2. Inheritance

By Mdm Wong Kwee Yin

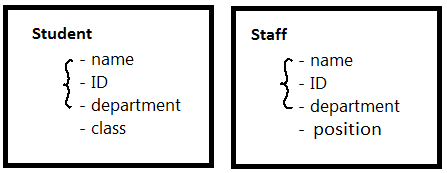
2.1 A Real Life Scenario

We will first consider a real world example which would be able to take advantage of the inheritance structure in Java programming.

**Example**

Singapore Polytechnic Library serves 2 types of members, ***students*** and***staff.***

Assuming that in our program, we are interested in capturing the following information:

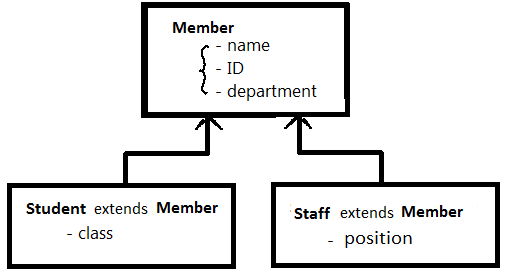


To model ***students*** and ***staff*** in the program, the most straight forward way is to create a class for ***Student*** and another class for ***Staff***.

However, one can easily see that there would be *many common characteristics* between these 2 classes.

This is the more efficient way:

* To define all these *common characteristics* in a separate class, called ***Member***.
* Next, to define class ***Staff*** and ***Student***as ***subclass*** of ***Member*** which makes class ***Staff*** and ***Student*** automatically ***inherit*** all the characteristics specified in ***Member*** without further specification of these characteristics.
* On top of that, class ***Staff*** and ***Student***can have their own individual unique characteristics defined within themselves.

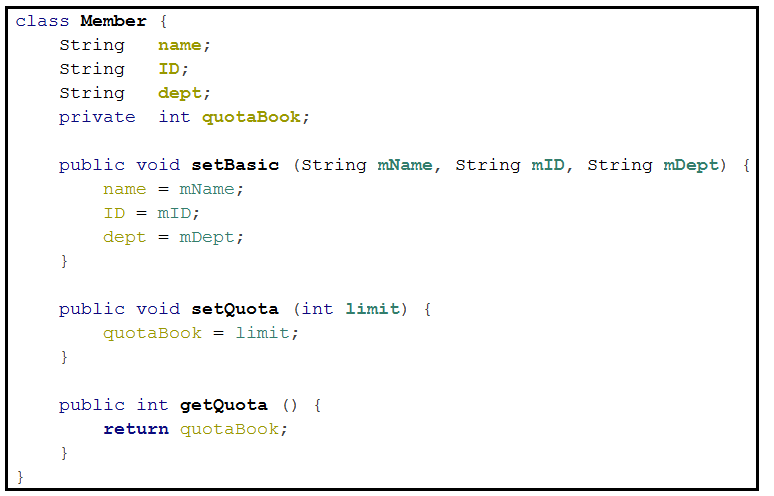


2.2 Java Implementation of the Example of Inheritance

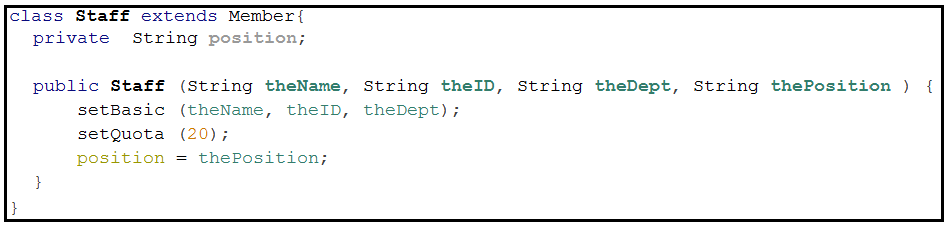
Class ***Member*** is created just as any classes we have learnt in topic 1.

For illustration purposes, we define the class ***Member*** as below:

(Assuming that the program also tracks the ***quota*** for members.)



Next, we define class ***Staff*** as below:



The important keyword is ***extends***.

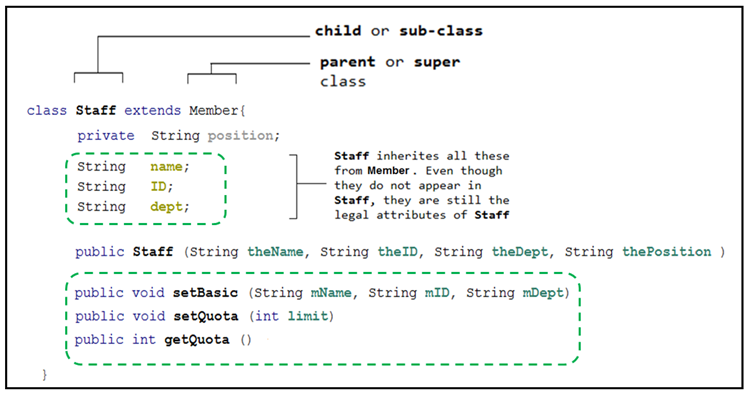
The important keyword is ***extends***.

The important keyword is ***extends***.

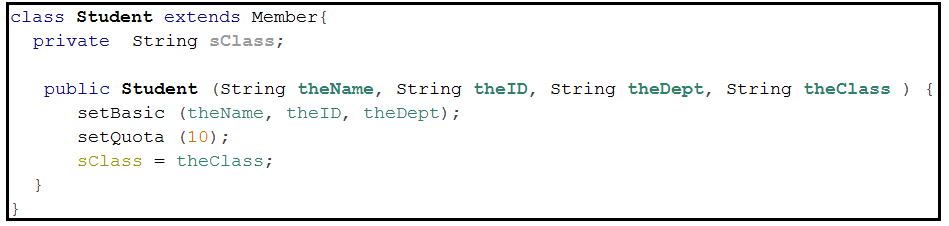
The constructor has access to the 2 methods, ***setBasic()*** and ***setQuota()***, that it inherits from class ***Member***. Quota for staff is set to 20.

The variable that is *unique* to staff, ***position***, is defined within class ***Staff***.

Effectively, this is what class ***Staff***:



Likewise, we define class ***Student*** as below:



The important keyword is ***extends***.

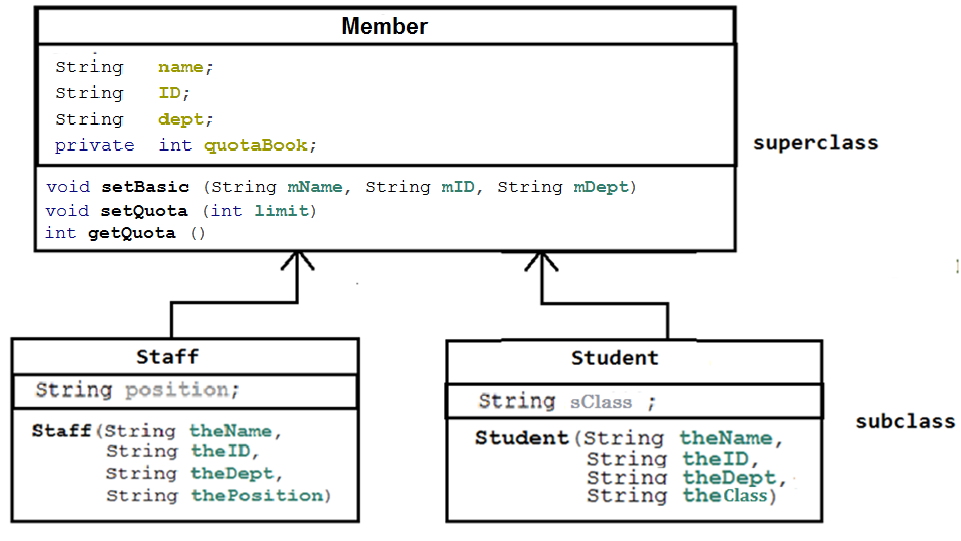
The important keyword is ***extends***.

