle brackets <Integer>. You need to put the following import statements at the top of your Lesson4.java file so that the Java compiler knows about HashSet.

```
import java.util.HashSet;
```

make hashset

```
HashSet<Integer> set1 = new HashSet<Integer>();
```

add value to hash set

```
set1.add(5);
```

try to add duplicate value to hash set

```
set1.add(5);
```

try to add another duplicate value to hash set

```
set1.add(5);
```


print out hash set

```
System.out.println(set1);
```

[5]

Print out hash set size

```
System.out.println(set1.size());
```

1

Check if the set contains a certain value using the **contains** method. The contains method returns true if the set contains the item.

```
System.out.println(set1.contains(5));
```

true

remove a item by value from hash set

```
set1.remove(5);  
System.out.println(set1);
```

[]

Union of two sets using addAll function

```
Set<Integer> s3 = new Set<Integer> ();  
s3.add(1);  
s3.add(2);  
s3.add(3);
```

```
Set<Integer> s4= new Set<Integer> ();  
s4.add(5);  
s4.add(6);  
s4.add(7);
```

```
s3.addAll(s4);  
System.out.println(s3);
```

```
[1,2,3,4,5,6]
```

Intersection of two sets using retainAll() function

```
Set<Integer> s3 = new Set<Integer> ();  
s3.add(1);  
s3.add(2);  
s3.add(3);
```

```
Set<Integer> s4= new Set<Integer> ();  
s4.add(5);  
s4.add(2);  
s4.add(3);
```

```
s3.retainAll(s4);  
System.out.println(s3);
```

```
[2,3]
```

set of letters

Put all letters from a string into a set, and print out the set, and discover a new word.

```
String s2 = "tomorrow";  
System.out.println(s2);  
HashSet<Character> set2 = new HashSet<Character>();  
for (int i=0;i< s2.length();i++)  
    set2.add(s2.charAt(i));  
  
System.out.println(set2);
```

```
tomorrow  
[r, t, w, m, o]
```

Try lots of different word, don't stop until a new word is found.

HASHMAP

A **HashMap** associates a key with a value. A HashMap can have many keys and corresponding values. Think of a HashMap like a telephone book with the name as the key and the telephone number as the value. HashMap are analogous to Dictionaries in other programming languages. Values in a hash map are stored in a calculated location based on the key.

A HashMap needs to know what type of data type it will use, the data type is specified inside triangle brackets <String, String>. The first data type is for the key and the second data type is for the value.

You need to put the following import statements at the top of your Lesson4.java file so that the Java compiler knows about the HashMap.

```
import java.util.HashMap;
```

make empty map

```
HashMap<String,String> map1 = new HashMap<String,String>();
```

add key and values to HashMap

```
map1.put("name","Tom");  
map1.put("email","tom@mail.com");  
map1.put("phone","123-4567");
```

print out HashMap

```
System.out.println(map1);
```

```
[name=Tom, email=tom@mail.com, phone=123-4567]
```

Print out number of entries in HashMap

```
System.out.println(map1.size());
```

```
3
```

Get values From a HashMap for key and printout values.

```
System.out.println(map1.get("name"));  
System.out.println(map1.get("email"));  
System.out.println(map1.get("phone"));
```

```
Tom  
tom@mail.com  
123-4567
```

Print out keys and value pairs from a HashMap using **entrySet**

```
System.out.println(map1.entrySet());
```

```
[name=Tom, email=tom@mail.com, phone=123-4567]
```

Print keys of a HashMap using **keySet**

```
System.out.println(map1.keySet());
```

```
[name,email, phone]
```

print values of a HashMap using **values**

```
System.out.println(map1.values());
```

```
[Tom, tom@mail.com, 123-4567]
```

Test if a hash map contains a key

```
System.out.println(map1.containsKey("name") );
```

```
true
```

Test if a hash map contains value

```
System.out.println(map1.containsValue("Tom") );
```

```
true
```

Type all the above examples in your file lesson6.java and, make sure you get the same results.

LESSON 6 HOMEWORK Part2

Question 7

Make an ArrayList called animals of your 5 favorite animals like: cat, dog, tiger, monkey and mouse.

Make an ArrayList called sounds of your 5 favorite animals sounds: "meow", "bark", "roar", "ee ee ee" and "squeak squeak"

In a loop ask what sound each animal make?

EXAMPLE: What sound does a cat make?

Using the values in the sound ArrayList, Tell them if that are right or wrong, and keep track of the correct answers.

At the end of the program tell them their score.

Question 8

Make an ArrayList called list1 of your favorite animals like: elephant, cat and dog. Print out the list of animals.

Ask the user of your program to type in name of one of the animal names from your list, that they don't like.

Remove the animal from this list and put into another list called list2.

Then ask them to type in the name of an animal they do like. Add this name to list1 and to list2.

Then print out the animals list1 and list2

Question 9

From question5 put all the animals from animal list1 and list2 into a set called set1. Then take all the animals from list1 and list2 that are common between them and put into another set called set2. Print out both sets.

Question 10

Make a HashMap map1 of your favorite animal kinds like (cat, dog, tiger). Give each animal a name (Tom, Sally, Rudolf) . Use the animal name as the key and the animal kind as the value.

Example: fluffy cat

Then make another HashMap map2 , use the animal kind (cat, dog, tiger). as the key and the animal sound (meow,bark,roar) as the value.

Example: cat meow

Use the same animal kind's that were in the first HashMap.

Print out the keys ((Tom, Sally, Rudolf) of the first HashMap.

Ask the user to type in one of the animal names.

Get the animal kind ((cat, dog, tiger) from the first HashMap map1.

Print to the screen the name of the animal and what kind of animal it is like:

Fluffy is a cat

Ask the user what kind of sound a cat makes?

From the second HashMap get the sound that animal makes. If they guess the correct sound then tell them correct or tell them out what sound animal make's like: Cat's meow

Put your code in the same file Homework6.java

LESSON 7 OVERLOADING , OVERRIDING, INTERFACES AND GENERICS

Overloading

Overloading lets you have methods with the same name but has different parameter lists. Do not confuse **overloading** with **overriding** with. **Overriding** means same method name but same parameter list. Overloading means same method name but different parameter list signature. When you make different constructors all with different parameters this is **overloading** as follows:

```
// default constructor
public Person(){
    this.name = "";
    this.age = 0;
}

// initializing constructor
public Person(String name, int age){
    this.name = name;
    this.age = age;
}
```

The default constructor initialize to default values, the initializing constructor initializes to values passed to it,

copy constructor

A copy constructor copies the values from an existing object. The copy constructor initializes variables from a another known same class object and in our case would be another Person object.

```
// copy constructor
public Person(Person p){
    this.name = p.name;
    this.age = p.age;
}
```


copy constructor for a derived class.

In this situation the derived copy constructor passes the derived object to the super class by calling the super keyword. The derived object contains a super class object internally.

```
// derived class copy constructor
public Student(Student s){
    super(s);
    this.studentID = s.studentID;
}
```

Todo:

Add the copy constructor to your Person class and to your derived Student class from previous lessons. The Student derived class would have to pass the Student to the Person class using the **super** keyword.

Overriding

Overriding allows you to call the methods of a derived class over the methods of a super class. The derived method has the same name and parameter list signature of a super class method. You have all ready used an overridden method's in your Person and Student classes the **toString** method.

```
// return Person info as a string
public String toString() {
    String sout = "Nice to meet you " + this.name + "\n";
    sout += this.name + " You are " + this.age + " years old";
    return;
}
```

```
// return Student info as a string
public String toString() {
    String sout = super.toString() + "\n";
    sout += " Your student id number is " + this.idnum;
    return sout;
}
```

There is another important **overridden** method you should know about, the **equals** method. It returns true if two Objects are equal either by same memory location or optionally by value.

Object class equals method:

```
// compare two objects having the same memory address
public boolean equals(Object obj) {
    return (this == obj);
}
```

Right now it just compare's the address of two objects. You can overload it to compare if two object values are equal instead. We will make an equals method for the Person class to test if two Person objects have the same name and age.. We need to use the **instanceof** operator to test if the obj parameter is a Person Object. Inside the if statement we compare if two person objects names and ages are equal.

Person class equals method:

```
public boolean equals(Object obj)
{
    If obj != null)
    {
        if(obj instanceof Person)
        {
            Person p = (Person) obj;

            // return if person name and age are equal
            return name.equals(p.name) && age == p.age
        }
    }
    return false;
}
```

To do

Add an equals method to your Person class to test if two person names are equal.

Equals method with the class name as a parameter

For convenience you can also make an equals method that takes a Person parameter instead of an Object parameter. This equals method does not override the Object equals method, it is just for convenience only.

equals method using Person parameter:

```
public boolean equals(Person p)
{
    If(p != null)
        // return if person names are equal
        return name.equals(p.name) && age == p.age
    else return false;
}
```

This is a very interesting situation the first equals method **overrides** the equals method of the Object class (same method name same parameters). The two equals method the Object parameter equals method and the Person parameter equals method **overloads** each other. (Same method name different parameters).

To do

Add the Person parameter equals method to your Person class.

If the equals method is called with a Object argument then the equals method with the Object parameter is called.

public boolean equals(Object obj)

If the equals method is called with a Person object argument then the equals method with the Person object parameter is called.

public boolean equals(Person p)

Todo

Make a new class file called Lesson7.java with a main method. Make some Person Objects with same name and some with different names. Using the two Person objects call the Person object **equals** method. Print out the results using System.out.println();

Adding equals methods to a derived class

We can also add the equals methods to a derived class. This can be a little tricky to do because we have more things to compare. For the derived class equals methods we call the equals methods from the super class and compare the result with the values in the derive class. When we call the equals method from the super class we pass the derive object to it. Note: the derive Student object also contains the base class Person object. For the student derived class we compare the Person name with the StudentID of the Student class. Here is the Student equals method using the Object parameter.

```
// Student class equals method using Object parameter:
public boolean equals(Object obj)
{
    If(obj != null)
    {
        if(obj instanceof Student)
        {
            Student s2 = (Student) obj;

            return super.equals(s2) && idnum.equals(s2.idnum);
        }
    }
    return false;
}
```

Here is the Student equals method using the Student parameter.

```
// equals method using Student parameter:
public boolean equals(Student s2)
{
    If (s2 != null)
        return super.equals(s2) && idnum.equals(s2.idnum);
    else return false;
}
Todo
```

Add both **equals** method to the Student class. the object parameter equals method and the Student parameter equals method.

INTERFACES

The purpose if the interface is to specify what methods a class should have and to represent objects that implement it. The interface has only **pure abstract** methods. **Pure abstract** methods are method definitions that have no programming statements that end in a semi colon. The code is to be implemented by the class that implements the interface.

The interface becomes the super class for the class it implements. An **interface** starts with the interface keywords and then contains all the pure abstract method definitions:

```
interface interface_name
{
    pure_abstract_method_definition(s);
}
```

Calculator Interface

We have made a sample interface for you, that defines the methods for a simple calculator class. You can put the ICalculator interface in a file called ICalculator.java

```
/*
* ICalculator interface
* specifies the method a ICalculator should have
*/
interface ICalculator {
    public double add(double a, double b);
    public double sub(double a, double b);
    public double mult(double a, double b);
    public double divide(double a, double b);
}
```

Note: the **public** modifier is optional, some java compilers make you remove it.

To use an interface the class must implement it using the keyword **implements**. The class must write code for all the methods that are specified in the interface.

```
public class class_name implements interface_name
{
    methods_to_implement
}
```

A class implementing the ICalculator interface is as follows: You can put into an class file called MyCalculator.java

```
/*
 * class MyCalculator implements ICalculator interface
 * implements the methods of the ICalculator interface
 */
public class MyCalculator implements ICalculator{

    public double add(double a, double b){
        return a + b;
    }
    public double sub(double a, double b){
        return a - b;
    }
    public double mult(double a, double b){
        return a * b
    }
    public double divide(double a, double b){
        return a/b;
    }
}
```

To do

Add a main method to the MyCalculator Java file. You will have to instantiate a MyCalculator object in your main class because the MyCalculator class methods are not static. We represent the MyCalculator by the **ICalculator** interface.

ICalculator calc = new Calculator();

Test each method in the calculate object inside a System.out.println statement.

Comparable Interface

We have already used java classes that implement the **Comparable** interface. The String and Integer class all implement the Comparable Interface having the **compareTo** method.

```
// Comparable interface:
public interface Comparable
{
    int compareTo(Object obj2);
}
```

The **compareTo** method returns a negative number if the current executing object value is less than the parameter obj2, returns a positive number if the object value is greater than the parameter obj2 value and returns 0 if the object value is equal to the parameter obj2 value

To use the **compareTo** method in your class you must implements the Comparable interface.

```
public class MyClass implements Comparable
```

Add a compareTo method to your Person class, so you can compare person names. Your person class must now implement the Comparable interface using the **implements** keyword.

```
public class Person implements Comparable {
```

You will need to use the **instanceof** operator as we did above in the **equals** method to test if the obj parameter is a Person Object.

```
// compare two persons
public int compareTo(Object obj)
{
    if(obj != null)
    {
        if(obj instanceof Person)
        {
            Person p2 = (Person) obj;
            return name.compareTo(p2.name);
        }
    }
    return 0;
}
```


Todo

In your Lesson7.java main method, make some persons with different names. Using the compareTo method print out the results of comparing different and same Person objects, using System.out.println().

Adding the compareTo method to a derived class

We can also add the compareTo methods to a derived class. This can be a little tricky to do because we have more things to compare. In the **compareTo** method we only need to compare a value in the derived class when the result of the super compareTo method is equal. (a 0 value).

```
// compare to method of Student derived class
public int compareTo(Object obj)
{
    if(obj != null)
    {
        if(obj instanceof Student)
        {
            Student s2 = (Student) obj;
            if( super.compareTo(s2) == 0)
                return idnum.compareTo(s2.idnum);
            else
                return super.compareTo(s2)
        }
    }
    return 0;
}
```

The Student derived class definition would now be

public class Student extends Person implements Comparable

when it implements the Comparable interface.

To do

Add the compareTo method to your derived Student class. In your Lesson7 main instantiate some Student objects and compare them.

GENERICS

Generics allow a class to have a specified data type when it is declared and instantiated. The specified data type in the Generic class all inherit from the Object class. The Object class is the super class of all classes, and can represent any other class. The Object class has a few methods available to do things with, just methods like equals, toString() etc. But no compares. The specified data type in a Generic is actually an Object data type. Generics are very awkward and frustration to use in Java. You will see soon.

We have already used Generics with the ArrayList, HashSet and HashMap when you specified the data types in diamond <> brackets to specify what data type they should use. Example specifying the data the for an ArrayList,

```
ArrayList<Integer> list1 = new ArrayList<Integer>();
```

This means the ArrayList will hold an Integer data type that allows you to use int numbers.

You define a Generic class like this:

```
access_modifier class_name<T>
{
    T variable name;
    additional variables
    constructors
    T getters
    setters(T)
    equals
    toString
}
```

The **T** can represent any class that the generic class will use.

