CHAPTER I

INTRODUCTION *JAVA* AND *OBJECT ORIENTED PROGRAMMING*

1.1 Practical Purpose

1. Students are starting to get to know the Java programming language
2. Students can use the Java programming language to create simple programs
3. Students can understand the concept of *object oriented programming (OOP)* in Java
4. Students can understand the features that can be used in Java along with the advantages of the Java language compared to other programming languages.

1.2 Material

Java is a programming language that can create all forms of applications, desktop, web, mobile and others, as created using other conventional programming languages. Java programming language is *object-oriented programming (OOP)* and can be run on various operating system platforms. The development of Java is not only focused on one operating system but is developed for various operating systems and is open source. With the slogan *"Write once, run anywhere"* .

1.2.1 Basic Syntax

Java can be defined as a collection of objects that communicate via invoking each other's methods. Some terms that need to be understood in Java, include *objects, classes*, *methods* and *instance variables*.

- **Object** – Objects have states and behaviors, for example, A dog has states: colour, breed and maybe nickname/name. Likewise, a dog also has several behaviors such as wagging its tail, barking and eating. An object is an instance of a class.
- **Class** – A class can be defined as a template/blueprint that describes the behavior/state that supports the object to be created
- **Method** – A method can be defined as a behavior. A class can contain several methods. This method is the placed where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variable** – Each object has its unique set of instance variables. An object's state is created by the value assigned to these instance variables.

Let's look at the following example program that prints the words Hello World.

```java
public class HelloWorld {
    /* This is my first java program.
     */
    public static void main(String args[]) {
        System.out.println("\nHello World"); // prints Hello World
    }
}
```

Now

Let's practice how to save, compile and run the program.

- Save the file as: 'HelloWorld.java'.
- Open a command prompt window and navigate to the directory where the program is running. Assume in C:\.
- Type 'javac HelloWorld.java' and press enter to compile the code. If there are no errors in your code, then the command prompt will take you to the next line
- Now type 'java HelloWorld' to run the program.
- You will be able to see 'Hello World' printed on the screen.

About Java program, it is very important to keep in mind the following points:

- **Case Sensitivity** – Java is case sensitive, which means identifier *Hello* and *hello* would have different meaning in Java.
- **Class Names** – For all class names, the first letter should be in uppercase, and if it consists of several words, the first letter of the word should be uppercase.
  Example: class MyFirstJavaClass
- **Method Names** – All method names should start with a lowercase letter. If it consists of several words, the first letter in the method name must be uppercase.
  Example: public void myMethodName()
- **Program File Name** – The name of the program file should exactly match the class name. When saving the file, it is usually saved using the class name (Remember Java is case sensitive) and ending in '.java'. If the file name and the class name are different then the compiler will not be able to compile it.
  Example: For example 'MyFirstJavaProgram' is the name of the class, then the file should be saved with the name 'MyFirstJavaProgram.java'
- **public static void main(String args[])** – Java program processing starts from the main() method which is a mandatory part of every Java program.

1.2.2 Java Primitive Types

The primitive or built-in variable types in Java are shown in Table 1.1

Table 1.1. Java Primitive Types

Type	Size (bits)	Minimum	Maximum	Example
bytes	8	-2^{7}	$2^{7}-1$	b = 100
short	16	-2^{15}	$2^{15}-1$	s = 30000
int	32	-2^{31}	$2^{31}-1$	i = 100000000
long	64	-2^{63}	$2^{63}-1$	l = 100000000000000
float	32	-10^{38}	$10^{38}-1$	f = 1.456
double	64	-10^{308}	$2^{308}-1$	d = 1.4567891012345678
char	16	0	$2^{16}-1$	c = 'c'
boolean	1	-	-	a = false b = true

1.2.3 Variables, Constants and Assignments

- Declaration of a variable of a specified type

```
type identifiers;
int options;
```

- Declaration of multiple variables of the same type, separated by commas

```
type identifier1, identifier2, … , identifierN;
double sum, difference, product, quotient;
```

- Variable declaration and assign an initial value

```
type identifier = initialValue;
int magicNumber = 88;
```

- Declare multiple variables and assign initial values

```
type identifier1 = initValue1, … , identifierN= initValue2;
String greetingMsg = ″Hi! ″, quitMsg = ″Bye! ″;
```

- Declare constants

```
final type identifier = value; //needs to be initialized
final double PI = 3.1415926;
```

- Assign a literal value (from right hand side - RHS) to a variable (from left hand side - LHS)

```
variable = literalValue;
number = 88;
```

- Evaluate the RHS expression and assign the result to the LHS variable

```
sum = sum + number;
```

1.2.4 Comments

There are two comments, namely *inline comments* and *block comments*. *Inline comments* are comments that are in one line, using the // sign.