The important keyword is ***extends***.

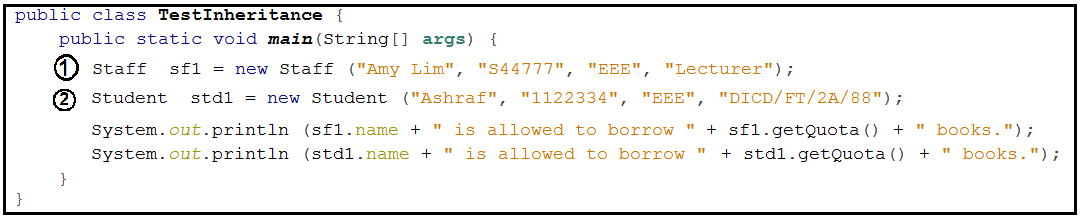
The constructor also has access to the 2 methods, ***setBasic()*** and ***setQuota()***, that it inherits from class ***Member***. Quota for staff is set to 10

The variable that is *unique* to student, ***sClass*** (for storing information of the student’s class) is defined within class ***Student***.

At this point, the 3 classes are:



All the 3 classes can be tested using the ***main()*** below:

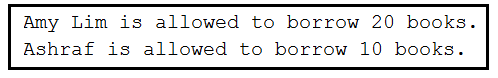




constructs a new ***Staff*** object. The name of the object is ***sf1.***

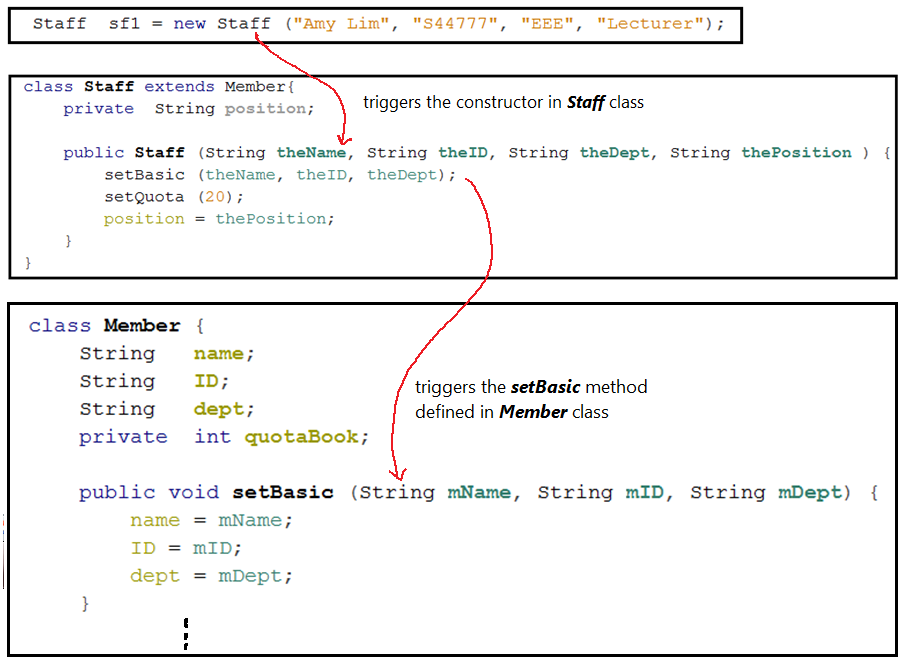
constructs a new ***Student***. The name of the object is ***std1***.

Running the above code will produce this output:



It is important to fully understand the ***flow of the program*** here.

Always start tracing from the ***main(***), the entry point of the program.



Essentially, the ***setBasic()*** initialises:

***sf1.***name = “Amy Lim”

***sf1.***ID = “S44777”

***sf1.***dept = “EEE”

Also, the ***setQuota(20)*** initialises the following inherited attribute for ***sf1*** object:

quotaBook = 20

However, ***sf1.***quotBook ***is illegal.***

Why?

Likewise, the following statement can be traced in the similar way:

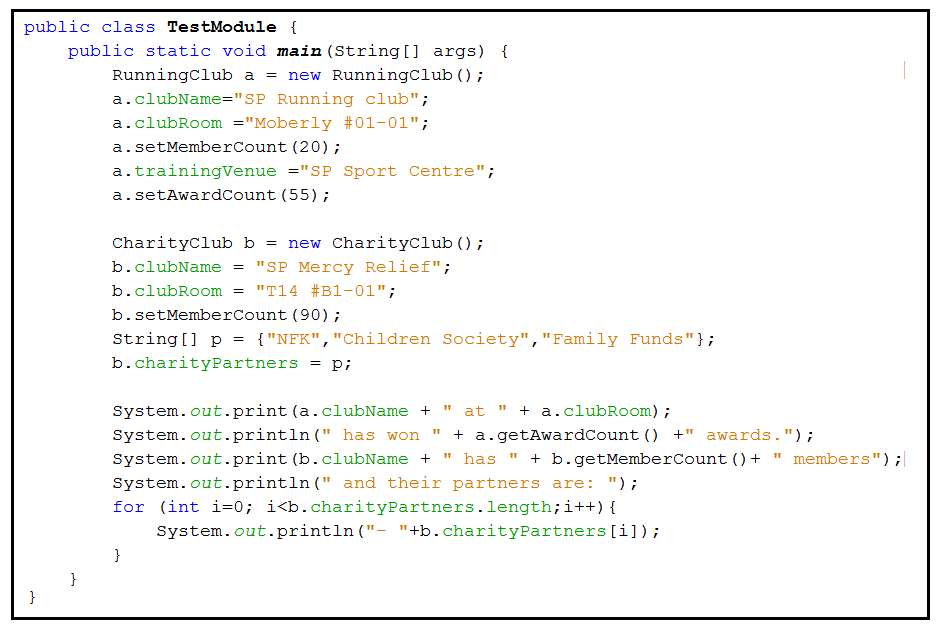
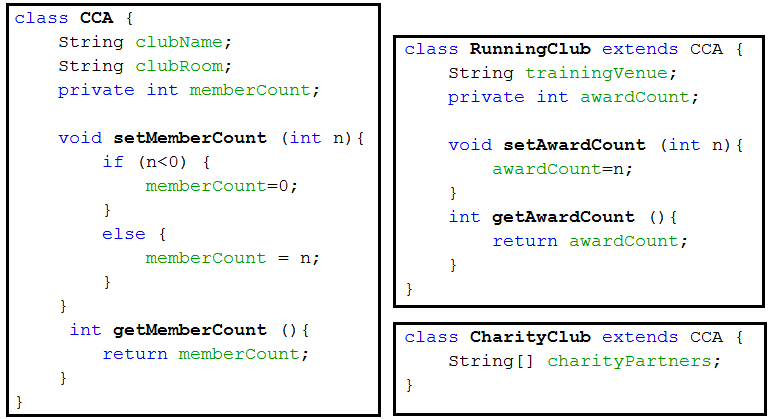


2.3 Thinking Time

**Activity 1**

Aim: To familiar with creating superclass and subclass by code tracing.

Trace the program and determine the output of the program.

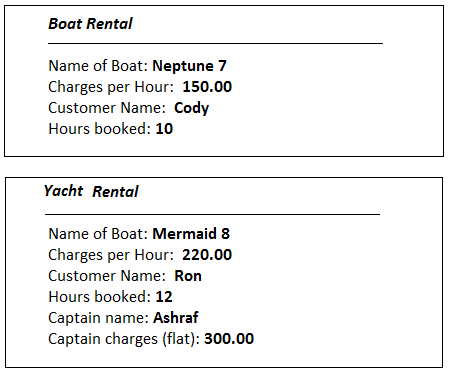


**Activity 2**

Aim: To familiar with creating superclass and subclass (in preparation for **mini-project**).