Here is a simple Generic class that holds a single generic variable **TItem**. The **T** represents any data type that will be specified when the constructor is called.

```
TItem<Integer> titem = new TItem<Integer>();
```

You can put the TItem into a class file called TItem.java

```
// Simple Generic class to hold a generic data type
```

```
public class TItem<T>  
{  
    private T item; // generic variable  
  
    // construct TItem  
    public TItem(T item)  
    {  
        this.item = item;  
    }  
  
    // return item  
    public T getItem()  
    {  
        return item;  
    }  
  
    // assign item  
    public void setItem(T item)  
    {  
        this.item = item;  
    }  
  
    // return data as a string  
    public String toString(){  
        return this.item.toString();  
    }  
}
```

Todo

Put the TItem class in a file called TItem.java, in your Lesson7.java file instantiate the TItem class like this

```
TItem<Integer> tItem = new TItem<Integer>(5);
```

Call the **getItem** method to print out the return value. Using the **setItem** method assign a new value to the TItem object like 8 then call the **getItem** method again to print out the return value. Finally print out the return value from the **toString** method.

Todo

Instantiate a TItem class with a String instead of a Integer.

Add a copy constructor to the TItem class.

```
// copy TItem  
public TItem(TItem<T> tItem)  
{  
    data = tItem.data;  
}
```

Adding a equals method to a generic class:

We now add three different equals method's to our TItem generic class

The first one receives an Object parameter.

```
public boolean equals(Object obj)
{
    if(obj != null)
    {
        if (obj instanceof TItem<?>)
        {
            TItem<?> t = (TItem<?>) obj;
            if (item.equals(t.item) )
            {
                return true;
            }
        }
    }
    return false;
}
```

The above equals method overrides the equals method of the Object class. Notice the **TItem<T>** has been replaced with **TItem<?>** the **?** is called a **wildcard** and accepts any class data type. This is needed since at compile time the object can be any class data type.

The second equals method takes a **TItem** object for convenience and compares to TItem objects. This is a standalone method but does not override the equals method in the Object class.

```
// return true if 2 Item have same data
public boolean equals(TItem<T> tItem){

    if(item != null)
        return this.item.equals(tItem.item);
    else return false;
}
```

The third one just takes a T data object not a TItem object. This is just to compare this TItem object with a T data object.

```
// return true if Item have same data
public boolean equals(T item){
    return this.item.equals(item);
}
```

The most appropriate equals method will be called by the compiler.

Note: With some Java compilers the **public boolean equals(T item)** method cannot coexist with the **public boolean equals(Object obj)** since the java compiler thinks the T is a Object.

Todo

Add the above equals method to the TItem class. In your Lesson7.java file, make another TItem object with another value, then print out if the two item objects are equal.

Make some more generic **TItem object's of type String, Double, Character** etc, and test if they are equal.

The Generic data type must be classes like Integer, Double, Float, Character not primitive data type int, double, float and char. You can still use primitive data type values. The java compiler will automatically convert a primitive data type to its corresponding Class. Example int 1234 get's converted to Integer, this is called **boxing**.

GENERIC INTERFACES

Interfaces can also be Generic to. The Comparator interface also has generic version where T can be any class that can be compared.

Comparable Generic interface

```
public interface Comparable<T>
{
    int compareTo(T obj);
}
```

With the use of Generics we can add a CompareTo method to our Person class that takes a Person rather than an Object.

```
// compare 2 person names
public int CompareTo(Person p)
{
    If(p != null)
        return this.name.compareTo(p.name);
    else return 0;
}
```

Our Person class definition would then look like this:

```
public class Person implements Comparable<Person>
```

Add this compareTo method to your Person class. You may need to remove or comment out the **public int compareTo(Object obj)** method if get a name clash error. Then in the Lesson7 main function try it out, compare two person objects.

Adding a compareTo method to a derived class that implements the Generic Comparable interface.

The **compareTo** method will now take a Student parameter rather than a Object parameter.

```
// compare to method of Student class using Student parameter
public int compareTo(Student s2)
{
    If(s2 != null)
    {
        If( super.compareTo(s2) == 0)
            return idnum.compareTo(s2.idnum);
        else
            return super.compareTo(s2)
    }
    return 0;
}
```

You need to rewrite the Student class definition as follows to use the Generic Comparable interface.

public class Student extends Person implements Comparable<Person>

rather than

public class Student extends Person implements Comparable<Student>

This is because the Generic Comparable interface cannot handle inheritance. You will get this error.

error: Comparable cannot be inherited with different arguments: <Student> and <Person>

GENERIC CLASSES IMPLEMENTING GENERIC INTERFACES

A generic class implementing the Comparable interface would be like this

```
public class MyGenericClass<T extends Comparable<T> > implements  
Comparable<T> {  
}
```

For our TItem class it would look like this:

```
public class TItem<T extends Comparable<T> > implements Comparable<T> {  
}
```

T needs to extends Comparable<T> because T does not know anything about the compareTo method and we need to tell the java compiler that T extends Comparable . We use the extends keyword instead of the implements keyword because we want to use the **compareTo** method of the Comparable object represented by T

implements Comparable<T> lets the compareTo method in the **TItem** class to have a **T** parameter.

Todo

Have your TItem class implement the Comparable interface to compare the T data. Add a compareTo method to your TItem class that will compare T data.

```
public int compareTo(T data)  
{  
    // compare the 2 data elements and return the result  
    return this.data.compareTo(data);  
}
```

In your Lesson7.java file use your TItem objects to compare values. Print out the results using System.out.println statements.

Using the Generic compareTo on A Generic class

Using the generic compareTo on a generic classes is a little awkward since you must compare a generic class with a generic data type. Your TItem class definition will now look like this.

```
public class TItem<T extends Comparable<? super T>> implements  
Comparable<TItem<T>>
```

? is called a wildcard and the super keyword means that ? is to accept super types of T. If T is a Integer then T would inherit the compareTo method.

Todo

Add the compareTo method to the TItem class to compare two TItem Objects. You need to update you class definition as above. Your second compareTo method will now look like this:

```

public int compareTo(TItem<T> tItem2)
{
    If(tItem2 != null)
        // compare the 2 TItem class objects using the data element
        return this.data.compareTo(tItem2.data);
    else return 0;
}

```

Todo

In your Lesson7.java file use your TItem objects to compare other TItem objects. Print out the results using System.out.println statements.

Array of Generic Objects

This is always a difficult thing to do. It is difficult to do because people do not realize you cannot create an Array of generic Objects, you just need to make an array of Objects only without specifying the data type. Once you realize this all your problems are over. The compiler cannot make the array of your Generic Object because it does not know the data type yet until the program runs. It thinks the generic T is an Object not a specified Integer or String, so you do not need to specify the T when you create the array. You still need to specify the T data type when you define the array variable.

Create an array of generic TItem's:

```
TItem<Integer>[] items = new TItem[5];
```

Optionally you can type cast to your array data type:

```
TItem<Integer>[] items = (TItem<Integer>[])new TItem[5];
```

Note: you need to ignore any compiler warnings, or add code to suppress the error.

```
@SuppressWarnings("unchecked")
TItem<Integer>[] items = new TItem[5];
```

Once you have an array of TItems you can add TItem object to it

```
Items [0] = new TItem<Integer>(1);
```

Todo

Make an array of 5 Generic TItems and in a loop fill them with some values.
In another loop print out the values, compare adjacent values and test if they are equal.

Making our own Generic Interface

We will make a generic interface called TCalculator that can use any specified DataType T.

Make a interface called TCalculator.java as follows:

```
/*  
 * Generic TCalculator interface  
 * specifies the methods a Generic Calculator should have  
 */
```

```
public interface TCalculator<T> {
```

```
    public T add(T a, T b);  
    public T sub(T a, T b);  
    public T mult(T a, T b);  
    public T divide(T a, T b);
```

GENERIC CALCULATOR

We now make a Generic Calculator that implement the TCalculator.
We will name the calculator **MyTICalculator** and the file **MyTICalculator.java**
The solution is very awkward because the T does not have any arithmetic operators like add or subtract.

Also the compiler cannot convert T to its primitive data type to the arithmetic operation like +,-*./ . Instead we use the instance operator to find out what our data type and then type cast T to that data type.

```
/*  
 * generic class MyTICalculator that implements the methods  
 * of the TICalculator interface  
 */
```

```
public class TCalculator<T> implements TICalculator<T>{
```

```
    public T add(T a, T b){
```

```
        if(a instanceof Integer )  
            return (T)(Integer)((Integer)a + (Integer)b);
```

```
        else if(a instanceof Double )  
            return (T)(Double)((Double)a + (Double)b);
```

```
        else if(a instanceof String )  
            return (T)(String)((String)a + (String)b);
```

```
        else return null;
```

```
    }  
}
```

Although not a optimum solution all we are doing is checking what data type our data is and then type casting our data to that data type so we can use the + operator. Once we do the addition we type cast the data back to our generic T data type.

A different approach is to use overloading, that has separate methods for the different data types, where as the Generic method just returns null.

```

public class MyTCalculator<T> implements TCalculator<T>{

    public int add(int a, int b){

        return a + b;
    }

    public double add(double a, double b){

        return a + b;
    }

    public T add (T a, T b)
    {
        return null;
    }

    // etc
}

```

I am sure there are other solutions.

Todo

Type in the generic calculator and use the methods you like or a combination of both or come with another elegant solution of your own. Add the reset of the methods, sub, mult and divide. Make a main method to test all the methods, try with many different data types like Integer, Double and String. String may not work in all situations or solutions.

```

MyTCalculator<Integer> calc = new MyTCalculator<Integer>();
System.out.println(calc.add(3,4));

```

```

MyTCalculator<Double> calc2 = new MyTCalculator<Double>();
System.out.println(calc2.add(3.7,4.4));

```

```
MyTCalculator<String> calc3 = new MyTCalculator<String>();  
System.out.println(calc3.add("happy", "sad"));
```

You should get something like this:

```
7  
8.1  
happysad
```

LESSON 7 HOMEWORK

Question 1

Add the copy constructor to your **Profession** class and to your derived **Payroll** class from previous lesson 3. The Payroll derived class would have to pass the Payroll object to the Profession class using the super keyword. In your main method of Homework7.java make a few Professions and Payrolls and print them out, then make a copy of each one using the copy constructor.

Question 2

Add the **equals** and **compare To** methods to your **Profession** and **Payroll** class from Lesson 3. The **Profession** class should compare the profession and/or salary, and the **Payroll** class can compare the bonus and optionally the result from the **Profession** class. In your main method of Homework7.java make a few **Profession** and **Payroll's** and print them out, then check if they are equal using the equals method and compare the objects using the **compareTo** method, then print out the results.

Question 3

Make a Interface called **IAAnimal** that has a void method **sound()**.

```
interface IAnimal {  
  
    public void sound();  
}
```

Put the interface in a java file called IAnimal.java.

Make a class called Cat that implements the IAnimal interface. It has a constructor that receives a name, a sound method that prints out “meow” and a toString() method the prints out the name of the animal and type. Put this in a java file called Cat.java.

Make a class called Dog that implements the IAnimal interface. It has a constructor that receives a name, a sound method that prints out “bark” and a toString() method the prints out the name of the animal and type. Put this in a java file called Dog.java.

Make a class called Tiger that implements the IAnimal interface. It has a constructor that receives a name, a sound method that prints out “roar” and a toString() method the prints out the name of the animal and type. Put this in a java file called Tiger.java.

Make a class of your favorite Animal that implements the IAnimal interface. It has a constructor that receives a name, a sound method that prints out “????” and a toString() method the prints out the name of the animal and type. Put this in a java file called ????.java.

Your toString method should return the animal name what kind of animal it is like:

“I am Fluffy and I am a cat”

In your main method of Homework7.java make the 4 four animals and print out what each animal says by first calling the toString method and then the sound method. You may want to put all the animals in a IAnimal array and print in a loop.

```
IAnimal[] animals = new IAnimal[4];
```

This is possible because an interface can represent any class that implements it.

An example output is:

```
"I am Fluffy I am a cat"  
"meow"
```

Question 4

Add the **compare To** methods to your **Profession** and **Payroll** class from Question2 , using the Generic Comparable interface **Comparable<Profession>**. Each will receive a **Profession** and **Payroll** parameter respectively. The **Profession** class should compare the profession and/or salary and the **Payroll** class can compare the bonus and optionally the result from the **Profession** class. The derived class Payroll must implement **Comparable<Profession>** not **Comparable<Payroll>** to avoid a name clash.

Check if the homework 7 program still work's the same.

Question 5

Make generic Animal class called TAnimal that can hold any type of the animals Cat, Dog, Tiger, etc. from Question 3 that implements the **IAnimal** interface also from Question 3

```
public class TAnimal<T extends IAnimal> implements IAnimal
```

The TAnimal class should have a generic instance variable T animal and T Animal getters and setters. We need to use **<T extends IAnimal>** because we want the compiler to know that T will represent a IAnimal so we can call the sound method from the animal instance variable. The IAnimal interface specifies the sound method() print out a animal sound.


```
interface IAnimal {  
    public void sound();  
}
```

In the main method of Homework 7.java make some animals and print them out with a sound like this:

```
TAnimal<Cat> cat = new TAnimal<Cat>(new Cat("Tom"));  
cat.sound();
```

Make an array of Generic TAnimals. When you make a Generic array you do not specify the data type but have to suppress any warnings.

```
@SuppressWarnings("unchecked")  
TAnimal[] animals = new TAnimal[3];
```

Put some IAnimals in the animals array. You can use the animals you made above or make some new ones

```
animals[0] = new TAnimal<Cat>(new Cat("tom"));  
animals[1] = new TAnimal<Dog>(new Dog("bill"));  
animals[2] = new TAnimal<Tiger>(new Tiger("george"));
```

Print out the animal and then call the sound method.

```
for(int i=0;i<animals.length;i++)  
{  
    System.out.println(animals[i].toString());  
    animals[i].sound();  
}
```

You should get something like this:

```
"I am Fluffy I am a cat"  
"meow"
```

Question 6

Make a generic TAnimal interface that has an additional method called talk that takes a Generic parameter that let's two animals to talk to each other. TAnimal extends IAnimal interface from Question 5.

```
interface TAnimal<T> extends IAnimal{  
  
    public void talk(T animal2);  
}
```

where the IAnimal interface from Question 3 is:

```
interface IAnimal {  
  
    public void sound();  
}
```

Make a generic class called TalkingTAnimal that extends TAnimal that can hold any type of the animals Cat, Dog, Tiger, etc. from Question 3 that implements the IAnimal interface.

```
public class TalkingTAnimal<T extends IAnimal> extends TAnimal<T>  
    implements TAnimal<TalkingTAnimal<T>>
```

<T extends IAnimal> means we want the compiler that **T** represents a **IAnimal** so we can call the sound method from it.

extends TAnimal<T> means we want to use all the methods from the super class TAnimal

TAnimal<TalkingTAnimal<T>> means we want the talk method parameter to be a TalkingTAnimal of Animal type **T**, so we can call the sound from it.

The TalkingTAnimal class should have a generic instance variable **T animal** and T getters and setters.

The talk method just lets each animal talk to each other by each calling the sound method(). It could say

“meow”

“ my name is Tony”

You need to make two talk methods, one that receives a T and a the other one receives TalkingTAnimal<T> .