Example :

```
// this is a comment
```

Block Comments are comments that explain more than one line, using the sign

```
/*....
....
....*/
```

Example :

```
/*this is comment 1
this is comment 2
This is a comment 3 */
```

1.2.5 Operators

Java supports several arithmetic operators as shown in Table 1.2

Table 1.2. Arithmetic Operators

Operator	Description	Usage	Example
`*`	Multiplication	`expr1 * expr2`	`2*3` →`6` `3.3 *1.0` →`3.3`
`/`	Distribution	`expr1 / expr2`	`1/2` →`0` `1.0 / 2.0` →`0.5`

`%`	Modulus (Remainder)	`expr1 % expr2`	`5 % 2` →`1`
			`-5 % 2` →`-1`
			`5.5 % 2.2` →`1.1`

`+`	Sum (or *unary positive)*	`expr1 + expr2`	`1 + 2` →`3` `1.1 + 2.2` →`3.3`
`-`	Subtraction (or *unary negative)*	`expr1 - expr2`	`1 - 2` →`-1` `1.1 - 2.2` →`-1.1`

Besides the usual simple *assignment operator* (=) previously described, Java also provides *compound assignment operators* as listed in Table 1.3. Java can also perform increment/ decrement operations, as in C++.

Table 1.3. Compound Assignment

Operator	Description	Usage	Example
=	Assignment Assign the value of the LHS to the variable at the RHS	`var = expr`	`x = 5`

+=	Compound addition and assignment	`var += expr`
		same as
		`var = var + expr`
			`x += 5`
			same as
			`x = x + 5`

-=	Compound subtraction and assignment	`var -= expr`
		same as
		`var = var - expr`
			`x -= 5`
			same as
			`x = x - 5`

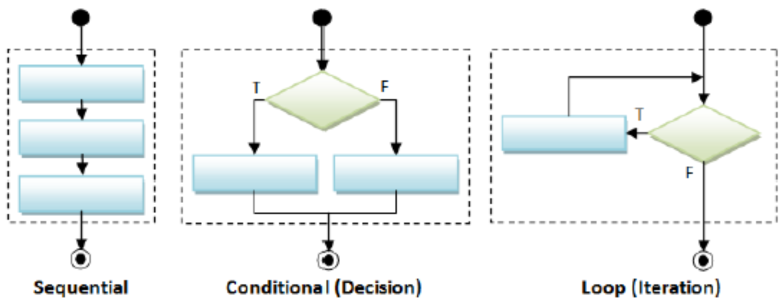
*=	Compound multiplication and assignment	`var *= expr`
		same as
		`var = var * expr`
			`x *= 5`
			same as
			`x = x * 5`

/=	Compound division and assignment	`var /= expr`
		same as
		`var = var / expr`
			`x /= 5`
			same as
			`x = x / 5`

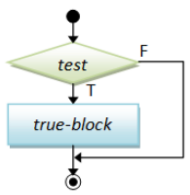
%=	Compound remainder (modulus) and assignment	`var %= expr`
		same as
		`var = var % expr`
			`x %= 5`
			same as
			`x = x % 5`

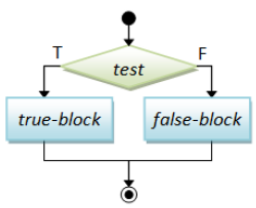
1.2.6 Flow Control

There are three basic *flow controls*, namely *sequential*, *conditional* (or *decision*), and *loop* (or *iteration*), as illustrated in the example below.



	Syntax	Example

	// if-then
	if(booleanExpression){
	  true-block ;
	}
		// if-then
		if (mark >= 50){
		  System.out.println("Congratulations!");
		  System.out.println("Keep it up!");
		}

	// if-then-else
	if(booleanExpression){
	  true-block ;
	} else {
	  False-block ;
	}
		// if-then-else
		if (mark >= 50){
		  System.out.println("Congratulations!");
		  System.out.println("Keep it up!");
		} else {
		  System.out.println("Try Harder!");
		}

1.2.7 Input

Java, like other languages, also supports input, output and error streams. We can input with the keyboard via ***System.in*** (standard input device). As for the output, it can be done with ***System.out*** (standard output device) and error with ***System.err*** (standard error device) automatically from the console. The following is an example program to calculate the ideal fat-burning heart rate.