A boat rental company provides 2 type of rental.

The data to be captured by the program and sample data are shown below:



**[ PLEASE DO NOT USE CONSTRUCTORS IN CLASSES FOR THIS ACTIVITY]**

(a) Identify the suitable data to be defined in a superclass.

(b) Create a new Java Project.

Create the superclass with variables representing respective data in (a).

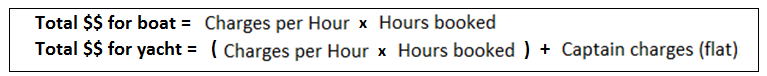
(**Please do not use any constructors for this Activity**)

(c) Create the subclass for ***Yacht***. Define the variables unique to the subclass.

(d) In the ***main()***, create one object for a boat and another object for a yacht, using the appropriate class defined.

(e) In the ***main()***, initialise all the variables for the objects created in (d) with the sample values given above.

(f) In both the superclass and the subclass, create a ***method*** each to *compute and return* the total costs. Calculation of total cost:



(g) In the ***main()***, call (or invoke) the method defined in (f) via the objects in (d) and display the return computed total cost for the objects.

2.4 Another Advantage of Object Oriented Programming

In previous topic, we listed the advantages of Object Oriented Programming as:

* Class and object models the ***real-life*** object closely with necessary attributes and behaviours (methods).
* Attributes and methods of objects are ***encapsulated*** (hidden)within the class (easier to associate all data pertaining to one particular object)
* ***Easier i***n maintenance (as in adding new attributes or methods)

With inheritance, what it means is that a ***superclass*** can be used in creating multiple subclasses by the keyword ***extends***. Characteristics defined in a ***superclass*** are ***reusable*** right away in its ***subclasses*** without further specifications.

This greatly reduces the development time in big projects where codes in existing classes can be reused to expand a program easily, in a very structured manner.

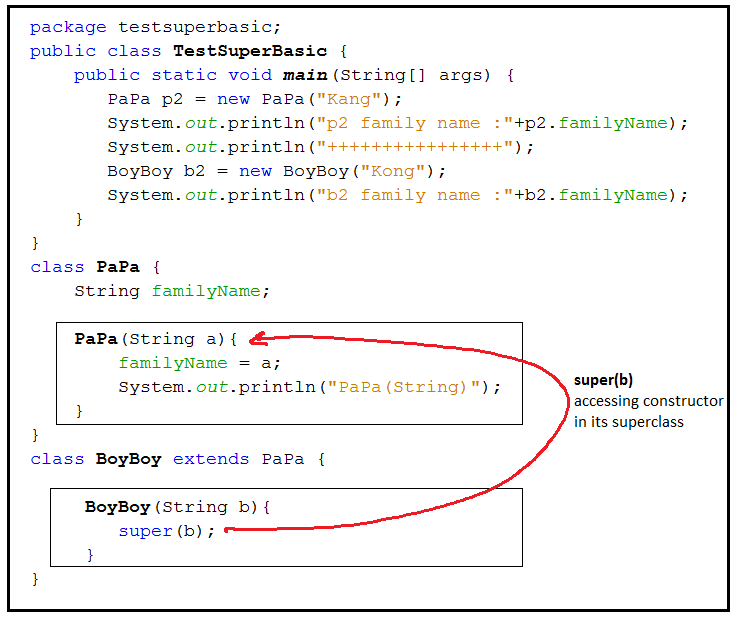
Inheritance gives rise to this important advantage of ***Object Oriented Programming (OOP)***

* ***OOP*** code is much more ***reusable***

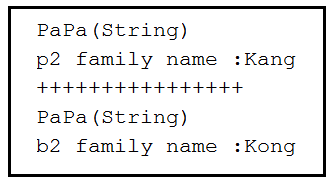
2.5 Self Revision

2.6 Keyword super

The keyword ***super*** allows the subclass to access superclass’ methods, including constructors.



Output:



**Matching Super Call**

A ***super*** call from the *subclass* will activate the ***matching superclass*** methods/constructors.

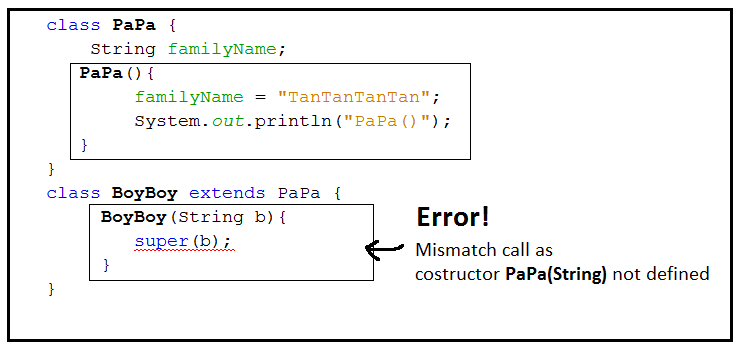
|  |  |
| --- | --- |
| **call from subclass** | **Matching constructor in superclass** |
| ***super()*** | e.g. ***PaPa()*** |
| ***super(String)*** | e.g. ***PaPa(String)*** |
| ***super(int, String)*** | e.g. ***PaPa(int, String)*** |

**e.g.**

The match is determined by the number and data types of the arguments of the methods.

**Missing Matching Methods/Constructors in Superclass**

The will result in ***error*** in the program.



**Auto *super()* Call**

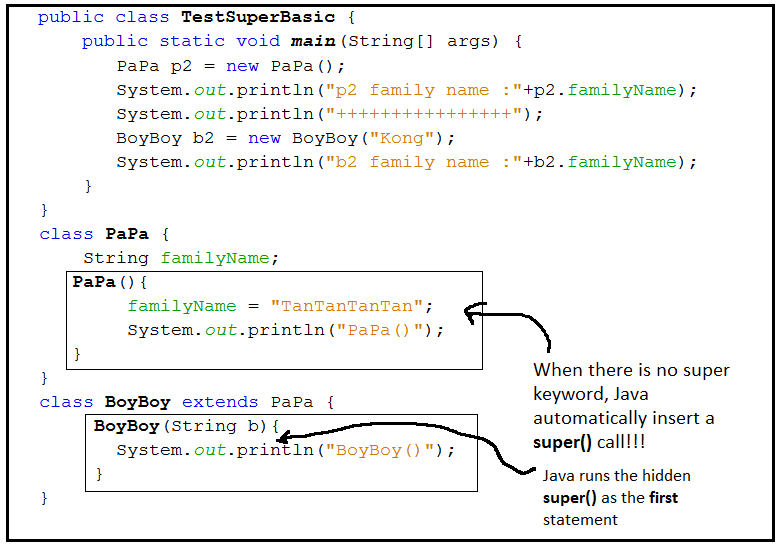
This is special.

This is special.

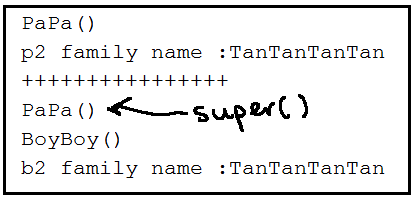
This is special.

