KATA PENGANTAR

Assalamu'alaikum Wr. Wb.

Alhamdullilah, atas berkat rahmat Allah Yang Maha Kuasa dengan didorongkan oleh keinginan luhur memperluas wawasan dalam pengembangan pengetahuan tentang Algoritma dan Struktur Data I, maka buku petunjuk (modul) praktikum ini disusun/dibuat.

Buku petunjuk (modul) Praktikum Pemrograman II ini dibuat untuk membantu jalannya Praktikum Pemrograman II prodi S1 Ilmu Komputer. Buku petunjuk (modul) ini dibuat sedemikian rupa sehingga dapat dengan mudah dipahami dan dipelajari oleh mereka yang belum pernah mengenal Algoritma dan Struktur Data sekalipun, dan bagi mereka yang pernah mengenal membaca buku ini akan menyegarkan ingatan.

Terima kasih kami ucapkan kepada semua pihak yang telah membantu dan mendukung pembuatan buku ini.

Wassalamu'alaikum Wr. Wb.

LABORATORIUM KOMPUTER DASAR
DIKE F.MIPA UGM

DAFTAR ISI

CHAPTER 1 – INTRODUCTION TO JAVA

1.1 LEARNING OBJECTIVES

1 Students began to recognize the Java Programming
2 Students can use the Java language to create programs
3 Students understand the terms on Java

1.2 MATERIALS

Java is a programming language that can make all forms of application, desktop, web, mobile and other, as made by using conventional programming language to another. Java programming language is object oriented (OOP-Object Oriented Programming), and can run on various operating systems. Java development is not only focused on one operating system, but is developed for a variety of operating systems and is open source. ***"Write once, run anywhere"***.

1..2.1 BASIC SYNTAX

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instance variables mean.

Object − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, 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 the object of its type supports.

Methods − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

Instance Variables − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

Let us look at a simple code that will print the words Hello World.

```
public class MyFirstJavaProgram {
   /* This is my first java program.
    */

   public static void main(String []args) {
      System.out.println("\nHello World"); // prints Hello World
      System.out.println("\nI Am Julia Robert ");
      System.out.println("\my Hobbies are Writing, Reading, Biking ");
      System.out.println("\nmy Favorite soccor is Messi")
      System.out.println("\nHow do you do today guyyy");
   }
}
```

Say Hello

InputExample.java | JavaApplication1.java

Source | History

```
1  package javaapplication1;
2
3  public class JavaApplication1 {
4
5      public static void helloGood(String stringTime, String stringName) {
6          System.out.println("\nGood " + stringTime);
7          System.out.println("\nMy name is " + stringName);
8      }
9
10     public static void myFavorites() {
11         System.out.println("my my hobbies are Writing, Reading, Biking ");
12         System.out.println("my favorite soccors are Messi, Cristian Ronaldo");
13         System.out.println("my best places to visit are Yogyakarta, Barobudur");
14         System.out.println("my ranking films are: \n1.Time Track, \n2.Superman");
15     }
16
17     public static int theTotalOfSalesToday(int[] sales) {
18         int sum = 0;
19         for (int itemSales : sales) {
20             sum += itemSales;
21         }
22         return sum;
23     }
24
25     public static void main(String[] args) {
26         System.out.println("Hello Yogyakarta");
27         System.out.println("How do you do today guyyy");
28         helloGood("morning", "Yulia Robert");
29         myFavorites();
30         int[] sales1 = {40, 22, 15, 29, 39, 22, 30, 35};
31         System.out.println("Total sales this weeks: " + theTotalOfSalesToday(sales1));
32     }
33 }
```

Output - JavaApplication1 (run)

```
run:
Hello Yogyakarta
How do you do today guyyy

Good morning

My name is Yulia Robert
my my hobbies are Writing, Reading, Biking
my favorite soccors are Messi, Cristian Ronaldo
my best places to visit are Yogyakarta, Barobudur
my ranking films are:
1.Time Track,
2.Superman
Total sales this weeks: 232
BUILD SUCCESSFUL (total time: 0 seconds)
```

Display my favorites and my perfromance by using methods

Let's look at how to save the file, compile, and run the program. Please follow the subsequent steps −
Open notepad and add the code as above.

- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

About Java programs, 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 Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
  Example: class MyFirstJavaClass
- **Method Names –** All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
  Example: public void myMethodName()
- **Program File Name –** Name of the program file should exactly match the class name.
  When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).
  Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as '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 PRIMITIVES TYPE

Table 1.1 – Java Primitives Type

TYPE	DESCRIPTION	
byte	Integer	8-bit signed integer The range is `[-2^7, 2^7-1] = [-128, 127]`
short		16-bit signed integer The range is `[-2^15, 2^15-1] = [-32768, 32767]`
int		32-bit signed integer The range is `[-2^31, 2^31-1] = [-2147483648, 2147483647]` (≈9 digits)
long		64-bit signed integer The range is [-2^63, 2^63-1] = [-9223372036854775808, +9223372036854775807] (≈19 digits)
float	Floating-Point Number	32-bit single precision floating-point number (≈6-7 significant decimal digits, in the range of `±[≈10^-45, ≈10^38]`)
double		64-bit double precision floating-point number (≈14-15 significant decimal digits, in the range of `±[≈10^-324, ≈10^308]`)

char	Character Represented in 16-bit Unicode `'\u0000'` to `'\uFFFF'`. Can be treated as 16-bit unsigned integers in the range of `[0, 65535]` in arithmetic operations.
boolean	Binary Takes a value of either `true` or `false`. The size of `boolean` is not defined in the Java specification, but requires at least one bit.

1.2.3 VARIABLE, CONSTANTS, AND ASSIGNMENTS

1. Declare a variable of a specified type

```
type identifier;
int option;
```

2. Declare multiple variables of the same type, separated by commas

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

3. Declare a variable and assign an initial value

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

4. Declare multiple variables with initial values

```
type identifier1 = initValue1, ..., identifierN = initValueN;
String greetingMsg = "Hi!", quitMsg = "Bye!";
```

5. Declare Constants

```
final double PI = 3.1415926;  // Need to initialize
```

6. Assign the literal value (of the RHS) to the variable (of the LHS)

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

7. Evaluate the expression (RHS) and assign the result to the variable (LHS)

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

1.2.4 COMMENTS

Ada dua komentar yaitu inline comment dan block comment.
Inline Comment yaitu komentar yang berada dalam satu baris, menggunakan tanda //
Contoh :

```
// ini komentar
```

Block Comment yaitu komentar yang penjelasannya lebih dari satu baris, menggunakan tanda

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

Contoh :

```
/*ini komentar 1
```

```
ini komentar 2
Ini komentar 3 */
```

1.2.5 OPERATORS

Java supports the following arithmetic operators:

Table 1.2 – Arithmetic Operators

Operator	Description	Usage	Examples
*	Multiplication	expr1 * expr2	2 * 3 → 6 3.3 * 1.0 → 3.3
/	Division	expr1 / expr2	1 / 2 → 0 1.0 / 2.0 → 0.5
%	Remainder (Modulus)	expr1 % expr2	5 % 2 → 1 -5 % 2 → -1 5.5 % 2.2 → 1.1
+	Addition (or unary positive)	expr1 + expr2 +expr	1 + 2 → 3 1.1 + 2.2 → 3.3
-	Subtraction (or unary negate)	expr1 - expr2 -expr	1 - 2 → -1 1.1 - 2.2 → -1.1

Besides the usual simple assignment operator (=) described earlier, Java also provides the so-called *compound assignment operator*s as listed:

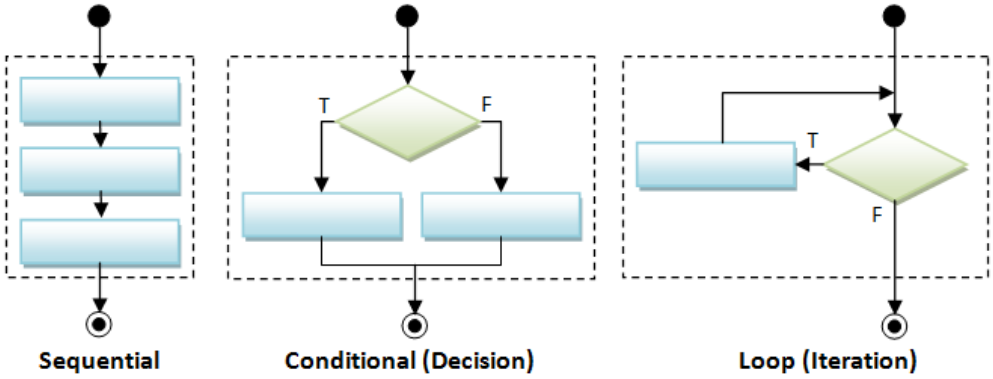
Table 1.3 – Compound Assignment

Operation	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

Java also have logical operators, bitwise operators, increment/decrement,etc, which have the similar syntax with c++ that you have learned in the previous semester.

1.2.6 FLOW CONTROL

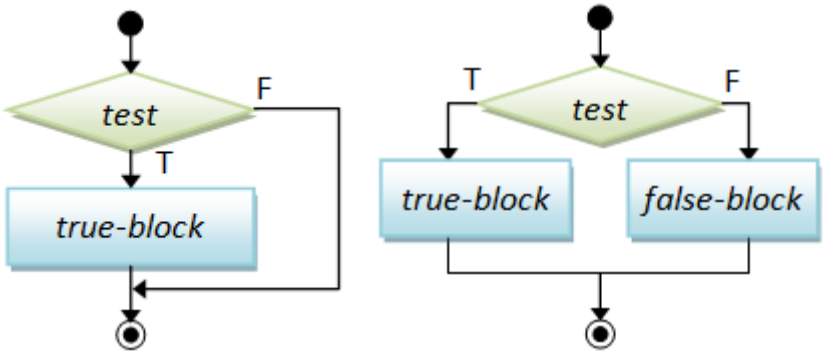
There are three basic flow control constructs - sequential, conditional (or decision), and loop (or iteration), as illustrated below.



Syntax	Example

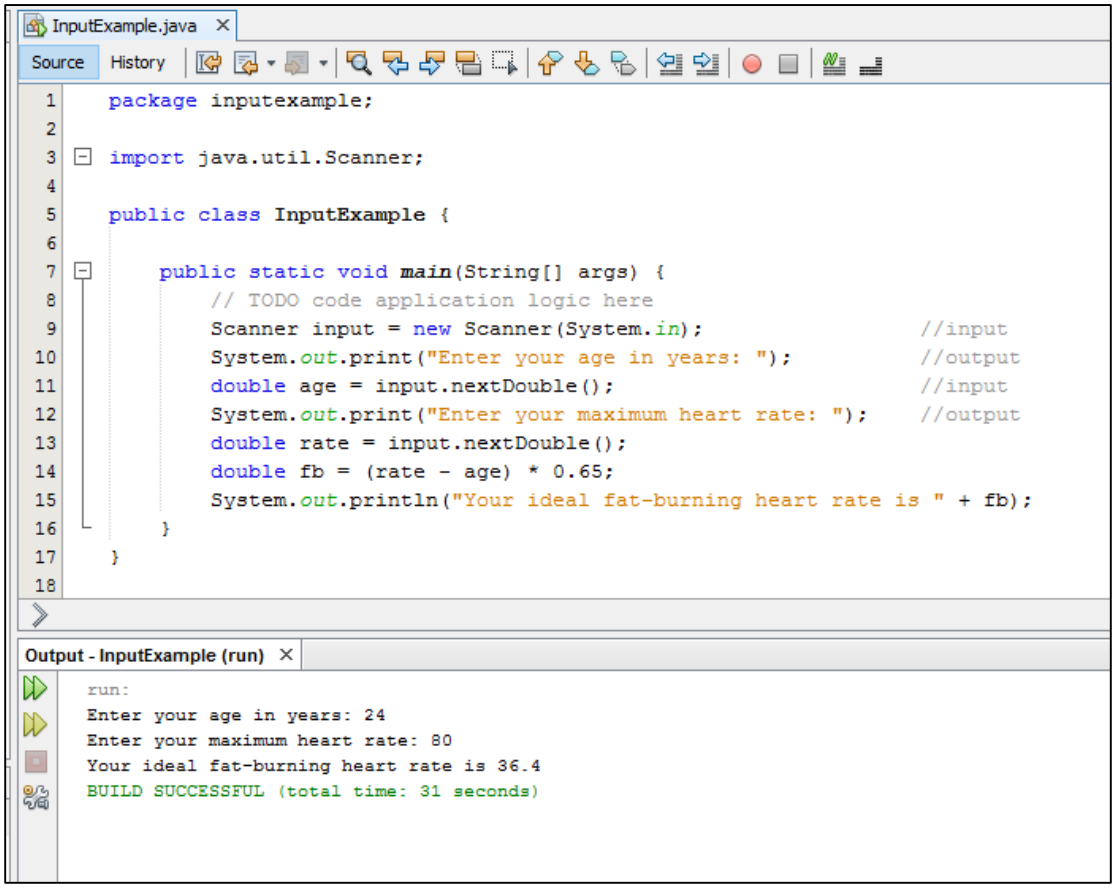
// if-then
if (booleanExpression **) {**
  true-block **;**
}
	if (mark >= 50) {
	  System.out.println("Congratulation!");
	  System.out.println("Keep it up!");
	}

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



1.2.7 INPUT

Java, like all other languages, supports three standard input/output streams: System.in (standard input device), System.out (standard output device), and System.err (standard error device). The System.in is defaulted to be the keyboard; while System.out and System.err are defaulted to the console. They can be re-directed to other devices, e.g., it is quite common to redirect System.err to a disk file to save these error message.
You can read input from keyboard via System.in (standard input device).



Fat Burning Hear Rate

1.2.8 INPUT FROM TEXT FILE

Other than scanning System.in (keyboard), you can connect your Scanner to scan any input source, such as a disk file or a network socket, and use the same set of methods nextInt(), nextDouble(), next(), nextLine() to parse the next int, double, String and line. For example,

```
Scanner in = new Scanner(new File("in.txt"));  // Construct a Scanner to scan
a text file
// Use the same set of methods
int anInt = in.nextInt();          // next String
```

```
double aDouble = in.nextDouble(); // next double
String str = in.next();           // next int
String line = in.nextLine();      // entire line
```

To open a file via new File(filename), you need to handle the so-called FileNotFoundException, i.e., the file that you are trying to open cannot be found. Otherwise, you cannot compile your program. There are two ways to handle this exception: throws or try-catch.

To run the program below, create a text file called in.txt containing:

Knowledge
“Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution.”
— Albert Einstein

“The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge.”
— Stephen Hawking

“The more you know, the more you realize you know nothing.”
— Socrates

“Tell me and I forget. Teach me and I remember. Involve me and I learn.”
— Benjamin Franklin

“Real knowledge is to know the extent of one’s ignorance.”
— Confucius

“If people never did silly things, nothing intelligent would ever get done.”
— Ludwig Wittgenstein

```java
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class FromText {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        String folder, fileName;
        folder = "E:";
        fileName = "in.txt";
        FileReader FR = new FileReader(folder + fileName); // could throw FileNotFoundExceptions
        int temp = FR.read(); //could throw IOException
        while (temp != -1) //-1 = EOF (End Of File)
        {
            System.out.print((char) temp); // typecast to char in order to print char, not a number
            temp = FR.read(); //could throw IOException
        }
    }
}
```

Output - Praktikum (run)

```
run:
Knowledge
□Imagination is more important than knowledge.  For knowledge is limited, whereas imagination embraces the entire world,
□ Albert Einstein

□The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge.□
□ Stephen Hawking

□The more you know, the more you realize you know nothing.□
□ Socrates

□Tell me and I forget.  Teach me and I remember.  Involve me and I learn.□
□ Benjamin Franklin

□Real knowledge is to know the extent of one□s ignorance.□
□ Confucius
```

CHAPTER 2 – OBJECT ORIENTED PROGRAMMING

2.1 LEARNING OBJECTIVES

1. Praktikkan paham perbedaan bahasa Java dengan bahasa pemrograman lain
2. Praktikkan paham keunggulan dari bahasa Java
3. Praktikkan paham fitur-fitur yang bisa digunakan pada java

2.2 MATERIALS

Java is designed to be a simple language, minimize mistakes, but still formidable. That is what distinguishes it from other programming languages. A Java application written in Java and take advantage of Java APIs (Application Programming Interface). Java API provides a collection of ready-made classes that facilitate the writing of the application.

2.2.1 CLASS , OBJECT, ATRIBUT DAN 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 as follows: a jelly mold bias produces many jelly.

Locusts are like objects that have a name, eyes, legs, wings, color, type. Locusts can also fly and perch. Eyes, legs and wings of color in the world of programming is also called attributes or properties. While the activity is flying and landed in the world of programming called the method.

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

Gambar 2.1 Sejumlah kalimat di Java yang tidak bisa digunakan sebagai nama class.

Modifier

Modifier is used to determine the relationship of an element class with elements of other classes. And modifier itself has several types according to their access, namely:

• Public: all the elements contained in a class (method, object, etc.) can be freely accessed by all other classes are in one package or not.

• Protected: all the elements contained in a class (method, object, etc.) can be accessed by all other classes are in one package and class parts / derivatives of the initial class albeit different package.

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

• Private: all the elements contained in a class (method, object, etc.) can be accessed by the class itself.

Access Modifiers	Same Class	Same Package	Subclass	Other packages
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

Gambar 2.2 Gambaran Modifiers

- ***static***
  Static adalah salah satu jenis modifier di Java yang digunakan agar suatu atribut atau pun method dapat diakses oleh kelas atau objek tanpa harus melakukan instansiasi terhadap kelas tersebut.
  Method main adalah salah satu contoh method yang mempunyai modifier static.

- ***final***
  Final adalah salah satu modifier yang digunakan agar suatu atribut atau method bersifat final atau tidak bisa diubah nilainya. Modifier ini digunakan untuk membuat konstanta di Java.

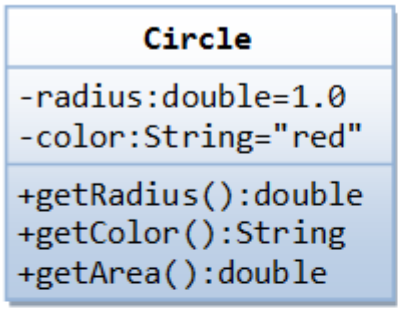
- ***abstract***
  Abstract adalah modifier yang digunakan untuk membuat kelas dan method abstrak.

2.2.2 DEFINING CONSTRUCTOR

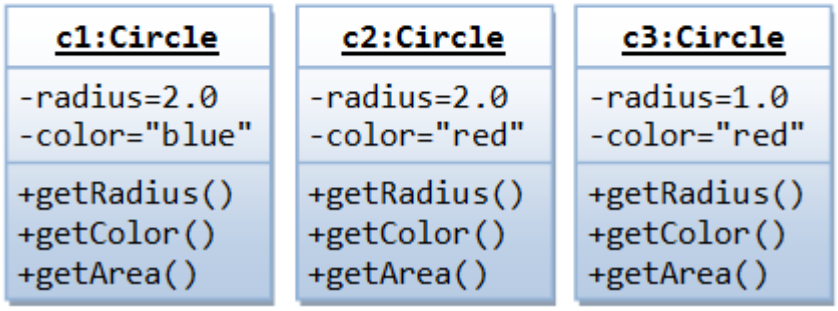
Constructor is a method that is automatically invoked / executed at a class diinstansi. Or in other words constructor is a method that first run when an object is first created. If a class is not found automatically constructor Java will create a default constructor.

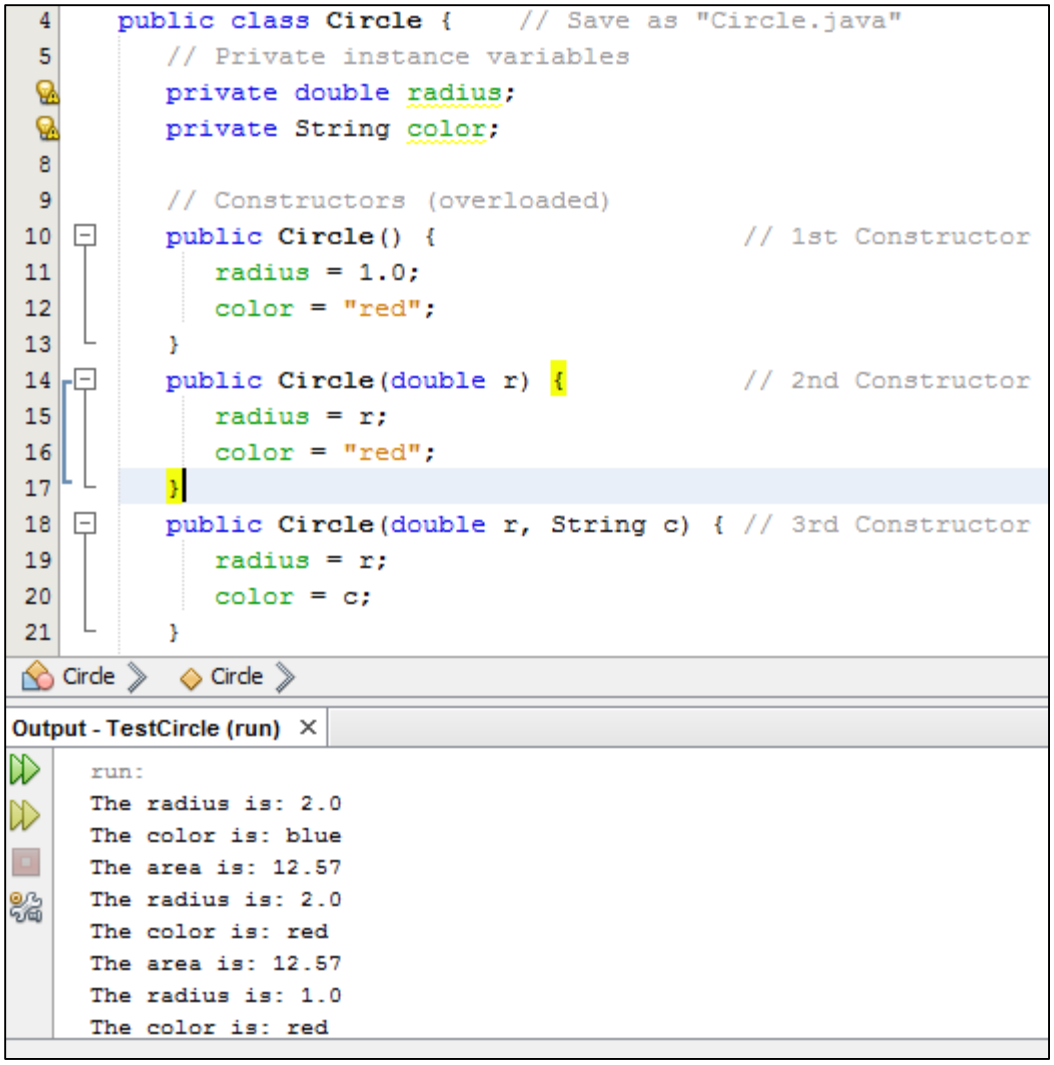
Here is an example of class and constructor.