FatBurningHeartRate.java ×

Source History

```java
package fatburningheartrate;

import java.util.Scanner;

public class FatBurningHeartRate {
    public static void main(String args[]) {
        // TODO code application logic here
        Scanner input = new Scanner(System.in);

        System.out.println("Enter your age in years: ");        //output
        double age = input.nextDouble();                        //input
        System.out.println("Enter your maximum heart rate: ");  //output
        double rate = input.nextDouble();                       //input

        double fb = (rate-age) * 0.65;

        System.out.println("Your ideal fat-burning heart rate is " + fb);
    }
}
```

Output - Run (fatburningheartrate) ×

```
Enter your age in years:
24
Enter your maximum heart rate:
80
Your ideal fat-burning heart rate is 36.4
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  30.744 s
```

1.2.8 Object Oriented Programming (OOP) in Java

Java is designed to be a simple language, minimizing errors, but still being robust. This is what distinguishes Java from other programming languages. A Java application is written in the Java language and utilizes the Java API (Application Programming Interface). The Java API provides a collection of ready-to-use classes that make writing applications easier.

Class, Object, Properties, Method

Class is a mold, template, prototype, the place of the object, while the ***object*** is the content of the class itself. One class can have more than one object or many. A simple example is a jelly mold that can produce a lot of jellies. Another example, locusts are like objects that have a name, eyes, legs, wings, color, type. Locuts can also fly and perch. The eyes, wings and colors of locuts in the programming world are also called ***attributes*** **or** ***properties***. While the activity is to fly and perch in the world of programming called the ***method***.

Table 1.4 Some words in Java that cannot be used as *class names*

Reserved Words				
abstract assert boolean break bytes case catch char class const continue	default do double else enum extends false final finally float for	goto if implements import instanceof int interface long native new null	package private protected public return short static strictfp super switch	synchronized this throw throws transient true try void volatile while

Modifier

Modifier is used to determine the relationship between a class element and other class elements. Based on the access control, *modifier* can be divided into 4 types, namely:

- ***Public*** : all elements contained in a class (method, object, etc.) can be freely accessed by all other classes that are in one package or not.
- ***Protected*** : all elements contained in a class (method, object, etc.) can be accessed by all other classes that are in one package and classes are part/derived of the initial class even though they are in different packages.

- ***Default*** : all elements contained in a class (method, object, etc.) can be accessed by all other classes that are in one package .
- ***Private*** : all elements contained in a class (method, object, etc.) can only be accessed by the class itself.

A summary of the differences between each *modifier* can be seen in Table 1.5.

Table 1.5 Modifiers

Visibility	public	protected	default	private
Same class	Yes	Yes	Yes	Yes
Class in the same package	Yes	Yes	Yes	No
Subclass in the same package	Yes	Yes	Yes	No
Subclass outside the same package	Yes	Yes	No	No
Non-subclass outside the same package	Yes	No	No	No

Previously we discussed the types of *access modifiers,* then we will discuss several types of *non-access modifiers,* namely:

- ***static*** is a type of modifier in Java that is used so that an attribute or method can be accessed by a class or object without having to instantiate that class. The main method is an example of a method that has a static modifier.
- ***final*** is one of the modifiers used so that an attribute or method is final or cannot be changed its value. This modifier is used to create constants in Java.
- ***abstract*** is a modifier used to create abstract classes and methods.

1.2.9 *Constructors*

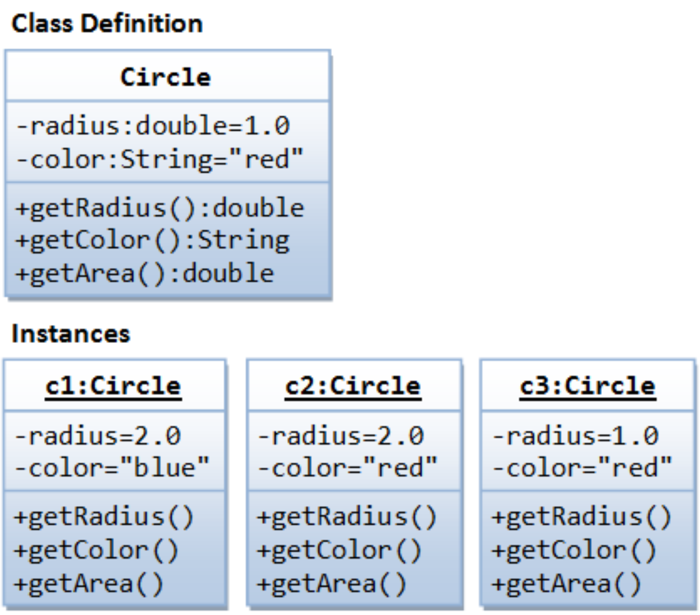
Constructor is a method that is automatically called/executed when a class is instantiated. Or in other words, the *constructor* is a method that is first executed when an object is first created. If in a class there is no constructor then Java will automatically create a default constructor.

1.2.10 *Encapsulation*

Encapsulation is a wrapper, wrapping here is intended to maintain a program process from being accessed arbitrarily or intervened by other programs. The concept of encapsulation is very important to maintain the program needs to be accessible at any time while maintaining the program.