When there is ***no super keyword*** defined in a ***subclass*** constructor, Java

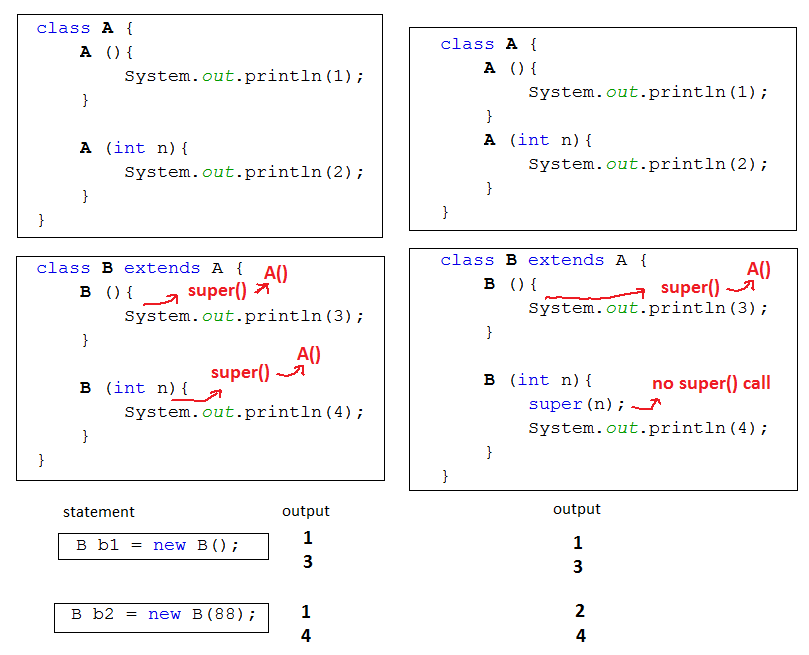
* ***automatically*** assume a ***super()*** and activate the matching constructor in the ***superclass***
* and runs the ‘*hidden*’ ***super()*** as the ***1st statement*** (see code and output below)

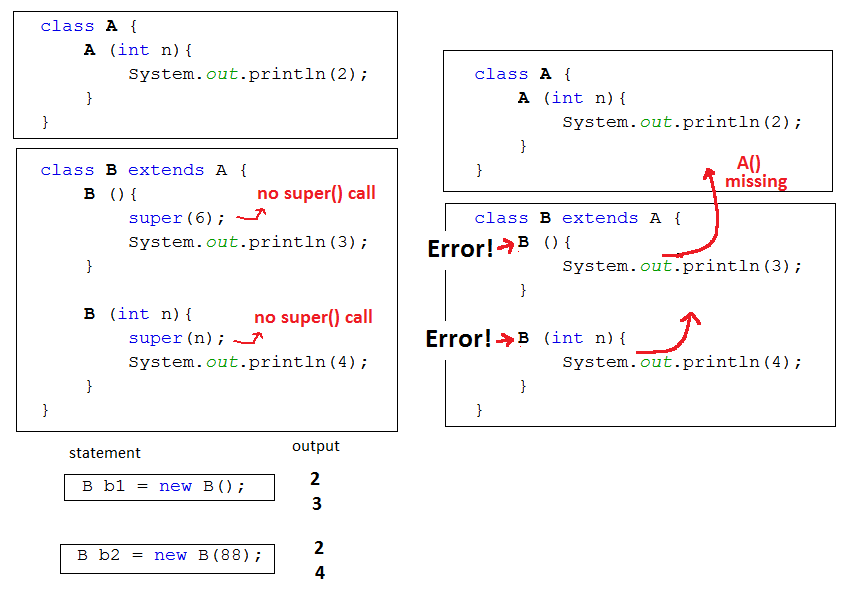


Output:



**Summary of Chain of super call**

****

****

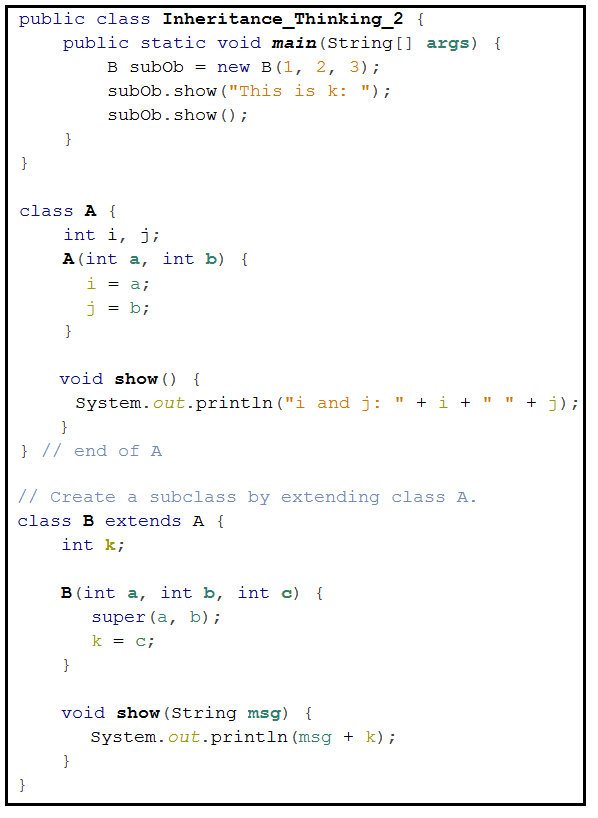
**Another example**:

2.7 Thinking Time

**Activity 3**

Aim: To familiar with chain of super call of superclass in subclass.

Determine the output of the program tracing the code below.

******

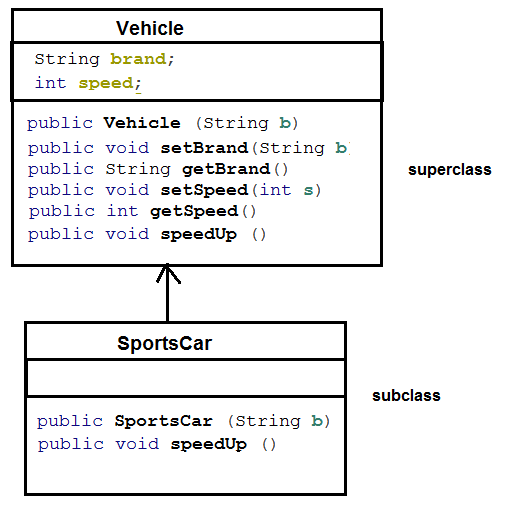
2.8 Method Overriding

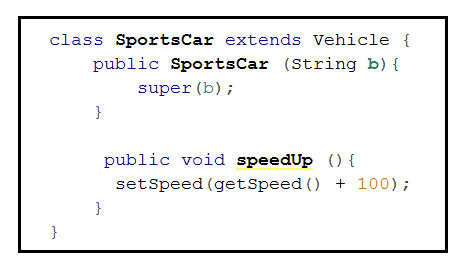
When a ***method*** with the *same name* and *type signature* appearing both in the superclass and the subclass, the method in the ***subclass*** is said to be ***overriding*** the method in the ***superclass***.

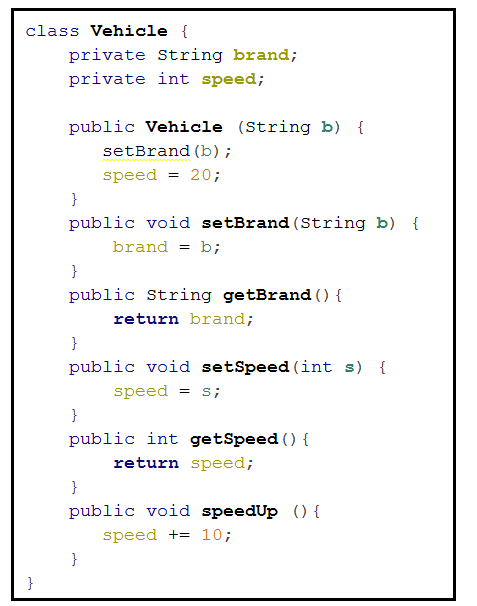
**Example**

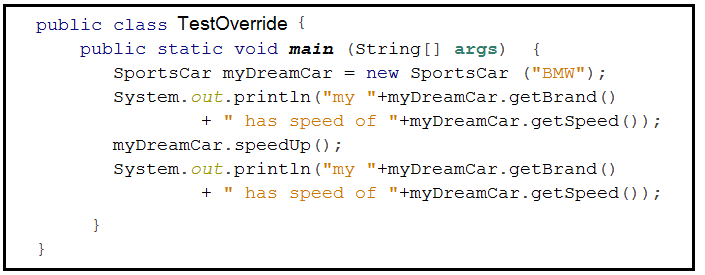
In this example, the superclass is ***Vehicle*** and the subclass is ***SportsCar***.