Class Definition



Instances





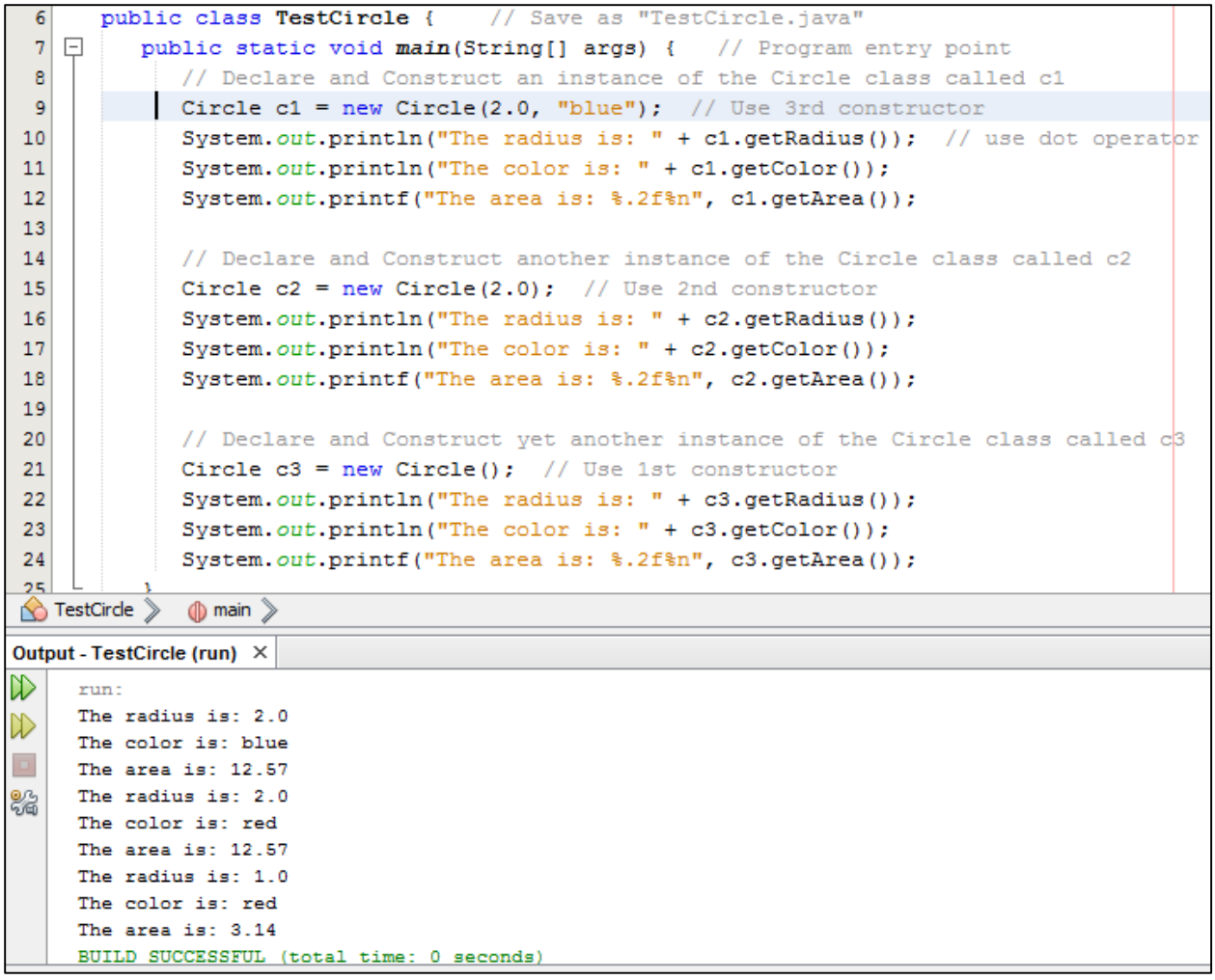
```
4   public class Circle {    // Save as "Circle.java"
5      // Private instance variables
6      private double radius;
7      private String color;
8
9      // Constructors (overloaded)
10     public Circle() {                    // 1st Constructor
11        radius = 1.0;
12        color = "red";
13     }
14     public Circle(double r) {            // 2nd Constructor
15        radius = r;
16        color = "red";
17     }
18     public Circle(double r, String c) { // 3rd Constructor
19        radius = r;
20        color = c;
21     }
```

Circle › Circle

Output - TestCircle (run)

```
run:
The radius is: 2.0
The color is: blue
The area is: 12.57
The radius is: 2.0
The color is: red
The area is: 12.57
The radius is: 1.0
The color is: red
```

Berikut kelas untuk melakukan testing.



```
public class TestCircle {    // Save as "TestCircle.java"
   public static void main(String[] args) {   // Program entry point
      // Declare and Construct 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 Construct 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 Construct yet another instance of the Circle class called c3
      Circle c3 = new Circle();  // Use 1st 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());
   }
}
```

TestCircle › main

Output - TestCircle (run)

```
run:
The radius is: 2.0
The color is: blue
The area is: 12.57
The radius is: 2.0
The color is: red
The area is: 12.57
The radius is: 1.0
The color is: red
The area is: 3.14
BUILD SUCCESSFUL (total time: 0 seconds)
```

2.2.3 ENCAPSULATION

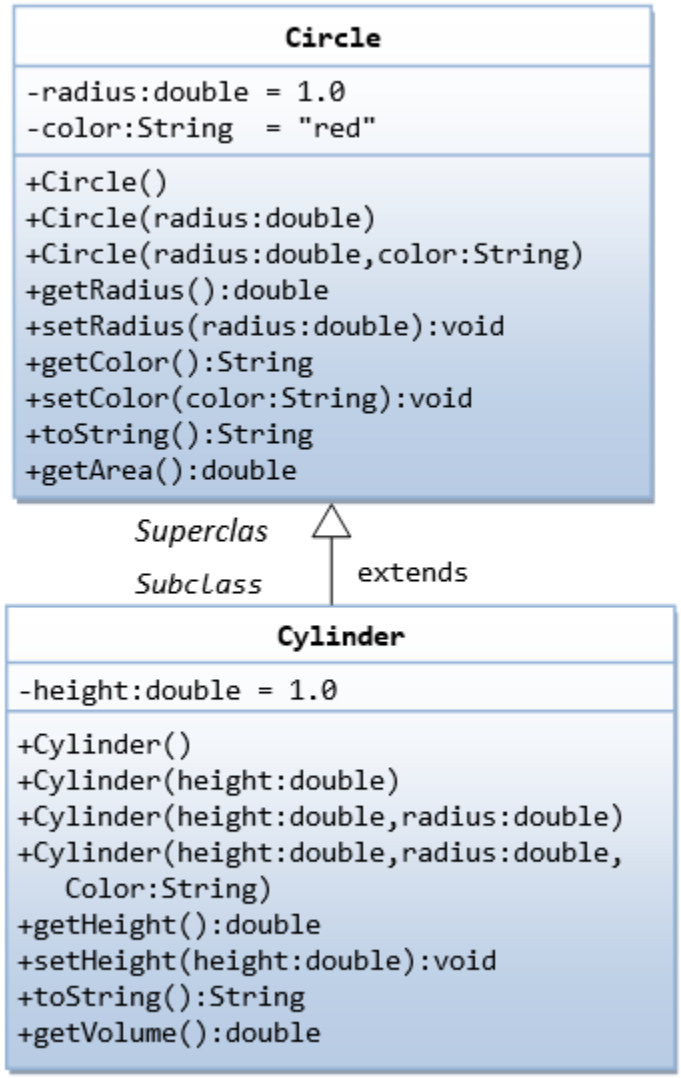
Encapsulation is wrapping, wrapping herein is intended to maintain a process that can not be accessed program arbitrarily or intervention by other programs. The concept of encapsulation is very important to maintain the program needs to be accessible at any time, while keeping the program.

In daily life encapsulation can be exemplified as the electric current in the generator, and a rotation system generator to produce electric current. Working electric current does not affect the working of the system of rotation of the generator, and vice versa. Because in the electric current, we do not need to know how the rotation system performance generator, if the generator rotates backward or forward or even oblique. Similarly, in the system of rotation of the generator, we do not need to know how the electric current, whether lit or not.

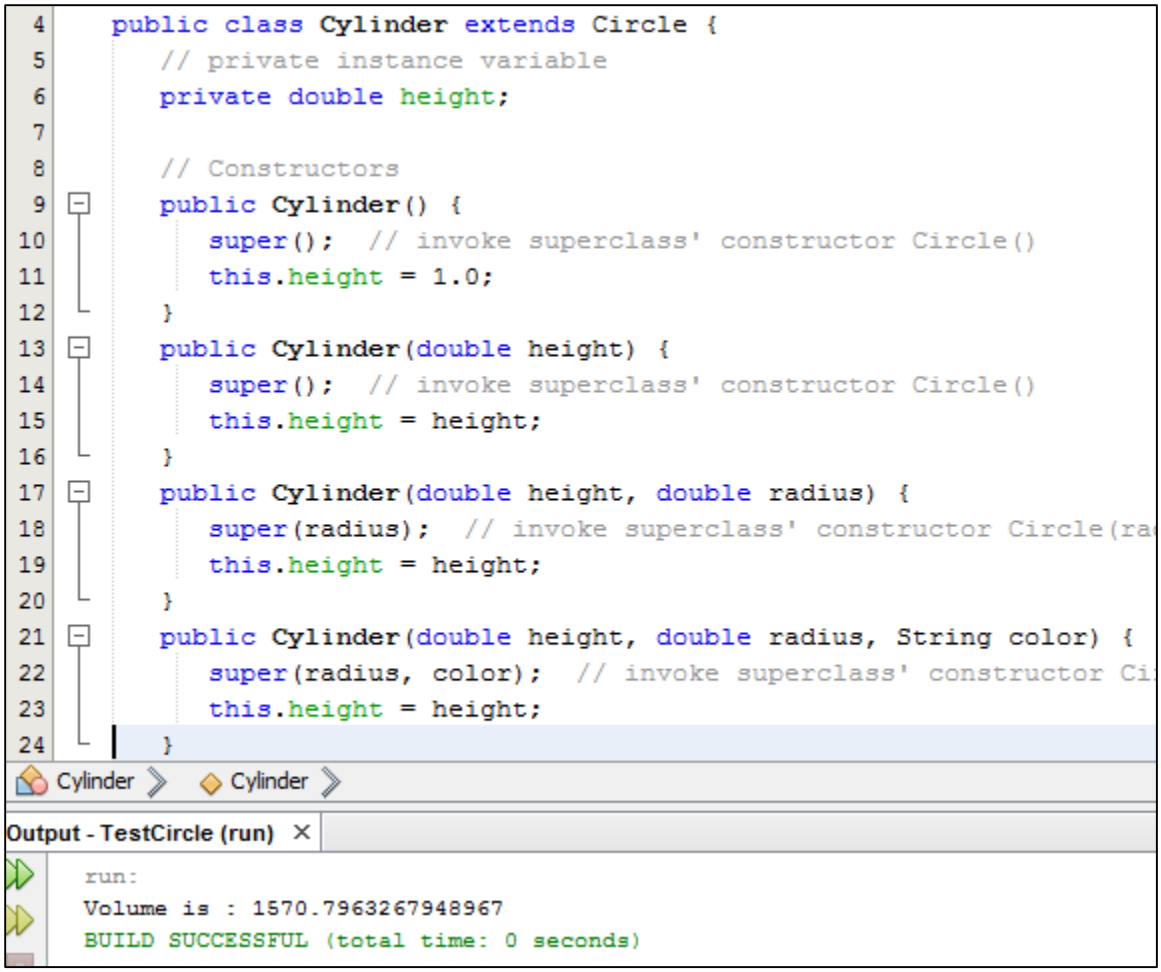
That concept of encapsulation work, he would protect a program of access to or intervention of other programs that influence it. It is very maintaining the integrity of the program that was created with the concept and plans that have been determined from the outset. An example of this can be seen in the Circle class.

2.2.4 INHERITANCE

Inheritance (inheritance / succession), this is a characteristic of OOP are not contained in the old style procedural programming. In this case, inheritance aims to form a new object that has the same properties as or similar to previously existing objects (inheritance). Object derivatives may be used Markowitz derivative object again and so on. Any changes to the parent object, the object will also change its derivatives. The composition of the parent object by object derivative called a hierarchy of objects. Or Inheritance is the inheritance of properties of an object to the object derivatives.



In this example, we derive a subclass called Cylinder from the superclass Circle, which we have created in the previous chapter. It is important to note that we reuse the class Circle. Reusability is one of the most important properties of OOP. (Why reinvent the wheels?) The class Cylinder inherits all the member variables (radius and color) and methods (getRadius(), getArea(), among others) from its superclass Circle. It further defines a variable called height, two public methods - getHeight() and getVolume() and its own constructors, as shown:



```
4   public class Cylinder extends Circle {
5       // private instance variable
6       private double height;
7
8       // Constructors
9       public Cylinder() {
10          super();  // invoke superclass' constructor Circle()
11          this.height = 1.0;
12      }
13      public Cylinder(double height) {
14          super();  // invoke superclass' constructor Circle()
15          this.height = height;
16      }
17      public Cylinder(double height, double radius) {
18          super(radius);  // invoke superclass' constructor Circle(ra
19          this.height = height;
20      }
21      public Cylinder(double height, double radius, String color) {
22          super(radius, color);  // invoke superclass' constructor Ci
23          this.height = height;
24      }
```

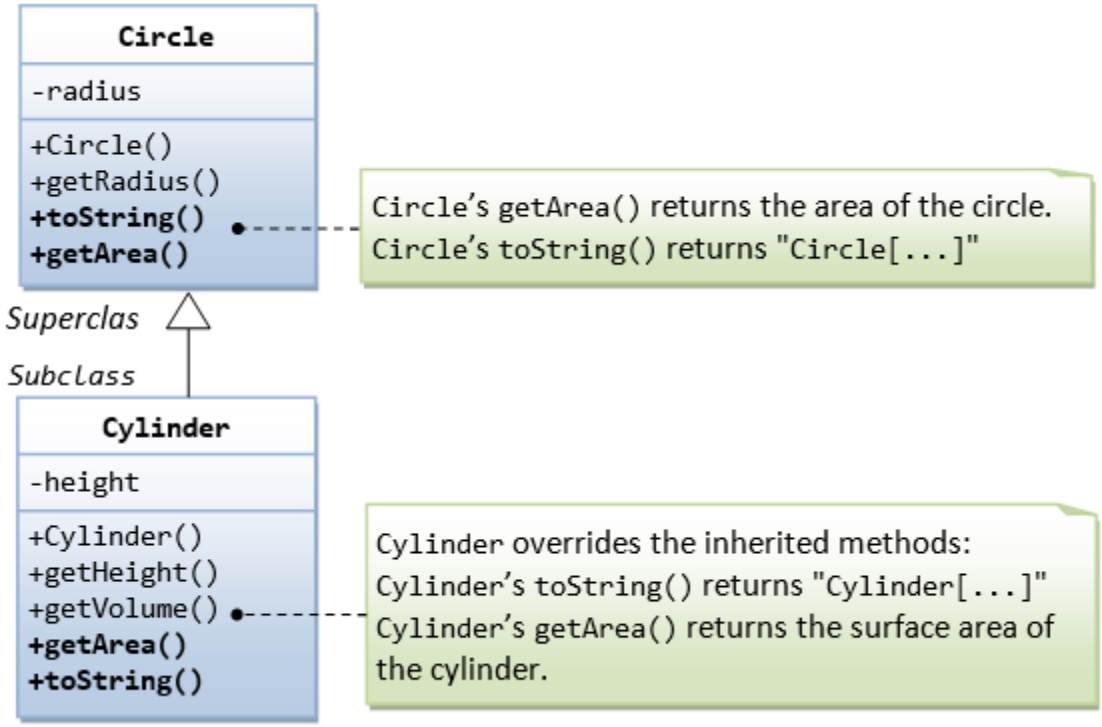
Cylinder > Cylinder >

Output - TestCircle (run)

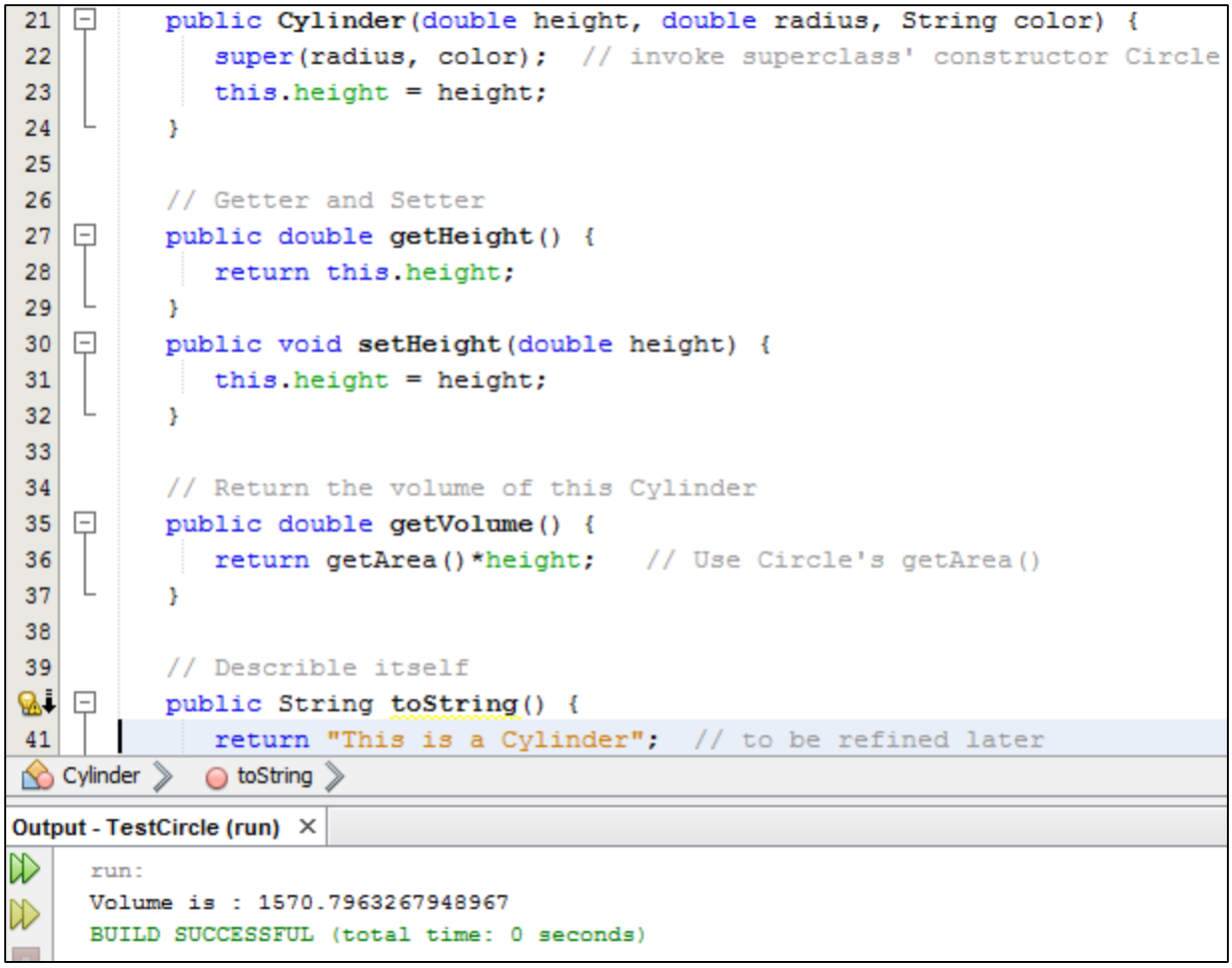
```
run:
Volume is : 1570.7963267948967
BUILD SUCCESSFUL (total time: 0 seconds)
```

2.2.5 POLYMORPHISM

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



In our earlier example of Circle and Cylinder: Cylinder is a subclass of Circle. We can say that Cylinder "is-a" Circle (actually, it "is-more-than-a" Circle). Subclass-superclass exhibits a so called "is-a" relationship.



```
21      public Cylinder(double height, double radius, String color) {
22          super(radius, color);  // invoke superclass' constructor Circle
23          this.height = height;
24      }
25
26      // Getter and Setter
27      public double getHeight() {
28          return this.height;
29      }
30      public void setHeight(double height) {
31          this.height = height;
32      }
33
34      // Return the volume of this Cylinder
35      public double getVolume() {
36          return getArea()*height;    // Use Circle's getArea()
37      }
38
39      // Describle itself
        public String toString() {
41          return "This is a Cylinder";  // to be refined later
```

Cylinder > toString >

Output - TestCircle (run)

```
run:
Volume is : 1570.7963267948967
BUILD SUCCESSFUL (total time: 0 seconds)
```

Via substitutability, we can create an instance of Cylinder, and assign it to a Circle (its superclass) reference, as follows:

```
// Substitute a subclass instance to a superclass reference
Circle c1 = new Cylinder(1.1, 2.2);
```

You can invoke all the methods defined in the Circle class for the reference c1, (which is actually holding a Cylinder object), e.g.

```
// Invoke superclass Circle's methods
c1.getRadius();
```

This is because a subclass instance possesses all the properties of its superclass.

However, you CANNOT invoke methods defined in the Cylinder class for the reference c1, e.g.

```
// CANNOT invoke method in Cylinder as it is a Circle reference!
c1.getHeight();  // compilation error
c1.getVolume();  // compilation error
```

This is because c1 is a reference to the Circle class, which does not know about methods defined in the subclass Cylinder.

c1 is a reference to the Circle class, but holds an object of its subclass Cylinder. The reference c1, however, retains its internal identity. In our example, the subclass Cylinder overrides methods getArea() and toString(). c1.getArea() or c1.toString() invokes the overridden version defined in the subclass Cylinder, instead of the version defined in Circle. This is because c1 is in fact holding a Cylinder object internally.

```
c1.toString();  // Run the overridden version!
```

```
c1.getArea();   // Run the overridden version!
```

2.2.6 ABSTRACT

An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body). You use the keyword abstract to declare an abstract method.