In daily life, encapsulation can be considered as an electric current in a generator, and a generator rotation system to produce an electric current. The work of the electric current does not affect the work of the generator rotation system, and vice versa. Because in the electric current, we do not need to know how the performance of the generator rotation system is, whether the generator rotates backwards or forwards or even obliquely. Likewise in the generator rotation system, we do not need to know how the electric current is, whether it is on or not. That's the working concept of encapsulation, it will protect a program from access or intervention from other programs that affect it. This really maintains the integrity of the program that has been made with the concepts and plans that have been determined from the beginning.

Examples of using *constructors* and *encapsulation* can be seen in the following Circle class creation .



The following is a *Circle class* which is saved as **Circle.java**

```java
package circle;

public class Circle {  // Save as Circle.java
    // Private instance variables
    private double radius;
    private String color;

    // Constructors
    public Circle(){
        radius = 1.0;   // 1st Constructor
        color = "red";
    }

    public Circle(double r){
        radius = r;   // 2nd Constructor
        color = "red";
    }

    public Circle(double r, String c){
        radius = r;   // 3rd Constructor
        color = c;
    }

    public double getRadius(){
        return this.radius;
    }

    public String getColor(){
        return this.color;
    }

    public double getArea(){
        final double pi = 3.14;
        return pi*radius*radius;
    }
}
```

Here's a class to perform testing which is saved as **TestCircle.java**

```java
package circle;

public class TestCircle { // Save as TestCircle.java
    public static void main(String[] args) {    // Program entry point
        // Declare and Contruct an Instance of the Circle Class Called c1
        Circle c1 = new Circle(2.0, "blue");     // Use 3rd Constructor
        System.out.println("The radius is: " + c1.getRadius());     //use dot operator
        System.out.println("The color is: " + c1.getColor());
        System.out.printf("The area is: %.2f%n", c1.getArea());

        // Declare and Contruct another Instance of the Circle Class Called c2
        Circle c2 = new Circle(2.0);     // Use 2nd Constructor
        System.out.println("The radius is: " + c2.getRadius());
        System.out.println("The color is: " + c2.getColor());
        System.out.printf("The area is: %.2f%n", c2.getArea());

        // Declare and Contruct yet another Instance of the Circle Class Called c3
        Circle c3 = new Circle();     // Use 2nd Constructor
        System.out.println("The radius is: " + c3.getRadius());
        System.out.println("The color is: " + c3.getColor());
        System.out.printf("The area is: %.2f%n", c3.getArea());
    }
}
```

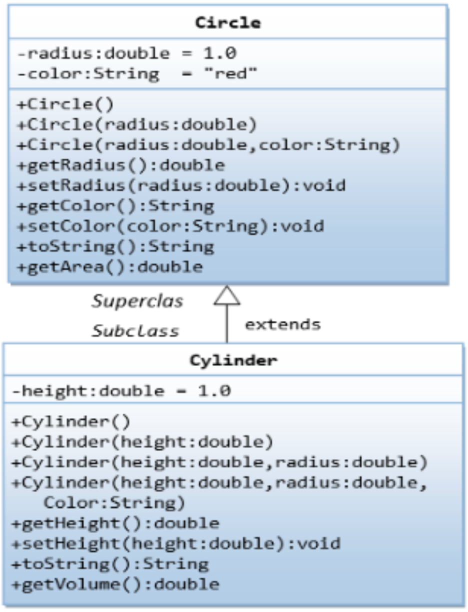
Here are the results when the program is run

```
Output – Run (circle)
The radius is: 2.0
The color is: blue
The area is: 12.56
The radius is: 2.0
The color is: red
The area is: 12.56
The radius is: 1.0
The color is: red
The area is: 3.14
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  3.947 s
```

1.2.11 Inheritance

Inheritance (succession) is a characteristic of OOP that is not found in old-style procedural programming. In this case, inheritance aims to form a new object that has the same or similar properties as an existing object (inheritance). Inheritance can be interpreted as the inheritance of the properties of an object to its derivative objects. Derived objects can be used to form child objects again and so on. Any changes to the parent object will also change the child object (its derivatives). The composition of the parent object with its child objects is called a hierarchy of objects. Inheritance is the inheritance of properties of an object to the object derivatives

The following is an example of using inheritance. In this example, we derive a subclass called Cylinder from the Circle superclass that we have created earlier. It should be noted that we can reuse the Circle class over and over again. Reusability is one of the most important properties of OOP. The Cylinder class inherits all the variables (*radius* and *color*) and methods (*getRadius(), getArea(),* and others) from the Circle superclass. It further defines a variable called *height* and two public methods – *getHeight()* and *getVolume()* and its own constructors as shown:



Create a *Cylinder class* and save it as **Cylinder.java**

```
package circle;

public class Cylinder extends Circle{
    // private instance variables
    private double height;

    //Constructors
    public Cylinder(){
        super();  //invoke superclass' constructor Circle
        this.height = 1.0;
    }

    public Cylinder(double height){
        super();  //invoke superclass' constructor Circle
        this.height = height;
    }
```

```java

    public Cylinder(double height, double radius){
        super(radius); //invoke superclass' constructor Circle(radius)
        this.height = height;
    }

    public Cylinder(double height, double radius, String color){
        super(radius, color);  //invoke superclass' constructor Circle(radius)
        this.height = height;
    }

    //Getter and Setter
    public double getHeight(){
        return this.height;
    }

    public void setHeight(double height){
        this.height = height;
    }

    //Return the volume of this Cylinder
    public double getVolume(){
        return getArea()*height;     // Use Circle's getArea()
    }
}
```

Then add a few lines of code following on **Circle.java**

```java
    public void setRadius(double radius){
        this.radius = radius;
    }

    public void setColor(String color){
        this.color = color;
    }

    public String toString(){
        return "This is a Circle";
    }
```

Next, to do testing, add a line of code following on **TestCircle.java**

```java

        // Declare and Contruct an Instance of the Cylinder Class Called c4
        Cylinder c4 = new Cylinder(500);     // Use Cylinder
        System.out.println("\nThe radius is: " + c4.getRadius());   // Invoke superclass Circle's methods
        System.out.println("The color is: " + c4.getColor());       // Invoke superclass Circle's methods
        System.out.printf("The area is: %.2f%n", c4.getArea());     // Invoke superclass Circle's methods
        System.out.println("The height is: " + c4.getHeight());
        System.out.printf("The volume is: %.2f%n", c4.getVolume());

```

The following is the result when the program is run

Output – Run (circle)

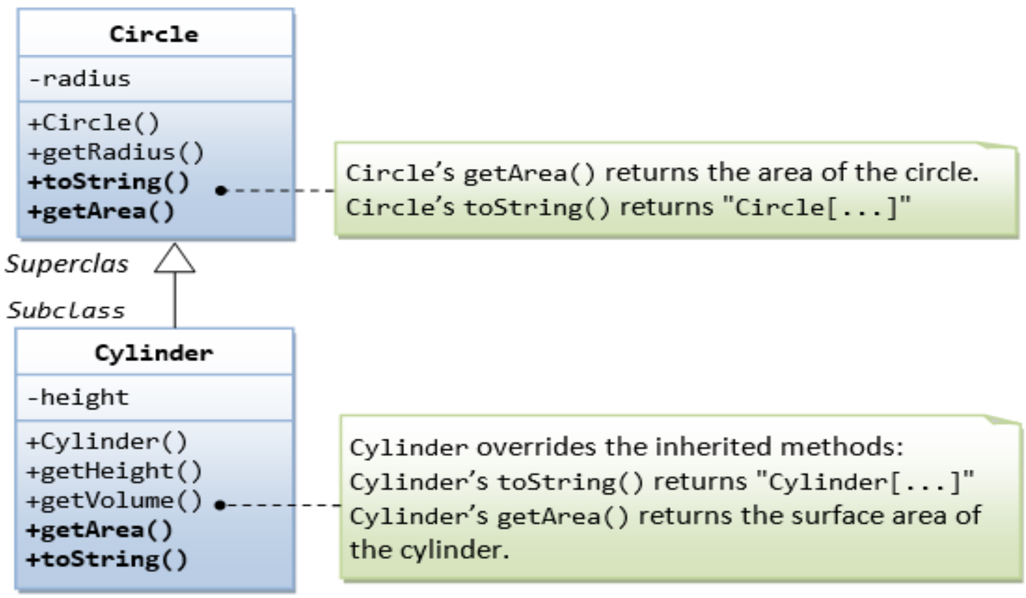
```
The radius is: 2.0
The color is: blue
The area is: 12.56
The radius is: 2.0
The color is: red
The area is: 12.56
The radius is: 1.0
The color is: red
The area is: 3.14

The radius is: 1.0
The color is: red
The area is: 3.14
The height is: 500.0
The volume is: 1570.00
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.299 s
```

1.2.12 Polymorphism

Polymorphism is an action that allows programmers to convey certain messages out of the object hierarchy, in which different objects give responses/feedback to the same message according to the nature of each object. Polymorphic can mean many forms, meaning that we can override, a method, which is derived from the parent class (super class) where the object is derived, so that it has different behavior.

In the previous Circle and Cylinder examples: Cylinder is a subclass of Circle. We can say that " Cylinder is a Circle" . The subclass-superclass relationship can be said to be an "is a" relationship . The following is an example of using polymorphism.