The T would represent a IAnimal like Cat or Dog where as the TalkingTAnimal<T> would represent a TalkingAnimal that could be a Cat or Dog.

```
public void talk(T animal2)  
    {  
        sound();  
        animal2.sound();  
    }
```

```
public void talk(TalkingTAnimal<T> animal2)  
    {  
        sound();  
        animal2.sound();  
    }
```

In the main method of Homework 7.java make some animals and print them out talking to each other like this:

```
TalkingTAnimal<Cat> cat = new TalkingT<Cat>(new Cat("tom"));  
TalkingTAnimal<Cat> cat2= new TalkingT<Cat>(new Cat("sue"));  
  
cat.talk(cat2);  
cat2.talk(cat);
```

You should get something like this

```
Meow  
My name is tom and I am a Cat  
Meow  
My name is sue and I am a Cat
```

LESSON 8 Enhanced Loops, Iterators, Comparators and Anonymous Functions and Inner classes

For loops print out values from lists. We can do this 3 different ways. Index for loop inside an index counter, enhanced for loop that can travel through each item in the list using a variable value and a loop that uses an iterator to traverse through the list by extracting the next value per iteration of the loop. Iterators are used allot in Java programming. You need to have the ArrayList and Iterator import statement at the top of your java program to use the ArrayList and Iterator. Note we are now using the List interface to represent our ArrayList object. Many people in programming recommend this as the correct way to program, to have a class that implements an interface to be represented by their interface name rather than their class name. We need to do the same. You will need the following imports at the top of your program.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

// make an ArrayList of values
List<Integer> list1 = new ArrayList<Integer>();
list1.add(1);
list1.add(2);
list1.add(3);
list1.add(4);
list1.add(5);

// using an index loop
for (int i=0;i<list1.size();i++)
    System.out.print (list1.get(i) + " ");
System.out.println("");

// use an enhanced for loop
for(int v: list1)
    System.out.print (v + " ");
System.out.println("");
```

```
// use an iterator loop
Iterator<Integer> itr = list1.iterator();
while(itr.hasNext())
{
    int x = itr.next();
    System.out.print (x + " ");
}

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

1 2 3 4 5

We can also use a **for loop** to print out the items stored in a Hash Set. Printing values in a HashSet is more difficult since the HashSet values are not stored in sequential memory locations. The Hash set values are not always in order. We only used enhanced for loop that can travel through each item in the list using a variable value and a loop that uses an iterator to traverse through the list by extracting the next value per iteration of the loop. You need to have the HashSet and iterator import statement at the top of your java program to use the ArrayList and Iterator. Note we are now using the Set interface to represent our HashSet object. Many people in programming recommend this as the correct way to program, we need to do the same. You would need the following imports at the top of your program

```
import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

// put values in a HashSet
System.out.println("print out values in a hashset");
Set<Integer> set1 = new HashSet<Integer>();
set1.add(1);
set1.add(2);
set1.add(3);
set1.add(4);
set1.add(5);
```

```
// use a enhanced for loop
for(int v: set1) System.out.print (v + " ");
System.out.println("");
```

```
// use an iterator loop
Iterator<Integer> itr2 = set1.iterator();
while(itr2.hasNext())
{
    x = itr2.next();
    System.out.print (x + " ");
}
System.out.println("");
```

1	2	3	4	5
---	---	---	---	---

We also use for loops can to print out HashMaps. The main purpose of loops is to print out the HashMap in order by key or order by values. We first print out dictionary by keys unsorted. You need to have the HashMap and iterator import statement at the top of your java program to use the HashMap, Iterator and Collections objects. The Collections is used for sorting. Note we are now using the Map interface to represent our HashMap object. Many people in programming recommend this as the correct way to program, we need to do the same. You would need the following imports at the top of your program

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Collections;
```

```
// make empty map
Map<String,String> map1 = new HashMap<String,String>();
```

```
// add values to HashMap
map1.put("name","Bill");
map1.put("age","24");
map1.put("studentid","S1234");
```

```
// print out map using entry set
// An Entry class contains the key and value
for(Map.Entry<String, String> set:map1.entrySet())
{
    System.out.println(set);
}
System.out.println("");
```

```
age=24
studentid=S1234
name=Bill
```

```
// print out keys of HashMap
for(String key: map1.keySet())
{
    System.out.print (key + " ");
}
System.out.println("");
```

```
name
age
studentid
```

```
// print HashMap by values of HashMap
for(String value: map1.values())
{
    System.out.print (value + " ");
}
System.out.println("");
```

```
name Bill
age 24
studentid S1234
```

```
// print HashMap sorted by key
// get list of keys
List<String> keys = new ArrayList<String>(map1.keySet());
```

```
// sort list of keys
Collections.sort(keys);
```

```
age: 24
studentid: S1234
name: Bill
```

```

//print hashmap sort by key
for(String key: keys)
{
    String value = map1.get(key);
    System.out.print (key + ":" + value + " ");
}
System.out.println("");

// print out HashMap sorted by value

// get list of values
List<String> values = new ArrayList<String>(map1.values());

// sort list of values
Collections.sort(values);

# for each sorted value find the corresponding key
for(String value: values)
{
    for(String key: map1.keySet())
    {
        if(map1.get(key).equals(value))
        {
            System.out.print(key + " : " +value );
            break;
        }
    }
}

```

age: 24 name: Bill studentid: S1234

Print HashMap sorted by value using EntrySet

To print a Hash Map sorted by values we have to put the EntrySet on an ArrayList. then sort the ArrayList by values. An Entry Set is a set that contains an Entry object containing both the key and value. The Entry object is specified in the Map Interface. We need to use the Entry object so that we can sort both the value and corresponding key together at the same time

Here are the steps:

Step 1: get the EntrySet from the HashMap

Obtain an Entry Set from the HashMap.

```
Set<Map.Entry<String, String>> set = map1.entrySet();
```

Step 2: make an ArrayList of Entry Sets

Put the entry set in a ArrayList.

```
List<Map.Entry<String, String>> list = new ArrayList<Entry<String, String>>(set);
```

Step 3: make the anonymous class Comparator

We sort the ArrayList by value, using the static sort method from the Collections class and using our own Comparator. A Comparator is an interface that has a compare method to compare 2 objects.

```
int compare(T obj1, T obj2);
```

The **compare** method returns a negative integer, zero, or a positive integer as the first argument obj1 is less than, equal to, or greater than the second argument obj2.

We use an anonymous class for our Comparator. An anonymous class is a class with no name that implements a known interface. The interface states what class the anonymous class is as well as to specify what methods the anonymous class must implement. Anonymous classes are quite handy and used everywhere in Java. The syntax for an Anonymous class is:

```
new interface_name () {  
  
    implement interface_methods  
}
```

We construct our anonymous class as follows:

```
Comparator c = new Comparator<Map.Entry<String, String>>() {  
  
    public int compare(Map.Entry<String, String> e1,  
        Map.Entry<String, String> e2) {  
        return e2.getValue().compareTo(e1.getValue());  
    }  
};
```

We have implemented the **compare** method that compares two Map.Entry objects by value.

Step 4: Sort the ArrayList using Collection.sort and our anonymous comparator

The Collections sort method takes an extra parameter as a Comparator object as a custom sort. We put in our anonymous comparator class as an argument to the Collection sort method like this.

```
Collections.sort(list,c);
```

Our anonymous class comparator sorts the Entry objects by values using its compare method.

Step 5: print out the sorted ArraySet

```
for (Map.Entry<String, String> entry : list) {  
    System.out.println(entry.getKey() + " = " + entry.getValue());  
}  
System.out.println("");
```

age: 24 idnum: S1234 name: tom

You need to put the following import statements on top of your Java class so that the compiler will recognize the HashMap, Entry, Comparator and Collections classes.

```
import java.util.Map.Entry;  
import java.util.Comparator;  
import java.util.Collections;
```

Here is the Complete Program:

```
// print HashMap sorted by value  
  
// Entry set contains both key and values,  
// so we can sort key and value together  
Set<Map.Entry<String, String>> set = map1.entrySet();  
List<Map.Entry<String, String>> list  
    = new ArrayList<Map.Entry<String, String>>(set);  
  
// anonymous comparator class to sort array list of entries  
Comparator c = new Comparator<Map.Entry<String, String>>() {  
    public int compare(Map.Entry<String, String> e1,  
        Map.Entry<String, String> e2) {  
        return e1.getValue().compareTo(e2.getValue());  
    }  
};  
  
// sort Entry key value pairs by value using anonymous Comparator  
Collections.sort(list,c);
```

Lastly, we print out the sorted list.

```
for (Map.Entry<String, String> entry : list) {  
    System.out.println(entry.getKey() + " = " + entry.getValue());  
}  
System.out.println("");
```

age: 24 idnum: S1234 name: tom

Recapping: here are the collection classes and their corresponding interfaces:

Interface	class implementing interface
List	ArrayList
Set	HashSet
Map	HashMap

Inner classes

Classes may contain other classes. The classes inside a class are known as a **inner classes** where as the class containing the inner class is known as a **outer** class. The advantage of inner classes is that they can access private variables and methods of the Outer class that they reside in. There are basically four ways of creating inner classes:

- 1) Nested Inner class
- 2) Outer Method Inner class
- 3) Anonymous inner classes
- 4) Static nested Inner classes

Nested Inner class

A nested inner class is contained in a outer class.

```
// outer class
class Outer {

    // nested inner class
    class Inner {
        public void print() {
            System.out.println("I am a Inner class method");
        }
    }
}
```

Instantiating a Inner class outside the Outer class

You instantiate a nested Inner class using an instance of the Outer class. You must instantiate the inner class with an instance of the outer class because the nested inner class belongs to the outer class.

```
public static void main(String[] args){
{
Outer.Inner inner = new Outer().new Inner();
inner.print();
}
}
```

Calling the print method on the inner object would display the following:

I am a Inner class method

Instantiating a Inner class inside the Outer class

In this situation the Outer class object is already created, you can create the Inner class without explicitly stating the outer class object reference.

```
// outer class
class Outer {

// nested inner class
class Inner {
public void print() {
System.out.println("I am a Inner class method");
}
}
}
```

```
// outer class method
void outerMethod() {
    System.out.println("inside outerMethod");
    Inner inner = new Inner();
    inner.print();
}
```

To run the inner class methods we must instantiate a outer class object then call the outer class outerMethod. Note: for the inner class to use the variables of the outer methods the variables inside the outer method must be declared **final**.

```
public static void main(String[] args){
{
    Outer outer = new Outer();
    outer.outerMethod();
}
}
```

inside outerMethod
I am a Inner class method

Instantiating Inner class declared inside a Outer Method

In this situation the Inner class is defined inside a method of the outer class. The inner class is also instantiated and run from inside the outer class method. Note: for the inner class to use the variables of the outer methods the variables inside the outer method must be declared **final**.

```

// outer class
class Outer {

    // outer class method
    void outerMethod() {
        System.out.println("inside outerMethod");

        // Inner class inside outer class method
        class Inner {
            void print() {
                System.out.println
                ("I am a Inner class method called inside a outer class method ");
            }
        }

        // instantiate a inner class inside a outer class method
        Inner inner = new Inner();
        inner.print();
    }
}

```

To run the inner class methods we must instantiate a outer class object then call the outer class outerMethod.

```

public static void main(String[] args){
{
    Outer outer = new Outer();
    outer.outerMethod();
}
}

```

The output as follows because the outerMethod of the outer object instantiates a inner object and then calls the print method of the inner object.

I am a Inner class method called inside a outer class method

Anonymous inner classes

Anonymous class is a class with no name. It can be created 2 different ways.

- 1) A Anonymous class is created from a known base class

```
class Base
{
    public void print()
    {
        System.out.println("I am a base class method");
    }
}

class AnonymousDemo
{
    public void createAnonymousClass()
    {
        // create a anonymous class from the base class
        Base base = new Base() {
            public void print() {
                super.print(); // calls method from base class
                System.out.println("I am a method in an anonymous class");
            }
        };
        base.print(); // call print method from anonymous class
    }
}
```

You would run like this, create the **AnonymousDemo** and the call the print method from the anonymous class.

```
public static void main(String[] args)
{
    AnonymousDemo demo = new AnonymousDemo();
    demo.createAnonymousClass();
}
}
```


You would get something like this.

I am a base class method
I am a method in an anonymous class

2) An implementation of a specified interface

```
interface IBase
{
    public void print();
}

class AnonymousIDemo
{
    public void createAnonymousClass()
    {
        // create a anonymous class from the IBase interface
        IBase ibase = new IBase() {
            public void print() {
                System.out.println("I am a method in an anonymous class");
            }
        };

        ibase.print(); // call print method from anonymous class
    }
}
```

You would run like this, create the **AnonymousIDemo** and the call the print method from the anonymous class.

```
public static void main(String[] args)
{
    AnonymousIDemo demo = new AnonymousIDemo();
    demo.createAnonymousClass();
}
}
```

You would get something like this:

I am a base class method
I am a method in an anonymous class

You should now realize these Anonymous classes have no name because they are created from a base class or from an interface .

Static nested classes

Static classes are inner classes that can be instantiated without an instance of the outer class. Static inner classes cannot use the variables of an outer class unless they are declared static. Static inner class may be easier to use outside the outer class because you can easily access them by only using the outer class name.

```
// outer class
class SOuter {

    // nested inner class
    static class SInner {
        public void print() {
            System.out.println("I am a static Inner class method");
        }
    }
}
```

When instantiating a static inner class outside the outer class you use the outer class name not a outer class object.

```
public static void main(String[] args)
{
    Souter.SInner sinner = new SOuter.SInner();
    sinner.print();
}
}
```

I am a static Inner class method

You can also instantiate inner class inside the outer class.

```
// outer class
class SOuter {

    // nested inner class
    static class SInner {
        public void print() {
            System.out.println("I am a static Inner class method");
        }
    }

    // outer class method
    void outerMethod() {
        System.out.println("inside outerMethod");
        SInner sinner = new SInner();
        sinner.print();
    }

    public static void main(String[] args){
        {
            SOuter outer = new SOuter();
            soutter. outerMethod();
        }
    }
}
```

inside outerMethod
I am a static Inner class method

To do: try all The inner class variations in your Lesson 8.java file.

Lesson 8 Home Work

Sentence Generator

A Sentence is composed of the following:

<article><adjective><noun><adverb><verb><article><adjective><noun>

Make an ArrayList<String> of articles like: "a", "an" and "the"

Then make an ArrayList<String> of adjectives like: "fat", "big", "small"

Then make an ArrayList<String> of nouns like: "cat", "rat", "house"

Then make an ArrayList<String> of adverbs like: "slowly", "gently", "quickly"

Then make an ArrayList<String> of verbs like "ate", "sat on", "pushed"

Make a HashMap<String,ArrayList<String> > called words to hold all the lists:

```
words .put( "articles",articles);
```

```
words .put( "adjectives",adjectives);
```

```
words .put( "nouns",nouns);
```

```
words .put( "adverbs",adverbs);
```

```
words .put( "verbs",verbs);
```

Next make an array of Strings or a ArrayList<String> called keys

Containing "articles", "adjectives", "nouns", "adverbs", "verbs", "articles", "adjectives", "nouns" **which** are HashMap keys used to make a sentence:

Finally make a sentence using the HashMap entries, using the parts of speech as the HashMap keys and by selecting random words from the HashMap ArrayList values.