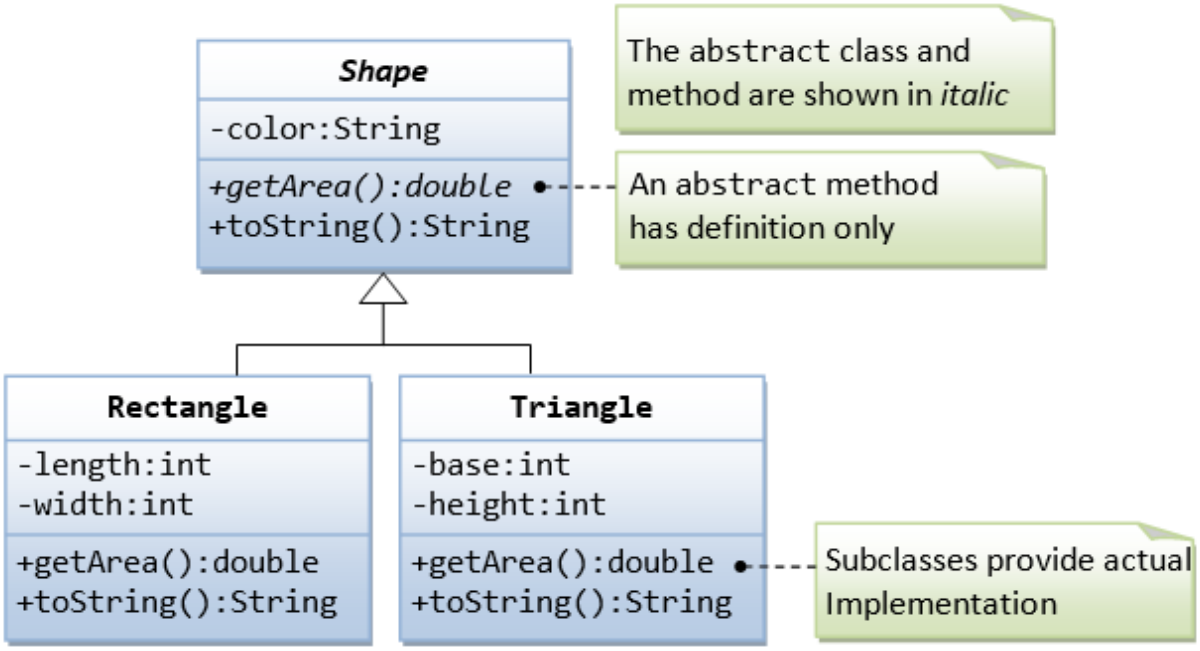
For example, in the Shape class, we can declare abstract methods getArea(), draw(), etc, as follows:

```
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 subclasses must implement a method called getArea()
   abstract public double getArea();
}
```

Shape > Shape

```
Output - TestShape (run)
run:
Rectangle[length=4,width=5,Shape of color="red"]
Area is 20.0
Triangle[base=4,height=5,Shape of color="blue"]
Area is 10.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Implementation of these methods is NOT possible in the Shape class, as the actual shape is not yet known. (How to compute the area if the shape is not known?) Implementation of these abstract methods will be provided later once the actual shape is known. These abstract methods cannot be invoked because they have no implementation.



Here is the implementation from above Diagram.
Shape.java

```
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 subclasses must implement a method called getArea()
    abstract public double getArea();
}
```

Shape › Shape

Output - TestShape (run)

```
run:
Rectangle[length=4,width=5,Shape of color="red"]
Area is 20.0
Triangle[base=4,height=5,Shape of color="blue"]
Area is 10.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Triangle.java

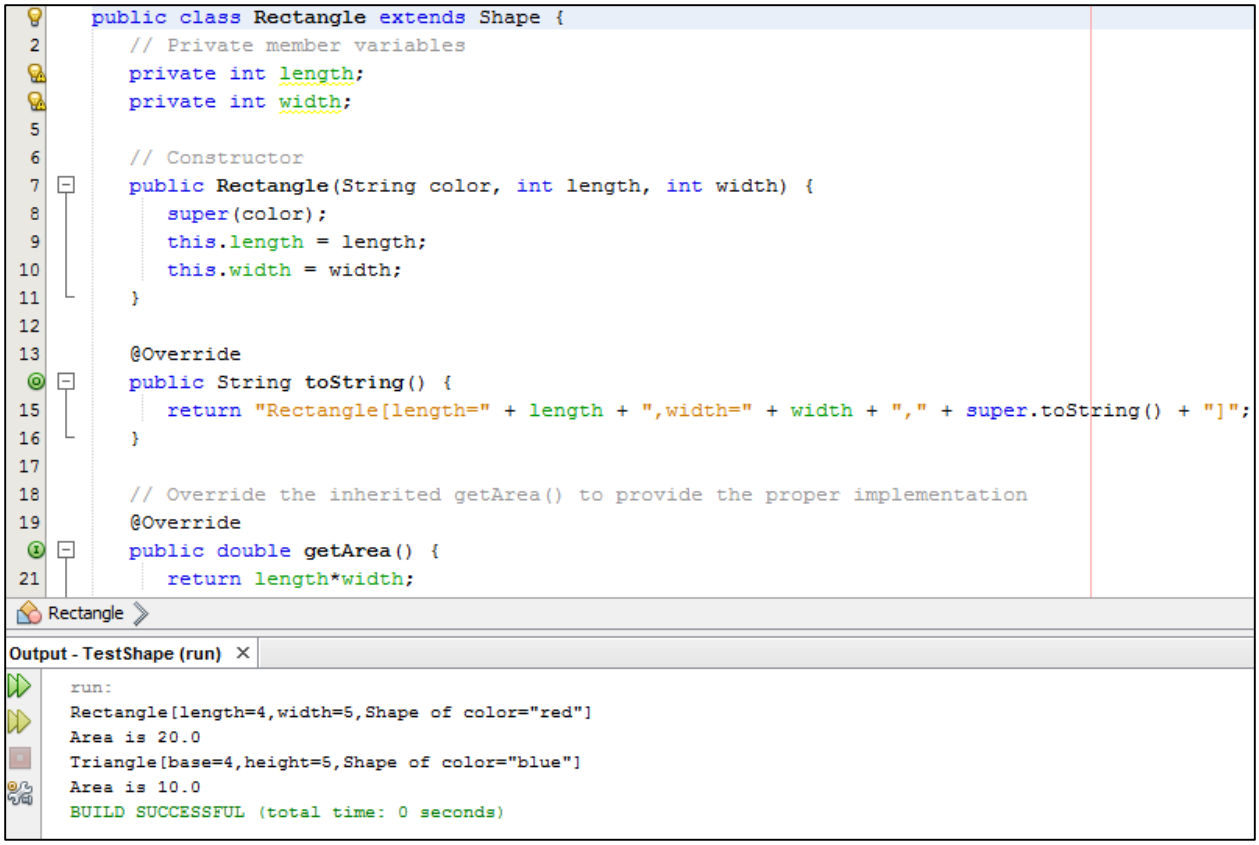
```java
public class Triangle extends Shape {
    // Private member variables
    private int base;
    private int height;

    // 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;
```

Triangle > toString

Output - TestShape (run)

```
run:
Rectangle[length=4,width=5,Shape of color="red"]
Area is 20.0
Triangle[base=4,height=5,Shape of color="blue"]
Area is 10.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Rectangle.java



```
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;
```

Rectangle

Output - TestShape (run)

```
run:
Rectangle[length=4,width=5,Shape of color="red"]
Area is 20.0
Triangle[base=4,height=5,Shape of color="blue"]
Area is 10.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

TestShape.java

```java
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("blue", 4, 5);
        System.out.println(s2);
        System.out.println("Area is " + s2.getArea());

    }
}
```

TestShape > main

Output - TestShape (run)

```
run:
Rectangle[length=4,width=5,Shape of color="red"]
Area is 20.0
Triangle[base=4,height=5,Shape of color="blue"]
Area is 10.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

2.2.7 INTERFACE

A Java interface is a 100% abstract superclass which define a set of methods its subclasses must support. An interface contains only public abstract methods (methods with signature and no implementation) and possibly constants (public static final variables). You have to use the keyword "interface" to define an interface (instead of keyword "class" for normal classes). The keyword public and abstract are not needed for its abstract methods as they are mandatory.

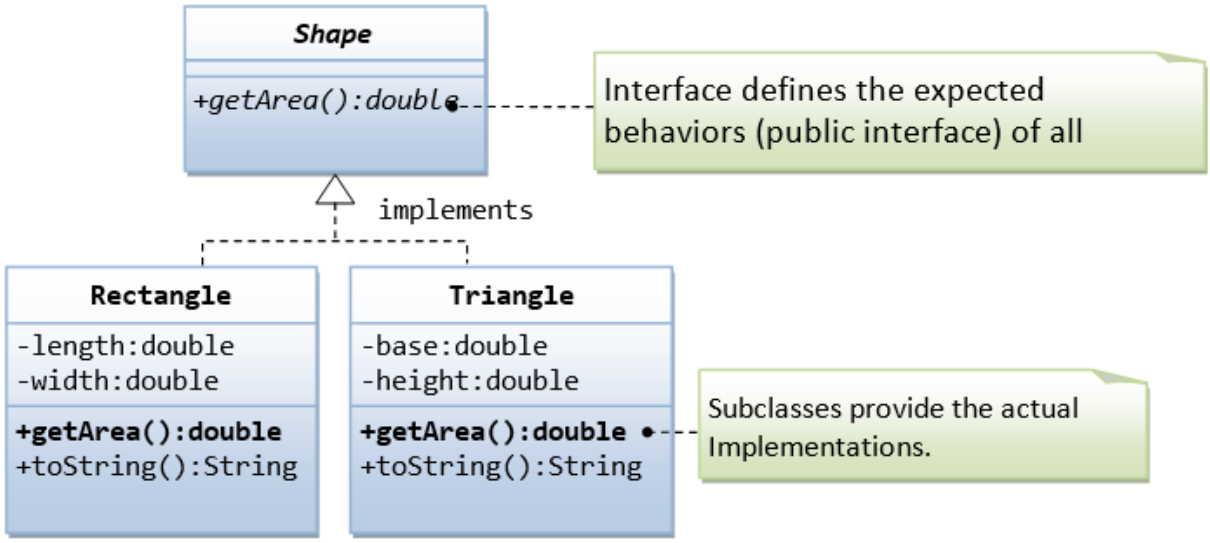
Similar to an abstract superclass, an interface cannot be instantiated. You have to create a "subclass" that implements an interface, and provide the actual implementation of all the abstract methods.

Unlike a normal class, where you use the keyword "extends" to derive a subclass. For interface, we use the keyword "implements" to derive a subclass.

An interface is a contract for what the classes can do. It, however, does not specify how the classes should do it.

An interface provides a form, a protocol, a standard, a contract, a specification, a set of rules, an interface, for all objects that implement it. It is a specification and rules that any object implementing it agrees to follow.

In Java, abstract class and interface are used to separate the public interface of a class from its implementation so as to allow the programmer to program at the interface instead of the various implementation.




```
/*
 * The interface Shape specifies the behaviors
 * of this implementations subclasses.
 */
public interface Shape {
   // Use keyword "interface" instead of "class"
   // List of public abstract methods to be implemented by its subclasses
   double getArea();
}
```

```
// The subclass Rectangle needs to implement all the abstract methods in Shape
public class Rectangle implements Shape {
   // 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;
   }
}
```

```
// The subclass Triangle need to implement all the abstract methods in Shape
public class Triangle implements Shape {
   // 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;
   }
}
```

A test driver is as follows:

```
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle(1, 2);  // upcast
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle(3, 4);  // upcast
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());

      // Cannot create instance of an interface
      //Shape s3 = new Shape("green");   // Compilation Error!!
   }
}
```

2.3 EXERCISES

1. Create a program that defines an animal, and there are three types of animal, carnivores, herbivores, omnivores, then run eat method, when we test carnivore eat method the results will be "Eating meat", herbivores eat method will appear "Eating Plant ", omnivores eat method will appear" Eating meat and plants "

2. From the problem number one explain about and give example abstraction, polymorphisms, inheritance and encapsulation.

2.4 HOMEWORK

1. Create a miniature hotel reservation program, so there is a hotel, consist of at least 10 rooms, and then when we do a method (free of its method name) into one of the rooms, the room that has initial status is “free” will change to “Reserved by (the name of booker)”

   Example :

   ```
   before hoho.getRoom001(“Joni”)
   room001 : free

   after hoho.getRoom001(“Joni”)
   room001 : reserved by Joni
   ```

   ```
   before hoho.getRoom009(“Parker”)
   room009 : free

   after hoho.getRoom009(“Parker”)
   room009 : reserved by Parker
   ```

CHAPTER 3 – ARRAY & LINKED LIST

3.1 LEARNING OBJECTIVES

1. Students can understand about array and linked list
2. Students are able to declare and use arrays and linked lists
3. Students can know when to use arrays and when to use linked lists

3.2 MATERIALS

3.2.1 Array

Array is one of the basic data structures available, and is a collection of many variables with the same data type, indicated by an index for each element. We can think of arrays as tables, where each element is stored in each table space. There are 2 types of array: one-dimensional array called vector and two-dimensional array called matrix.

In Java, arrays are a primitive type that is treated as objects. To create an array, you must use the *new* operator:

```
int[] intArray;          // defines a reference to an array
intArray = new int[100];// creates the array, and sets intArray to refer to it
```

The above declaration can be shortened into one statement:

```
int[] intArray = new int[100];
```

This declaration will create an array with *null* elements. If you want to insert elements at the start of declaration, you can do this:

```
int[] intArray = {0,3,6,9,12,15,18,21,24,27}; // declaring the contents of array
```

Accessing an element in the array requires square brackets, similar to other languages. Note that the array index starts with 0, meaning the first element is on index 0.

```
temp = intArray[3];      // get contents of fourth element of array
intArray[7] = 66;        // insert 66 into the eighth cell
```

Here is an example of array implementation, named *Array.java*. The program shows how an array can be used to store and display data.

Start Page | Array.java

Source | History

```
1   // Array.java
2   // Demonstrates how to create an array and stores and display array elements
3   package array;
4   import java.io.IOException;
5   import java.util.Scanner;
6
7   class Array {
8       public static void main(String[] args) throws IOException{
9           int size;                           // size of array that we want
10          Scanner in = new Scanner(System.in);// set up the input function
11          System.out.print("Enter the size of the array: ");
12          size = in.nextInt();                // insert the desired array size
13
14          int[] arr;                          // reference
15          arr = new int[size];                // make array
16
17          for(int j=0; j<size; j++){          // for each array space
18              System.out.print("Enter number:\t");
19              arr[j] = in.nextInt();          // insert a number
20          }
21
22          for(int j=0; j<size; j++)           // for each element in array
23              System.out.print(arr[j] + " "); // print the element
24          System.out.println("");
25      }  // end main()
26  }  // end class Array
27
```

The program takes the desired size of the array and the array elements as the input, and then displays the elements in one line as the output. For example, if the desired array size is 5, and the inserted array elements are 65, 76, 1, 900, and 55, then the result will look like below:

Output - Array (run)

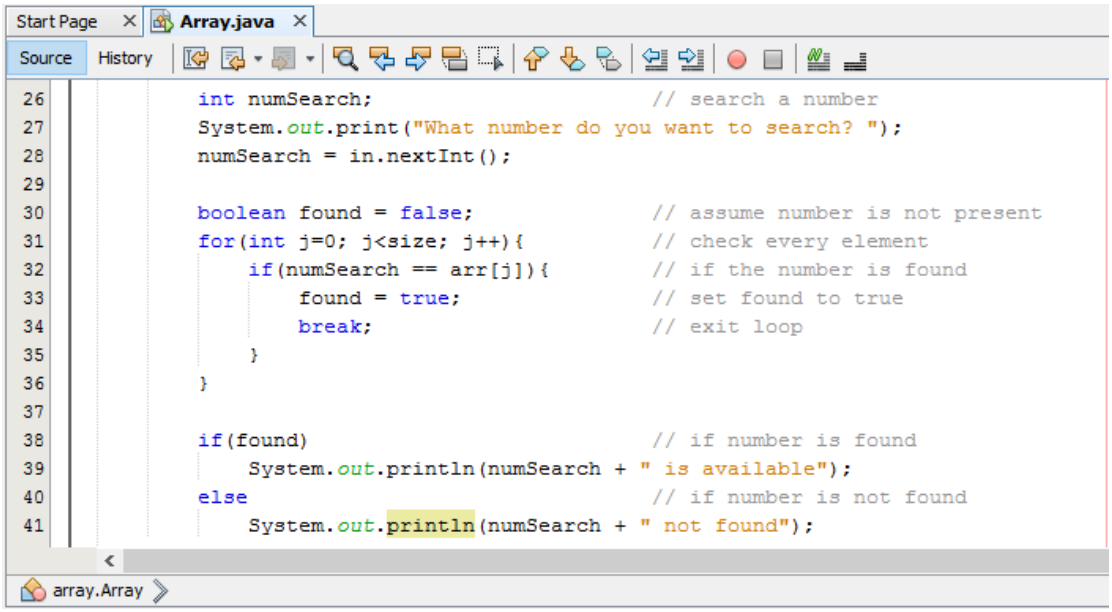
```
run:
Enter the size of the array: 5
Enter number:   65
Enter number:   76
Enter number:   1
Enter number:   900
Enter number:   55
65 76 1 900 55
BUILD SUCCESSFUL (total time: 17 seconds)
```

3.2.2 Array Processing

After the array is stored, we can then utilize the array for data processing. Some examples are search and replacement.

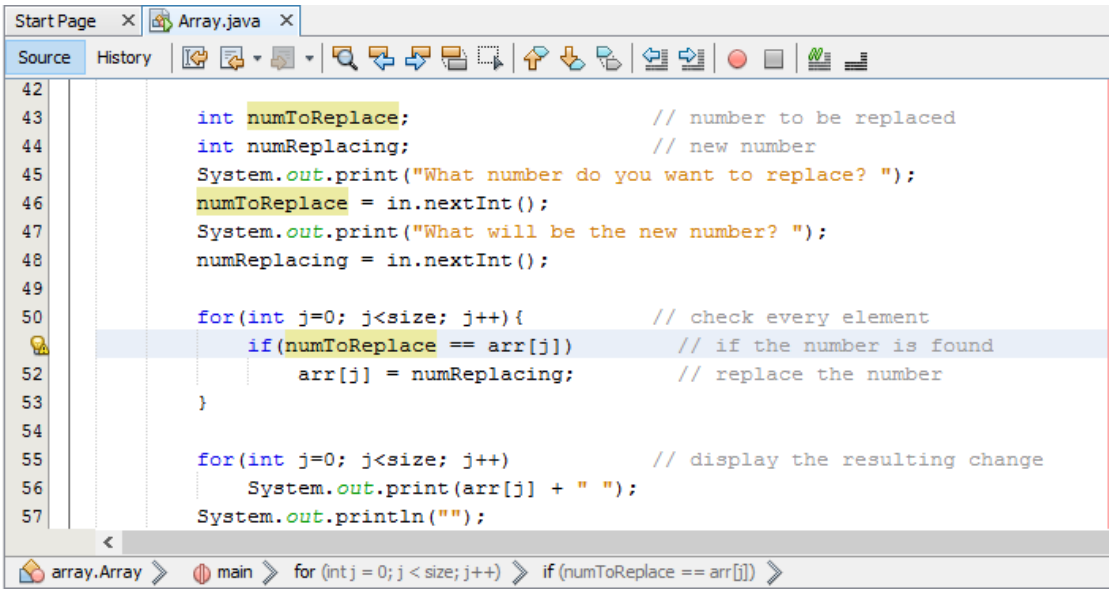
Data search means finding whether an element is stored in the array. For example, in an array containing the numbers [20, 21, 22, 24], searching for the number 23 would return the value FALSE, while searching for the number 24 would return TRUE.

The code for data search would look like this:



Meanwhile, data replacement means replacing a value in the array with another array. Case in point: the number 5 in an array [5, 10, 15, 20, 25, 30] can be replaced with the number 6, resulting in the array now containing [6, 10, 15, 20, 25, 30].

The code for data replacement would look like this:



The resulting total code becomes like this:

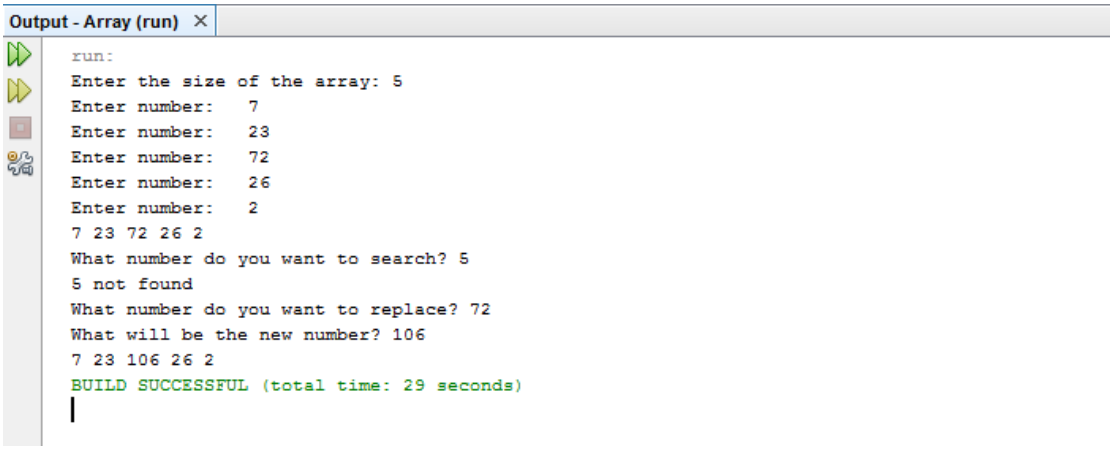
Start Page | Array.java

Source | History

```java
// Array.java
// Demonstrates how to create an array and stores and display array elements
package array;
import java.io.IOException;
import java.util.Scanner;

class Array {
    public static void main(String[] args) throws IOException{
        int size;                           // size of array that we want
        Scanner in = new Scanner(System.in);// set up the input function
        System.out.print("Enter the size of the array: ");
        size = in.nextInt();                // insert the desired array size

        int[] arr;                          // reference
        arr = new int[size];                // make array

        for(int j=0; j<size; j++){          // for each array space
            System.out.print("Enter number:\t");
            arr[j] = in.nextInt();          // insert a number
        }

        for(int j=0; j<size; j++)           // for each element in array
            System.out.print(arr[j] + " "); // print the element
        System.out.println("");

        int numSearch;                      // search a number
        System.out.print("What number do you want to search? ");
        numSearch = in.nextInt();

        boolean found = false;              // assume number is not present
        for(int j=0; j<size; j++){          // check every element
            if(numSearch == arr[j]){        // if the number is found
                found = true;               // set found to true
                break;                      // exit loop
            }
        }

        if(found)                           // if number is found
            System.out.println(numSearch + " is available");
        else                                // if number is not found
            System.out.println(numSearch + " not found");

        int numToReplace;                   // number to be replaced
        int numReplacing;                   // new number
        System.out.print("What number do you want to replace? ");
        numToReplace = in.nextInt();
        System.out.print("What will be the new number? ");
        numReplacing = in.nextInt();

        for(int j=0; j<size; j++){          // check every element
            if(numToReplace == arr[j])        // if the number is found
                arr[j] = numReplacing;        // replace the number
        }

        for(int j=0; j<size; j++)           // display the resulting change
            System.out.print(arr[j] + " ");
        System.out.println("");

    }  // end main()
}  // end class Array
```

For the

output, if we start with an array of size 5 containing [7, 23, 72, 26, 2], search for the number 5 in the array, and replace 72 with 106, it should look like this:



3.2.3 Linked List

Similar to array, linked list is another basic data structure for data storage and processing. However, the difference lies in the use of memory: whereas array stores its elements in a large memory block, linked list stores each element on separate memory spaces called nodes. These spaces are then connected to each other using pointers, hence the name “linked list”.

A linked list require two types of variables:

1. Node
   The main body of the list, every node contains the stored element and the pointer to the next node in the list
2. Pointer
   The head of the list, the pointer contains the memory address of the next node.

In Java, implementation of node and pointer can be done like this:

```
7   class Node{
8
9       public int iData;               // data item
10      public Node next;               // next link in list
11
12      public Node(int id){ // constructor
13          iData = id;                 // initialize data
14      }                               // ('next' is automatically set to null)
15
16      public void displayLink(){      // display ourself
17          System.out.print("{" + iData + "} ");
18      }
19  } // end class Node
```

After the node and pointer class is implemented, the linked list class is implemented:

```
21  class LinkListInit
22  {
23      private Node first;             // ref to first node on list
24
25      public LinkListInit()           // constructor
26      {
27          first = null;               // no items on list yet
28      }
29
30      public boolean isEmpty()        // true if list is empty
31      {
32          return (first==null);
33      }
34                                      // insert at start of list
35      public void insertFirst(int id)
36      {...5 lines }
41                                      // insert at end of list
42      public void insertLast(int id)
43      {...13 lines }
56
57      public Node deleteFirst()       // delete first item
58      {...6 lines }
64
65      public Node deleteLast()        // delete last node
66      {...11 lines }
77
78      public void displayList()
79      {...10 lines }
89  } // end class LinkList
```

You may notice that there are linked list operations present. These operations helps us to process and manipulate the linked list. The methods can be seen below:

1. *insertFirst()* and *insertLast()*

The *insertFirst()* and *insertLast()* methods of *LinkListInit* insert a new node at the beginning and the end of the list respectively.