The overriding method (the one in the subclass) is called ***SpeedUp()***.

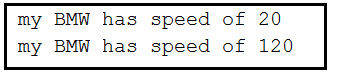








Output:

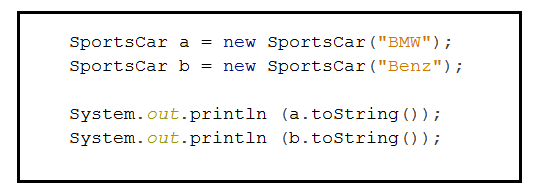


**Another example**: [*http://www.java2s.com/Book/Java/Class-Definition/What\_is\_Method\_Overriding.htm*](http://www.java2s.com/Book/Java/Class-Definition/What_is_Method_Overriding.htm)

2.9 Default toString() in all Java Classes

Every class defined in Java has a method called ***toString().***   
Invoking this method returns the system generated ID for each object, which is unique.

Example:

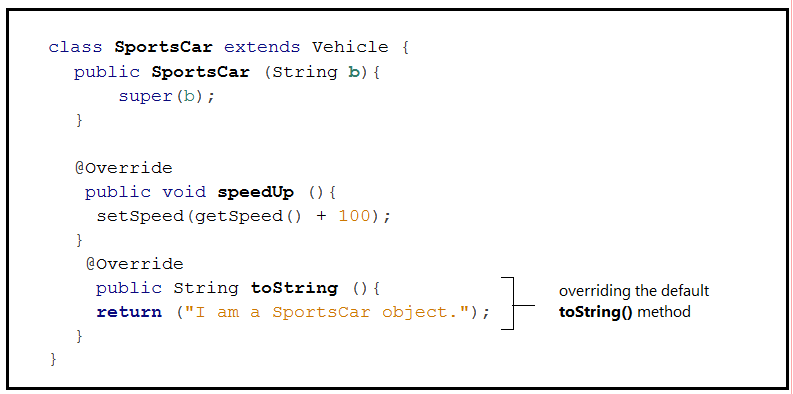


Output is:

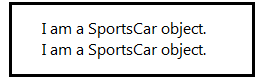


[*http://www.java2s.com/Tutorial/Java/0100\_\_Class-Definition/Inheritanceconstructorsandarguments.htm*](http://www.java2s.com/Tutorial/Java/0100__Class-Definition/Inheritanceconstructorsandarguments.htm)

If we define an overriding toString () method in SportsCar Class,

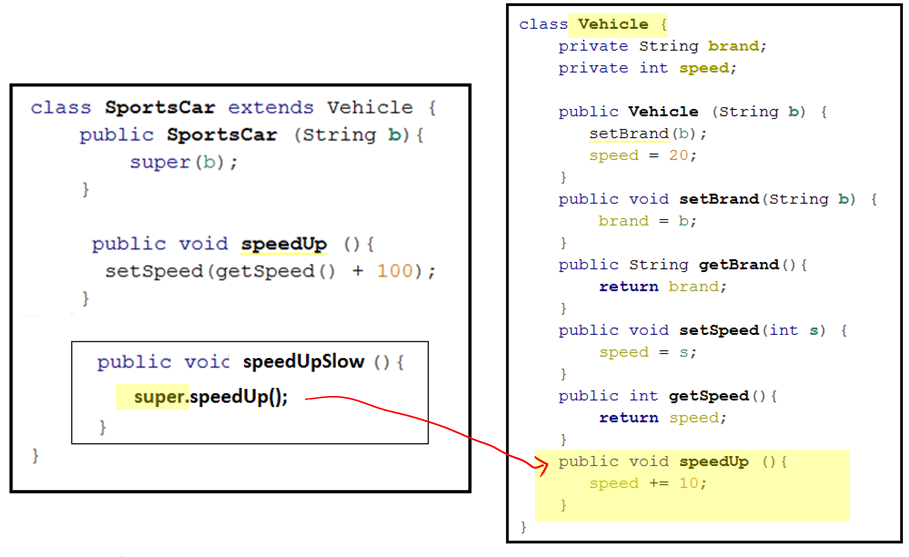


Output becomes:



Using the ***super keyword***, the ***subclass*** can activate the ***methods*** in the ***superclass.***

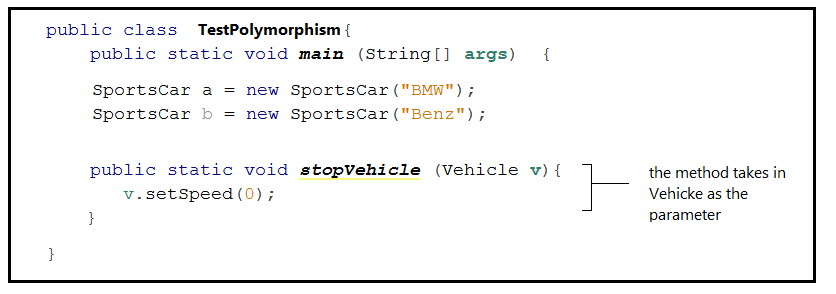
**Example:**



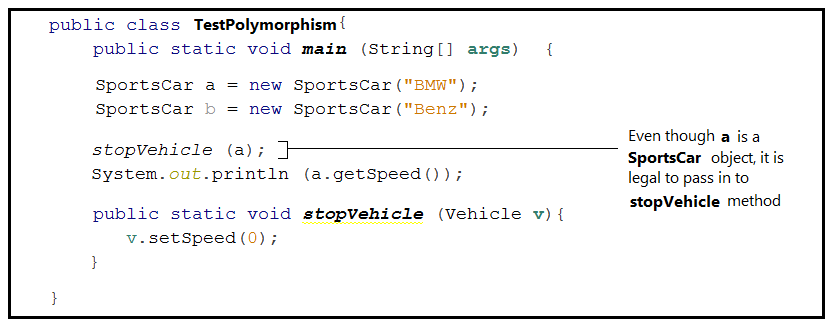
2.10 Polymorphism

An object of a subclass can be used by any codes designed to use with an object of its superclass.

For instance, the following ***stopVehicle*** method takes in a ***Vehicle*** parameter.





Since ***SportsCar*** is a subclass of ***Vehicle***, ***stopVehicle*** method will also accept a ***SportsCar*** parameter :



Output is:





This feature is known as the ***polymorphism*** (a Greek word meaning ‘many forms’).

**Another example**: [*http://java2s.com/Tutorials/Java/Java\_Class\_Creation/Polymorphism\_in\_Java\_language.htm#Polymorphism*](http://java2s.com/Tutorials/Java/Java_Class_Creation/Polymorphism_in_Java_language.htm)

2.11 Access Modifiers

There many are many access modifiers in Java.

Let us look at these four:

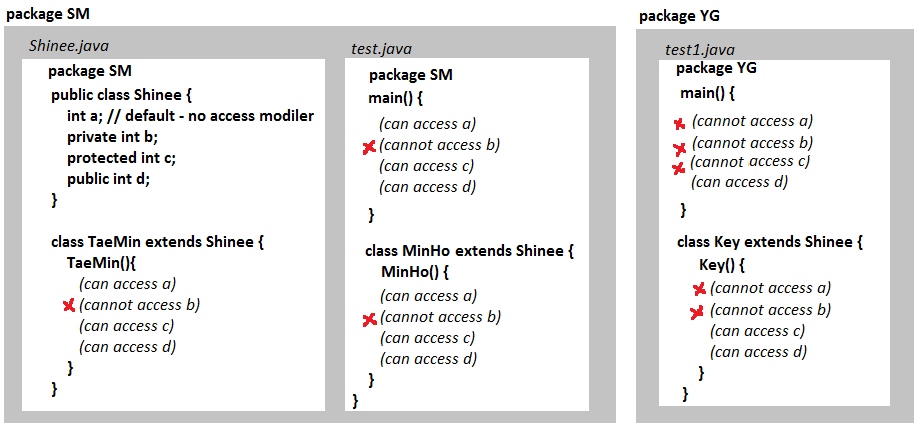
* ***default***
* ***public***
* ***private***
* ***protected***(subtle difference from ***private***)

and their implications on:

* ***variable***
* ***method/constructor***
* ***class***

**Modifiers on Variables**

Modifier ***default, private, protected and public*** on ***variables*** in programs within and outside the package as well as ***subclasses :***

******

*Note:* The difference between ***default*** and ***protected*** is very subtle.  
It lies in the accessibility in programs in another package.