```
String sentence = "";
```

```
for(String key: keys)
```

```
{
```

```
    int r = (int)(Math.random()*words.get(key).size());
```

```
    sentence += words.get(key).get(r) + " ";
```

```
}
```

Then print out the sentence.

`System.out.println(sentence);`

You should get something like this:

The big cat slowly ate the small rat

Which has picked random words from the dictionary sentence structure:

`<article><adjective><noun><adverb><verb><article><adjective><noun>`

You can put your code in your homework8.java file or make a Sentence.java file

Animal Zoo

Make a class called Zoo that has a Inner class called Animal. You can make any Kind of Inner class you want. Make a HashMap of your favorite animals. The key should be the animal kind like cat, dog or tiger, the value should be what sound the animal makes meow, bark, tiger. Sort all animals by what sound they make. Finally print out the animal kind and value sorted by what sound they make.

LESSON9 FILE ACCESS

Create a new java class file called Lesson9.java and type in all the following examples.

File Access

Java has extensive file objects for reading and writing to file. We concentrate on the most used.

Write characters to a file

The **FileWriter** class is used to write characters one by one sequentially to a file.

```
try
{
// write chars to a file
FileWriter fw = new FileWriter("data1.txt");

String s = "Hello";

for(int i=0;i<s.length();i++)
{
char c = s.charAt(i);
fw.write(c);
}

fw.write('\n');
}

catch(IOException ex)
{
System.out.println(ex.getMessage());
}
```



hello

Read characters from a file

The **FileReader** class is used to read characters one by one sequentially from a file. The File class is used to store a file name. The **try** block is used to catch any errors like file not found, corrupt file etc. The catch statement is used to catch the errors. When a program encounters abnormal operation, an Exception is thrown. The try block initiates the operation so that the catch block can catch the exception and report the exception. Failure to include a try catch block would result in immediate program termination.

```
try{
    FileReader fr = new FileReader("data1.txt");
    int ch = fr.read();

    while(ch != -1)
    {
        // convert int to char
        System.out.print((char) ch);
        ch = fr.read();
    }

    fr.close();
}

catch(IOException ex)
{
    System.out.println(ex.getMessage());
}
```



hello

Write lines to a file

The **PrintWriter** class is used to write lines one by one sequentially to a file.

```
// write lines to a file
try{
    PrintWriter pw = new PrintWriter("data2.txt");
    pw.println("hello there");
    pw.println("goodbye now");
    pw.close();
}

catch(IOException ex)
{
    System.out.println(ex.getMessage());
}
```

Hello there
goodbye now

Read line by line from a file

The **Scanner** class is used to read lines one by one sequentially from a file.

```
try{

// Open the file for read
Scanner fsc = new Scanner(new File("data2.txt"));

// for each line in file
while(fsc.hasNextLine())
{
    String line = fsc.nextLine();
    System.out.println(line);
}
```



```
// close the file
fsc.close();
}

catch(IOException ex)
{
    System.out.println(ex.getMessage());
}
```

Hello there
goodbye now

Just read words from a file

The **Scanner** class can also be used to read words one by one sequentially from a file.

```
try{

// Open the file for read
Scanner fsc = new Scanner(new File("data2.txt"));

// for each word in file
while(fsc.hasNext())
{
    String word = fsc.next();
    System.out.println(word);
}

// close the file
fsc.close();
}
```

Hello
there
goodbye
now

```
catch(Exception ex)
{
    System.out.println(ex.getMessage());
}
```

Write lines to a csv file (comma separated values)

The **PrintWriter** class is used to write lines one by one sequentially to a file.

```
// write lines to a file
try{
    PrintWriter pw = new PrintWriter("data.csv");
    pw.println("one,two,three,four");
    pw.close();
}

catch(IOException ex)
{
    System.out.println(ex.getMessage());
}
```

```
one,two,three,four
```

Read a csv file.

A csv file is a file where data is stored row by row in columns separated by commas. The **Scanner** class is used again to read lines one by one sequentially from a file. We use the split method from the String class to separate each line into an array of words. Each word is called a **token**.

data.csv:

```
one,two,three,four
five,six,seven,eight
```

```

// read csv file
// Open the file for read
try
{
    Scanner fsc = new Scanner(new File("data.csv"));

    // read line one at a time
    // till the file is empty
    while (fsc.hasNextLine())
    {
        line = fsc.nextLine();
        String[] words = line.split(",");

        for(String word: words)
        {
            System.out.println(word);
        }
    }

    System.out.println("");
    fsc.close();
}

catch(FileNotFoundException ex)
{
    System.out.println(ex.getMessage());
}

```

Output token words

```

One
two
three
four

```

Append line to end of file

We can also write lines to the end of a file (append) using the **PrintWriter** class and open the file with the **FileWriter** class set to append with the true argument.

```
// write lines to end of a file
try{
PrintWriter pw2 = new PrintWriter(new FileWriter("data2.txt",true));
    pw2.write("tomorrow");
    pw2.close();
}
catch(IOException ex)
{
    System.out.println(ex.getMessage());
}
}
```

Read the lines back from data2.txt and you will get:

hello
goodbye
tomorrow

Write A Object to a File

You can write objects to a file as long as they implement the **Serializable** interface located in

java.io.Serializable;

Most of the built in Java objects already implement the **Serializable** interface like the String class.

The first step is to make a Java class to implement the **Serializable** interface. We will make a Book class that store's a title and description.

```
import java.io.Serializable;

class Book implements Serializable
{
    private String title;
    private String author;

    public Book(String title, String author)
    {
        this.title = title;
        this.author = author;
    }

    public String toString()
    {
        return "Book " + title + " written by " + author;
    }
}
```

If your class does not implement the Serializable interface you will get the following message when you run the program:

writing aborted; java.io.NotSerializableException: Book

You can put the book class at the top of your Lesson9.java file if you do not include the public modifier, or you can out it into a java file called Book.java.

The next thing we make a Book object from the Book class definition.

```
Book book = new Book("Wizard of OZ","L. Frank Baum");
```

We can now write out the book object to a file called "test.bin". This would be a binary file that has un readable character in it.

```

// write to output stream
try {
    FileOutputStream fos = new FileOutputStream("Test.bin");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(book);
    oos.close();
}
catch (Exception ex) {
    System.out.println(ex.getMessage());
}

```

The ObjectOutputStream object oos is used to write the object to a file

We can now read in the Book object from the file using the ObjectInputStream object ois.

```

// read object
try {
    FileInputStream fis = new FileInputStream("test.bin");
    ObjectInputStream ois = new ObjectInputStream(fis);
    Book b2 = (Book)ois.readObject();
    ois.close();
    System.out.println(b2); // print out book object
}
catch (Exception ex) {
    System.out.println(ex.getMessage());
}

```

The output is as follows:

Book Wizard of OZ written by L. Frank Baum

LESSON 9 HOMEWORK

Question 1

Write a small 5 line story to a file called “story.txt” then close the file. Then open the story text file that has the small story in it, and count the number of letters, words, sentences and lines. Words are separated by spaces and new lines. Sentences are separated by periods “.” or other punctuation like “?”.

Lines are separated by ‘\n’. Words may contain numbers and punctuation like apple80 and don’t.

Print a report to the screen: the number of letters, words, sentences and lines. Also write the report to a file called report.txt. Open the report file and display the report file lines to the screen. Call your java file Homework9.java.

Question 2

Write a program that writes out another Java program to a file, then open up the file you wrote that contains the Java program and execute it.

Algorithm:

1. Open up a file with PrintWriter with a java extension like “Test.java”
 2. Write lines to a file with print statements like:
 3. `pw.write(“System.out.println(“I like programming”)”)`
you need to alternate between hardcoded double quotes and specified “ double quotes
 4. close the file
 5. open the Test.java file in your Java IDE and run the Java file.
- It should print out

I like programming

You can put your code in your java file Homework9.java or make a new Java File called Homework9Question2.java

Bonus:

Add more print , input, etc Java statements, you could even write out the question code so you have a program the writes out itself!

LESSON 10 Abstract Classes and Methods, Polymorphism and Java Objects

Abstract classes and methods

A method with no code is called an **abstract method**. A class that contains an abstract method is known as an **abstract class**. An abstract class may contain ordinary methods and abstract methods. An abstract method has a method signature but no underlying code. Signature refers to the method definition heading. The code for an abstract method is to be implemented in the derived class. Here is an abstract method to calculate weekly pay.

```
// calculate weekly pay  
public abstract double weeklyPay();
```

If a class has an abstract method then it must be marked abstract. Here is an abstract Employee class that has an abstract method **weeklyPay**.

```
public abstract class Employee  
{  
    // calculate weekly pay  
    public abstract double weeklyPay();  
}
```

In run time the code in the derived class is executed. When a class contains the same name and signature as a method of the base class this is known as **overriding**. Here is a derived class **Manager** that implements the **weeklyPay** method of the super class **Employee**.

```
public class Manager extends Employee {  
  
    // Constants  
    public static final int WEEKS_IN_YEAR = 52
```

```

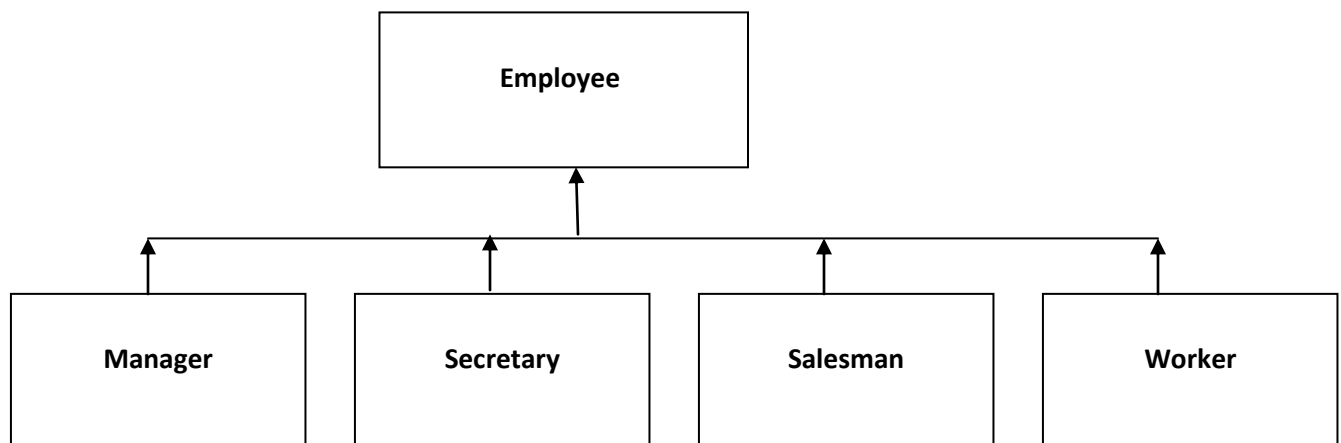
// calculate weekly pay
public double weeklyPay()
{
    double pay = getSalary() / WEEKS_IN_YEAR;
    return pay;
}
}

```

Abstract classes are never instantiated because they have incomplete (missing) code. All the code is to be implemented in the derived classes. Only the derived classes are instantiated.

Polymorphism

Polymorphism is another powerful concept in Object Oriented Programming. Polymorphism lets one super class represent many other different types of derived classes and then execute a method from each one that will produce a different behaviour. We can have an Employee super class to represent different kinds of Employees. Managers, Secretaries, Salesman and Workers derived classes.



We make our Employee class abstract meaning it can never be instantiated. We also need an abstract method to calculate the employee's wages for the week. An abstract method is a method that is defined in the super class that has a

method header definition but no programming statements. The programming statements will be defined in the derived class.

This is known as method **overriding** since it has the same method name and identical parameter data types. We also have method **overloading** where we have same method name but different parameter data types.

Here is our Employee super abstract class.

```
/*
 * Employee abstract class
 */

public abstract class Employee {

    private String name;
    private String id;
    private double salary;

    // construct Employee with name, id and salary
    public Employee(String name,String id,double salary)
    {
        this.name=name;
        this.id=id;
        this.salary = salary;
    }

    // return employee name
    public String getName()
    {
        return this.name;
    }

    // return employee id
    public String getID()
    {
        return this.id;
    }
}
```

```

    }
    // return employee yearly salary
    public double getSalary()
    {
        return this.salary;
    }

    // assign name
    public void setName(String name)
    {
        this.name=name;
    }

    // assign id
    public void setID(String id)
    {
        this.id=id;
    }

    // assign salary
    public void setSalary(double salary)
    {
        this.salary = salary;
    }

    // calculate weekly pay
    public abstract double weeklyPay();

    // return employee info
    public String toString()
    {
        return name + " " + id + " $" + salary;
    }
}

```

We now need to make the derived classes. You can put them in the same file as the Employee class or in separate files. If you put them into the Employee.java file, then the derived classes cannot have the public access modifier. We will have 4 derived classes. Each derived class will calculate the pay for the week differently, calculated from the yearly salary. Each derived class will calculate the weekly pay separately as follows:

Derived class	How to calculate weekly pay
Manager	Divide yearly salary by number of weeks in year
Secretary	Divide yearly salary by number of weeks in year plus \$100 bonus
Salesman	Divide yearly salary by number of weeks in year plus sales Commission rate
Worker	Divide yearly salary by number of weeks plus any overtime time and a half

It is probably best to put each derived class in a separate file. This way it makes thing easier these classes to use in other projects. Our Manager class as do the other class uses constants to store and represent herd coded values. We do not want numbers in our program, since they may have to be changed later. Constants just store a value. Once declared and initialized the value cannot be changed. Constants have the keyword final, Constants are also declared as **static** so that they only consume once space in memory.

```
public static final int WEEKS_IN_YEAR = 52;
```

This way many objects can share the same constant, and no duplicate memory space is used.

```
/*  
 * Manager derived class  
 *  
 */
```

```

public class Manager extends Employee {

    // Constants
    public static final int WEEKS_IN_YEAR = 52;

    // construct Manager with name, id and salary
    public Manager(String name,String id,double salary)
    {
        super(name,id,salary);
    }

    // calculate weekly pay
    public double weeklyPay()
    {
        double pay = getSalary() / WEEKS_IN_YEAR;
        return pay;
    }

    // return manager info
    public String toString()
    {
        return "Manager " + super.toString();
    }
}

/*
 * Secretary derived class
 *
 */

```

```

public class Secretary extends Employee {

    // Constants
    public static final int WEEKS_IN_YEAR = 52;
    public static final double BONUS = 100;

```

```

// construct Secretary with name, id, salary
public Secretary(String name,String id,double salary)
{
    super(name, id, salary);
}

// calculate weekly pay
public double weeklyPay()
{
    double pay = getSalary()/ WEEKS_IN_YEAR + BONUS;
    return pay;
}

// return employee info
public String toString()
{
    return "Secretary " + super.toString();
}
}

/*
 * Salesman derived class
 *
 */

public class Salesman extends Employee {

    // Constants
    public static final double COMMISSION_RATE = .25;

    // weekly sales
    private double sales;

```

```

// construct Salesman with name, id, salary and sales
public Salesman(String name,String id,double salary, double sales)
{
    super(name, id, salary);
    this.sales = sales;
}

// calculate weekly pay
public double weeklyPay()
{
    double pay = getSalary() + sales * COMMISSION_RATE;
    return pay;
}

// return employee info
public String toString()
{
    return "Salesman " + super.toString() + " Sales: " + sales;
}
}

/*
 * Worker derived class
 */
public class Worker extends Employee {

    // Constants
    public static final int WEEKS_IN_YEAR = 52;
    public static final int HOURS_IN_WEEK = 40;
    public static final double OVERTIME_RATE = 1.5;

    // hours overtime
    private int overtime;

    // construct Worker with name, id, salary

```



```

public Worker(String name,String id,double salary, int overtime)
{
    super(name, id, salary);
    this.overtime = overtime;
}

// calculate weekly pay
public double weeklyPay()
{
    double pay_rate = getSalary()/ WEEKS_IN_YEAR /
        HOURS_IN_WEEK;
    double pay = getSalary()/ WEEKS_IN_YEAR
        + overtime * pay_rate * OVERTIME_RATE;
    return pay;
}

// return employee info
public String toString()
{
    return "Worker " + super.toString();
}
}

```

Polymorphism is like a giant else-if statement. If it is a Salesman derived class object, then calculate weekly wage using sales and commission. Our first step is to make an array to hold Employee derived objects.