```
35      public void insertFirst(int id)
36      {                               // make new node
37          Node newNode = new Node(id);
38          newNode.next = first;       // newNode --> old first
39          first = newNode;            // first --> newNode
40      }
41                                      // insert at end of list
42      public void insertLast(int id)
43      {
44          if(first == null)           // if the list is empty
45              insertFirst(id);        // create first node
46          else                        // if not
47          {
48              Node temp = first;      // start from first node
49              while(temp.next != null)// until we are at the last node
50              {
51                  temp = temp.next;   // keep going
52              }
53              temp.next = new Node(id); // add new node at the end
54          }
55      }
```

2. deleteFirst() and *deleteLast()*

The *deleteFirst()* and *deleteLast()* methods destroys the first and last node respectively. This is done by moving the first node reference forward for the former, and setting the second last node to point to null for the latter.

```
57      public Node deleteFirst()       // delete first item
58      {                               // (assume list not empty)
59          Node temp = first;          // save reference to node
60          first = first.next;         // delete it: first --> next node
61
62          return temp;                // return deleted node
63      }
64
65      public Node deleteLast()        // delete last node
66      {
67          Node temp = first;          // start from first node
68          while(temp.next.next != null)// till we are at the second last node
69          {
70              temp = temp.next;       // keep going
71          }
72          Node toReturn = temp.next;  // store the last node
73          temp.next = null;           // delete reference to last node
74
75          return toReturn;            // return deleted node
76      }
```

3. displayList()

This method displays the contents of the list, from first node to last node.

```
78      public void displayList()
79      {
80          System.out.print("List (first-->last): ");
81          Node current = first;       // start at beginning of list
82          while(current != null)      // until end of list
83          {
84              current.displayLink();  // print data
85              current = current.next; // move to next link
86          }
87          System.out.println("");
88       }
89  }  // end class LinkList
```

Here is the full implementation of *LinkList.java*.

Start Page | LinkList.java

Source | History

```
1   // LinkList.java
2   // demonstrates linklist
3   package linklist;
4   import java.io.IOException;
5   import java.util.Scanner;
6
7   class Node{
8
9       public int iData;               // data item
10      public Node next;               // next link in list
11
12      public Node(int id){ // constructor
13          iData = id;                 // initialize data
14      }                               // ('next' is automatically set to null)
15
16      public void displayLink(){      // display ourself
17          System.out.print("{" + iData + "} ");
18      }
19  } // end class Node
20
21  class LinkListInit
22  {
23      private Node first;             // ref to first node on list
24
25      public LinkListInit()           // constructor
26      {
27          first = null;               // no items on list yet
28      }
29
30      public boolean isEmpty()        // true if list is empty
31      {
32          return (first==null);
33      }
34                                      // insert at start of list
35      public void insertFirst(int id)
36      {                               // make new node
37          Node newNode = new Node(id);
38          newNode.next = first;       // newNode --> old first
39          first = newNode;            // first --> newNode
40      }
41                                      // insert at end of list
42      public void insertLast(int id)
43      {
44          if(first == null)           // if the list is empty
45              insertFirst(id);        // create first node
46          else                        // if not
47          {
48              Node temp = first;      // start from first node
49              while(temp.next != null)// until we are at the last node
50              {
51                  temp = temp.next;   // keep going
52              }
53              temp.next = new Node(id); // add new node at the end
54          }
55      }
56
57      public Node deleteFirst()       // delete first item
58      {                               // (assume list not empty)
59          Node temp = first;          // save reference to node
60          first = first.next;         // delete it: first --> next node
```

```
61
62          return temp;                // return deleted node
63      }
64
65      public Node deleteLast()        // delete last node
66      {
67          Node temp = first;          // start from first node
68          while(temp.next.next != null)// till we are at the second last node
69          {
70              temp = temp.next;       // keep going
71          }
72          Node toReturn = temp.next;  // store the last node
73          temp.next = null;           // delete reference to last node
74
75          return toReturn;            // return deleted node
76      }
77
78      public void displayList()
79      {
80          System.out.print("List (first-->last): ");
81          Node current = first;       // start at beginning of list
82          while(current != null)      // until end of list
83          {
84              current.displayLink();  // print data
85              current = current.next; // move to next link
86          }
87          System.out.println("");
88      }
89  }  // end class LinkList
90
91  class LinkList
92  {
93      public static void main(String[] args) throws IOException{
94          LinkListInit theList1 = new LinkListInit();  // make new list
95          LinkListInit theList2 = new LinkListInit();  // make new list
96          Scanner in = new Scanner(System.in); // set scanner
97          int nodeNum1;     // the number of integers for first list
98          int nodeNum2;     // the number of integers for second list
99          int tempNum;     // temporary holder for integer
100
101         System.out.print("First list size? ");
102         nodeNum1 = in.nextInt(); // insert first list size
103
104         for (int i=0; i<nodeNum1; i++){
105             System.out.print("Insert number: ");
106             tempNum = in.nextInt(); // insert number
107             theList1.insertLast(tempNum); // on the end of list
108         }
109         theList1.displayList();      // display list
110
111         System.out.print("Second list size? ");
112         nodeNum2 = in.nextInt(); // insert amount of integers
113
114         for (int i=0; i<nodeNum2; i++){
115             System.out.print("Insert number: ");
116             tempNum = in.nextInt(); // insert number
117             theList2.insertFirst(tempNum); // on the start of list
118         }
119         theList2.displayList();      // display list
120
```

```
121             System.out.println("\nDeleting first node of first list");
122             theList1.deleteFirst();      // deleting the first node of first list
123             theList1.displayList();      // first list after deletion
124
125             System.out.println("\nDeleting last node of second list");
126             theList2.deleteLast();       // deleting the last node of second list
127             theList2.displayList();      // second list after deletion
128
129         }  // end main()
130     }  // end class LinkList
131
```

In this program, two lists are declared; the first list has the integers inserted at the end, while the second list has the integers inserted at the beginning. The first node of the first list and the last node of the second list are then deleted.

An example of the program's output is such:

```
Output - LinkList (run)
run:
First list size? 3
Insert number: 32
Insert number: 65
Insert number: 876
List (first-->last): {32} {65} {876}
Second list size? 4
Insert number: 54
Insert number: 9
Insert number: 58
Insert number: 4
List (first-->last): {4} {58} {9} {54}

Deleting first node of first list
List (first-->last): {65} {876}

Deleting last node of second list
List (first-->last): {4} {58} {9}
BUILD SUCCESSFUL (total time: 17 seconds)
```

3.2.4 Built-in Linked List

Another implementation of linked list in Java is the use of the built-in LinkedList class. The class contains many operations, including the aforementioned ones, without having to build new classes.

Rewriting the program to use LinkedList class would result in this:

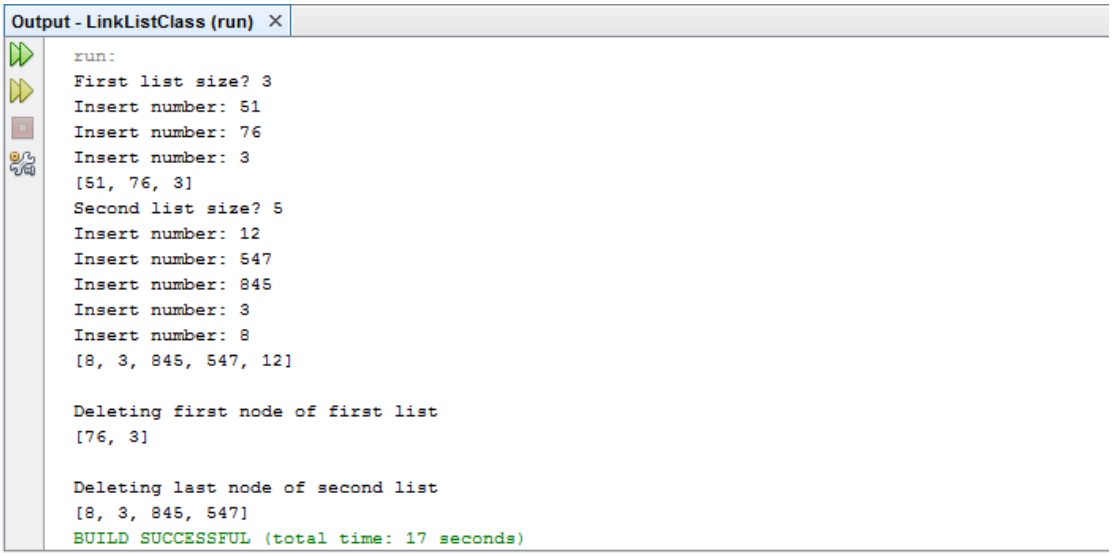
Start Page | LinkListClass.java

Source | History

```
1  //LinkListClass.java
2  //demonstrates the use of the built-in LinkedListClass
3  package linklistclass;
4  import java.io.IOException;
5  import java.util.LinkedList; // important for LinkedListClass
6  import java.util.Scanner;
7
8  class LinkListClass {
9      public static void main(String[] args) throws IOException {
10         LinkedList theList1 = new LinkedList();  // make new list
11         LinkedList theList2 = new LinkedList();  // make new list
12         Scanner in = new Scanner(System.in); // set scanner
13         int nodeNum1;     // the number of integers for first list
14         int nodeNum2;     // the number of integers for second list
15         int tempNum;     // temporary holder for integer
16
17         System.out.print("First list size? ");
18         nodeNum1 = in.nextInt(); // insert first list size
19
20         for (int i=0; i<nodeNum1; i++){
21             System.out.print("Insert number: ");
22             tempNum = in.nextInt(); // insert number
23             theList1.addLast(tempNum); // on the end of list
24         }
25         System.out.println(theList1);      // display list
26
27         System.out.print("Second list size? ");
28         nodeNum2 = in.nextInt(); // insert amount of integers
29
30         for (int i=0; i<nodeNum2; i++){
31             System.out.print("Insert number: ");
32             tempNum = in.nextInt(); // insert number
33             theList2.addFirst(tempNum); // on the start of list
34         }
35         System.out.println(theList2);      // display list
36
37         System.out.println("\nDeleting first node of first list");
38         theList1.removeFirst();      // deleting the first node of first list
39         System.out.println(theList1);      // first list after deletion
40
41         System.out.println("\nDeleting last node of second list");
42         theList2.removeLast();       // deleting the last node of second list
43         System.out.println(theList2);      // second list after deletion
44     }
45
46 }
47
```

linklistclass.LinkListClass > main >

The output would be like this:



3.3 Exercises

1. Using arrays, create a program that will sum all of the numbers inserted by the user. Note that the program will read the amount of numbers to be inserted, and the inserted numbers
2. Do the same as above question, but use linked lists
3. Create an array that stores 10 numbers, then triples each number
4. Write a delete() function for the first implementation of linked list that can delete any number on the list

CHAPTER 4 – STACK & QUEUE

4.1 LEARNING OBJECTIVES

1. Students can understand about stack and queue
2. Students are able to declare and use stacks and queues
3. Students can know when to use stacks and when to use queues

4.2 MATERIALS

4.2.1 STACK

A stack is a linear data structure that allows access to only one data item: the last item inserted. If you remove this item, then you can access the next-to-last item inserted, and so on. Think of stack like a pile of letters: you process each letter from the top, and you can only add a letter to the top of the pile. A stack is said to be a Last-In-First-Out (LIFO) storage mechanism, because the last item inserted is the first one to be removed.

There are four basic operations on stack:

1. Initialize stack

   Stack initialization means creating an empty stack, or a stack that has no elements in it

```
7   class StackInit                     // stack initiation template
8   {
9       private final int maxSize;            // size of stack array
10      private final int[] stackArray;
11      private int top;                // top of stack
12
13      public StackInit(int s)         // constructor
14      {
15          maxSize = s;                // set array size
16          stackArray = new int[maxSize];   // create array
17          top = -1;                   // no items yet
18      }
```

2. Push

   Push is the operation of adding an element to the top of the stack. The algorithm is as follows:

   a. Create a new data that will be pushed into the stack
   b. Push the data into the stack
   c. Modify the top pointer to point to the newly pushed data

```
20      public void push(int j)      // put item on top of stack
21      {
22          stackArray[++top] = j;      // increment top, insert item
23      }
```

3. Pop

   Pop is the operation of removing an element from the stack. Assuming the stack contains several elements, the algorithm is as follows:

   a. Take and remove the top data from the stack
   b. Modify the top pointer to point to the next top data

```
25      public double pop()             // take item from top of stack
26      {
27          return stackArray[top--];   // access item, decrement top
28      }
```

4. isEmpty

isEmpty is the operation of checking whether the stack contain elements. The operation will return true if the stack is empty, and will return false otherwise.

```
30      public boolean isEmpty()        // true if stack is empty
31      {
32          return (top == -1);
33      }
```

Stack can be implemented using two methods: array and linked list. Here is a full implementation example of stack using array, named *Stack.java*. The program stores the input numbers in a stack, then displays the numbers that have been stored.

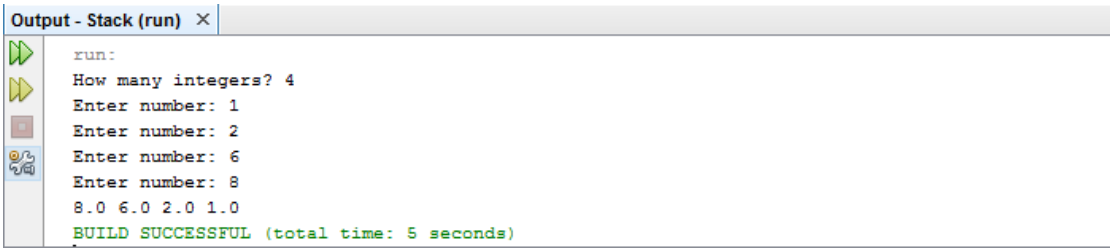
Start Page | Stack.java | LinkListClass.java

Source | History

```
1   // Stack.java
2   // demonstrates stacks
3   package stack;
4   import java.io.IOException;         // for I/O
5   import java.util.Scanner;
6
7   class StackInit                     // stack initiation template
8   {
9       private final int maxSize;            // size of stack array
10      private final int[] stackArray;
11      private int top;                // top of stack
12
13      public StackInit(int s)         // constructor
14      {
15          maxSize = s;                // set array size
16          stackArray = new int[maxSize];   // create array
17          top = -1;                   // no items yet
18      }
19
20      public void push(int j)       // put item on top of stack
21      {
22          stackArray[++top] = j;      // increment top, insert item
23      }
24
25      public double pop()             // take item from top of stack
26      {
27          return stackArray[top--];   // access item, decrement top
28      }
29
30      public boolean isEmpty()        // true if stack is empty
31      {
32          return (top == -1);
33      }
34
35  } // end class StackInit
36
37  class Stack
38  {
39      public static void main(String[] args) throws IOException
40      {
41          int stackSize;  // stack size
42          int stackNum;   // number to be inserted in stack
43          Scanner in = new Scanner(System.in);
44
45          System.out.print("How many integers? ");
46          stackSize = in.nextInt();           // insert stack size
47
48          StackInit theStack = new StackInit(stackSize);// make new stack
49
50          for(int i=0; i<stackSize; i++){
51              System.out.print("Enter number: ");
52              stackNum = in.nextInt();        // insert number
53              theStack.push(stackNum);        // push element onto stack
54          }
55
56          while(!theStack.isEmpty())          // until it is empty,
57          {                                   // delete item from stack
58              double value = theStack.pop();
59              System.out.print(value);        // display the popped item
60              System.out.print(" ");
61          }
62
63          System.out.println("");
64      }  // end main()
65
66  }  // end class Stack
```

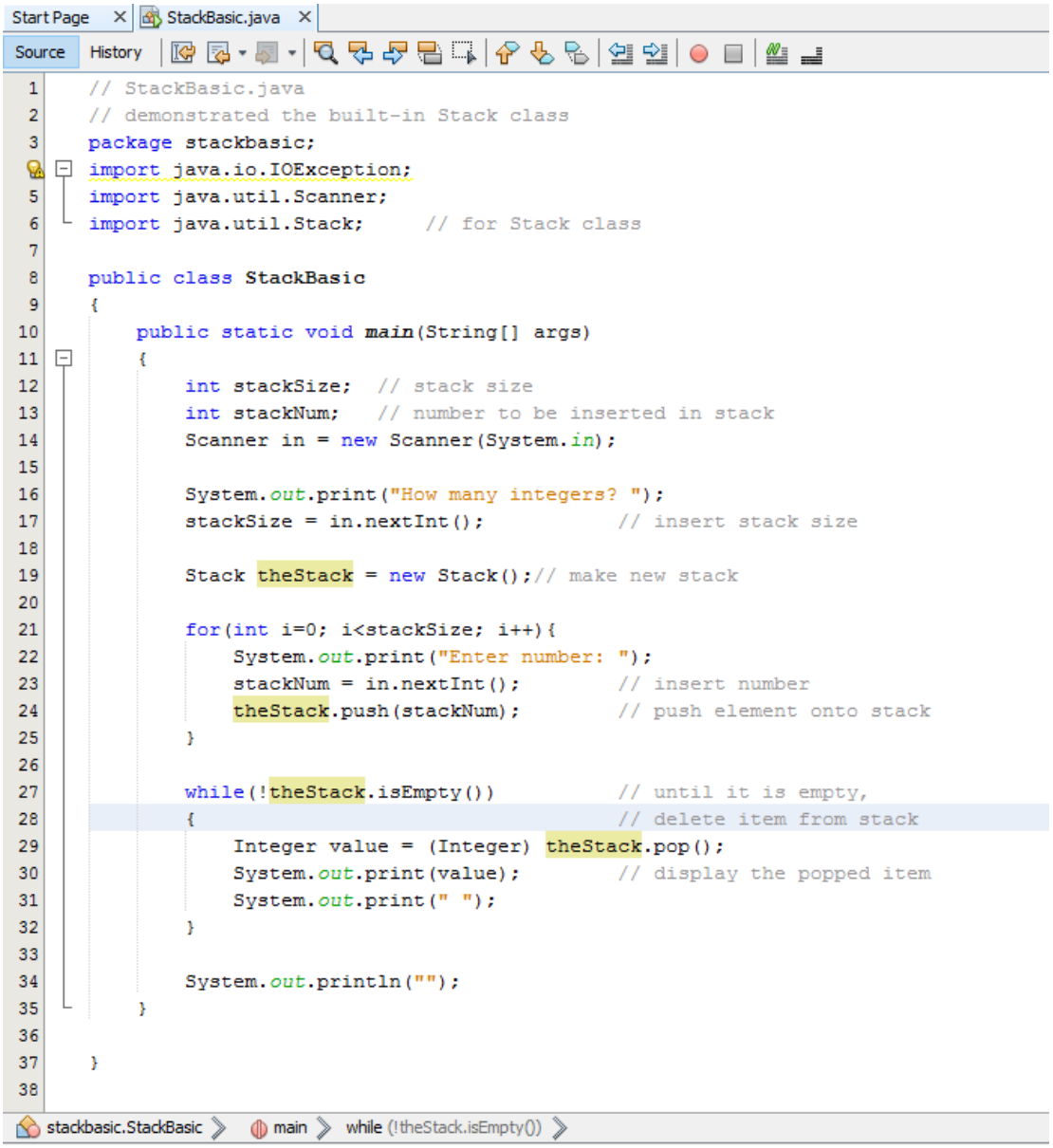
stack.Stack > main >

This is an example of the output of the program.



4.2.2 Stack Implementation

Java already has a built-in Stack class, imported through Java.util.Stack. Here is the example of the program similar to the previous one but using the Stack class.



Here is the output example.

4.2.3 Queue

A queue is a linear data structure similar to stack, but the addition and deletion of data item is done on opposite ends. Items can be inserted on one end, and removed from the other end. Think of queue like supermarket queues: you enter from one end of the queue and exit from the other end. A queue is said to be a First-In-First-Out (FIFO) storage mechanism, because the first item inserted is the first one to be removed.

There are five basic operations on queue:

1. Initialize queue

   Queue initialization means creating an empty queue, or a queue that has no elements in it

```
15      public QueueInit(int s)             // constructor
16      {
17          maxSize = s;
18          queArray = new int[maxSize];
19          front = 0;
20          rear = -1;
21          nItems = 0;
22      }
```

2. Enqueue

   Enqueue is the operation of adding a new element in the rear end of the queue. The algorithm is as follows:

   a. Create a new data that will be inserted into the queue
   b. Insert the new data into the queue
   c. Modify the last pointer to point to the newly inserted data

```
24      public void enqueue(int j)       // put item at rear of queue
25      {
26          if(rear == maxSize-1)        // deal with wraparound
27              rear = -1;
28
29          queArray[++rear] = j;        // increment rear and insert
30          nItems++;                    // one more item
31      }
```

3. Dequeue

   Dequeue is the operation of removing an element from the front end of the queue. Assuming the queue contains several elements, the algorithm is as follows:

   a. Take and remove the frontmost data from the queue
   b. Modify the first pointer to point to the next front data

```
33      public int dequeue()              // take item from front of queue
34      {
35          int temp = queArray[front++];   // get value and incr front
36          if(front == maxSize)            // deal with wraparound
37              front = 0;
38          nItems--;                       // one less item
39          return temp;
40      }
```

4. isEmpty

   isEmpty is the operation of checking whether the queue contain elements. The operation will return true if the queue is empty, and will return false otherwise.