In the *Circle class,* change the *modifier* from *private* to *public.*

```
    // Change from private to public
    public double radius;
    public String color;

```

Then add a few lines of code following on **Cylinder.java**

```

    public double getArea(){
        final double pi = 3.14;
        return 2*pi*radius*radius + 2*pi*height;
    }

    // Define itself
    public String toString(){
        return "This is a Cylinder";     //to be refined later
    }
}

```

Next, add a few lines of code below in TestCircle.java and examine the output from lines of code 33-38.

```
30
31     // Declare and Contruct an Instance of the Cylinder Class Called c4
32    Circle c5 = new Cylinder(1.1,2.2);      // Use Cylinder
33    System.out.println("\nThe radius is: " + c5.getRadius());    // Invoke superclass Circle's methods
34    System.out.println("The color is: " + c5.getColor());       // Invoke superclass Circle's methods
      System.out.println("The height is: " + c5.getHeight());        //compilation error
      System.out.printf("The volume is: %.2f%n", c5.getVolume());     //compilation error
37    System.out.printf("The area is: %.2f%n", c5.getArea());      // Run the overridden version!
38    System.out.println(c5.toString());                          // Run the overridden version!
39
```

In this example, we create an instance of the Cylinder class and assign it to a Circle (superclass), as shown below:

```
// Substitute a subclass instance to a superclass reference
Circle c5 = new Cylinder(1.1, 2.2);
```

We can invoke all the methods defined in the Circle for the reference `c5`(which are derived from Cylinder object), for example

```
// Invoke Superclass Circle's methods
System.out.println("\nThe radius is: " + c5.getRadius());
System.out.println("The color is: " + c5.getColor());
```

This is because subclass instances possess all the properties of its superclass. However, we cannot invoke methods defined in the Cylinder class for the reference `c5`. Example :

```
//CANNOT invoke method in Cylinder as it is a Circle reference!
//compilation error
System.out.println("The height is: " + c5.getHeight());
System.out.printf("The volume is: %.2f%n", c5.getVolume());
```

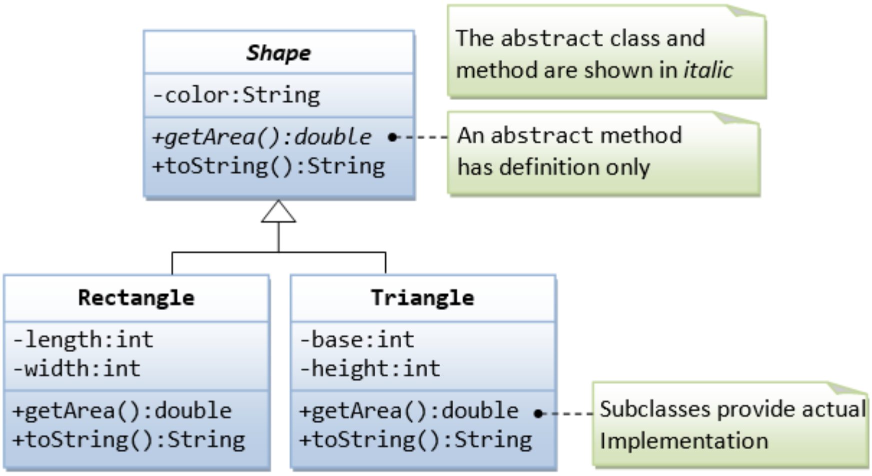
This is because `c5` refers to the Circle class, not the Cylinder class, which does not know about methods defined in the Cylinder subclass.

Although `c5` refers to the Circle class, but holds an object of its Cylinder subclass. Thus, even though `c5` references and maintains its internal identity, the Cylinder subclass overrides the method `getArea()` and `toString().` In this example, `c5.getArea()` or `c5.toString()` requests the overridden version defined in the Cylinder subclass instead of the version defined in the Circle class. This is because `c5` is in fact holding a Cylinder object internally.

```
// Run the overridden version!
System.out.printf("The area is: %.2f%n", c5.getArea());
System.out.println(c5.toString());
```

1.2.13 Abstract

An *abstract* method is a method with only a signature or a representation (e.g., method name, argument list and return type) without implementation (i.e., the method's body). We can use the abstract keyword to declare an abstract method. The following is an example of using abstract, in this diagram it is shown that there is an abstract class, *Shape*, and two subclasses that show the implementation of the abstract class namely *Rectangle* and *Triangle.*



Based on the example above, we create a class *Shape* first and save it as Shape.java, in this class we can declare abstract method *getArea(), draw(),* etc.

```java
package shape;

abstract public class Shape {
    //Private member variable
    private String color;

    //Constructor
    public Shape (String color){
        this.color = color;
    }

    @Override
    public String toString(){
        return "Shape of color=\"" + color + "\"";
    }

    //All shape subclass must implement a method called getArea()
    abstract double getArea();
}
```

Implementation of this method is not possible in the *Shape* class, because there are several unknown factors, e.g.how to calculate the area of a shape if the shape is not known. Implementation of these abstract methods will be provided later once the actual shape is known. This abstract method cannot be invoked because it has no real implementation. Therefore, we need to implement another subclass like the following.