```

// make an array of employees
Employee[] employees = new Employee[4];

```

Next, we will add with the derived objects to the Array. Each derived object gets a name, employee id, yearly salary, the salesman gets the sales for the week and the worker get the number of over time hours for the week.

```
// fill array with derived objects
employees[0] = new Manager("Tom Smith","E1234",100000);
employees[1] = new Secretary("Mary Smith","E5678",40000);
employees[2] = new Salesman("Bob Smith","E1111",20000,10000);
employees[3] = new Worker("Joe Smith","E2222",30000,5);
```

Next, we loop through the array printing out the employee info and the calculated weekly pay. Notice the weekly pay is different for each employee type, this is what we want, automatic selection. This is polymorphism in action.

```
// loop through employee array
// print out employee info
// calculate weekly pay.

for(int i=0;i<employees.length;i++)
{
    // print out employee info
    System.out.println(employees[i].toString());

    // calculate weekly pay
    double pay = employees[i].weeklyPay();

    // print weekly pay
    System.out.println("My weekly pay is: $" +
        String.format("%.2f",pay));
}
```

To do:

Add an abstract **raise** method to the Employee class to give each employee different types of raises. This method will return a double. Some could be percentages, others can be a fixed value, how many sales or how many over time hours worked..

In another loop in the main method give each employee a raise and then recalculate their weekly pay and print out the results. Use an enhanced for loop.

Abstract classes implementing interfaces

We can make an abstract class called **Animal** the implements the **IAAnimal** interface from previous lessons. The IAnimal interface contains the sound method, what sound a animal makes.

interface IAnimal:

```
{  
    public String sound();  
}
```

The abstract Animal class will implement the IAnimal interface. The sound method specified in the IAnimal interface will be implemented in the classes derived from the abstract Animal class. The abstract Animal class will also have a abstract method called getType that will return the type of Animal. The abstract method getType() will also be implemented by the derived class of the Animal class. Note: The sound method cannot be an abstract method in the Animal class, it can only be implemented as a non-abstract method.

class abstract Animal implements IAnimal

```
{  
    private String name;  
  
    public Animal(String name)  
    {  
        this.name = name;  
    }  
  
    public abstract String getType();  
  
    public String toString()  
    {  
        return "my name is: " + name;  
    }  
}
```

We then can make a derived Cat class that inherits the Animal class.

```

public class Cat extends Animal
{
    public Cat(String name)
    {
        super(name);
    }

    public String sound()
    {
        return "meow"
    }

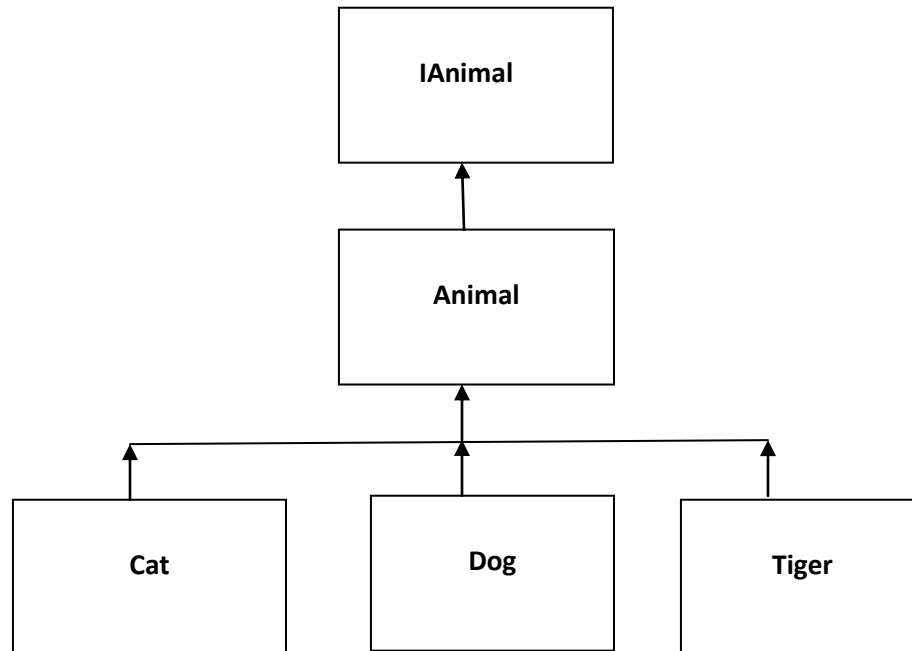
    public String getType()
    {
        return "cat";
    }

    public String toString()
    {
        return "I am a cat" + super.toString();
    }
}

```

To do:

Make derived classes Dog and Tiger, and in a main method make an array of IAnimals interfaces and print out what animal it is what name, type of animal and what sound it makes. Put the main method in a java class called Animals and in a file called Animals.java.



Java Built in Classes and Objects.

Java has lots of built in objects, meaning classes already written for you that you can use right away. We cannot cover them all, but we can cover a few of them. We have already covered the String object and the File Access objects; all these objects have lots of methods to do many different operations. The Math class is used a lot for mathematical calculations, The Math class is not considered an Object but just a class because all its methods are static.

The random method of the math class is used to generate a random number between 0 and 1.0;

```
double d = Math.random();  
System.out.println(d);
```

Generates a double number between 0 and 1.0 like 0.6549805947125389
We can multiply the random() method by another to get larger random number.

```
Double d = Math.random() * 10;  
System.out.println(d);
```

Generates a double number between 0 and 10 like 6.87654343233444
You can **typecast** the double number to an int to get an integer result like this:

```
int x = (int)Math.random() * 10;  
System.out.println(x);
```

Here we generate a random number between 0 and 9 like 7; The random methods generate a decimal number between 0 and 1.0. We multiply this number by 10 and convert to a int number using (int). This is known as **type casting**.
You will be using the random method of the Math class allot.

Here are some of the math class common used fields and methods

Following are the fields for java.lang.Math class

Math.E – the base of the natural logarithms.

Math.PI – the ratio of the circumference of a circle to its diameter.

Here are the Math Class methods:

Method	Description
static double abs(double a)	returns the absolute value of a double value.
static double acos(double a)	returns the arc cosine of a value; the returned angle is in the range 0.0 through pi
static double asin(double a)	returns the arc sine of a value; the returned angle is in the range -pi/2 through pi/2.
static double atan(double a)	returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2
static double atan2(double y, double x)	returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).
static double ceil(double a)	returns the smallest double value that is greater than or equal to the

	argument and is equal to a mathematical integer.
static double cos(double a)	returns the trigonometric cosine of an angle
static double floor(double a)	returns the largest double value that is less than or equal to the argument and is equal to a mathematical integer
static double log(double a)	returns the natural logarithm (base e) of a double value.
static double log10(double a)	returns the base 10 logarithm of a double value
static double max(double a, double b)	returns the greater of two double values
static double min(double a, double b)	returns the smaller of two double values
static double pow(double a, double b)	returns the value of the first argument raised to the power of the second argument
static double random()	method returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
static long round(double a)	returns the double value that is closest in value to the argument and is equal to a mathematical integer
static double sqrt(double a)	returns the correctly rounded positive square root of a double value.
static double tan(double a)	returns the trigonometric tangent of an angle
static double toDegrees (double angrad)	converts an angle measured in radians to an approximately equivalent angle measured in degrees
static double toRadians (double angdeg)	converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Random Class

The Random class is also used to generate Random numbers. The Random class has many methods to generate Random numbers.

You must instantiate a Random object before you can use it because the Random class methods are not static.

```
Random rgen = new Random();
```

you need to put the following import statement on top of your Java file, so the Java compiler will know about it.

```
import java.util.Random;
```

You can generate double random number using the nextDouble method

```
double d = rgen.nextDouble();
```

generates a double number between 0 and 1.0 like 0.6549805947125389

You can generate integer random number using the nextInt method. This method allows you to specify the upper bound -1

```
int r = rgen.nextInt(10)
```

Generates a random number between 0 and 9 like 7

enum

enums allow you to associate a constant with a label and conveniently group the labels together under a common purpose. This is quite an advantage to group a collection of related constants together under a common name. A good example are the days of the week: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.

A enum lets you list all the labels and automatically assign constants to them. The constants are String representation of the label. The constants let you compare the label's.

A **enum** is little bit like a class since you can have methods to do operations on the labels. Here is a enum for the days of the week:

```
enum Weekdays{  
    Sunday, Monday, Tuesdays, Wednesday, Thursday, Friday, Saturday  
}
```

An enum is analogous to using constant's, you use the labels just like constants. The labels are actually string values. Sunday would get the string value "Sunday".

The advantage of enum is that all the weekdays get to be in a Weekdays group. This makes your program more professional because you now know what group a label belongs to. When you use enums Java forces you to use the enum name with the label., Example: Weekdays.Monday;

You would use the Weekdays enum like this:

```
// make a day to represent weekday Monday  
Weekdays day1 = Weekdays.Monday;  
System.out.println(day1); // Monday  
  
// make a day to represent weekday Tuesday  
Weekdays day2 = Weekdays.Tuesday;  
System.out.println(day2); // Tuesday  
  
// We can compare 2 weekdays are equal like this:  
if(day1 == day2)  
{  
    System.out.println(day1 + " equals " + day2);  
}
```

```

else{
    System.out.println(day1 + " not equals " + day2);
    // Monday not equals Tuesday
}

// get a weekday from a string
Weekdays day3 = Weekdays.valueOf("Wednesday");
System.out.println(day3); // Wednesday

// print weekday string
System.out.println(Weekdays.Wednesday); // Wednesday

// get index of weekday
int index = Weekdays.valueOf("Monday").ordinal(); // 1
System.out.println(index); // 1

// to print out a list of Weekdays
for (Weekdays day : Weekdays.values()) {
    System.out.println(day);
}

```

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

Packages

Packages represent another organizational unit. Packages allow you to group common purpose classes together under a common name. For example: You may have input and output classes under a io package. Data classes under a data package, display classes under a View class, operational classes under a Controller package. Larger programming projects must use packages to organize their classes into computational units.

Without packages the project would be unorganized and difficult to manage. Unfortunately small programming projects find packages a nuisance and frustrating and the cause of all compilation errors. A small program in a package may be difficult to compile and run.

Also using a class from another project may be difficult to use, since its package name may be different from your class package name. When a package is used the java file must be inside a folder with the same package name. Package names are all lower case as to match the folder name that houses it.

To declare class in a package:

```
package mypackage;  
  
public class MyClass  
{  
    public MyClass()  
    {  
    }  
    public String toString()  
    {  
        return "I like packages";  
    }  
}
```

For another class to use the packaged class you must import the package at the very top of your program:

```
import mypackage.MyClass  
  
public class Test  
{  
    public static void main(String[] args)  
    {  
        MyClass mc = new MyClass()  
        System.out.println(myClass.toString());  
    }  
}
```

For a class to import a package the folder holding the package must be in the same folder as the class using the package.

Lambda and Functional Interfaces

A Java lambda expression is just like a method which can be created without belonging to any class. They are anonymous methods (methods without names) and are used to implement a method defined by a **functional interface**. A functional interface is an interface that contains one and only one abstract method.

```
interface Operation {  
    int calculate(int n);  
}
```

Lambda expressions use the arrow operator `->` to specify the lambda expression. It divides the lambda expressions in two parts:

`(n) -> n*n`

The left side specifies the parameters required by the expression, which could also be empty if no parameters are required. `() -> System.out.println("Hello");`

The right side is the lambda body which specifies the actions of the lambda expression which is a programming statement using the parameters (if any) from the left hand statement like the `n*n`.

We declare and define a lambda expression using the functional `Operation` interfaces defined above that increments a number as follows:

```
Operation inc = (n) -> n+1; // means return n+1
```

We declare and define a lambda expression that decrements a number as follows:

```
Operation dec = (n) -> n-1; // means return n-1
```

We now increment and decrement a number using the lambda expression and call the calculate method defined in the functional interface:

```
int result = inc.calculate(5);  
System.out.println(result); // 6  
  
result = dec.calculate(5);  
System.out.println(result); // 4
```

We can make another functional interface that has a method to add two numbers as follows:

```
interface Operation2 {  
    int calculate(int n, int n2);  
}
```

We then declare and define a lambda expression that adds two numbers as follows:

```
Operation2 add = (a, b) -> a + b;
```

We now add two numbers using the lambda expression and call the calculate method defined in the functional interface

```
result = add.calculate(3,4);  
System.out.println(result);
```

It seems to be a three step process:

(1) Define the functional interface:

```
interface Operation {  
    int calculate(int n);
```

(2) Write the lambda expression?:

```
Operation inc = (n) -> n+1;
```

(3) Use the lambda expression to calculate a value:

```
int result = add.calculate(3,4);
```

A lambda expression may also include additional blocks of code to do additional calculations. Here is a lambda expression that sums the numbers between two values. Notice we put the block of code between 2 curly brackets ending in a semi colon. { };

```
Operation2 sum = (a, b) -> {  
  
    int v = 0;  
  
    for(int i=a;i<=b;i++)  
    {  
        v += i;  
    }  
  
    return v;  
}; // don't forget the semi-colon  
  
result = sum.calculate(1,5);  
System.out.println(result); // 15
```

TO DO:

Define a functional interface that can do operations on a input string. Write a lambda expression to print out a string message and then use the lambda expression to print out somebody's name. Next write a lambda expression to reverse the string. You will need to declare your functional interface outside (above) the main function() since you cannot declare a interface inside another method just inside another class. Very awkward don't you think?

You can also have Generic functional interfaces as follows:

```
interface TOperation2<T>  
{  
    T calculate(T n, T n2);  
  
}
```

We now write our Generic lambda expression as follows using a **Double**

```
TOperation2<Double> tadd = (a, b) -> a + b;
```

We then call the calculate method to do the addition:

```
double result2 = tadd.calculate(3.7,4.8);  
System.out.println(result2); // 8.5
```

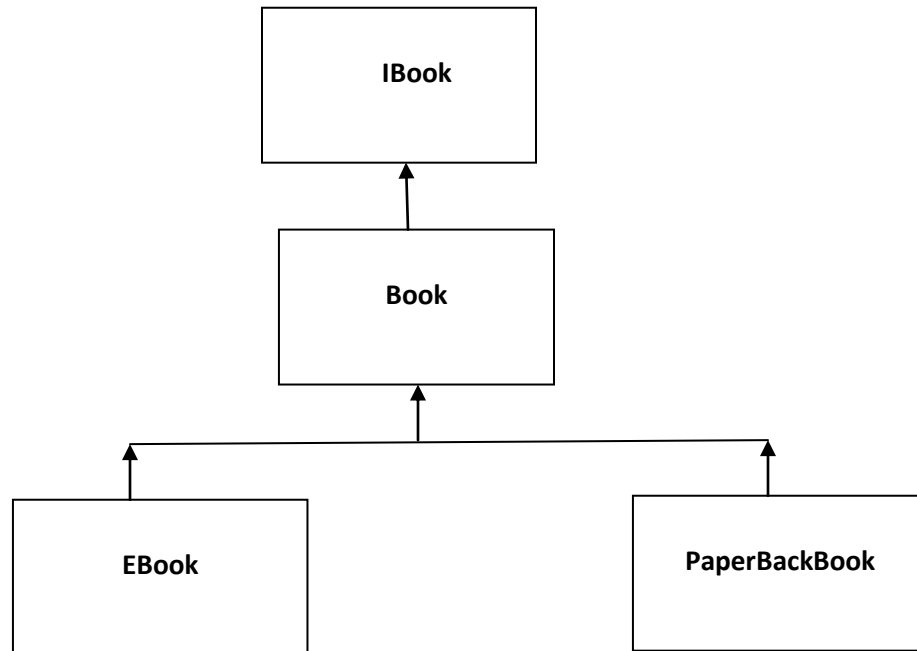
TO DO:

Use the generic functional interface that can join two Strings together.

LESSON 10 HOMEWORK

Question 1

Make a book interface call **IBook** that has a method to calculateTotalCost to purchase a book. Make a Abstract Book class called **Book** that implements the **IBook** interface and has the abstract method calculateTax and calculateShipping. A Book has a title, cost and a toString() method. Make derived classes **EBook** and **PaperBackBook**. A EBook has no shipping cost but a PaperBackBook has a shipping cost. A EBook has 10% tax rate but a PaperBackBook has no tax. Make a class called Homework10 with a main method. In the main method make an array of IBook interfaces. Populate the array with one EBook and one PaperBook. In a loop print out the book's detail, calculate the shipping and tax and add to the cost as final total cost. Print out the cost, tax, shipping charges and final cost using the calculateTotalCost method.



Question 2

Use a generic functional interface to make a generic calculator we did from Lesson 9. Call your homework program LambdaGenericCalculator.java

```
/*  
 * Generic TCalculator interface  
 * specifies the methods a Generic Calculator should have  
 */
```

```
public interface TCalculator<T> {
```

```
    public T add(T a, T b);  
    public T sub(T a, T b);  
    public T mult(T a, T b);  
    public T divide(T a, T b);
```

```
}
```


LESSON 11 RECURSION

When a function calls itself it is known as **recursion**. Recursion is analogous to a while loop. Most while loop statements can be converted to recursion, most recursion can also be converted back to a while loop.

The simplest recursion is a function calling itself printing out a message.

```
public static void print_message()  
{  
    System.out.println("I like programming");  
    print_message();  
}
```

I like programming
I like programming
I like programming
I like programming
I like programming
...

Unfortunately this program will run forever, so you will need to stop the program somehow while it is running.

We can add a counter **n** to it so it can terminate at some point.

```
public static void print_message(int n)  
{  
    if(n > 0)  
    {  
        System.out.println("I like programming");  
        print_message(n-1);  
    }  
}
```

Now the program will print the message **n** times

Every time the `print_message` function is called **n** decrements by 1. When **n** is 0 the recursion stops. You may place the statement `System.out.println("I like programming\n")` before or after the recursive call. If you put it before then the message is printed first before each recursive call.

If you put after than the message is printed after all the recursive calls are made. There is quite a difference in program execution. The operation is very similar to the following while loop:

```
n = 5
while(n > 0)
{
    System.out.println("I like programming\n");
    n--;
}
```

You should now run the recursion function

You would call the function like this:

```
print_message(5);
```

It will print I like programming 5 times.

```
I like programming
I like programming
I like programming
I like programming
I like programming
```

Recursion is quite powerful, a few lines of code can do so much.

Our next example will count of numbers between 1 and n. This example may be more difficult to understand, since recursion seems to work like magic, and operation runs in invisible to the programmer.

```
public static int countn(int n)
{
    if(n == 0)
    {
        return 0;
    }
    else
    {
        return countn(n-1) + 1;
    }
}
```

count(5) would return 5 because $1 + 1 + 1 + 1 + 1 = 5$

You can run it in a program like this:

```
System.out.println(countn(5) ); // 5
```

When (n == 0) this is known as the base case. When n == 0 the recursion stops and 0 is return to the last recursive call. Otherwise the **countn** function is called and n is decremented by 1.

It works like this:

```
main calls countn(5) with n = 5
countn(5) calls countn(4) with n=4
countn(4) calls countn(3) with n=3
countn(3) calls countn(2) with n = 2
countn(2) calls countn(1) with n = 1
countn(1) calls countn(0) with n = 0
```

```
countn(0) returns 0 to count(1) since n == 0
countn(1) add's 1 to the return value 0 and then returns 1 (0 + 1) to count(2)
countn(2) add's 1 to the return value 1 and then returns 2 (1 + 1) to count(3)
countn(3) add's 1 to the return value 2 and then returns 3 (2 + 1) to count(4)
countn(4) add's 1 to the return value 3 and then returns 4 (3 + 1) to count(5)
countn(5) add's 1 to the return value 4 and then returns 5 (4 + 1) to main()
```

```
main() receives 5 from count(5) and prints out 5
```

The statement **return countn(n-1) + 1** is used to call the function recursively and also acts as a place holder for the value returned by the called function **countn**. The returned value is then added to 1 and then returned.

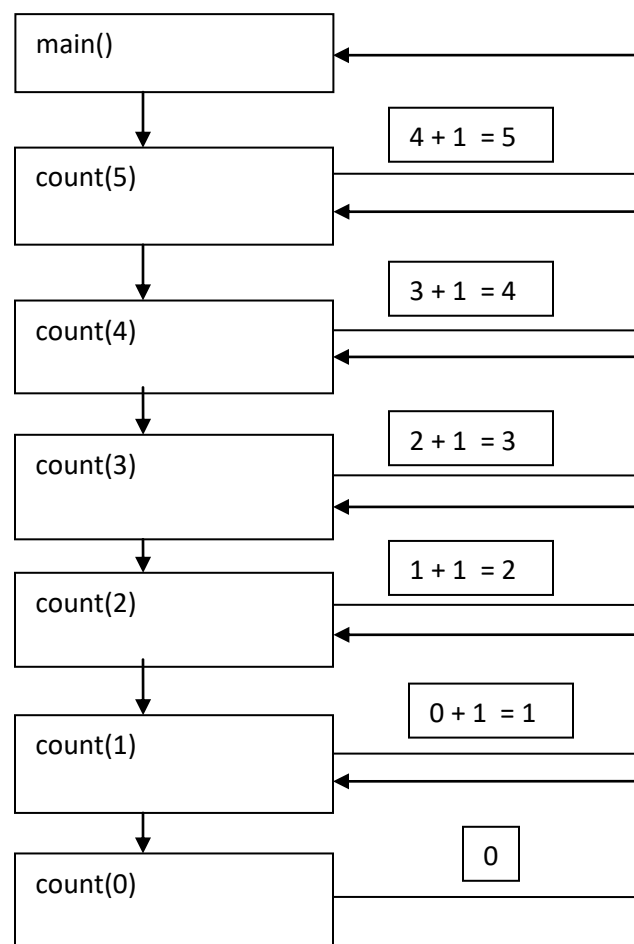
The value returned from the recursive **countn** function is the previous value and the 1 is the present value to be added to the previous value.

We could rewrite the recursive part as follows:

```
int x = countn(n-1);  
return x + 1;
```

x will now receive the return value from the countn recursive function call and 1 will be added to the return value and this new value will be returned to the caller.

If you can understand the above then you understand recursion. If you cannot then maybe the following diagram will help you understand.



You probably don't need to understand how recursion works right away. Sometime you just need to accept things for now then understand later. One day it will hit you when you are thinking about something else. Basically recursion works like this:

For every recursive function call the parameter and local variables are stored. Technically they are stored in temporary memory called a stack. Every time the function returns the previous numbers that were stored are restored and now become the current number, to be used to do a calculation. The numbers are restored in **reverse** order.

Function call/ return	N
call count(n-1)	5
call count(n-1)	4
call count(n-1)	3
call count(n-1)	2
call count(n-1)	1
count(n-1) returns 0	0
count(n-1) returns 0 + 1	1
count(n-1) returns 1 + 1	2
count(n-1) returns 2 + 1	3
count(n-1) returns 3 + 1	4
count(n-1) returns 4 + 1	5

The thing to remember about recursion is it always return's back where it is called. A recursive function call behaves the same way as a non-recursive function, program execution resumes right after the recursive function call.

Here are some more recursive function examples:

```
// Sum of numbers between 1 to n
public static int sumn(int n)
{
    if(n == 0)
    {
        return 0;
    }
    else
    {
        return sumn(n-1) + n;
    }
}
```

sumn(5) would return 15

You can run it in a program like this:

```
System.out.println(sumn(5) ); // 15
```

It works similar to countn instead of adding 1 its adds n's.

0+1+2+3+4+5 = 15

Our counter n serves 2 purposes a recursive counter and a number to add.

The previous value is returned from the recursive sumn function and the present value is n. The value n is restored in reverse order that is saved in. Every time the recursive sumn function is called it is saved. Every time the recursive sumn function is returned n is restored in traverse order.

Main calls sumn(5)

n	Call sumn(n-1)	Previous	present	calculated	Returned value
5	Call sumn(4) (5-1)				
4	Call sumn(3) (4-1)				
3	Call sumn(2) (3-1)				
2	Call sumn(1) (2-1)				
1	Call sumn(0) (1-1)				
0	Return 0				
1	Return sumn(0) +1	0	1	0 + 1	1
2	Return sumn(1)+2	1	2	1+2	3
3	Return sumn(2)+3	3	3	3+3	6
4	Return sumn(3)+4	6	4	6+4	10
5	Return sumn(4)+5	10	5	10+5	15 to main

Main receives 15 from sum(5)

The sumn function just returns the value 0 when n is 0 and other times just calls itself then adds the value n to the returned value. Basically sumn is just calling itself and adding n to the returned values. For each recursive call sumn(n-1) is called first after each recursive call the n is added and a value is returned,. N is added in reverse order because sumn(n-1) is called first n times.

Multiply numbers 1 to n (factorial n)

We can also make a **multn** function which multiplies n rather than adding n. This is basically factorial n.

```
public static int multn(int n)
{
    if(n ==0)
    {
        return 1;
    }

    else
    {
        return multn(n-1) * n;
    }
}
```

multn(5) would return 120

since $1*2*3*4*5 = 120$

Our base case returns 1 rather than 0 or else our result would be 0;

You can run it in a program like this:

```
System.out.println(multn(5) ); // 120
```

Power x^n

Another example is to calculate the power of a number x^n

In this case we need a base parameter b and an exponent parameter n .

```
public static int pown(int b, int n)
{
    if(n ==0)
    {
        return 1;
    }

    else
    {
        return pown(b,n-1) * b;
    }
}
```

pown(2,3) would return 8 because $2*2*2=8$ since $2^3=8$

You can run it in a program like this:

```
System.out.println(pown(2,3) ); // 8
```

Every time a recursive call is made the program stores the local variables in a call stack. Every time recursive call finishes executing, the save local variables disappear and the previous local variables are available. These are the ones present before the recursive function was called. These save variables may now be used in the present calculations.

For the above example $2^3=8$ the call stack would look like this.

						n=0	
						b=2	1
						n=1	n=1
						b=2	b=2
							2
						n=4	n=2
						b=2	b=2
							n=2
						b=2	b=2
							4
						n=5	n=5
						b=2	b=2
						n=3	n=3
						b=2	b=2
							n=3
						b=2	b=2
							8

Every time the recursive function finished executing it returns a value. Each returning value is multiplied by the base b. In the above case the returning values are 1,2,4 and 8

The return value is the value from the previous function multiplied by b (2)

return pown(b,n-1) * b;

the function first returns 1 then $1 * b = 1 * 2 = 2$ then $2 * 2 = 4$ and finally $4 * 2 = 8$

efficient power x^n

A more efficient version of pown can be made relying on the fact then even n can return $b * b$ rather than just return $* b$ for odd n

```
public static int pown2(int b,int n)
{
    if (n == 0)
    {
        return 1;
    }
}
```

```

if (n % 2 == 0)
{
    return pown2(b, n-2) * b * b;
}

else
{
    return pown2(b, n-1) * b;
}
}

```

Operation is now much more efficient $1 * 2 * 4 = 8$

You can run it in a program like this:

```
System.out.println(pown2(2,3) ); // 8
```

Summing a sequence

Adding up all the numbers in a sequence

$n \quad (n * (n + 2)) / 2$

0	0
1	1
2	4
3	7
4	12
5	17

Total:	42

```

public static int seqn(int n)
{
    if(n == 0)
    {
        return 0;
    }

    else
    {
        int x = (n * (n + 1))/ 2;
        System.out.println( x );
        return x + seqn(n-1);
    }
}

```

seqn(5) would return 35 because $0 + 1 + 3 + 6 + 10 + 15 = 35$

You can run it in a program like this:

```

System.out.println(seqn(5) ); // 35

```

You can print out the sequence by modifying the **seqn** function like this:

```

public static int seqn2(int n)
{
    if(n == 0)
    {
        return 0;
    }
    else
    {
        int x = (n * (n + 1))/ 2;
        System.out.println( x );
        return x + seqn2(n-1);
    }
}

```

You can run it in a program like this:

```
System.out.println(seqn2(5) );
```

You will get an output like this:

```
15  
10  
6  
3  
1  
35
```

The sequence printed backwards and the final sum is 35

To do:

Try this formula: $f(n-1) + 2 * (n-1)$

Fibonacci sequence

Recursion is ideal to directly execute recurrence relations like Fibonacci sequence

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,

In mathematical terms, the sequence f_n of Fibonacci numbers is defined by the recurrence relation.

$$f_n = f_{n-1} + f_{n-2}$$

with seed values

$$f_0 = 0 \text{ and } f_1 = 1.$$

A **recurrence relation** is an **equation** that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

```
public static int fib(int n)
{
    if (n == 0)
    {
        return 0;
    }

    else if (n == 1)
    {
        return 1;
    }

    else
    {
        return fib(n-1) + fib(n-2);
    }
}
```

Notice The recursive statement is identical to the recurrence relation

fib(10) would return 55 because $21 + 34 = 55$

You can run it in a program like this:

```
System.out.println(fib (10) ); // 55
```

Combinations

We can also calculate combinations using recursion.

Combinations are how many ways can you pick r items from a set of n distinct elements.

Call it **nCr** (n choose r)

5C2 (5 choose 2) would be

Pick two letters from set $S = \{A, B, C, D, E\}$

Answer: $\{A, B\}, \{B, C\}, \{B, D\}, \{B, E\}, \{A, C\}, \{A, D\}, \{A, E\}, \{C, D\}, \{C, E\}, \{D, E\}$

There are 10 ways to choose 2 letters from a set of 5 letters. The combination formula is

$$nC_r = n! / r!(n-r)!$$

The Recurrence relation for calculated combinations is:

base cases:

$$nC_n = 1$$

$$nC_0 = 1$$

recursive case:

$$nC_r = nC_{r-1} + nC_{r-2} \text{ for } n > r > 0$$

Our recursive function for calculating combinations is:

```
public static int combinations(int n, int r)
{
    if (r == 0 || n == r)
    {
        return 1;
    }

    else
    {
        return combinations(n-1, r) + combinations(n-1, r-1);
    }
}
```

combinations(5,2) would return 10

You can run like this:

```
System.out.println(combinations(5,2) ); // 10
```

Print a string out backwards

With recursion printing out a string backwards is easy, it all depends where you put the print statement. If you put **before** the recursive call then the function prints out the characters in reverse, since n goes from n-1 to 0. If you put the print statement **after** the recursive call then the characters are printed not in reverse since n goes from 0 to n-1.