```
42      public boolean isEmpty()        // true if queue is empty
43      {
44          return (nItems==0);
45      }
```

5. isFull

   The opposite of isEmpty, the operation will return true if the queue is empty, and will return false otherwise.

```
47      public boolean isFull()         // true if queue is full
48      {
49          return (nItems==maxSize);
50      }
```

Queue can be implemented in two ways: array and linked list. Here is a full example of queue implementation using array, named *Queue.java*. The program implements a queue using a class *QueueInit* and operated based on our commands to create a queue accordingly.

Start Page | Queue.java

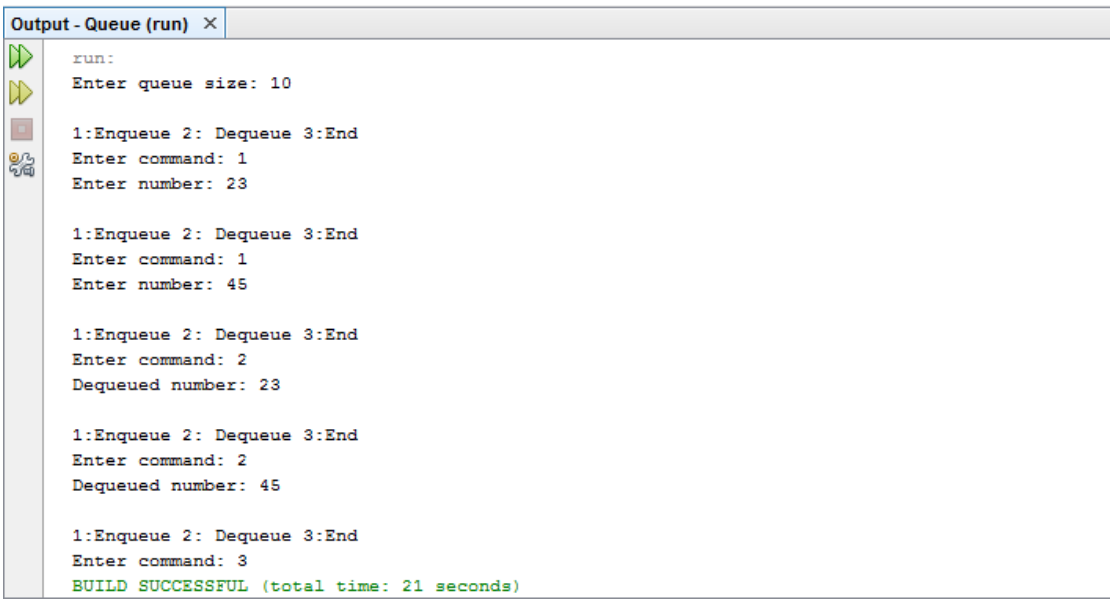
Source | History

```
1   // Queue.java
2   // Demonstrates queues
3   package queue;
4   import java.io.IOException;                 // for I/O
5   import java.util.Scanner;
6
7   class QueueInit
8   {
9       private int maxSize;
10      private int[] queArray;
11      private int front;
12      private int rear;
13      private int nItems;
14
15      public QueueInit(int s)              // constructor
16      {
17          maxSize = s;
18          queArray = new int[maxSize];
19          front = 0;
20          rear = -1;
21          nItems = 0;
22      }
23
24      public void enqueue(int j)        // put item at rear of queue
25      {
26          if(rear == maxSize-1)         // deal with wraparound
27              rear = -1;
28
29          queArray[++rear] = j;         // increment rear and insert
30          nItems++;                     // one more item
31      }
32
33      public int dequeue()              // take item from front of queue
34      {
35          int temp = queArray[front++];    // get value and incr front
36          if(front == maxSize)             // deal with wraparound
37              front = 0;
38          nItems--;                        // one less item
39          return temp;
40      }
41
42      public boolean isEmpty()          // true if queue is empty
43      {
44          return (nItems==0);
45      }
46
47      public boolean isFull()           // true if queue is full
48      {
49          return (nItems==maxSize);
50      }
51
52      public int size()                 // number of items in queue
53      {
54          return nItems;
55      }
56  }  // end class QueueInit
57
```

```
58  class Queue
59  {
60      public static void main(String[] args) throws IOException
61      {
62          int queueSize;                          // for queue size
63          int numTemp;                          // for inserted number
64          int numChoice = 0;                        // for command
65          Scanner in = new Scanner(System.in);    // for input
66
67          System.out.print("Enter queue size: ");
68          queueSize = in.nextInt();
69
70          QueueInit theQueue = new QueueInit(queueSize);// set queue
71
72          while(numChoice != 3)
73          {
74              System.out.println("\n1:Enqueue 2: Dequeue 3:End");
75              System.out.print("Enter command: ");
76              numChoice = in.nextInt();
77              if(numChoice == 1){
78                  if(theQueue.isFull())
79                      System.out.println("Queue is full");
80                  else
81                  {
82                      System.out.print("Enter number: ");
83                      numTemp = in.nextInt();
84                      theQueue.enqueue(numTemp);
85                  }
86              }
87              else if(numChoice == 2){
88                  if (theQueue.isEmpty())
89                      System.out.println("Queue is empty");
90                  else{
91                      numTemp = theQueue.dequeue();
92                      System.out.println("Dequeued number: " + numTemp);
93                  }
94              }
95              else if (numChoice != 3){
96                  System.out.println("Wrong command");
97              }
98          }
99      }  // end main()
100 }  // end class Queue
101
```

queue.Queue > main > while (numChoice != 3) > if (numChoice == 1) > if (theQueue.isFull()) else >

The output may look like this:



4.2.4 Queue Implementation

Queue are used normally in situations where elements may come one by one, and processed one by one, thus requiring the use of a queue. One example may be the banking queue, where people wait in a line to get a banking service. Here is an example of such implementation, called *BankService.java*. This program allows you to add customers to a queue, remove customers from the queue, and check the current situation of the queue. Note that this program used LinkedList class to serve as the queue.

Start Page | BankService.java

Source | History

```
1   // BankService.java
2   // demonstrates an application of queue using LinkedList
3   package bankservice;
4   import java.util.Scanner;           // for input
5   import java.util.LinkedList;        // for linked list
6   import java.io.IOException;         // for I/O
7   import java.util.Iterator;          // for list iteration
8
9   class Customer{                     // customer object constructor
        private String name;            // customer name
        private double idCust, balance; // customer id and balance
12
13          // create customer
14      public Customer(String _name, double _idCust, double _balance){
15          name = _name;
16          idCust = _idCust;
17          balance = _balance;
18      }
19
20      public String getName(){        // return customer name
21          return name;
22      }
23
24      public double getIdCust(){      // return customer id
25          return idCust;
26      }
27
28      public double getBalance(){     // return customer balance
29          return balance;
30      }
31  } // end class Customer
32
33  class BankServiceStart{             // queue constructor
        private LinkedList<Customer> list;  // declare linked list for queue
35
36      public BankServiceStart(){
37          list = new LinkedList<>();      // create linked list
38      }
39
40      public boolean isEmpty(){           // true if queue is empty
41          return(list.isEmpty());
42      }
43
44      public void enqueue(Customer item){ // add customer to queue
45          list.addLast(item);
46      }
47
48      public void dequeue(){              // remove customer from queue
49          list.removeFirst();
50      }
51
52      public Customer callCust(){         // get front customer name
53          return list.getFirst();
54      }
55
```

```
56      public void displayQueue(){         // display the current queue
57          Iterator<Customer> i = list.iterator(); // for iterating
58          if(!i.hasNext())                        // if empty
59              System.out.println("Queue is currently empty");
60          else {                                  // if not empty
61              while(i.hasNext()){                 // until the last customer
62                  Customer tempCust = i.next();
63                  System.out.println("Name: " + tempCust.getName()
64                                     + "\nCustomer ID: " + tempCust.getIdCust()
65                                     + "\nBalance: " + tempCust.getBalance());
66                  // print customer info
67              }
68          }
69      }
70  } // end class BankServiceStart
71
72  class BankService {
73
74      public static void main(String[] args) throws IOException{
75          String name_m;              // for getting customer name
76          double idCust_m = 0;        // for getting customer id
77          double balance_m;           // for getting customer balance
78          int queueCom = 0;           // for queue command
79
80          Scanner in = new Scanner(System.in);    // for input
81
82          BankServiceStart bankQueue = new BankServiceStart();    // create queue
83
84          // creating a bank system
85          System.out.println("Welcome to Rajasa Bank Queueing System!");
86
87          while(queueCom != 4){       // while we are not done
88              System.out.println("What would you like to do?"); // ask command
89              System.out.println("1:Check queue, "
90                      + "2:Add customer to queue, "
91                      + "3:Call customer from queue, "
92                      + "4:Finish");
93              queueCom = in.nextInt();    //enter command
94
95              switch(queueCom){
96                  case 1:      // if we want to check queue
97                      bankQueue.displayQueue();   // show queue
98                      break;
99                  case 2: // if we want to enter customer
100                     System.out.print("Customer name: ");
101                     name_m = in.next();         // enter customer name
102                     idCust_m++;                 // automatic id
103                     System.out.print("Balance: ");
104                     balance_m = in.nextDouble();// enter customer balance
105
106                     Customer newCust = new Customer(name_m,
107                             idCust_m, balance_m);// create customer object
108                     bankQueue.enqueue(newCust);      // insert to queue
109                     System.out.println(name_m
110                             + " has been added to queue!");
111                     break;
112                 case 3: // if we want to call customer
113                     if(bankQueue.isEmpty()) // if empty
```

```
114                         System.out.println("Queue is empty");
115                     else{                   // if not empty
116                         Customer toServe = bankQueue.callCust();
117                         System.out.println("Calling " + toServe.getName());
118                         bankQueue.dequeue();
119                     }
120                     break;
121                 case 4: // when we are done
122                     System.out.println("Thank you for using the system!");
123                     break;
124                 default:
125                     System.out.println("Incorrect command. Try again.");
126             }
127         }
128
129     } // end main()
130 }     // end class BankService
131
```

bankservice.Customer

This is the example of the output.

```
Output - BankService (run)
run:
Welcome to Rajasa Bank Queueing System!
What would you like to do?
1:Check queue, 2:Add customer to queue, 3:Call customer from queue, 4:Finish
1
Queue is currently empty
What would you like to do?
1:Check queue, 2:Add customer to queue, 3:Call customer from queue, 4:Finish
2
Customer name: Jenny
Balance: 20000
Jenny has been added to queue!
What would you like to do?
1:Check queue, 2:Add customer to queue, 3:Call customer from queue, 4:Finish
2
Customer name: Henry
Balance: 50000000
Henry has been added to queue!
What would you like to do?
1:Check queue, 2:Add customer to queue, 3:Call customer from queue, 4:Finish
1
Name: Jenny
Customer ID: 1.0
Balance: 20000.0
Name: Henry
Customer ID: 2.0
```

```
Balance: 5.0E7
What would you like to do?
1:Check queue, 2:Add customer to queue, 3:Call customer from queue, 4:Finish
3
Calling Jenny
What would you like to do?
1:Check queue, 2:Add customer to queue, 3:Call customer from queue, 4:Finish
1
Name: Henry
Customer ID: 2.0
Balance: 5.0E7
What would you like to do?
1:Check queue, 2:Add customer to queue, 3:Call customer from queue, 4:Finish
4
Thank you for using the system!
BUILD SUCCESSFUL (total time: 53 seconds)
```

4.3 EXERCISES

1. Create a program to reverse the order of a string of characters entered by the user (eg. REVELATION -> NOITALEVER)
2. Write a program that checks whether a string is palindrome (palindrome is a string that reads the same forward and backward, eg TACOCAT)
3. Draw a sequence diagram showing the contents of the queue q, after each command line is done
   q.enqueue(10);
   q.enqueue(20);
   q.dequeue();
   q.enqueue(30);
4. Create a program that calculates the parking fee that each car must pay in a parking lot. Each car has the information of the car model and its parking time. Assume the rate is Rp. 2000 per hour.

4.4 HOMEWORK

Create a simulation of queue on Banking Service.

CHAPTER 5 – ADVANCED SORTING

5.1 OBJECTIVES

In this laboratory you will:

1. Create an implementation of merge-sort
2. Compare merge-sort with other slow sorting method

5.2 MATERIALS

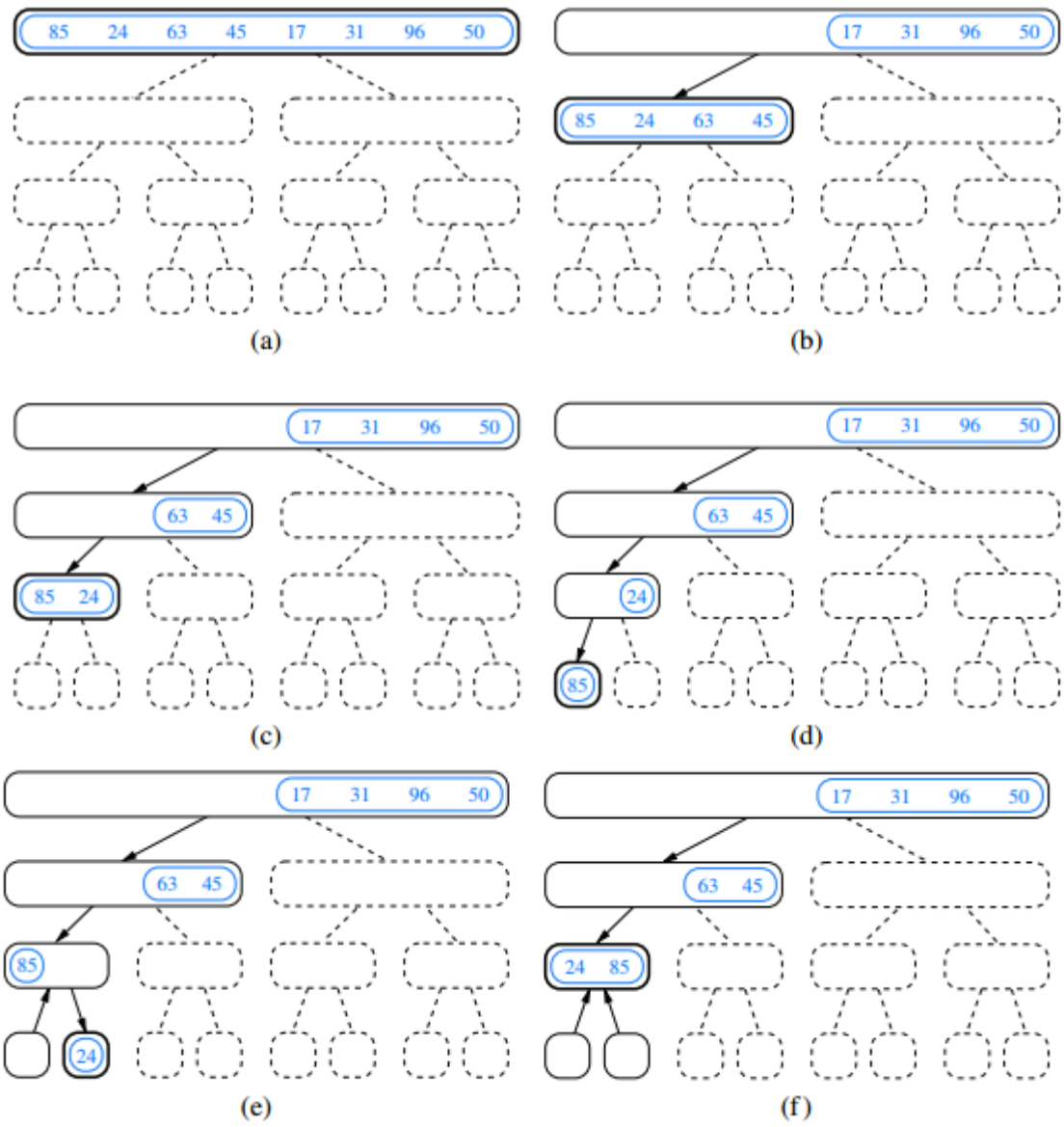
5.2.1 MERGE-SORT OVERVIEW

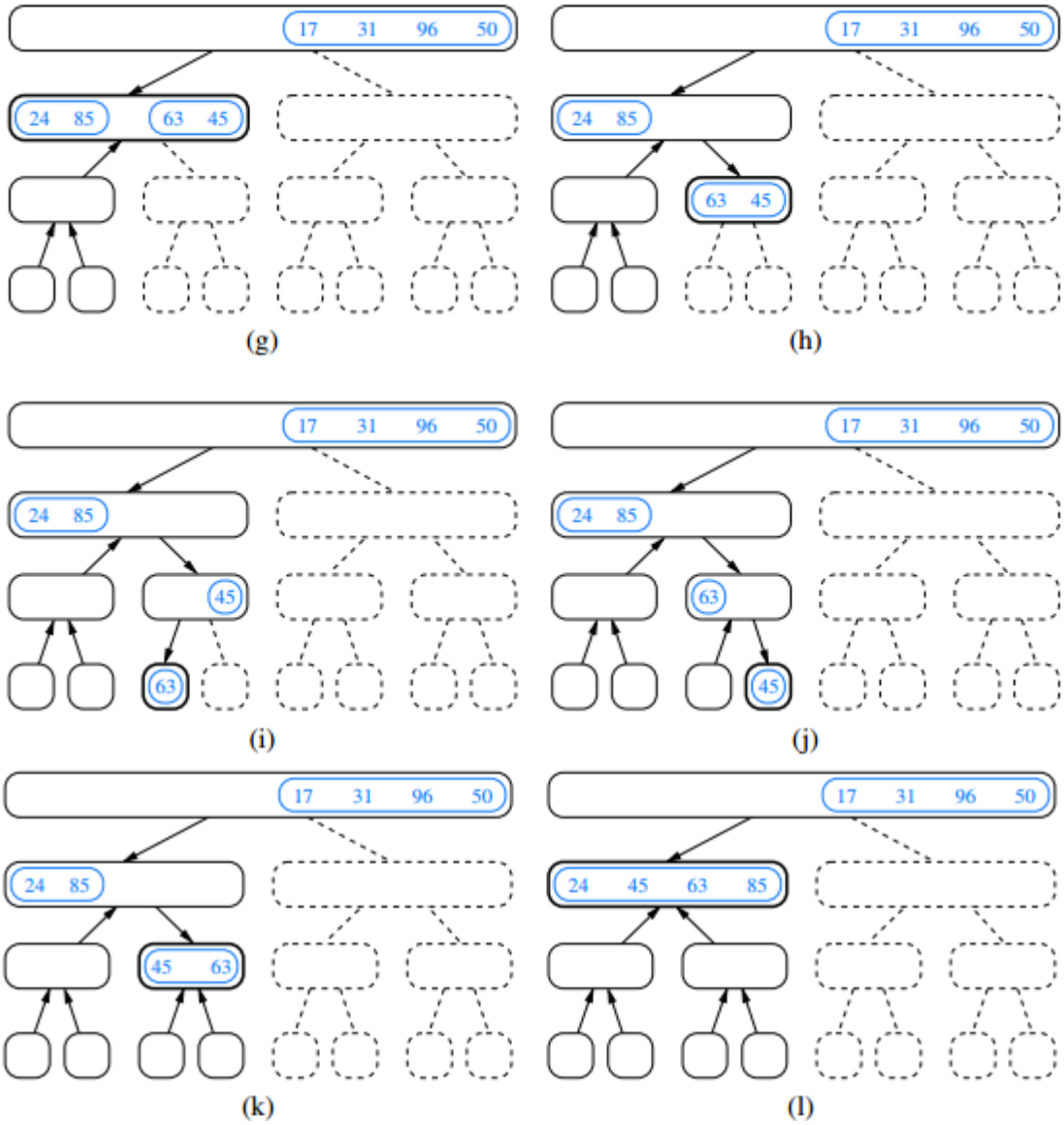
We first describe the merge-sort algorithm at a high level, without focusing on whether the data is an array or linked list. (We will soon give concrete implementations for each.) To sort a sequence S with n elements using the three divide-and-conquer steps, the merge-sort algorithm proceeds as follows:

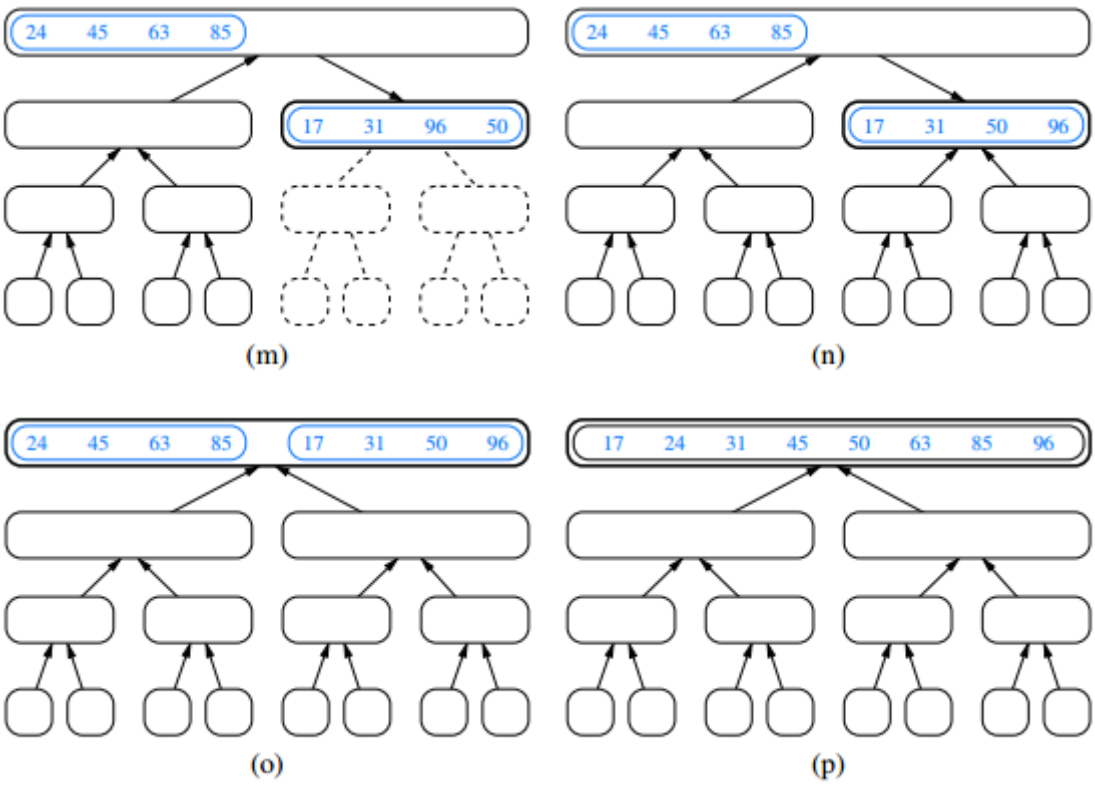
1. Divide: If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S1 and S2, each containing about half of the elements of S; that is, S1 contains the first $\lfloor n/2 \rfloor$ elements of S, and S2 contains the remaining $\lceil n/2 \rceil$ elements.
2. Conquer: Recursively sort sequences S1 and S2.
3. Combine: Put the elements back into S by merging the sorted sequences S1 and S2 into a sorted sequence.

In reference to the divide step, we recall that the notation $\lfloor x \rfloor$ indicates the floor of x, that is, the largest integer k, such that $k \leq x$. Similarly, the notation $\lceil x \rceil$ indicates the ceiling of x, that is, the smallest integer m, such that $x \leq m$.

Figure below show how merge-sort work.


85 24 63 45 17 31 96 50
(a)
17 31 96 50
85 24 63 45
(b)
17 31 96 50
63 45
85 24
(c)
17 31 96 50
63 45
24
85
(d)
17 31 96 50
63 45
85
24
(e)
17 31 96 50
63 45
24 85
(f)


17 31 96 50
24 85 63 45
(g)
17 31 96 50
24 85
63 45
(h)
17 31 96 50
24 85
45
63
(i)
17 31 96 50
24 85
63
45
(j)
17 31 96 50
24 85
45 63
(k)
17 31 96 50
24 45 63 85
(l)



MERGE-SORT IMPLEMENTATION USING QUEUE

Code below is an implementation of merge-sort using queue.