In this file external to the package, ***protected*** variable is accessible in subclass but not program.

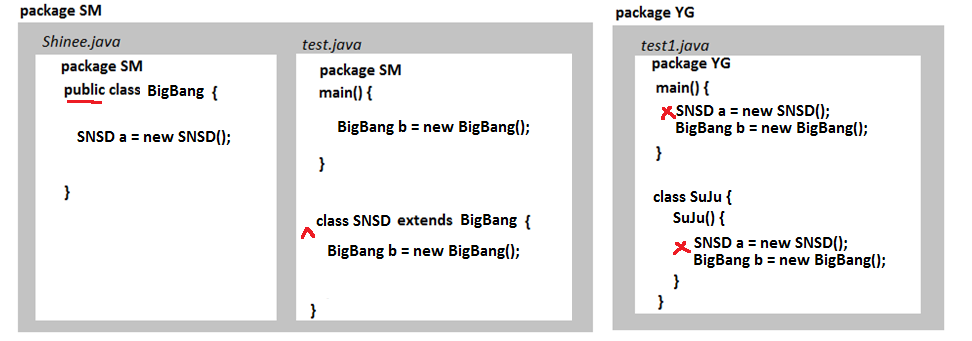
**Modifiers on Method/Constructor**

Similarly, the same accessibility applies to access modifiers on ***methods/constructors***.



**Modifiers on Class**

* ***default*** modifier on class – available only within the same package
* ***public*** modifier on class – available within or in different package
* ***private, protected*** – not applicable



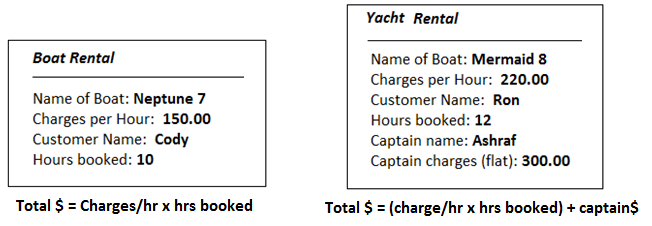
2.12 Thinking Time

**Activity 4**

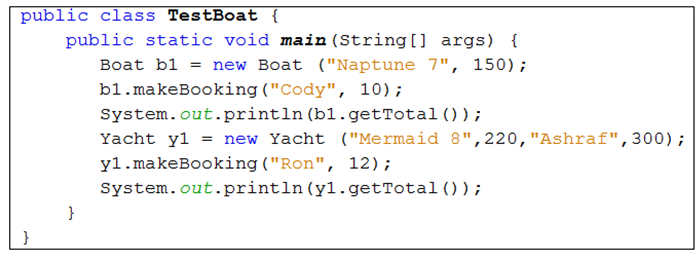
Aim: To familiar with overriding methods in subclass.

A boat rental company provides 2 type of rental.

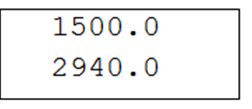
The data to be captured by the program, sample data and cost calculations are shown below:



The ***main()***:



The definitions of class ***Boat*** and ***Yacht*** are on next page.

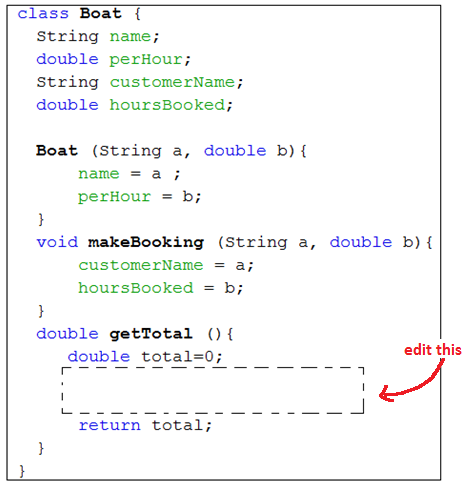
The output (you can verify the calculations of total cost base on the sample data and formulas):

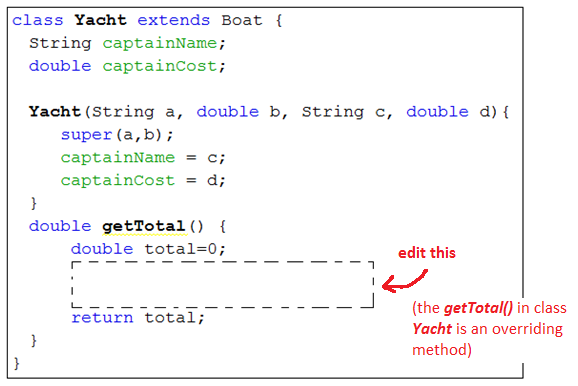
Use the partial code on next page, edit the following methods to calculate the total cost (see output) of the bookings base on the information given above:

* method ***getTotal()*** in class ***Boat***
* ***overriding*** method ***getTotal()*** in class ***Yacht***

Class ***Boat*** is the ***superclass***.

Class ***Yacht*** is the ***subclass*** of ***Boat.***

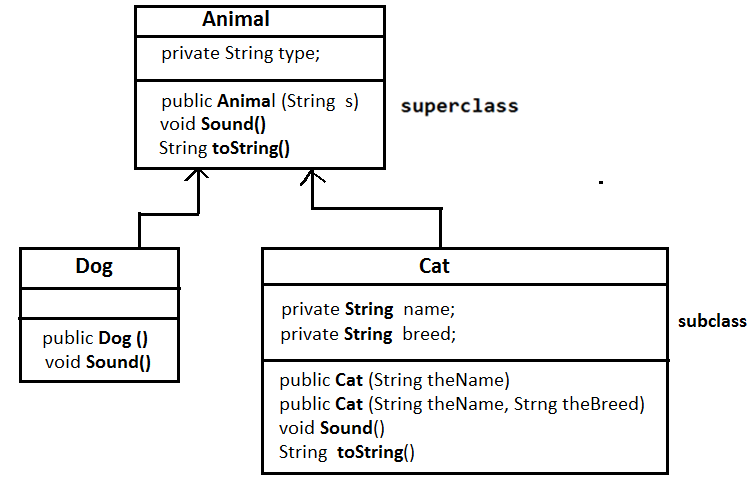




**Activity 5**

Aim: To familiar with overriding methods in subclass.