The following is a *Triangle subclass* which is saved as **triangle.java**

```java
package shape;

public class Triangle extends Shape{
    //Private member variables
    private int base;
    private int height;

    //Constructor
    public Triangle(String color, int base, int height){
        super(color);
        this.base=base;
        this.height=height;
    }

    @Override
    public String toString(){
        return "Triangle[base=" + base + ",height=" + height + "," + super.toString() + "]";
    }

    //Override the inherited getArea() to provide the proper implementation
    @Override
    public double getArea(){
        return 0.5*base*height;
    }
}
```

The following is a *Rectangle subclass* saved as **Rectangle.java**

```java
package shape;

public class Rectangle extends Shape{
    //Private member variables
    private int length;
    private int width;

    //Constructor
    public Rectangle(String color, int length, int width){
        super(color);
        this.length=length;
        this.width=width;
    }

    @Override
    public String toString(){
        return "Rectangle[length=" + length + ",width=" + width + "," + super.toString() + "]";
    }

    //Override the inherited getArea() to provide the proper implementation
    @Override
    public double getArea(){
        return length*width;
    }
}
```

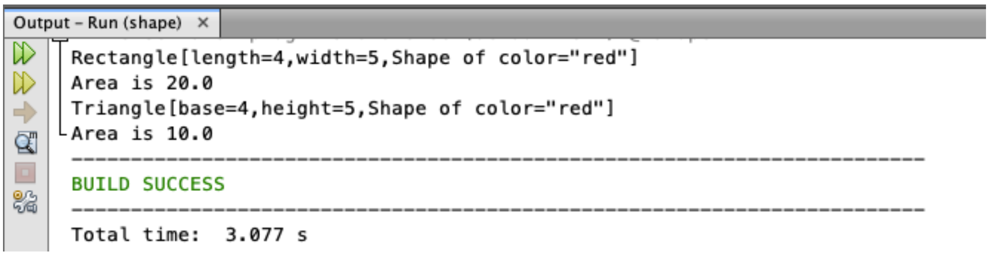
Here's a class to perform *testing* which is stored as **TestShape.java**

```java
package shape;

public class TestShape {
    public static void main(String[] args){
        Shape s1 = new Rectangle("red", 4, 5);
        System.out.println(s1);
        System.out.println("Area is " + s1.getArea());

        Shape s2 = new Triangle("red", 4, 5);
        System.out.println(s2);
        System.out.println("Area is " + s2.getArea());
    }
}
```