```
// reverse a string  
public static void print_reverse(String s, int n)  
{  
    if(n == 0)  
    {  
        System.out.println(s.charAt(0));  
    }  
  
    else  
    {  
        System.out.print(s.charAt(n));  
        print_reverse(s, n-1);  
    }  
}
```

You would call the print_reverse function like this

```
string s = "hello";  
print_reverse(s, s.length()-1);
```

olleh

Check if a string is a palindrome

A palindrome is a word that is spelled the same forward as well as backwards:
Like "radar" and "racecar"

```
// return true if string is a palindrome otherwise return false
public static boolean is_palindrome( String s, int i, int j)
{
    if (i >= j)
    {
        return true;
    }

    else
    {
        if (s.charAt(i) != s.charAt(j))
            return false;
        else
            return is_palindrome(s,i+1, j-1);
    }
}
```

You would call the **is_palindrome** function like this:

```
string s2 = "radar";
System.out.println(s2 + " " + (is_palindrome(s2, 0,s2.length()-1)) );
```

Radar true

```
string s3 = "apple";
System.out.println(s3 + " " + (is_palindrome(s3, 0,s3.length()-2)));
```

apple false

Permutations

Permutations are how many ways you can rearrange a group of numbers or letters. For example for the string “ABC” the letters can be rearranges as follows:

ABC
ACB
BAC
BCA
CBA
CAB

Basically we are swapping character and then print them out
We start with ABC if we swap B and C we end up with ACB

```
// print permutations of string s
public static void printPermutations(char[] s, int i, int j)
{
    int k;
    char c;

    // print out permutation
    if (i == j)
    {
        System.out.println( new String(s) );
    }

    else
    {
        for (k = i; k <= j; k++) {

            // swap i and k
            c = s[i];
            s[i] = s[k];
            s[k] = c;
        }
    }
}
```

```

// recursive call
printPermutations(s, i + 1, j);

// put back, swap i and k
c = s[i];
s[i] = s[k];
s[k] = c;
}
}
}

```

You would call the print_reverse function like this:

```

char[] ca = {'A','B','C'};
printPermutations(ca, 0,ca.length-1);

```

ABC
ACB
BAC
BCA
CBA
CAB

Combination sets

We have looked at combinations previously where we wrote a function to calculate how many ways you can choose r letters from a set of n letters.

nCr = n choose r

Combinations allow you to pick r letters from set $S = \{A, B, C, D, E\}$

$n = 5$ $r = 2$ $nCr = 5C2$

Answer: $\{A, B\}, \{B, C\}, \{B, D\}, \{B, E\}, \{A, C\}, \{A, D\}, \{A, E\}, \{C, D\}, \{C, E\}, \{D, E\}$

We are basically filling a second character array with all possible letters up to r .

Start with ABCDE we would choose AB then AC then AD then AE etc.
We use a loop to traverse the letters starting at n =0, and fill the comb string.
When n = r we then print out the letters stored in the comb string

```
public static void printCombinations(char[] s, char combs[],
    int start, int end, int n, int r)
{
    int i = 0;
    int j = 0;

    // current combination is ready to be printed
    if (n == r)
    {
        for (j = 0; j < r; j++)
        {
            System.out.print(combs[j]);
        }
        System.out.println("");
        return;
    }

    // replace n with all possible elements.
    for (i = start; i <= end && end - i + 1 >= r - n; i++)
    {
        combs[n] = s[i];
        print_combinations(s, combs, i+1, end, n+1, r);
    }
}
```

You would call the print_combinations function like this:

```
char[] ca2 = {'A','B','C','D','E'};
char[] combs = new char[ca2.length+1] ;
int r = 2
printCombinations(ca2, combs,0,ca2.length-1,0,r);
```

AB
AC
AD
AE
BC
BD
BE
CD
CE
DE

The difference between combinations and permutations is that in a combination you can have different lengths within the set, where as in permutations they are the same length as the set but different arrangements.

Determinant of a matrix using recursion

In linear algebra, the determinant is a useful value that can be computed from the elements of a square matrix. The determinant of a matrix A is denoted $\det(A)$, $\det A$, or $|A|$

In the case of a 2×2 matrix, the formula for the determinant is:

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

For a 3×3 matrix A, and we want the formula for its determinant $|A|$ is

$$\begin{aligned}
 |A| &= \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\
 &= aei + bgf - ceg - bdi - afh
 \end{aligned}$$

Each determinant of a 2×2 matrix in this equation is called a "minor" of the matrix A. The same sort of procedure can be used to find the determinant of a 4×4 matrix, the determinant of a 5×5 matrix, and so forth.

Our code actually follows the above formula, calculating and summing the minors.

```
// calculate determinant of a matrix
public static float determinant(float matrix[][], int size)
{
    int c;
    float det=0;
    int sign=1;
    float[][] b=new float[3][3];
    int i,j;
    int m,n;

    // base case
    if(size == 1)
    {
        return (matrix[0][0]);
    }
    else
    {
        det=0;
        for(c=0; c<size; c++)
        {
            m=0;
            n=0;
            for(i=0; i<size; i++)
            {
                for(j=0; j<size; j++)
                {
                    b[i][j] = 0;
                    if(i!=0 && j!=c)
                    {
                        b[m][n] = matrix[i][j];
                        if(n<(size-2))
                        {
                            n++;
                        }
                    }
                }
            }
        }
    }
}
```

```

        else
        {
            n=0;
            m++;
        }
    }
}
}
det = det + sign*(matrix[0][c]*determinant(b,size-1));
sign = -1*sign; // toggle sign
}
}
return (det);
}

```

-306

You call and run the determinant function like this:

```
float m[3][3] = {{6,1,1},{4,-2,5},{2,8,7}};
```

```
System.out.println ("det = " + determinant(m,m.length) );
```

There are many more recursive examples, too numerous to present.
If you do all the following to do questions you will be a recursive expert.

LESSON 11 HOMEWORK

Question 1

Write a recursive function **int addNumbers(int[] a, int n)** adds up all numbers in an array and returns the sum.

Question 2

Write a recursive function **int searchNumber(int[] a, int n, int x)** that searches for a number(x) in an array and return the index (n) of the number if found otherwise returns -1 if not found.

Question 3

Write a recursive function **int largestNumber(int[] a, int n)**, that returns the largest number in an array,

Question 4

Write a recursive function **int smallestNumber(int[] a, int n)** that returns the smallest number in an array,

Question 5

Write a recursive function **boolean searchDigit(int number, int x)** that searches for a digit in an positive number and return true if the number is found otherwise returns false if not found.

Question 6

Write a recursive function called **int sumDigits (int number)** that adds up all the digits in a positive number of any lengths. The recursive function receives an int number and returns the sum of all the digits.

Question 7

Write a recursive function called **void printArrayReverse(int a[], int n)** that prints an array backwards. The recursive **printArrayReverse** method receives the array and the array length – 1 prints the array in reverse. Make sure your method prints a new line at the end

Question 8

Write a recursive function called **void printArray(int a[], int n)** that prints an array. The recursive **printArray** method receives the array and the array length – 1 and prints the array. Make sure your method prints a new line at the end.

Question 9

Write a recursive function called **void reverseString(String s)** that reverses a string in place. The recursive **reverseString** method receives the string and returns the string in reverse. No printing is allowed. You need to use the substring method since you cannot replace individual letters in a string.

Question 10

Write a recursive function called **void formatNumber(String s)** that can insert commas in a number. For example 1234567890 becomes 1,234,567,890

Question 11

Write a recursive function **boolean isEven(int n)** that return true if a number has even count of digits or false if the number of digits is odd. Hint: subtract by -2.

Question 12

Write a recursive function **void printBinary(int d)** that would print a decimal number as a binary number. A binary number just has digits 0 to 1. Where as a decimal number has digits 0 to 9. The decimal number 5 would be 0101 in binary, since $1*1 + 0*2 + 1*4 + 0*8$ is 10. We are going right to left. To convert a decimal number to binary You just need to take mod 2 of a digit and then divide the number by 2

$$5\%2 = 1 : 1$$

$$5/2 = 2$$

$$2\%2 = 0 : 0$$

$$2/2 = 1$$

$$1\%2 = 1 : 1$$

$$1/2 = 0$$

$$0\%2 = 0 : 0$$

We are done so going backwards
5 in binary is 0 1 0 1

Question 13

Write a recursive function **boolean isPrime(int number, int n)** that returns true if a number is prime otherwise false.

A prime number can only be divided evenly by itself. 2,3,5,7, are prime numbers. You can use the mod operator % to test if a number can be divided evenly by itself. $6 \% 3 = 0$, so 6 can be divided evenly by 3 so 6 is not a prime number.

Algorithm:

If the number is less than 2 then it is not a prime number.

If the number is 2 then it is a prime number.

If the number can be divided evenly by n then it is not a prime number

If n reached the number then it is a prime number.

Start n with 2 and increment to the number recursively.

Examples:

isPrime(10,2)	false	10 is not a prime number
isPrime(7,2)	true	7 is a prime number

Question 14

Make a recursive function called **partition(int[] a,int n)** that will partition an array in place into odd and even numbers

Example: **partition([1, 2, 3, 4, 5, 6], 0)**

Array before:

[1, 2, 3, 4, 5, 6]

Array After:

[1, 3, 5, 2, 4, 6]

Use your **printArray** method to print out the array before and after the partition method is called.

Put all your functions in a java file called Homework11.java Test all the recursive functions in the main function.

Lesson 12 Regular Expressions

Regular expressions let's you search for string patterns in a text string. Regular expressions are a little difficult to understand and use, but once you realize they are just using letters used to form a pattern that you can match.

The simple lest regular expression is a string of letters like "are" that you can use to determine if a text string contains this pattern.

Example: "Happy days are here again" contains the pattern "are".

Java has the following classes to work with Regular Expressions located in the **java.util.regex** package:.

Java Regular Expression classes.

Class	Description
Pattern	Defines a Pattern to be used in a search
Matcher	Used to search for a Pattern and contains search information
Pattern SyntaxException	Indicates a syntax error in a regular expression pattern

Pattern class methods

Method	Description
static Pattern compile (String pattern, int flag)	Contains a pattern to compile and a flag to indicate how to search like case insensitive search. Returns a Pattern object that can be used to validate string using the compiled regular expression pattern.
Matcher matcher(String text)	Creates and returns Matcher object that will be used to match the given input string text against the compiled regular expression pattern
String[] split(String text)	Split the given input string accordingly the compiled regular expression pattern

Pattern class flags

Flag	Description
CASE_INSENSITIVE	Ignore case of letters
LITERAL	Ignore special characters
UNICODE_CASE	Used with CASE_INSENSITIVE to ignore the case of letters outside the English alphabet

Matcher class methods

Method	Description
boolean find()	Returns true if the compiled pattern was found in the string otherwise false
String group(int n)	Is used to find matched subsequences
int groupCount()	Used to find the total amount of matched subsequences

Example using find

The **find ()** function searches the string for a match, and returns true object if there is a match.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class Lesson12 {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("like",
        Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher("I like Java");
        boolean matchFound = matcher.find();
        if(matchFound) {
            System.out.println("Match found");
        }
        else {
            System.out.println("Match not found");
        }
    }
}
```

Match found

How it works:

```
Pattern pattern = Pattern.compile("like", Pattern.CASE_INSENSITIVE);
```

Is used to compile the pattern “like” and returns a Pattern object. A case insensitive search is to be performed,

```
Matcher matcher = pattern.matcher("I like Java");
```

The matcher method is used to search for the pattern “like” in the string “I like Java” and returns a Matcher object containing information about the search.


```
boolean matchFound = matcher.find();
```

The find method returns true if the pattern “like” is found in the string “I like Java”

Using Split

Split is used to separate a string text into individual words separated by the regular expression pattern.

```
pattern = Pattern.compile(" ", Pattern.CASE_INSENSITIVE);  
String[] result = pattern.split("I like Java");  
  
for(String w: result)  
{  
    System.out.println(w);  
}
```



```
I  
like  
Java
```

How it Works

```
pattern = Pattern.compile(" ", Pattern.CASE_INSENSITIVE);
```

Is used to compile the pattern “ ” (empty space) and returns a Pattern object. A case insensitive search is to be performed,

```
String[] result = pattern.split("I like Java");
```

The split method is used to search for the pattern " " (empty space) in the string 'I like Java' and separate into individual words in a returned array.

```
for(String w: result)
{
    System.out.println(w);
}
```

Used to print out the individual words stored in the returned array.

Sequences, Metacharacters and Sets

Regular expression have **sequences**, **metacharacters** and **sets** to make regular expressions more powerful.

A **special sequence** is a \ followed a characters and has a special meaning, like \s to represent white space.

Metacharacters are characters with a special meaning, like + which means 1 or more matches.

A **set** is a set of characters inside a pair of square brackets [] with a special meaning, where [A-z] matches any letter A to Z.

Here are the tables of Sequence, metacharacters and sets:

Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning.

Sequence	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from	"\d"

	0-9)	
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Java\Z"

Metacharacters

Metacharacters are characters with a special meaning that allows the metacharacter to represent many other characters. Example the dot . can represent any character.

Metacharacter	Description	Example
[]	A set of characters from start to end separated by a -	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters like \()	"\d"
.	Any character (except newline character)	". "
^	Starts with	"^happy"
\$	Ends with	"days\$"
*	Zero or more occurrences	"a*"
+	One or more occurrences	"a+"
?	Zero or more occurrence	"a?"
{ }	Exactly the specified number of occurrences	"a{2}"
	Either or	"yes no"
()	Capture and group	"(\w+)"

Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning.

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., , (), \$, { } has no special meaning, so [+] means: return a match for any + character in the string

Using metacharacters

If you want to match a digit use: \d

If you want to match a uppercase letter use: [A-Z]

If you want to match a uppercase and lowercase letter use: [a-zA-Z]

If you want to match a space use: \s

To match 1 or more spaces: \s+

To match 0 or more spaces: \s*

To match 0 or 1 spaces: \s?