Crete a new Java project and code the 3 classes with the information given below:



class **Animal**

* + variable **type** : store the type of animal (e.g. “Dog”, “Cat” etc)
  + constructor **Animal(String s)** : sets **type** to the input parameter **s**
  + method **sound()** : prints message “**Sound of an animal**“
  + overriding method **String toString()** : prints the message “**This is a x**“ where **x** is the value of the variable**type** (e.g. “Dog”, “Cat” etc

class **Dog**

* + a subclass of **Animal**
  + no variables
  + constructor **Dog()**: sets **type** to **“Dog”** using the proper **super** statement
  + an overriding method **sound()**: prints message “**Woof Woof!**“

class **Cat**

* + a subclass of **Animal**
  + a private variable **name**: store name of the Cat
  + a private variable **breed**: store breed of the Cat
  + constructor **Cat(String theName)** : sets **type** to **“Cat”** using the proper **super** statement and **name** to the input parameter **theName**
  + constructor **Cat(String theName, String theBreed)** w sets **type** to **“Cat”** using the proper **super** statement, **name** to **theName** and **breed** to **theBreed** .
  + an overriding method **sound()**: prints message “**Miaoow!”**
  + an overriding method **String toString()**: prints this message “**This is a x, name y of breed z**“ where **x** is the **type**, **y** is the **name** and **z** is the **breed** of the **Cat**

Create a **main()** program that

* + creates a **Dog** object, say **d**
  + create a **Cat** object **c1** to model a cat called “**Kelly**”
  + create another **Cat** object **c2** to model a cat called “**Momo**” and of breed “**Persian**”
  + Create a variable ***Animal ref***
  + We want to demonstrate *Polymorphism* by showing that even though **ref** is created as **Animal** but it can be used to reference its subclasses (i.e. **Dog** or **Cat)**.
  + Now reference the **Dog** object **d** using **ref**

**ref = d;**

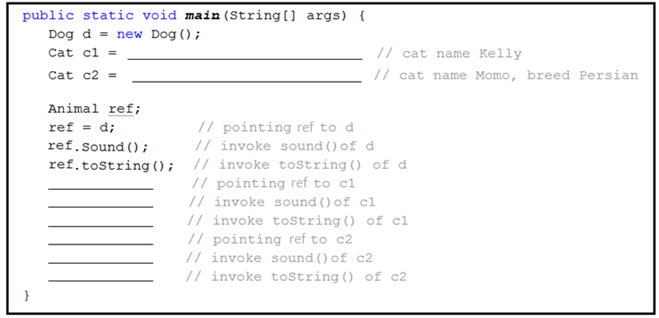
* Next, invoke the **sound()** and **toString()** of **ref**

**ref.toString();**

**ref.sound();**

* Next reference the **Cat** objects in the similar way and repeat to invoke **sound()** and **toString()** using **ref**

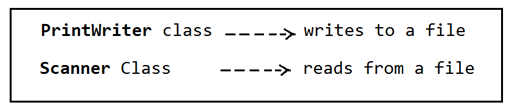
Partial code of the **main()** :



2.13 Additional Material Useful for Project Development

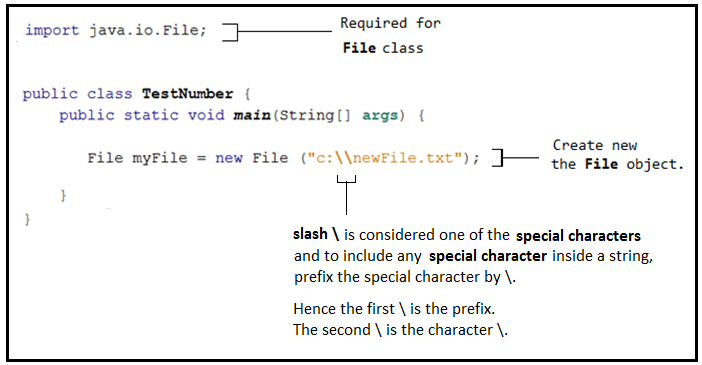
**Introduction – read/write Text File**

* ***Text files*** are normal files stored in our hard disks which can be opened and interpreted by text editors such as *NotePad.*
* Text files are not part of Java programs. They are external to Java Programs.
* Java program treats reading data from text files as ***Input*** operation.
* Java program treats writing data to text file as an ***Output*** operation.
* In order to perform ***input/output (I/O)***, we need to employ appropriate ***JDK I/O classes***. The 2 main JDK classes for writing/reading data to/from text files are:



**File Object – which file to work with**

* Before writing to a text file, first we need to ‘***prepare*’** the ***File object*** in the Java program so that the Java program would ‘*know*’ *which text file to work with*.
* A ***File*** object encapsulates the properties of a file or a path (directory).



**Another Example for having \ in String**

If the path of a file in the hard disk is (take note of all the “\”)

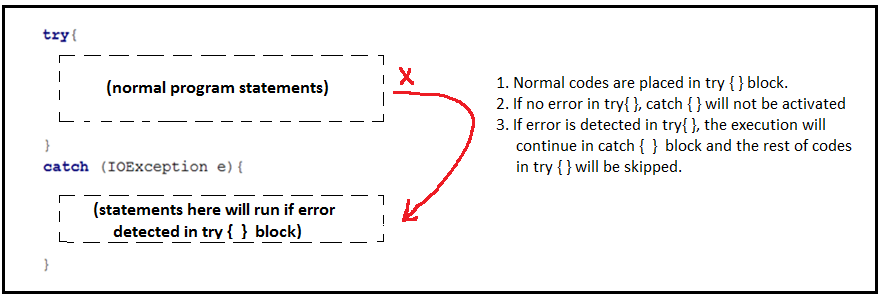
**D:\Users\s34337\Pictures\birthday.jpg**

then, to put this path in a ***String*** in Java, it would be



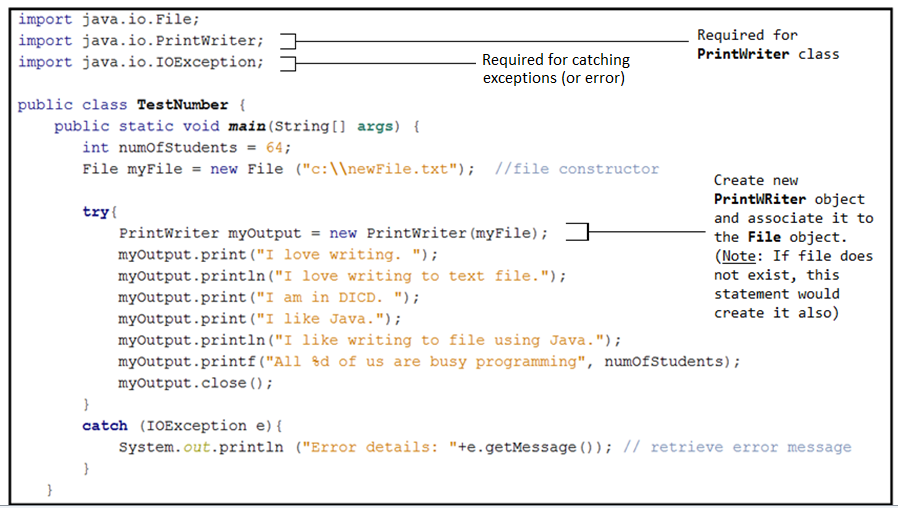
**try-catch Blocks and Exception**

* ***File I/O*** operation falls under the group of operations where Java insists that the programmer has to ‘***catch***’ the error (or ***exception***), if there is any.
* The type of error or ***exception*** to watch out for in ***File I/O*** operations is the ***IOException***, whichmainly covers errors in opening file, accessing file, file-not found etc.
* To get Java program to watch out for the ***IOException***, we need to put the statement(s) that would possibly generate the ***exception*** in a ***try { }*** block.
* A ***try { }*** block must be accompanied by a ***catch { }*** block. (See below)
* In Java term, when an ***error occurs*** – it is said that an ***exception has been thrown***.



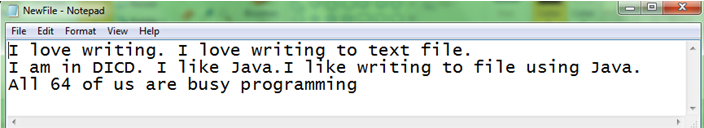
**Writing Data to Text File using PrintWriter**

We use the ***PrintWriter*** class to write to *text files* (see code below)



Check the text file “***c:\\newFile.txt***” using the ***NotePad***.