```
1  import java.util.ArrayDeque;
2  import java.util.Comparator;
3  import java.util.Queue;
4  public class Algorithm {
5      public static <K> void merge(Queue<K> S1, Queue<K> S2, Queue<K> S, Comparator<K> comp) {
6          while (!S1.isEmpty() && !S2.isEmpty()) {
7              if (comp.compare(S1.peek(), S2.peek()) < 0)
8                  S.add(S1.remove());
9              else
10                 S.add(S2.remove());
11         }
12         while (!S1.isEmpty())
13             S.add(S1.remove());
14         while (!S2.isEmpty())
15             S.add(S2.remove());
16     }
17     public static <K> void mergeSort(Queue<K> S, Comparator<K> comp) {
18         int n = S.size();
19         if (n < 2) return;
20         Queue<K> S1 = new ArrayDeque<>();
21         Queue<K> S2 = new ArrayDeque<>();
22         while (S1.size() < n/2)
23             S1.add(S.remove());
24         while (!S.isEmpty())
25             S2.add(S.remove());
26         mergeSort(S1, comp);
27         mergeSort(S2, comp);
28         merge(S1, S2, S, comp);
29     }
30 }
```

```java
package merge_sort;

import java.util.ArrayDeque;
import java.util.Comparator;
import java.util.Queue;

public class Merge_Sort {
    public static void main(String[] args) {
        Queue<Integer> myQueue = new ArrayDeque<>();

        myQueue.add(5);
        myQueue.add(1);
        myQueue.add(7);
        myQueue.add(100);
        myQueue.add(90);

        System.out.println("Before sorting =");

        myQueue.forEach((t) -> {
            System.out.print(t.toString() + " ");
        });System.out.println();

        Comparator<Integer> comp = new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                if(o1 < o2)
                    return -1;
                else
                    return 1;
            }
```

```
            myQueue.add(90);
16
17          System.out.println("Before sorting =");
18
19          myQueue.forEach((t) -> {
20              System.out.print(t.toString() + " ");
21          });System.out.println();
22
23          Comparator<Integer> comp = new Comparator<Integer>() {
24              @Override
25              public int compare(Integer o1, Integer o2) {
26                  if(o1 < o2)
27                      return -1;
28                  else
29                      return 1;
30              }
31          };
32
33          Algorithm.mergeSort(myQueue, comp);
34
35          System.out.println("After sorting =");
36
37          myQueue.forEach((t) -> {
38              System.out.print(t.toString() + " ");
39          });System.out.println();
40      }
41
42  }
```

```java
package merge_sort;

import java.util.ArrayDeque;
import java.util.Comparator;
import java.util.Queue;

public class Merge_Sort {
    public static void main(String[] args) {
        Queue<Integer> myQueue = new ArrayDeque<>();

        myQueue.add(5);
        myQueue.add(1);
        myQueue.add(7);
        myQueue.add(100);
        myQueue.add(90);

        System.out.println("Before sorting =");

        myQueue.forEach((t) -> {
            System.out.print(t.toString() + " ");
        });System.out.println();

        Comparator<Integer> comp = new Comparator<Integer>() {
```

```
Output - Merge_Sort (run)
run:
Before sorting =
5 1 7 100 90
After sorting =
1 5 7 90 100
BUILD SUCCESSFUL (total time: 0 seconds)
```

5.3 EXERCISES

1. Implement merge-sort using queue as shown in figure.

CHAPTER 6 – TREES

6.1 LEARNING OBJECTIVES

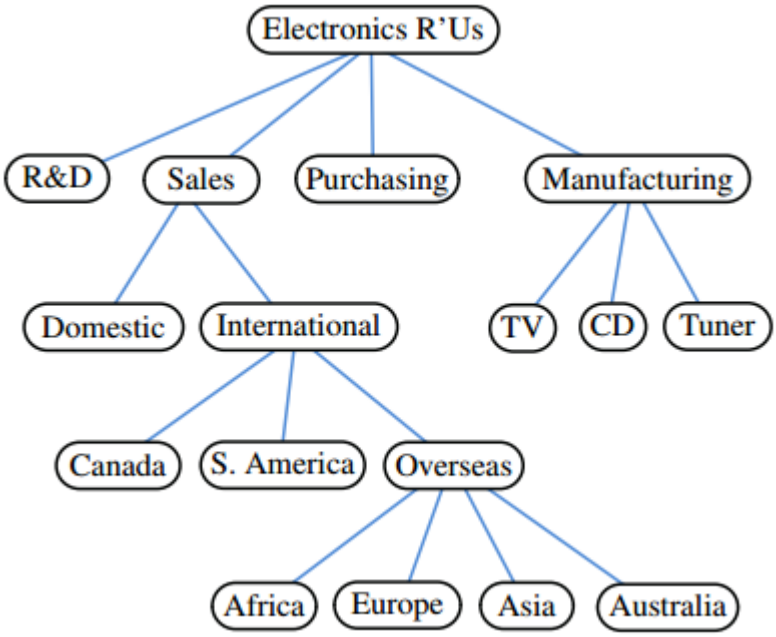
In this laboratory you:

1. Create an implementation of the Binary Tree.
2. Examine how an index can be used to retrieve records from a tree.

6.2 MATERIALS

6.2.1 TREE OVERVIEW

A tree is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements. A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines. (See Figure Below) We typically call the top element the root of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).



Formally, we define a tree T as a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following properties:

- If T is nonempty, it has a special node, called the root of T , that has no parent.
- Each node v of T different from the root has a unique parent node w; every node with parent w is a child of w.

Note that according to our definition, a tree can be empty, meaning that it does not have any nodes. This convention also allows us to define a tree recursively such that a tree T is either empty or consists of a node r, called the root of T , and a (possibly empty) set of subtrees whose roots are the children of r.

6.2.2 BINARY TREE

A binary tree is an ordered tree with the following properties:

- Every node has at most two children.
- Each child node is labeled as being either a left child or a right child.
- A left child precedes a right child in the order of children of a node.

The subtree rooted at a left or right child of an internal node v is called a left subtree or right subtree, respectively, of v. A binary tree is proper if each node has either zero or two children. Some people also refer to such trees as being full binary trees. Thus, in a proper binary tree, every internal node has exactly two children. A binary tree that is not proper is improper.

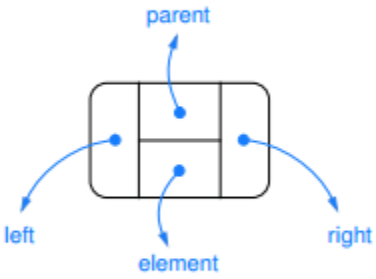
6.2.3 BINARY TREE ABSTRACT DATA TYPE

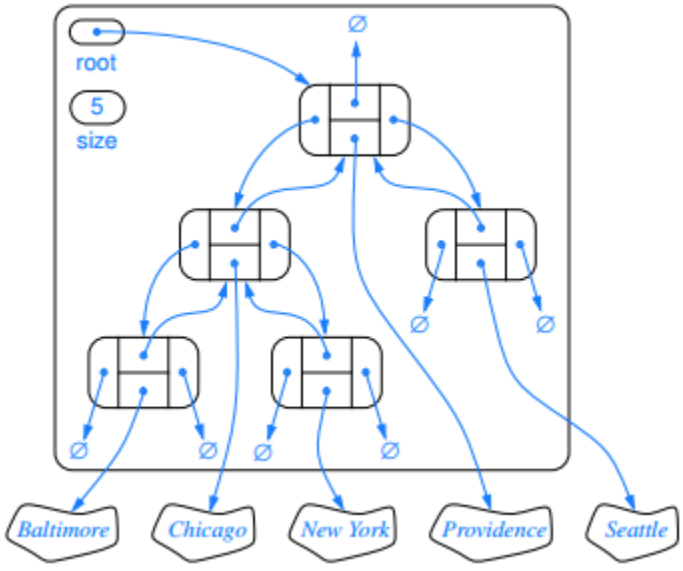
As an abstract data type, a binary tree is a specialization of a tree that supports three additional accessor methods:

- left(p): Returns the position of the left child of p (or null if p has no left child).
- right(p): Returns the position of the right child of p (or null if p has no right child).
- sibling(p): Returns the position of the sibling of p (or null if p has no sibling).

6.2.4 LINKED STRUCTURE FOR BINARY TREE

A natural way to realize a binary tree T is to use a linked structure, with a node (see Figure Below) that maintains references to the element stored at a position p and to the nodes associated with the children and parent of p. If p is the root of T , then the parent field of p is null. Likewise, if p does not have a left child (respectively, right child), the associated field is null. The tree itself maintains an instance variable storing a reference to the root node (if any), and a variable, called size, that represents the overall number of nodes of T .

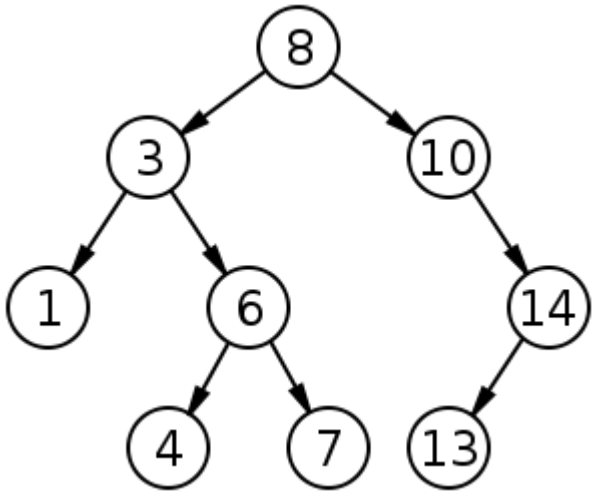




6.2.5 BINARY SEARCH TREE

Binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of containers: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast

lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

In the case of a linked binary search tree, we suggest that the following update methods be supported:

- addNode(node): Add a node to a binary search tree. If the tree is empty, then the node become the root.
- insertNode(): Insert the new node to the existing binary search tree.
- searchValue(root, value): a static method to check if a given value exist on a given binary search tree.

```
2   public class Node {
3           private int value;
4           public Node leftChild;
5           public Node rightChild;
6
7           Node(int value){
8               this.value = value;
9           }
10
11          public int getValue() {
12              return value;
13          }
14  }
15
```

```
2   public class BinaryTree {
3           public Node root;
4
5           public void addNode(Node node){
6                   if(root == null){
7                           root = node;
8                   }
9                   else{
10                          insertNode(root, node);
11                  }
12          }
13
14          private void insertNode(Node parent, Node node){
15                  if(parent.getValue() > node.getValue()){
16                          if(parent.leftChild == null){
17                                  parent.leftChild = node;
18                          }
19                          else{
20                                  insertNode(parent.leftChild, node);
21                          }
22                  }
23                  else{
24                          if(parent.rightChild == null){
25                                  parent.rightChild = node;
26                          }
27                          else{
28                                  insertNode(parent.rightChild, node);
29                          }
30                  }
31          }
```

```
33          public static boolean searchValue(Node root, int value){
34                  if(root == null){
35                          return false;
36                  }
37                  else{
38                          if(root.getValue() == value){
39                                  return true;
40                          }
41                          else if(root.getValue() > value){
42                                  return searchValue(root.leftChild, value);
43                          }
44                          else{
45                                  return searchValue(root.rightChild, value);
46                          }
47                  }
48          }
49  }
```

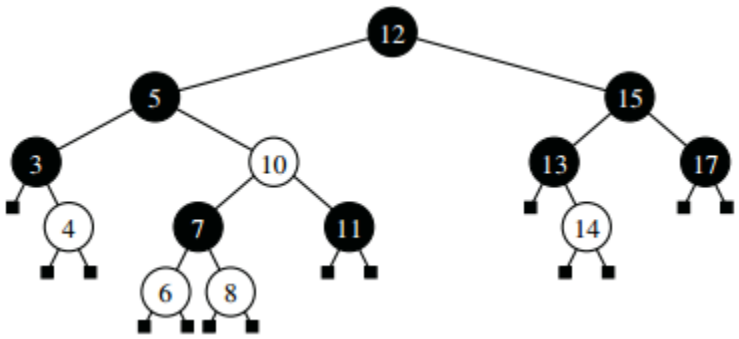
```
2   public class Main {
3
4       public static void main(String[] args) {
5           // TODO Auto-generated method stub
6           BinaryTree bt = new BinaryTree();
7
8           bt.addNode(new Node(5));
9           bt.addNode(new Node(4));
10          bt.addNode(new Node(6));
11          bt.addNode(new Node(7));
12          bt.addNode(new Node(3));
13
14          System.out.println(BinaryTree.searchValue(bt.root, 3));
15      }
16
17  }
18
```

6.2.6 RED-BLACK TREE

Formally, a red-black tree is a binary search tree with nodes colored red and black in a way that satisfies the following properties:

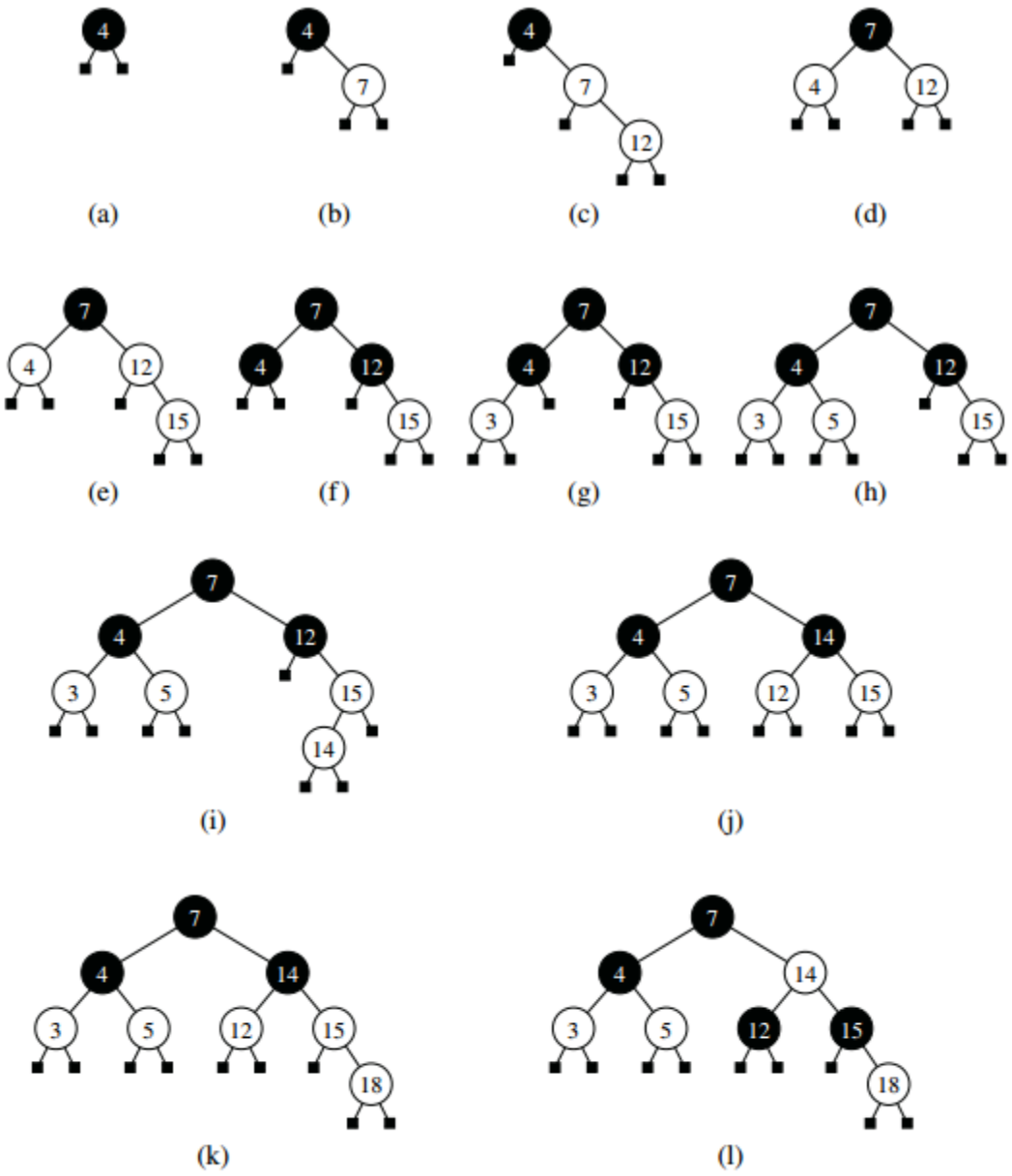
- Root Property: The root is black.
- External Property: Every external node is black.
- Red Property: The children of a red node are black.
- Depth Property: All external nodes have the same black depth, defined as the number of proper ancestors that are black.

An example of a red-black tree is shown in Figure Below



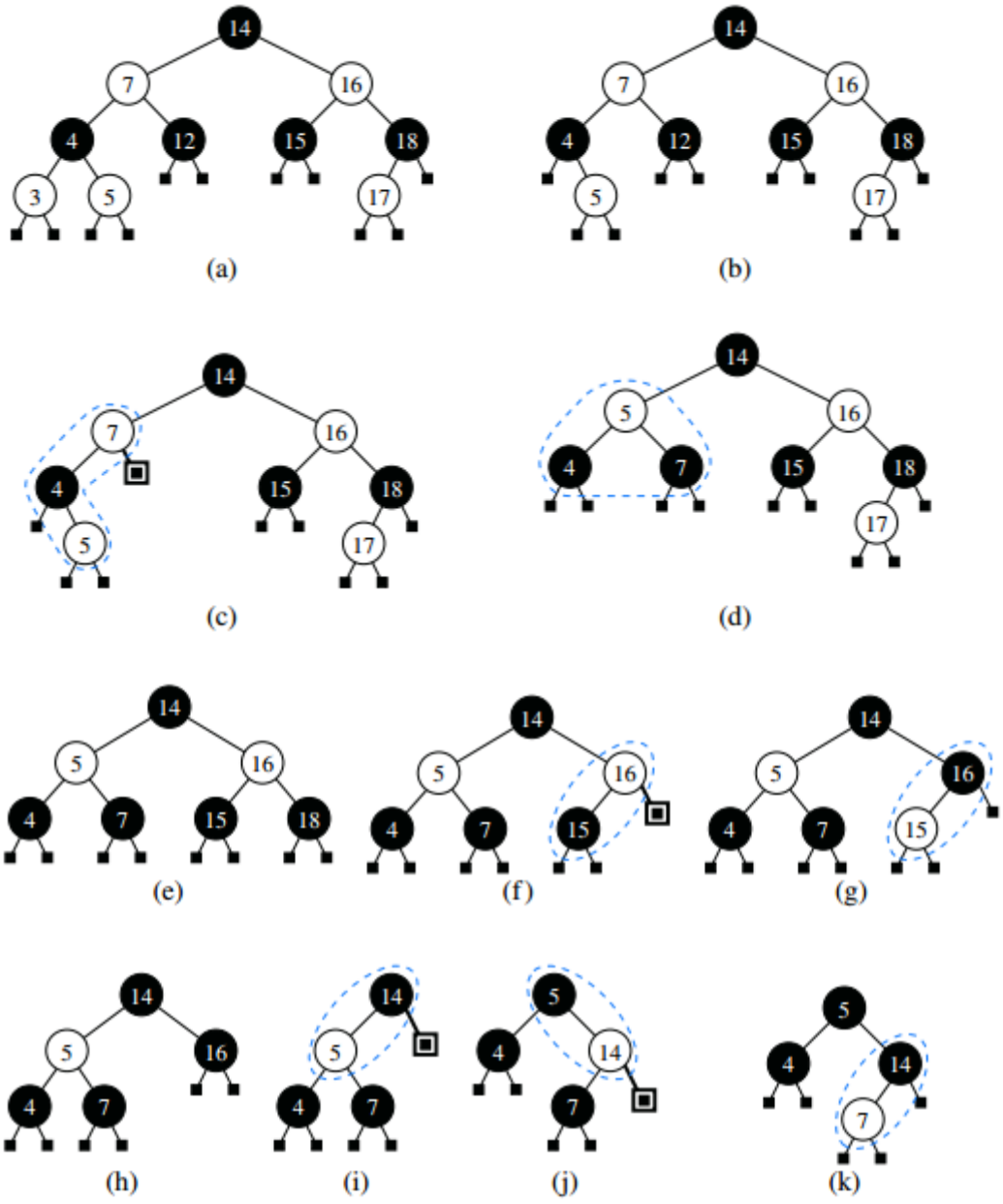
INSERTION RED-BLACK TREE

Figures Below show a sequence of insertion operations in a red-black tree.



DELETION RED-BLACK TREE

Figures Below show a sequence of deletion operations in a red-black tree.



6.3 EXERCISE

1. Implement a binary tree using linked structure that support all the update methods.

6.4 HOMEWORK

Implement insertion method for red-black tree

CHAPTER 7 – HASHING

7.1 LEARNING OBJECTIVES

1.1 Pratikan dapat mengenal dan menerapkan hashing dalam struktur data
1.2 Praktikan dapat mengkodekan hashing dalam konsep object oriented

7.2 MATERIALS

Hashing is a process to index, add, retrieve data from a database. By inserting hashing, finding, and deleting data in the database can be made quickly (complexity). In application hashing always use a hash table as a reference in inserting, finding, or deleting data in the data structure. Data can be stored in the hash table and the process of determining the location of the data in the hash table uses a hash function.

The structure of the hash table is just an array of fixed length, contains a key. A key is a string with the value of the association. Each key is mapped to some number in the range 0 to (table size-1) and placed in a cell.

Posisi	Kunci	Data
0	102300	
1	123600	
...	.....	.....
10		
10	432110	
10		
...	.....	.....
37	100237	
37		
37	546737	
...	.....	.....
77	100277	
77	767177	
77		
...	.....	.....
99	000199	
99	109899	
99	519299	
99		

Gambar 1-Hash Table

7.2.1 Hash Function

In general, a hash function (H) is a function to convert the set of keys recordings (K) into the address set reminders (position index array). There are two important aspects to consider when choosing a hash function. First, the hash function should be easy and quick to look for or calculated. Second, the hash function as much as possible to map the array index in the hash table is uniformly so that the collision (collision) can be minimized. Here are some commonly used hash functions:
Modular Method

In this way we can choose a perubah m of greater value than the number of keys in K, for example m, and usually have a prime number. The hash function is specified as:
H (k) = k mod m or H (k) = k mod m + 1
The first equation selected if the desired key address is 0 to m-1. The second equation is selected if the desired key address is 1 to m.

Midsquare Method
In this method, the key is known squared and the hash function is selected

H (k) = I

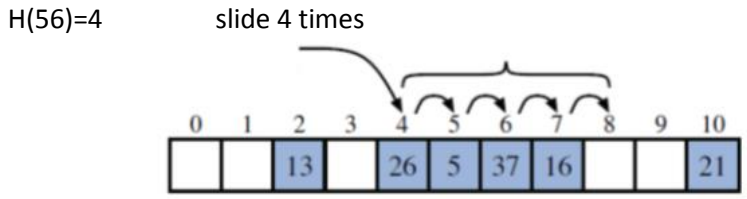
I value obtained with clear digits on both sides of k, with a note that the number of digits on the left and the right should be the same. If not the same, then the digit to the left as though the added amount of padding zero (0), so it will produce the correct address.

Digit Sum
In this function, keys that are known to be broken down into several groups, each consisting of several pieces of digits, for example two pieces. Then digits of existing group totaled.
H (10347) = 1 + 03 + 47 = 51
H (87 492) = 8 + 74 + 92 = 174 -> 1 + 74 = 75
H (34212) = 3 + 42 + 12 = 57
H (88 688) = 8 + 86 + 88 = 182 -> 1 + 82 = 83

7.2.2 Preventing Collision

Collision is a situation where if the position obtained from the hash function has been used, then another position should be found because the collision occurred. This collision can be solved by rehashing. Here are some ways to cope with collisions:

Linear Probing
If in case of a collision, then the solution is to find out if the array next to them already in use? If it is then it will look next to him again until the array is found that has not been used. As an example:



In the picture above will be made inserts, after having applied to a hash function H(k) then its generate the 4-th index. Because the index has been used then there was a collision. Then the next step is to find the next index that unused until you find the 8-th index.