To match 3 digits use: \d{3}

To match 1 or more letters: [a-zA-Z] +

To match 0 or more letters: [a-zA-Z]*

To match 0 or 1 letter: [a-zA-Z]?

Regular Expression examples using Sequences, Metacharacters and Sets

us zip code

use \d{5} to match 5 digits

note: we must use “ \\d{5} ” in the pattern string since ‘\’ is an escape character that needs to be escaped again to become just a forward slash

```
pattern = Pattern.compile("\\d{5}", Pattern.CASE_INSENSITIVE);
matcher = pattern.matcher("12345");
System.out.println(matcher.find()); // true
```

Canadian postal code

use [a-zA-Z] to represent all upper and lower case letters

```
pattern = Pattern.compile  
("[a-zA-Z]\\d[a-zA-Z]\\d[a-zA-Z]\\d",Pattern.CASE_INSENSITIVE);  
matcher = pattern.matcher("M2J2Y5");  
System.out.println(matcher.find()); // true
```

Phone number

(123) 456-7890

A phone number has 3 digits surrounded by round brackets: (123)

So we use \d to represent digits 0-9 and the use {3} for three digits contained in round brackets. **(\d{3})**

The round brackets are metacharacters that have to be escaped with forward slashes. **(\d{3})**

Continuing we have a space followed by three more digits \d{3} a hyphen – followed by 4 more digits \d{4} The following is a regular expression for a phone number.

(\d{3}) \d{3}-\d{4}

(123)		456	-	7890
(\d{3})		\d{3}	-	\d{4}

We only have 1 slight problem, many people will forget to enter a space after the round brackets so we use the metacharacter ? which means 0 or 1 character.


```

pattern = Pattern.compile
("(\\(\\d{3}\\) ?\\d{3}-\\d{4}",Pattern.CASE_INSENSITIVE);
matcher = pattern.matcher("(123) 456-7890");
System.out.println(matcher.find()); // true
matcher = pattern.matcher("(123)456-7890");
System.out.println(matcher.find()); // true

```

The final regular expression as follows to handle a space or empty space is:

(123)		456	-	7890
\\(\\d{3}\\)	?	\\d{3}-\\	-	\\d{4}

using groups ()

A group allows you to pick out and extract parts of the matching text.

Example: pick parts of the first and last name of a name

```

pattern = Pattern.compile
("(\\w+), (\\w+)",Pattern.CASE_INSENSITIVE);
matcher = pattern.matcher("Tom, Smith");
System.out.println(matcher.find());
for(int i=1;i<=matcher.groupCount();i++)
{
    System.out.printf("Group %d %s\\n",i,matcher.group(i));
}

```

```

Group 1 Tom
Group 2 Smith

```

The groupCount function tells us how many groups we have. The group function prints out the group test resulting from the search.

Using Java String class Regular Expressions

Using Regular Expressions with the Java string class may be a little easier to use, The String class has the following methods.

Method	Description
boolean matches (String pattern)	method takes a regular expression as parameter, and returns true if the regular expression matches the string, and false if not
String[] split (String pattern)	method takes a regular expression as parameter and splits the string at all positions in the string where the regular expression matches a part of the string. returns a String array with these substrings.
String replaceFirst ("from","to")	returns a new String with the first match of the regular expression passed as first parameter with the string value of the second parameter.
String replaceAll("from","to")	returns a new String with all matches of the regular expression passed as first parameter with the string value of the second parameter.

Example using matches

```
String text = "(123) 456-7890";  
System.out.println(text.matches("\\(\\d{3}\\) ?\\d{3}-\\d{4}")); // true
```

Example using split

```
text = "Happy days are here again";  
String[] words = text.split("\\s+");  
for(String w: words )  
{  
    System.out.println(w);  
}  
}
```

```
Happy  
days  
are  
here  
again
```

Example using replaceFirst

Replace all replaces the first occurrence in the string

```
text = "Happy days are here again";  
text = text.replaceFirst(" ", ",");  
System.out.println(text);
```

Happy,days are here again

Example using replaceAll

Replace all replaces all occurrences in the string

```
text = "Happy days are here again";  
text = text.replaceAll(" ", ",");  
System.out.println(text);
```

Happy,days,are,here,again

Regular Expression Homework

You can use String class or Regular Expression classes.

Question 1

Write the regular expression to validate a web page url like:
<http://www.cstutoring.com>

Question 2

Write the regular expression to validate a email and use a group the print out the username and host.

Example:

students@cstutoring.com
username: students
host: cstutoring.com

Question 3

Write a regular expression to split a line where the words are separated by commas that can contain one or many spaces. Print out the words in a loop each on a separate line.

hint: use split method

Example line:

Happy, days, are, here,again\n"

Example output:

Happy
days
are
here
again

Question 4

Write a regular expression to remove all the new lines from a string.

hint: use replaceAll method

Example:

Before:

happy days are\nhere again\n"

after:

happy days are here again"

Question 5

Write python code using a regular expression(s) to locate all words with a certain letter and print them out.

Hint #1: use split then match on each word.

Hint #2: use * metacharacter

Lesson 13 PROJECTS

Project 1 IntArray Class

Make an IntArray class to store a int values in an internal array called items. Make a default constructor that makes an empty array. Make another constructor that takes in the initial size of the array. Make another constructor the receives an ordinary array. You need to copy the elements in the receiving array to your internal array. Make another constructor that receives your IntArray. Again, you need to copy the elements in the receiving IntArray to your internal array. Make methods to access array elements by array index. Make operational methods to add items to the end, insert at a certain index, and remove at a certain index. When adding and inserting items the internal array should just increase in size by 1. When removing items from the internal array just shift the other vales down and set the last value to 0. Make operational method to sort the array ascending and search for values. Use bubble sort to sort the array and use binary search to search for items in the internal array when it is sorted. You can find the code for bubble sort and binary search on the internet. Lastly make a toString method to print out the array elements enclosed in square bracket's like this: [9 4 9 3 6 4 8] Make a TestArray class with a main method to test all the methods of your Array class or alternately for convenience put the main method inside your IntArry class.

Project 2 Int Matrix class

Make a Matrix class that has rows and column variables and an two-dimensional array of the specified rows and columns. Make a default constructor to make an empty Matrix of rows and columns. Have private variable to store rows and columns. Make another constructor the receives an ordinary two-dimensional array. You need to copy the elements in the receiving array. Make another constructor that receives your IntMatrix. Again, you need to copy the elements in the receiving IntMatrix. Make setters and getters to access the matrix elements. make a to string method that will print out the matrices Make operational methods to add, subtract, multiply, divide, transpose and rotate matrices by a specified rotation. Make a TestMatrix class with a main method to test all the methods of your Matrix class.

Project 3 Spelling Corrector

Read in a text file with spelling mistakes, find the incorrect spelled words and offer corrections. The user should be able to choose the correct choice from a menu. Look for missing or extra letters or adjacent letters on the keyboard. Download a word dictionary from the internet as to check for correct spelled words. Use a Hash table to store the words. Store the correct spelled file.

Project 4 MathBee

Make a Math bee for intermixed addition, subtraction, multiplication and division single digit questions. Use random numbers 1 to 9 and use division numbers that will divide even results. Have 10 questions and store results in a file. Keep track of the users score.

Project 5 Quiz App

Make a quiz app with intermixed multiple choice, true and false questions. You should have a abstract Question super class and two derived classes MultiipleChoice and TrueAndFalse. Each derived class must use the abstract methods to do the correct operation. Store all questions in one file. Store the results in another file indicating the quiz results.

Project 6 Phone Book App

Make a phone book app that uses a HashMap or ArrayList to store Phone numbers, emails and names. You need an Contact class to store name, emails and phone number. You should be able to view, add, delete, and search for contacts as menu operations. Contacts need to be displayed in alphabetically orders. Offer to lookup by name, email or by phone number. Contacts should be stored in a file, read when app runs, and saved with app finished running.

```
import java.util.Map.Entry;  
import java.util.Comparator;  
import java.util.Collections;
```

```

// print HashMap sorted by value

// Entry set contains both key and values,
// so we can sort key and value together
Set<Map.Entry<String, String>> set = map1.entrySet();
List<Map.Entry<String, String>> list
    = new ArrayList<Map.Entry<String, String>>(set);

// anonymous comparator class
Comparator c = new Comparator<Map.Entry<String, String>>() {
    public int compare(Map.Entry<String, String> e1,
        Map.Entry<String, String> e2) {
        return e2.getValue().compareTo(e1.getValue());
    }
};

// sort Entry key value pairs by value using anonymous Comparator
Collections.sort(list,c);

```

Lastly, we print out the sorted list.

```

for (Map.Entry<String, String> entry : list) {
    System.out.println(entry.getKey() + " = " + entry.getValue());
}
System.out.println("");

```

Project 7 Appointment App

Make an Appointment book app that uses a HashMap to store Appointments. You need an Appointment class to store name, description and time. You should be able to view, add and delete appointments as menu operations. Appointments need to be displayed in chronological orders. Appointments should be stored in a file, read when app runs, and saved with app finished running. Use the following code to sort a Hashmap by value.

```

import java.util.Map.Entry;

```

```

import java.util.Comparator;
import java.util.Collections;

// print HashMap sorted by value

// Entry set contains both key and values,
// so we can sort key and value together
Set<Map.Entry<String, String>> set = map1.entrySet();
List<Map.Entry<String, String>> list
    = new ArrayList<Map.Entry<String, String>>(set);

// anonymous comparator class
Comparator c = new Comparator<Map.Entry<String, String>>() {
    public int compare(Map.Entry<String, String> e1,
        Map.Entry<String, String> e2) {
        return e2.getValue().compareTo(e1.getValue());
    }
};

// sort Entry key value pairs by value using anonymous Comparator
Collections.sort(list,c);

```

Lastly, we print out the sorted list.

```

for (Map.Entry<String, String> entry : list) {
    System.out.println(entry.getKey() + " = " + entry.getValue());
}

System.out.println("");

```

Project 8 Fraction class

Make a **Fraction** class that stores an int numerator and denominator. Make two constructors, a default constructor that initializes the numerator to 0 and initializes the denominator to 1, and an initializing constructor that initializes the numerator and denominator with user values.

Make methods **add**, **subtract**, **multiply** and **divide**, that receives two Fraction classes and return a Fraction class result. Your empty add method may look like this:

```
public Fraction add(Fraction f2)
{
    Fraction f3 new Fraction();
    return f3.
}
```

Make a **toString** method that returns a String representation of a fraction class as a whole number like 5, when the denominator is 1 and a fraction like 2/3 when the denominator is not 1. Make a main or standalone Test class to test your Fraction class.

Bonus marks:

Make a **reduce** method using GCD that will reduce a fraction to lowest terms. You can make a iterative or recursive GCD private method as follows:

iterative GCD

```
static long gcd (long a, long b) {
    long r, i;
    while(b!=0){
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

recursive GCD

```
public int gcd(int a, int b) {
    if (b==0) return a;
    return gcd(b,a%b);
}
```

Also make a **equals** method and a **compareTo** methods to test if two fractions are equal and to compare fractions if they are less or greater to each other. Test these features to your testing.

Project 9 GenericArray Class

Make the IntArray class to be a Generic Matrix class TArray so that it can store any data type. You need your generic data type T to extend the Comparable interface like this

```
T extends Comparable<T>
```

or else you cannot compare values in your TArray class for sorting.

You need to instantiate generic array with a Comparable object to hold your array T items, since the java compiler cannot make a T object for you. You then need to type cast your created Comparable array to a T type array.

```
items = (T[])new Comparable[size];
```

Make a TestTArray class with a main method to test all the methods of your TArray class.

Project 10 Generic Matrix class

Make the IntMatrix class to be a Generic Matrix class TMatrix so that it can store any data type. You need your generic data type T to extend the Number interface like this

```
T extends Number<T>
```

or else you cannot do arithmetic operations in your TMatrix class. You will also need the Generic TCalculator class from Lesson6 as a private variable so it can do arithmetic operations for you.

```
private TCalculator<T> calc = new TCalculator<T>();
```

You need to instantiate generic two dimensional array with a Number object since the Java compiler cannot make a T object for you. You then need to type cast your created Number array to a T type array.

```
items = (T[][] )new Number[rows][cols];
```

Make a TestTMatrix class with a main method to test all the methods of your TMatrix class.

Project 11 Generic Arithmetic Classes

Make Generic Add, Sub, Mult and Divide Generic classes so each has a overloaded method to do arithmetic operations on any data type. Have an generic abstract super class called Operations to represent all the Operation classes. Incorporate the classes in you Generic Calculator. Rename you Generic calculator to TCalculator2.

Project 12 Grocery Store App

Make a Grocery Store App where Customers can purchase items. Preferred customers get a discount. After all items have been entered a receipt is printed.

Step 1: Item class

Make a Item class with private variables product name, quantity ordered, price and discountPrice.

If the item is not a discount item then the discount price is 0.

Make a constructor that will receive the item name, quantity, price and discount price.

Make getters and setters for each instance variable

Make a formatted toString method that will return item name price quantity discount price surrounded by round brackets and extension price like this:

Carrots 2 1.29 (.89) 2.58 (1.78)

Step 2: GroceryStore class

Make a GroceryStore class that will store items bought, total items bought, that total's the order and print out a receipt.

The Grocery Store class will store the customer's name, and all items bought in an ArrayList <Item> called items.

The Grocery store constructor will receive the customer name and create the ArrayList <Item> of items.

The grocery store will have a method to add an item object called add.

The grocery store class will also print out a receipt using a method called printReceipt.

The Grocery store class will have a getTotal method to return the total of all items. The getTotal method can also be used to print out the receipt total.

All instance variables are private and you cannot have any getters and setters.

Step 3: DiscountStore class

The DiscountStore class inherits from the GroceryStore class.

The DiscountStore receives the CustomerName and sends the CustomerName to the GroceryStore super class.

If the customer is a preferred customer then the DiscountStore class is used. The DiscountStore class will calculate discount percent, and count of discount items and total of all items using the discount price rather than the regular price. The discount class will override the getTotal class of the grocery store class.

The discount store class will also print a receipt showing the number of discount, items and the discount percent obtained.

Step 4 GroceryApp class.

The GroceryApp class is the main class where the cashier enters the customer items, bought.

The cashier will ask if the customer is a preferred customer. if it is a preferred customer then the DiscountStore class is used else the Grocery Store class is used.

The cashier will enter the items bought. Once all items have been entered then the receipt is printed out.

You will need to store a list of products in a file to simulate the entering of products or make an array of items like this:

```
Item[] items = {new Item("apples",2,1.26,1.08)
new Item("oranges",2,1.26,1.08),
new Item("carrots",2,1.26,1.08)
new Item("apple",2,1.26,1.08)};
```

The file format will be like this

Customer name

Preferred or not preferred

Number of products

Item name, quantity, price, discount price

Example file:

Tom Smith

Preferred

3

Carrots , 2.49, 1.78, 2

Fish,12.67, 11.89,3

Milk 4.89.3.75, 2

In either case the items will be added to the store.

The main method will have a menu as follows:

- (1) add items to Grocery store
- (2) add items to Discount store
- (3) print receipt
- (4) exit program

END