The following is the result when the program is run



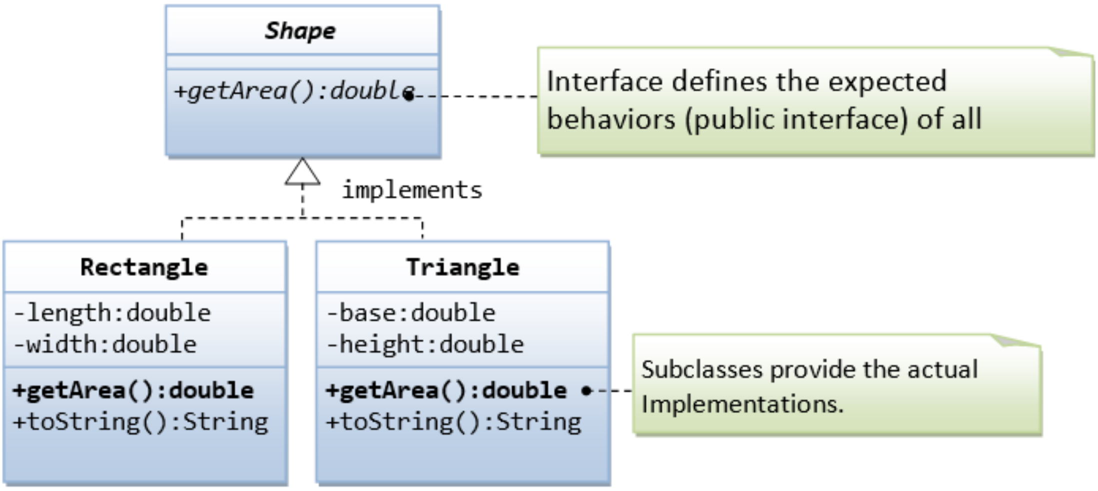
1.2.14 *Interface*

In simple terms, an object interface is a contract or method implementation agreement that contains what a class can do but does not specify how the class does it. For a class that uses the interface object, it must re-implement all the methods which are in the interface. In object-oriented programming, the term object interface is often shortened to interface just. An *interface* is a form, protocol, standard, contract, specification, set of rules for all objects that implement it. Like an abstract superclass, interfaces cannot be instantiated. If we have studied abstract class, then interface can be said to be another form, although the theoretical concept and purpose of use are different. Same as an abstract class, the interface also contains only public abstract methods (methods with signatures and no implementations) and possibly only constants or parameters (public static final variables) if any. The contents of the method will be recreated in the class that uses interfaces. If we consider abstract class as a framework or blueprint from other classes, then the interface is the implementation of a method that must be available in an object. Interface can't be called a framework class.

For example, if it is analogous to a computer, the interface can be exemplified with the mouse or keyboard. Inside the mouse interface, we can create methods like left_click(), right_click(), and double_click(). If the laptop class uses interface, then the class must regenerate the method left_click(), right_click(), double_click().

We have to use the *interface* keyword to define the interface (not the class keyword for normal classes). Public and abstract keywords are not required. On the interface, we must create a subclass that implements the interface and provides the actual implementation of all its methods. Unlike normal classes, we use the extended keyword to get the subclasses, for interfaces, we use the keyword *implements* to derive subclasses. In Java, abstract classes and interfaces are used to separate the public interface of a class from its implementation thus allowing programmers to program on interfaces rather than multiple implementations.

This diagram shows an example of using the interface.



Based on the example above, we create a *Shape2 class* first and save it as Shape2.java.

```java
/*
 * The interface Shape specifies the behaviors
 * of this implementations subclasses.
 */

package shape2;

public interface Shape2 {
    // Use keyword "interface" instead of "class"
    // List of public abstract methods to be implemented by its subclasses
    double getArea();
}

```

Then create a *Triangle subclass* which is saved as **Triangle.java**

```java
package shape2;

// The subclass Triangle need to implement all the abstract methods in Shape
public class Triangle implements Shape2 {
    // Private member variables
    private int base;
    private int height;

    // Constructor
    public Triangle(int base, int height) {
        this.base = base;
        this.height = height;
    }

    @Override
    public String toString() {
        return "Triangle[base=" + base + ",height=" + height + "]";
    }

    // Need to implement all the abstract methods defined in the interface
    @Override
    public double getArea() {
        return 0.5 * base * height;
    }
}
```

Next, make *Rectangle subclass* saved as **Rectangle.java**

```java
package shape2;

// The subclass Rectangle needs to implement all the abstract methods in Shape
public class Rectangle implements Shape2 {
    // using keyword "implements" instead of "extends"
    // Private member variables
    private int length;
    private int width;
    // Constructor
    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }
    @Override
    public String toString() {
        return "Rectangle[length=" + length + ",width=" + width + "]";
    }
    // Need to implement all the abstract methods defined in the interface
    @Override
    public double getArea() {
        return length * width;
    }
}
```

Here's a *class* to perform *testing* which is stored as **TestShape.java**

```java
package shape2;

public class TestShape {
    public static void main(String[] args) {
        Shape2 s1 = new Rectangle(1, 2); // upcast
        System.out.println(s1);
        System.out.println("Area is " + s1.getArea());
        Shape2 s2 = new Triangle(3, 4); // upcast
        System.out.println(s2);
        System.out.println("Area is " + s2.getArea());
        // Cannot create instance of an interface
        // Shape2 s3 = new Shape2("green"); // Compilation Error!!
    }
}
```

Here is the result when we run the program

```
Output - Run (shape2)
Rectangle[length=1,width=2]
Area is 2.0
Triangle[base=3,height=4]
Area is 6.0
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  3.000 s
```

1.3 Tasks

1. Create a program that defines an animal. These animals consist of 3 types, *carnivores*, *herbivores*, and *omnivores*. Then run the *eat method*, when the *eat method* on *carnivores* is executed it will display the result "Eat meat", when the *eat method* on *herbivores* is executed "Eat plants" will appear, when the *eat method* on *omnivores* is executed will appear "Eat meat and plants". Then, explain which ones are *abstract*, *polymorphisms* , *inheritance* or *encapsulation* .
2. Create a miniature hotel reservation program. For example, there is a hotel that consists of several rooms (at least 10), then when we run a *method* (free to define the method name) into a room, the room that has initial status is "free" changes to "reserved by (the name of the booker)".

   Example :

```
before hoho.getRoom001("Joni")
room001 : free
after hoho.getRoom001("Joni")
room001 : reserved by Joni
```

```
before hoho.getRoom009("Parker")
room009 : free
after hoho.getRoom001("Parker")
room009 : reserved by Parker
```