```
41      public void insert(int key, int[] Array){
42          int indexArr = modMethod(key,arraySize,ArrayData);
43          if(apaArrayPenuh(Array)==true){
44              System.out.println("Array penuh!");
45          }else{
46              while(ArrayData[indexArr]!=-1){ //menggunakan linear probing
47               indexArr+=1;
48               if(indexArr==arraySize){
49                   indexArr=0;
50               }
51              }
52              ArrayData[indexArr]=key;
53          }
54      }
```

The above method uses linear probing to resolve collisions. Linear probing applied to the loop that serves to check whether a neighboring array slot has not been filled, the loop will continue to run until it finds the index array that has not been filled, in turn, enter value.

Quadratic Probing

Much like with linear probing, only in the event of a collision as the name suggests is quadratic probing, it will shift in squares to find the array slot that has not been filled. For example,

k + 1, k + 4, k + 9, k + 16, ...

this brings the advantage that if the number of addresses available is prime and hash table size array is large, then the trial time that is needed to find an array with empty slot can be minimalized.

```
56      public void insert2(int key, int[] Array){
57          int indexArr = modMethod(key,arraySize,ArrayData);
58          int i=1;
59          if(apaArrayPenuh(Array)==true){
60              System.out.println("Array penuh!");
61          }else{
62              while(ArrayData[indexArr]!=-1){ //menggunakan quadratic probing
63               indexArr+=i*i;
64               if(indexArr==arraySize){
65                   indexArr=0;
66               }else if(indexArr>=arraySize){
67                   indexArr%=arraySize;
68               }
69               i++;
70              }
71              ArrayData[indexArr]=key;
72          }
73      }
```

The above method using quadratic probing. Control flow in code intended that when the index over the hash table size (array size) then the search will return to the zero index. Remember if the key modulo by the numbers that is not prime then at a certain moment will occur an infinite loop, because it will cause

the same pattern while searching the index array. Therefore it would be good if the modulo number converted into a larger primes but approaching the array size.

Double Hashing

In double hashing is used two pieces of the hash function to avoid collisions. It can simply be explained as follows. K determined from the key hash addresses his first, for example, H (k) = h. Then a second hash specified address, for example, H (k) = h '≠ m (where m is the number of hash addresses that can be generated from the first hash function). Thus, the search is performed in sequence at addresses

h, h + h ', h + 2h', h + 3h ', ...

```
75      public void insert3(int key, int[] Array){ //Double hashing
76          int index1 = modMethod(key,arraySize,ArrayData); //index fungsi hash 1
77          int index2 = (key+(key%10))%arraySize; //index fungsi hash 2
78          if(apaArrayPenuh(Array)==true){
79              System.out.println("Array penuh!");
80          }else{
81              int indexNext=index1;
82              while(ArrayData[indexNext]!=-1){
83                  int i=1;
84                  indexNext=index1+(i*index2);
85                  if(indexNext==arraySize){
86                    indexNext=0;
87                  }else if(indexNext>=arraySize){
88                    indexNext%=arraySize;
89                  }
90                  i++;
91              }
92              ArrayData[indexNext]=key;
93          }
94      }
```

From the above code can be observed that in the process of dealing with collisions this method uses a hash function that works to help determine how many shifts to be done. To avoid an infinite loop like quadratic probing, then how good it using primes as the modulo number.

```
98      public static void main(String[] args) {
99          // TODO code application logic here
100         Hashing theFunc = new Hashing(30);
101
102         theFunc.insert2(10, ArrayData);
103         theFunc.insert2(10, ArrayData);
104
105         for (int ArrayData1 : ArrayData) {
106             System.out.print(ArrayData1+" ");
107         }
108
109     }
```

```
Output - hashing (run)
run:
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 10 10 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 BUILD SUCCESSFUL (total time: 0 seconds)
```

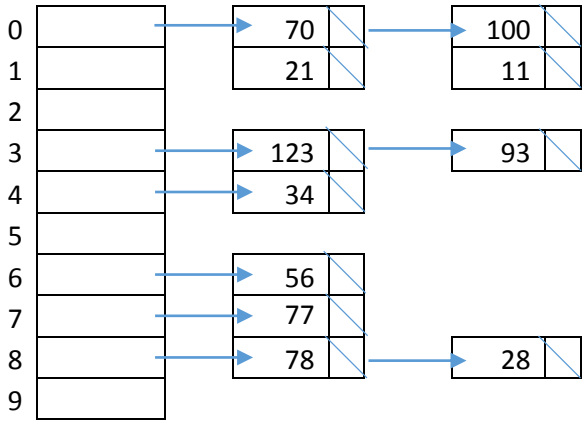
Chaining

Chaining is another method used to overcome the possibility of hash collisions address. In principle, this method utilizes linked list mounted on each array address hash table. Thus, a complete hash address and a linked list that stores the records that have the same hash address, the address will be stored in the same hash linked list corresponding to using the pointer as a liaison.

For example, if we have tapes recording key can be written as

34 56 123 78 93 70 100 21 11 77 28

And a hash function that is selected is k mod 10. Thus, the hash address will consist of ten addresses numbered 0 through 9. recordings above will be mapped as shown below



Because in this method chaining is the implementation of a single linked list it must get a node (a separate class) that contains key and value.

```
16    class LinkedHashEntry{
17        String key;
18        int value;
19
20        LinkedHashEntry next;
21        /* Constructor */
22
23        LinkedHashEntry(String key, int value){
24            this.key = key;
25            this.value = value;
26            this.next = null;
27        }
28    }
```

Then, because it will be implemented on the array also then made a hash table class, in which is contained method such as insert (), remove (), etc.

```
32  class HashTable{
34      private int TABLE_SIZE;
35      private int size;
37      private LinkedHashEntry[] table;
38      /* Constructor */
39      public HashTable(int ts){
40          size = 0;
41          TABLE_SIZE = ts;
42          table = new LinkedHashEntry[TABLE_SIZE];
43
44          for (int i = 0; i < TABLE_SIZE; i++){
45              table[i] = null;
46          }
47      }
48
49      /* Function to get number of key-value pairs */
50      public int getSize(){
51          return size;
52      }
53
54      /* Function to clear hash table */
55      public void makeEmpty(){
56          for (int i = 0; i < TABLE_SIZE; i++){
57              table[i] = null;
58          }
59      }
```

```
61      /* Function to get value of a key */
62      public int get(String key){
63          int hash = (myhash( key ) % TABLE_SIZE);
64          if (table[hash] == null){
65              return -1;
66          }
67          else{
68              LinkedHashEntry entry = table[hash];
69              while (entry != null && !entry.key.equals(key))
70                  entry = entry.next;
71              if (entry == null)
72                  return -1;
73              else
74                  return entry.value;
75          }
76
77      }
```

```
79      /* Function to insert a key value pair */
80      public void insert(String key, int value){
81          int hash = (myhash( key ) % TABLE_SIZE);
82          if (table[hash] == null)
83              table[hash] = new LinkedHashEntry(key, value);
84          else{
85              LinkedHashEntry entry = table[hash];
86              while (entry.next != null && !entry.key.equals(key))
87                  entry = entry.next;
88              if (entry.key.equals(key))
89                  entry.value = value;
90              else
91                  entry.next = new LinkedHashEntry(key, value);
92          }
93          size++;
94      }
```

```
98      public void remove(String key){
99          int hash = (myhash( key ) % TABLE_SIZE);
100         if (table[hash] != null){
101             LinkedHashEntry prevEntry = null;
102             LinkedHashEntry entry = table[hash];
103             while (entry.next != null && !entry.key.equals(key)){
104                 prevEntry = entry;
105                 entry = entry.next;
106             }
107
108             if (entry.key.equals(key)){
109                 if (prevEntry == null)
110                     table[hash] = entry.next;
111                 else
112                     prevEntry.next = entry.next;
113                 size--;
114             }
115         }
116     }
```

```
120     private int myhash(String x ){
121         int hashVal = x.hashCode( );
122         hashVal %= TABLE_SIZE;
123         if (hashVal < 0)
124             hashVal += TABLE_SIZE;
125         return hashVal;
126     }
127 
128     /* Function to print hash table */
129     public void printHashTable(){
130         for (int i = 0; i < TABLE_SIZE; i++){
131             System.out.print("\nBucket "+ (i + 1) +" : ");
132             LinkedHashEntry entry = table[i];
133             while (entry != null)
134             {
135                 System.out.print(entry.value +" ");
136                 entry = entry.next;
137             }
138         }
139     }
140 }
```

```
141    public class HashTablesChainingListHeadsTest {
142
143        public static void main(String[] args){
144
145        Scanner scan = new Scanner(System.in);
146
147        System.out.println("Hash Table Test\n\n");
148
149        System.out.println("Enter size");
150
151        /* Make object of HashTable */
152
153        HashTable ht = new HashTable(scan.nextInt());
154
155        char ch;
156
157        /*  Perform HashTable operations  */
158
159        do{
160
161            System.out.println("\nHash Table Operations\n");
162
163            System.out.println("1. insert ");
164
165            System.out.println("2. remove");
166
167            System.out.println("3. get");
```

```
169                 System.out.println("4. clear");
170
171                 System.out.println("5. size");
172
173
174                 int choice = scan.nextInt();
175
176                 switch (choice){
177
178                 case 1 :
179
180                     System.out.println("Enter key and value");
181
182                     ht.insert(scan.next(), scan.nextInt() );
183
184                     break;
185
186                 case 2 :
187
188                     System.out.println("Enter key");
189
190                     ht.remove( scan.next() );
191
192                     break;
```

```
194                 case 3 :
195
196                     System.out.println("Enter key");
197
198                     System.out.println("Value = "+ ht.get( scan.next() ));
199
200                     break;
201
202                 case 4 :
203
204                     ht.makeEmpty();
205
206                     System.out.println("Hash Table Cleared\n");
207
208                     break;
209
210                 case 5 :
211
212                     System.out.println("Size = "+ ht.getSize() );
213
214                     break;
215
216                 default :
217
218                     System.out.println("Wrong Entry \n ");
219
220                     break;
221
222                 }
```

```
224                 /* Display hash table */
225
226                 ht.printHashTable();
227
228                 System.out.println("\nDo you want to continue (Type y or n) \n");
229
230                 ch = scan.next().charAt(0);
231
232             } while (ch == 'Y'|| ch == 'y');
233         }
234     }
```

```
Hash Table Operations

1. insert
2. remove
3. get
4. clear
5. size
1
Enter key and value
Koko 35

Bucket 1 : 90
Bucket 2 :
Bucket 3 : 50 70 35
Do you want to continue (Type y or n)

n
BUILD SUCCESSFUL (total time: 2 minutes 6 seconds)
```

7.3 Exercise

Data given key: {10,20,30,40,50,8,60,15,30,45,11,13,17,19,100}, apply the hash function on the key on the array and hashing should be able to handle collision / crash , Make if the collision is avoided by the way,

a. linear probing
b. quadratic probing
c. Double hashing

CHAPTER 8 – HEAP

8.1 LEARNING OBJECTIVES

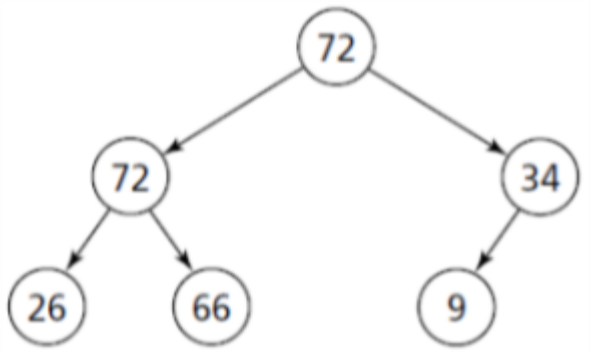
1. create an implementation of the Heap ADT using an array representation of a tree.
2. create a heap sort method based on the heap construction techniques used in your implementation of the Heap ADT.
3. create a heap sort method based on the heap construction techniques used in your implementation of the Heap ADT.
4. analyze where elements with various priorities are located in a heap.

8.2 MATERIALS

8.2.1 OVERVIEW

A heap is a binary tree that satisfies the following property:

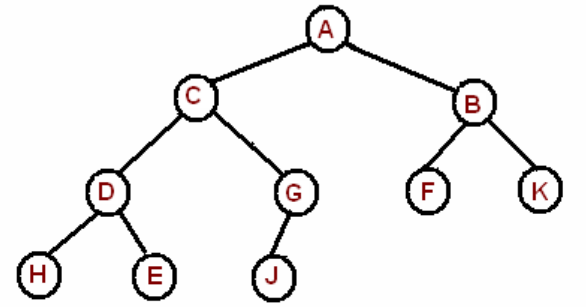
- Tree must be complete. The point is that each level of the tree should be filled (a parent should have the right leftchild and child), except at the lowest level may not be complete. If the lowest level is not complete then the child should be vacated is rightchild.
- For each element E, all of which are child values of E must be less than or equal with the value of E. Therefore, root will store the maximum value.



Gambar 1 – Heap Tree

8.2.2 Array Implementation

In indexing the array index, heap tree has rules to store value as follows:



0	1	2	3	4	5	6	7	8	9	10
	A	C	B	D	G	F	K	H	E	J

In this case the root is placed on the index "1" to facilitate the calculation of the index. Can be observed that for a node other than the root applicable,

- For each leftchild is at index 2 * k
- For each rightchild is at index 2 * k + 1
- And the parent is at index k / 2

Heap tree is formed of nodes that have a key value. Therefore, it needs a node class as the following code

```
8     class Node
9        {
10       private int iData;              // data item (key)
11    // -----------------------------------------------------------
12       public Node(int key)            // constructor
13          { iData = key; }
14    // -----------------------------------------------------------
15       public int getKey()
16          { return iData; }
17    // -----------------------------------------------------------
18       public void setKey(int id)
19          { iData = id; }
20    // -----------------------------------------------------------
21       }  // end class Node
```

After make the node class, then made a heap class

```
23      class Heap
24          {
            private Node[] heapArray;
            private int maxSize;            // size of array
27          private int currentSize;        // number of nodes in array
28      // --------------------------------------------------------------
29          public Heap(int mx)             // constructor
30              {
31              maxSize = mx;
32              currentSize = 0;
33              heapArray = new Node[maxSize];  // create array
34              }
35      // --------------------------------------------------------------
36          public boolean isEmpty()
37              { return currentSize==0; }
```

Once formed Heap class, of course we need a method to insert, remove, and to keep the heap property is always met.

8.2.3 Heap – insert

The new elements always be added to the heap at the bottom level. Property of the heap is always kept by comparing elements that are added to its parent and then swap any position when the element is larger than its parent. Keep in mind that the purpose of this property is to keep the value of the parent is always greater than the child.

```
39      public boolean insert(int key)
40          {
41          if(currentSize==maxSize)
42              return false;
43          Node newNode = new Node(key);
44          heapArray[currentSize] = newNode;
45          trickleUp(currentSize++);
46          return true;
47          }  // end insert()
48      // -------------------------------------------------------------
49      public void trickleUp(int index)
50          {
51          int parent = (index-1) / 2;
52          Node bottom = heapArray[index];
53
54          while( index > 0 &&
55                  heapArray[parent].getKey() < bottom.getKey() )
56              {
57              heapArray[index] = heapArray[parent];  // move it down
58              index = parent;
59              parent = (parent-1) / 2;
60              }  // end while
61          heapArray[index] = bottom;
62          }  // end trickleUp()
```

8.2.4 Heap – remove

In heap tree, when its applied to the method remove () then the node that has the largest value is deleted. Then the root will be replaced with the lower-right node. Then the tree heap will keep the property with the existing rule that would bring down the rightmost node to the most suitable position on the heap tree.

```
64      public Node remove()            // delete item with max key
65        {                             // (assumes non-empty list)
66        Node root = heapArray[0];
67        heapArray[0] = heapArray[--currentSize];
68        trickleDown(0);
69        return root;
70        }  // end remove()
71    // -------------------------------------------------------------
72      public void trickleDown(int index){
73        int largerChild;
74        Node top = heapArray[index];       // save root
75        while(index < currentSize/2)       // while node has at
76           {                               //    least one child,
77           int leftChild = 2*index;
78           int rightChild = leftChild+1;
79                                           // find larger child
80           if(rightChild < currentSize &&  // (rightChild exists?)
81                           heapArray[leftChild].getKey() <
82                           heapArray[rightChild].getKey())
83              largerChild = rightChild;
84           else
85              largerChild = leftChild;
86                                           // top >= largerChild?
87           if( top.getKey() >= heapArray[largerChild].getKey() )
88              break;
89                                           // shift child up
90           heapArray[index] = heapArray[largerChild];
91           index = largerChild;            // go down
92           }  // end while
93        heapArray[index] = top;            // root to index
94        }  // end trickleDown()
```

Delete on this method is delete the max, so if the method is called the key to the root will be lost because the key to the root key must be the greatest / maximum. After the root is removed then the trickle-down method will look for the right key to root replacement at the same time to maintain the heap property remains unfulfilled.

```
96      public boolean change(int index, int newValue)
97          {
98          if(index<0 || index>=currentSize)
99              return false;
100         int oldValue = heapArray[index].getKey(); // remember old
101         heapArray[index].setKey(newValue);  // change to new
102
103         if(oldValue < newValue)             // if raised,
104             trickleUp(index);               // trickle it up
105         else                                // if lowered,
106             trickleDown(index);             // trickle it down
107         return true;
108         }  // end change()
```

8.2.4 Print the heap tree

Each value / key from the heap tree stored in an array. Then to display the heap tree, the heap property is used.