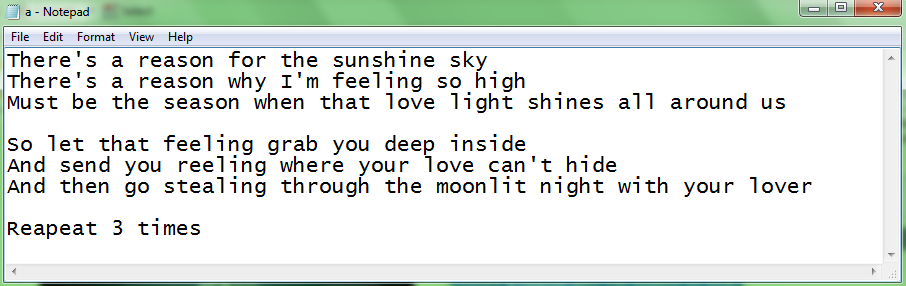
It shall look something like this:



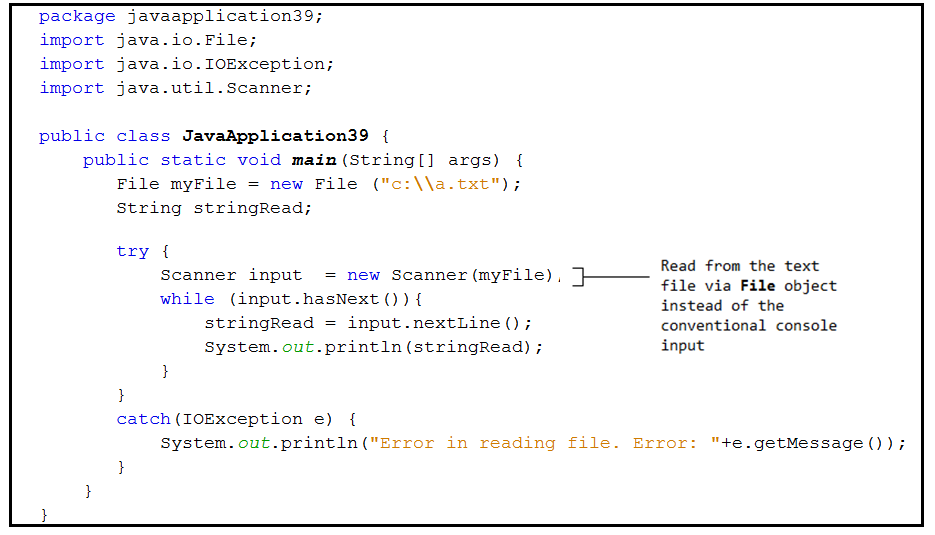
**Reading Data from Text File using Scanner**

Now, we will try the reverse – reading data from a text file.

Assuming that we have a text file **a.txt** with the contentof:

****

The following example reads the above text file *line by line* and *prints* out the text.



Note that there is a ***while loop*** in the code above.

Under what condition will the program *enter* the loop?

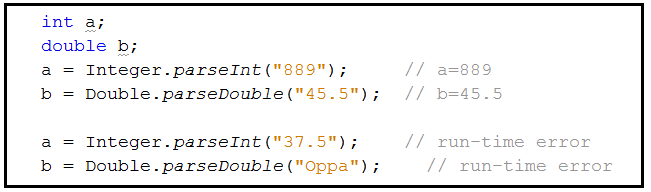
Under what condition will the program *exit* the loop?

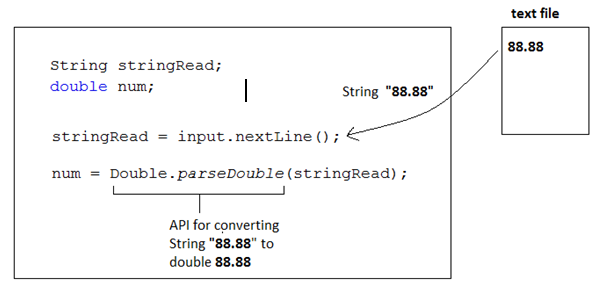
**Convert from String to Number**

***Integer.parseInt( )*** - It converts a ***String*** to an **int** provided that the ***String*** inside ( ) is possible to be converted to integer.

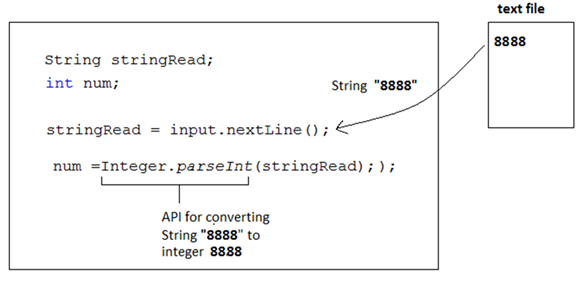
***Double.parseDouble( )*** - It converts a ***String*** to a **double** provided that the ***String*** inside ( ) is possible to be converted to double.

Examples:



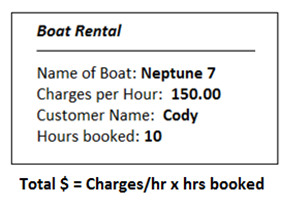
****Data read in from text file using  ***Scanner.nextLine();*** is always of ***String*** type.

We can convert the data read into numbers:

****

***+++++++++++++++++++ end of Java 1 revision ++++++++++++++++++++++***

1. Develop a Java program to model the Boat rental example.

****** Create a class ***Boat*** for to capture the following information:

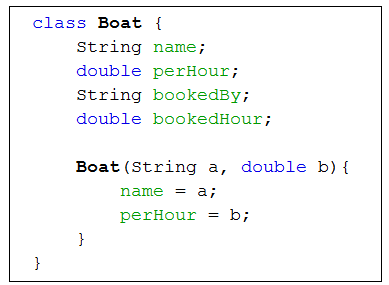
Create a constructor for the Boat with this signature.



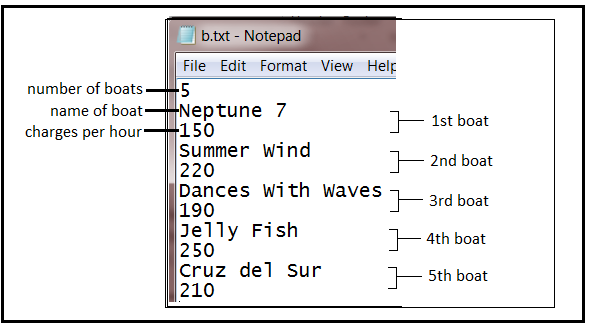
The constructor set the name of the boat to value of ***a*** and charges per hour to value of ***b***.

Use meaningful names for your variables.

Possible solutions for (1) :



2. Assuming that the company has only 5 boats and all the boat information , i.e. boat name and charges per hour, is stored in a text file **b.txt** (see below)



Extend from the code in (1).

(use ***Scanner***) Read in the data from the text file.

Make use of the first data (i.e. 5 ) to create an array of ***Boat***.

Create a new ***Boat*** object as the data of 1st boat is being read in.

Initialise the ***name*** and ***charges per hour*** for the 1st ***Boat*** object using the data read in.

Store the object ***Boat*** in the array created.

Repeat the above for all the 5 Boat objects.

After reading in all the data from the text file, display the content of the array.

Possible solutions for (2)



Inheritance & Polymorphism Summary

1. You can derive a new class from an existing class. This is known as class inheritance. The new class is called a subclass, child class or derived class. The existing class is called a superclass, parent class, or a base class. Subclasses usually have more functionality than their superclasses.
2. A constructor is used to construct an instance of a class.
3. To override a method, the method must be defined in the subclass using the same signature as in its superclass.
4. An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.
5. Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.
6. The keywords super and this are used to reference the superclass and the subclass, respectively.
7. The final modifier is used to prevent changes to a class, a method, or a variable. A final class cannot be extended. A final method cannot be overridden. A final variable is a constant.
8. An object of a subclass can be used by any code designed to work with an object of its superclass. This is known as polymorphism.
9. An interface is a classlike construct that contains only constants and abstract methods.
10. An interface cannot be instantiated.
11. A subclass can only extend one superclass, but it can implement many interfaces to achieve multiple inheritance.