```
110     public void displayHeap()
111         {
112         System.out.print("heapArray: ");     // array format
113         for(int m=0; m<currentSize; m++)
114             if(heapArray[m] != null)
115                 System.out.print( heapArray[m].getKey() + " ");
116             else
117                 System.out.print( "-- ");
118         System.out.println();
119                                               // heap format
120         int nBlanks = 32;
121         int itemsPerRow = 1;
122         int column = 0;
123         int j = 0;                            // current item
124         String dots = "................................";
125         System.out.println(dots+dots);        // dotted top line
126
127         while(currentSize > 0)                // for each heap item
128             {
129             if(column == 0)                   // first item in row?
130                 for(int k=0; k<nBlanks; k++)  // preceding blanks
131                     System.out.print(' ');
```

8.3 Exercise

Make heap tree for arrays with size 15 and data key:
Key = {78, 3, 9, 10, 23, 77, 34, 86, 90, 100, 20, 66, 94, 63, 97}

8.4 Homework

Make a heap sort to sort the data key on a heap of exercise above

CHAPTER 9 - GRAPHS

9.1 LEARNING OBJECTIVES

In this laboratory you will:
Create an implementation of the Graph

9.2 MATERIALS

9.2.1 OVERVIEW

Graphs are widely-used structure in computer science and different computer applications. We don't say data structure here and see the difference. Graphs mean to store and analyze metadata, the connections, which present in data. For instance, consider cities in your country. Road network, which connects them, can be represented as a graph and then analyzed. We can examine, if one city can be reached from another one or find the shortest route between two cities.
First of all, we introduce some definitions on graphs. Next, we are going to show, how graphs are represented inside of a computer. Then you can turn to basic graph algorithms.
There are two important sets of objects, which specify graph and its structure. First set is **V**, which is called **vertex-set**. In the example with road network cities are vertices. Each vertex can be drawn as a circle with vertex's number inside.

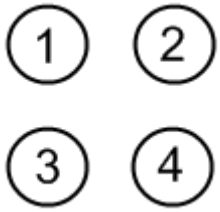


Figure 9.1 Vertices

Next important set is E, which is called edge-set. E is a subset of V x V. Simply speaking, each edge connects two vertices, including a case, when a vertex is connected to itself (such an edge is called a loop). All graphs are divided into two big groups: directed and undirected graphs. The difference is that edges in directed graphs, called arcs, have a direction. These kinds of graphs have much in common with each other, but significant differences are also present. We will accentuate which kind of graphs is considered in the particular algorithm description. Edge can be drawn as a line. If a graph is directed, each line has an arrow.

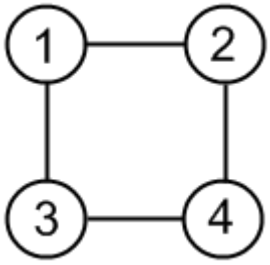


Figure 9.2 Undirected Graph

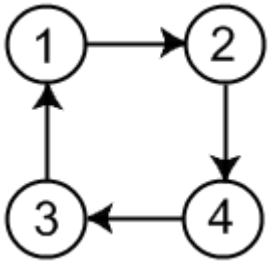


Figure 9.3 Directed Graph

9.2.2 GRAPH IMPLEMENTATION

Let's begin with the Vertex class. A Vertex is a a node in the graph, as described above. The vertex's neighbors are described with an ArrayList<Edge>. The purpose of using the incidence neighborhood (Edges) rather than the adjacency neighborhood (Vertices) was because path-finding and spanning tree algorithms need the edges to function. Note that this implementation does not allow for multiple Edge objects between the same pair of Vertex objects.

```
4   public class Vertex {
5
6       private ArrayList<Edge> neighborhood;
7       private String kota;
8
9       public Vertex(String label){
10          this.kota = label;
11          this.neighborhood = new ArrayList<Edge>();
12      }
```

Program Code 9.1 Vertex Class

Next, we examine the Edge class. An Edge models the adjacency relation between two vertices. So the Edge class has two vertices. In many situations, the notion of an edge weight is also important. Intuitively, the edge weight represents the distance between two vertices. Even in unweighted graphs, it is a commonly accepted notion that traversing two edges bears greater cost than staying at the current vertex. So the Edge class allows for a weight attribute, assuming by convention a uniform weight of 1 if no weight is specified. This allows for treating the graph as "unweighted" in a manner that is still consistent with the preconditions of many common graph algorithms.

It is also important to note that I deviate from a key expectation of the Comparable interface here. The Comparable interface expects that if the compareTo() method returns a value of 0, then the equals() method will return a value of true for the given parameter. I use the compareTo() method to strictly compare Edge weights, while the equals() method simply tests if the Edge references the same pair of Vertices. The design decision has two considerations. The first is to prohibit multiple edges for the same pair of vertices, to which the equals() method contributes. The second consideration is that the compareTo() method only cares about Edge weights in algorithmic considerations.

```
3   public class Edge implements Comparable<Edge> {
4
5       private Vertex one, two;
6       private int weight;
7
8       public Edge(Vertex one, Vertex two){
9           this(one, two, 1);
10      }
11
12      public Edge(Vertex one, Vertex two, int weight){
13          this.one = (one.getLabel().compareTo(two.getLabel()) <= 0) ? one : two;
14          this.two = (this.one == one) ? two : one;
15          this.weight = weight;
16      }
```

Program Code 9.2 Edge Class and Constructor

```
75      public int compareTo(Edge other){
76          return this.weight - other.weight;
77      }
```

Program Code 9.3 compareTo method

```
100     public boolean equals(Object other){
101         if(!(other instanceof Edge)){
102             return false;
103         }
104
105         Edge e = (Edge)other;
106
107         return e.one.equals(this.one) && e.two.equals(this.two);
        }
```

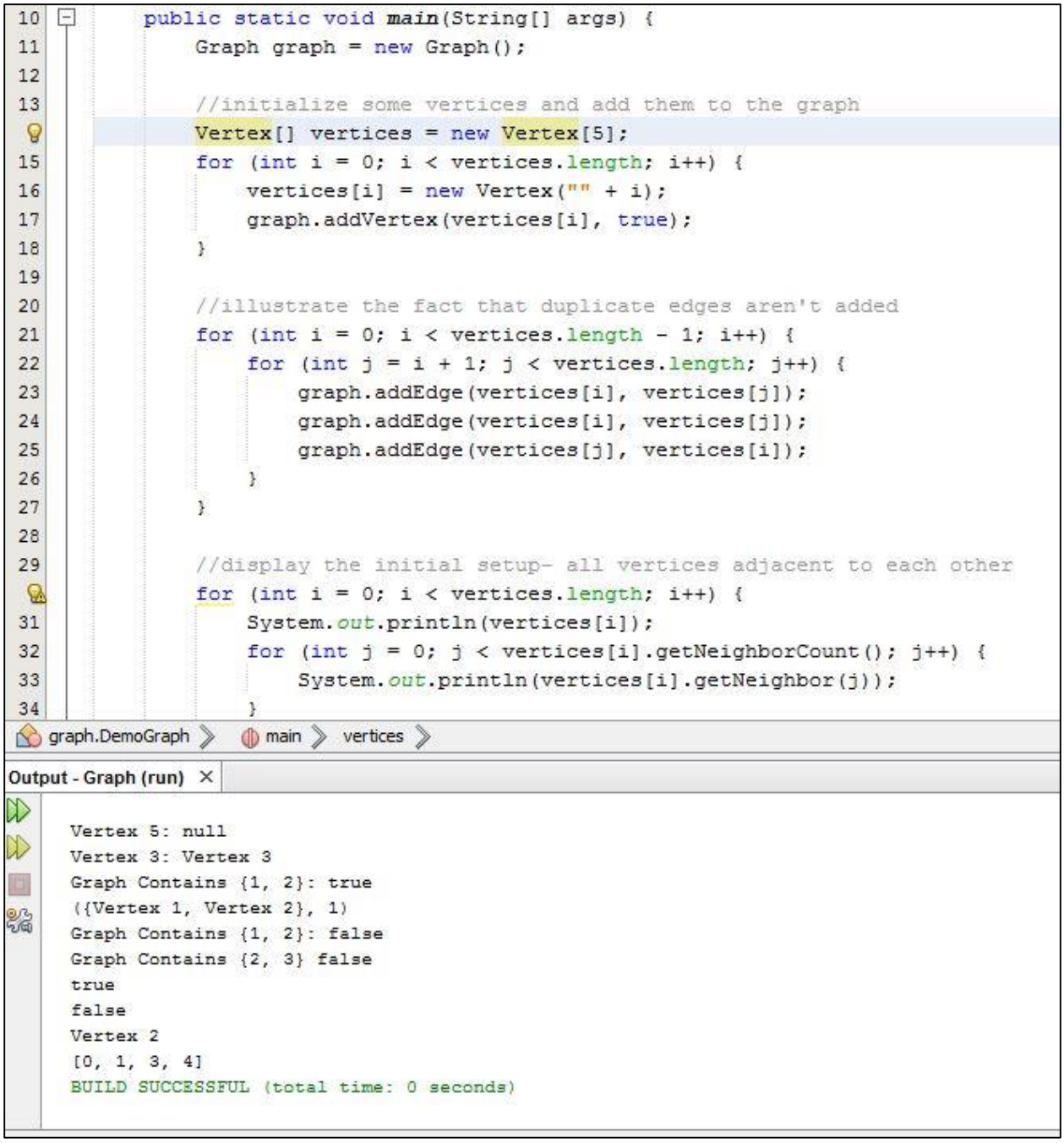
Program Code 9.4 equals method

Lastly, consider the Graph class implementation. We utilize an incidence list implementation, which is favorable for many (but not all) Graph algorithms. Basic operations of insertion, lookup, and removal are supported for both Vertex and Edge objects. Vertex objects are uniquely identified by their labels, and no two Vertex objects can share the same label. However, there is support to overwrite existing Vertex objects. Note that doing such will remove from the Graph all Edges associated with the existing Vertex.

No equals() method is provided for the Graph class. The notion of Graph equality is referred to as Graph isomorphism. The problem deciding if two Graphs are isomorphic is not known to be in the complexity class P. Therefore, it is not known if a polynomial time algorithm exists to determine if two graphs are isomorphic. For this reason, no equals() method is included to compare the two Graphs.

```
16  public class Graph {
17
        private HashMap<String, Vertex> vertices;
        private HashMap<Integer, Edge> edges;
20
21      public Graph(){
            this.vertices = new HashMap<String, Vertex>();
            this.edges = new HashMap<Integer, Edge>();
24      }
25
26      public Graph(ArrayList<Vertex> vertices){
            this.vertices = new HashMap<String, Vertex>();
            this.edges = new HashMap<Integer, Edge>();
29
            for(Vertex v: vertices){
31              this.vertices.put(v.getLabel(), v);
32          }
33      }
```

Here is a demo file to illustrate Graph class functionality, the graph that will be built is going to be:



```
10      public static void main(String[] args) {
11          Graph graph = new Graph();
12
13          //initialize some vertices and add them to the graph
            Vertex[] vertices = new Vertex[5];
15          for (int i = 0; i < vertices.length; i++) {
16              vertices[i] = new Vertex("" + i);
17              graph.addVertex(vertices[i], true);
18          }
19
20          //illustrate the fact that duplicate edges aren't added
21          for (int i = 0; i < vertices.length - 1; i++) {
22              for (int j = i + 1; j < vertices.length; j++) {
23                  graph.addEdge(vertices[i], vertices[j]);
24                  graph.addEdge(vertices[i], vertices[j]);
25                  graph.addEdge(vertices[j], vertices[i]);
26              }
27          }
28
29          //display the initial setup- all vertices adjacent to each other
            for (int i = 0; i < vertices.length; i++) {
31              System.out.println(vertices[i]);
32              for (int j = 0; j < vertices[i].getNeighborCount(); j++) {
33                  System.out.println(vertices[i].getNeighbor(j));
34              }
```

graph.DemoGraph > main > vertices

Output - Graph (run)

```
Vertex 5: null
Vertex 3: Vertex 3
Graph Contains {1, 2}: true
({Vertex 1, Vertex 2}, 1)
Graph Contains {1, 2}: false
Graph Contains {2, 3} false
true
false
Vertex 2
[0, 1, 3, 4]
BUILD SUCCESSFUL (total time: 0 seconds)
```

The full implementation of DemaGraph is shown below:

```
public class DemoGraph {

    public static void main(String[] args){
        Graph graph = new Graph();

        //initialize some vertices and add them to the graph
        Vertex[] vertices = new Vertex[5];
```

```
        for(int i = 0; i < vertices.length; i++){
            vertices[i] = new Vertex("" + i);
            graph.addVertex(vertices[i], true);
        }

        //illustrate the fact that duplicate edges aren't added
        for(int i = 0; i < vertices.length - 1; i++){
            for(int j = i + 1; j < vertices.length; j++){
               graph.addEdge(vertices[i], vertices[j]);
               graph.addEdge(vertices[i], vertices[j]);
               graph.addEdge(vertices[j], vertices[i]);
            }
        }

        //display the initial setup- all vertices adjacent to each other
        for(int i = 0; i < vertices.length; i++){
            System.out.println(vertices[i]);

            for(int j = 0; j < vertices[i].getNeighborCount(); j++){
                System.out.println(vertices[i].getNeighbor(j));
            }

            System.out.println();
        }

        //overwritte Vertex 3
        graph.addVertex(new Vertex("3"), true);
        for(int i = 0; i < vertices.length; i++){
            System.out.println(vertices[i]);

            for(int j = 0; j < vertices[i].getNeighborCount(); j++){
                System.out.println(vertices[i].getNeighbor(j));
            }

            System.out.println();
        }


        System.out.println("Vertex 5: " + graph.getVertex("5")); //null
        System.out.println("Vertex 3: " + graph.getVertex("3")); //Vertex 3

        //true
        System.out.println("Graph Contains {1, 2}: " +
                graph.containsEdge(new Edge(graph.getVertex("1"),
             graph.getVertex("2"))));

        //true
        System.out.println(graph.removeEdge(new Edge(graph.getVertex("1"),
             graph.getVertex("2"))));

        //false
```

```
        System.out.println("Graph Contains {1, 2}: " + graph.containsEdge(new
             Edge(graph.getVertex("1"), graph.getVertex("2"))));

        //false
        System.out.println("Graph Contains {2, 3} " + graph.containsEdge(new
             Edge(graph.getVertex("2"), graph.getVertex("3"))));

        System.out.println(graph.containsVertex(new Vertex("1"))); //true
        System.out.println(graph.containsVertex(new Vertex("6"))); //false
        System.out.println(graph.removeVertex("2")); //Vertex 2
        System.out.println(graph.vertexKeys()); //[3, 1, 0, 4]

    }
}
```

9.3 EXERCISES

1. Test the Graph Implementation according to code
2. Implement Depth First Search and Breadth First Search

CHAPTER 10 – WEIGHTED GRAPHS

10.1 LEARNING OBJECTIVES

In this laboratory you:

- Create an implementation of the Weighted Graph ADT to implement shortestPath.

10.2 MATERIALS

10.2.1 WEIGHTED GRAPH OVERVIEW

In the last chapter we saw that a graph's edges can have direction. In this chapter we'll explore another edge feature: weight. For example, if vertices in a weighted graph represent cities, the weight of the edges might represent distances between the cities, or costs to fly between them, or the number of automobile trips made annually between them (a figure of interest to highway engineers). When we include weight as a feature of a graph's edges, some interesting and complex questions arise. What is the minimum spanning tree for a weighted graph? What is the shortest (or cheapest) distance from one vertex to another? Such questions have important applications in the real world.

10.2.2 WEIGHTED GRAPH IMPLEMENTATION

In the previous chapter you have seen that every edge can be weighted, as you can see in this method:

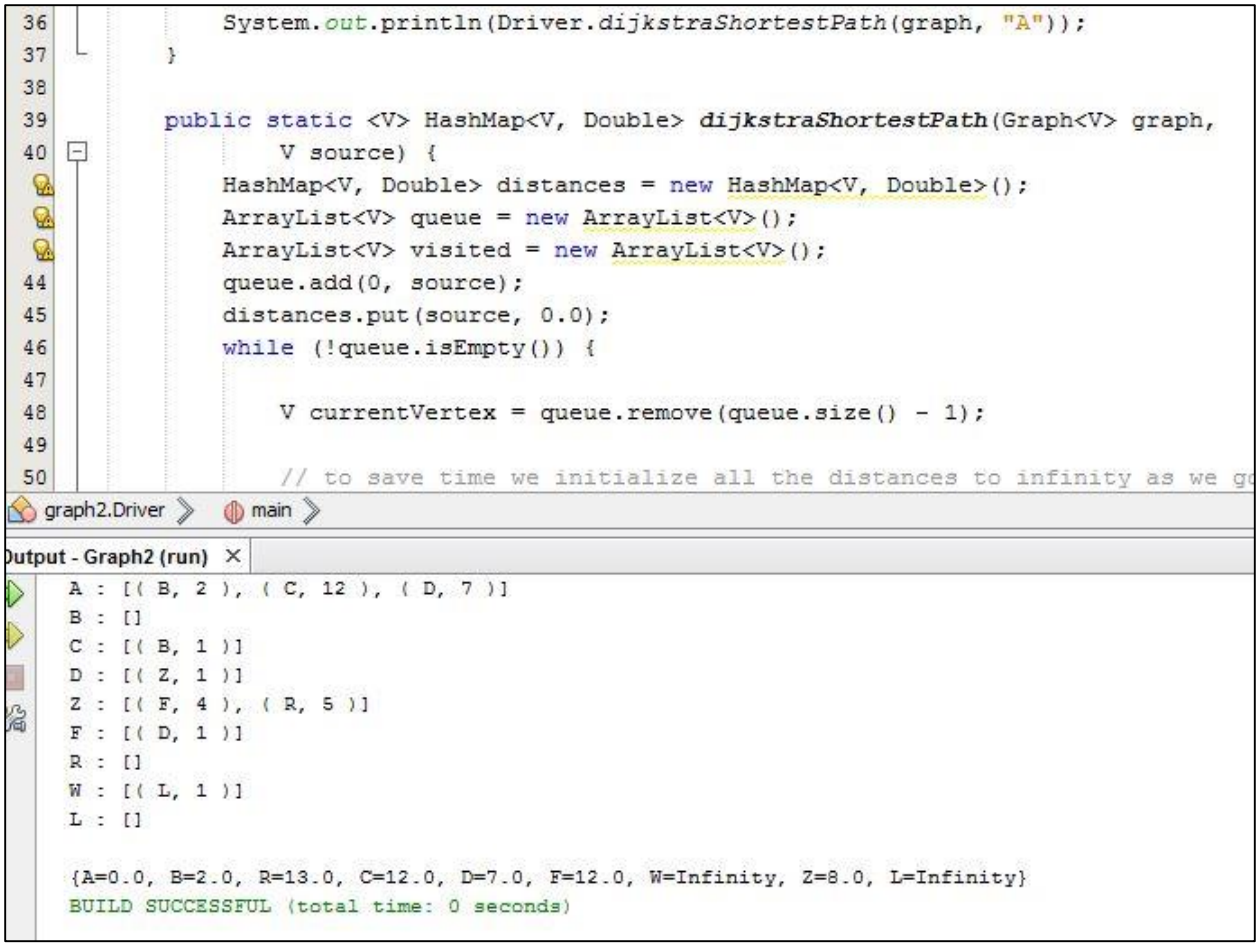
```
public boolean addEdge(V vertexOne, V vertexTwo, int weight) {
    if (directed) {
        return false;
    }

    if (!adjacencyList.containsKey(vertexOne)) {
        ArrayList<Edge<V>> tempList = new ArrayList<Edge<V>>();
        tempList.add(new Edge<V>(vertexTwo, weight));
        add(vertexOne, tempList);
        return true;
    }

    if (!adjacencyList.containsKey(vertexTwo)) {
        ArrayList<Edge<V>> tempList = new ArrayList<Edge<V>>();
        tempList.add(new Edge<V>(vertexOne, weight));
        add(vertexTwo, tempList);
        return true;
    }

    adjacencyList.get(vertexOne).add(new Edge<V>(vertexTwo, weight));
    adjacencyList.get(vertexTwo).add(new Edge<V>(vertexOne, weight));
    return true;
}
```

In the Graph class, add a method void shortestPath(Vertex v0) that implements a single-source shortest-path algorithm for the Graph.This method finds the cost of the shortest path between v0 and every other connected vertex in the graph, counting every edge as having unit cost. An outline of the algorithm

(Dijkstra's algorithm) is below. Note that Dijkstra's algorithm works for graphs with non-negative weight edges, but in this case all edges have the same weight.

```
Given: a graph G and starting vertex v0 in G

Initialize all vertices in G to be unmarked and have infinite cost

Create a priority queue, q, that orders vertices by lowest cost

Set the cost of v0 to 0 and add it to q

while q is not empty:
    let v be the vertex in q with lowest cost
    remove v from q
    mark v as visited
    for each vertex w that neighbors v:
        if w is not marked and v.cost + 1 < w.cost:
            w.cost = v.cost + 1
            add w to q

Output: the cost of each vertex v in G is the shortest distance from v0 to v.
```

Program for Djikstra:



```
36              System.out.println(Driver.dijkstraShortestPath(graph, "A"));
37          }
38
39          public static <V> HashMap<V, Double> dijkstraShortestPath(Graph<V> graph,
40                  V source) {
                HashMap<V, Double> distances = new HashMap<V, Double>();
                ArrayList<V> queue = new ArrayList<V>();
                ArrayList<V> visited = new ArrayList<V>();
44              queue.add(0, source);
45              distances.put(source, 0.0);
46              while (!queue.isEmpty()) {
47
48                  V currentVertex = queue.remove(queue.size() - 1);
49
50                  // to save time we initialize all the distances to infinity as we go
```

graph2.Driver > main >

Output - Graph2 (run)

```
A : [( B, 2 ), ( C, 12 ), ( D, 7 )]
B : []
C : [( B, 1 )]
D : [( Z, 1 )]
Z : [( F, 4 ), ( R, 5 )]
F : [( D, 1 )]
R : []
W : [( L, 1 )]
L : []

{A=0.0, B=2.0, R=13.0, C=12.0, D=7.0, F=12.0, W=Infinity, Z=8.0, L=Infinity}
BUILD SUCCESSFUL (total time: 0 seconds)
```

10.3 EXERCISES

Implement the shortest path problem in your java code.

10.4 HOMEWORK

Implement the shortest path on Map Direction Problem. Take the data from Google Maps API.