University of Malta

Junior College

Department of Computing and Information Technology

Programming Techniques

IT Advanced Level Course Notes

Riccardo Flask

CONTENTS

STRUCTURED DATA TYPES

VECTORS

Vectors are like arrays because they possess the following common properties:

- *homogeneous* - vectors typically store only one kind of element;
- *indexed* - vectors permit clients to access values using integers that indicate position;
- *efficient* - vectors provide fast access to values.

The Vector class is found in the java.util package, and extends java.util.Abstractlist. What makes vectors better than arrays is the fact that they are ***expandable***. The size of vectors can change (grow or shrink), as opposed to the fixed size of arrays. The vector class is also called a ***container***. To use vectors one should follow these steps:

1. Import the java class
    a. `import java.util.Vector;` //load vector package
2. Declare a variable to hold the vector
    a. `Vector v;` //declare a vector object
3. Declare a new vector object and assign it to the vector variable
    a. `v = new Vector();` // create an empty vector object

Vectors can be also given an initial value which determines their initial size:

`v = new Vector(5);` // create a vector object with initial capacity 5

The following methods can be used with Vectors:

1. `addElement` – to add elements to a vector
```
v.addElement(new Integer(1));
v.addElement(new Float(1.9999));
for (int i=2; i<10; i++) {
int lastInt = ((Number) v.lastElement()).intValue();
v.addElement (new Integer(i + lastInt)); } //adding elements recursively
```
2. `size()` – returns size of vector
```
v.size();
```
3. `removeElement` – to remove items from vector
```
v.removeElementAt(0); //removes an element stored at index 0 (first position)
```
4. `trimToSize` – At this point the size of Vector has not been altered, only the data has been removed. To resize the vector after deleting some items the method has to be used.
5. `elementAt` – retrieve a specific element from a vector
```
for (int i=0; i<v.size(); i++) {
System.out.println("v[" + i + "] = " + v.elementAt(i));
}
```
6. `setElementAt` – store a particular value in a particular location
```
v.setElementAt (x, i); //puts item x in position I (a[i] = x;
```

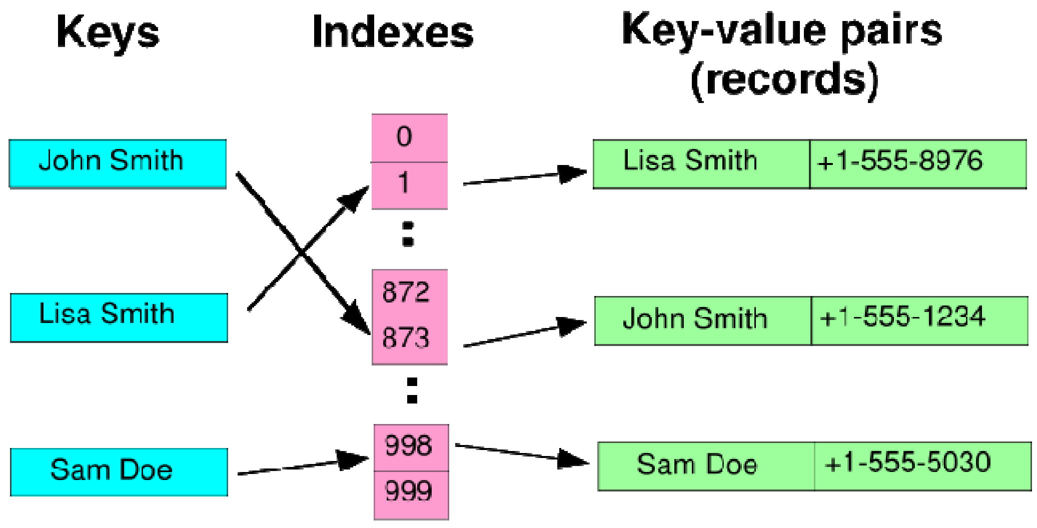
To print the contents of a vector, another method can be used using the Enumeration Interface.

```
// Traverse entire list, printing them all out
for (Enumeration e = myVector.elements(); e.hasMoreElements();)
```

```
{
        String myString = (String) e.nextElement();
        System.out.println(myString);
}
```

HASH TABLES

Hashtables are an extremely useful mechanism for storing data. Hashtables work by mapping a key to a value, which is stored in an in-memory data structure. Rather than searching through all elements of the hashtable for a matching key, a hashing function analyses a key, and returns an index number. This index matches a stored value, and the data is then accessed. This is an extremely efficient data structure.



Hashtables are supported by Java, in the form of the java.util.Hashtable class. Hashtables accept as keys and values any Java object. You can use a String, for example, as a key, or perhaps a number such as an Integer. However, you can't use a primitive data type, so you'll need to instead use Char, Integer, Long, etc.

```
// Use an Integer as a wrapper for an int

Integer integer = new Integer ( i );

hash.put( integer, data);
```

Data is placed into a hashtable through the put method, and can be accessed using the get method. It's important to know the key that maps to a value, otherwise it’s difficult to get the data back. If you want to process all the elements in a hashtable, you can always ask for an Enumeration of the hashtable's keys. The get method returns an object, which can then be cast back to the original object type.

```
// Get all values with an enumeration of the keys
```

```
   for (Enumeration e = hash.keys(); e.hasMoreElements();)
   {
       String str = (String) hash.get( e.nextElement() );
       System.out.println (str);
   }
```

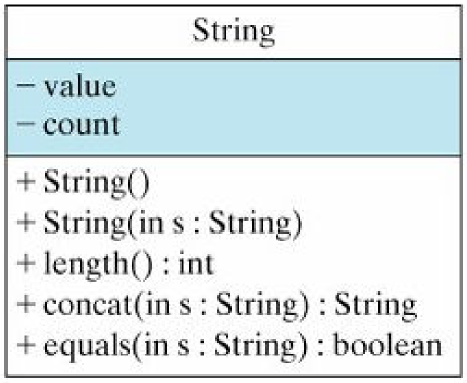
The following code adds one hundred strings to a hashtable. Each string is indexed by an Integer, which wraps the int primitive data type. Individual elements can be returned, or the entire list can be displayed. Note that hashtables don't store keys sequentially, so there is no ordering to the list.

```
import java.util.*;

public class hash {
  public static void main (String args[]) throws Exception {
    // Start with ten, expand by ten when limit reached
    Hashtable hash = new Hashtable(10,10);

    for (int i = 0; i <= 100; i++)
    {
        Integer integer = new Integer ( i );
        hash.put( integer, "Number : " + i);
    }

    // Get value out again
    System.out.println (hash.get(new Integer(5)));
    // Get value out again
    System.out.println (hash.get(new Integer(21)));

    System.in.read();

    // Get all values
    for (Enumeration e = hash.keys(); e.hasMoreElements();)
    {
        System.out.println (hash.get(e.nextElement()));
        }
    }
}
```

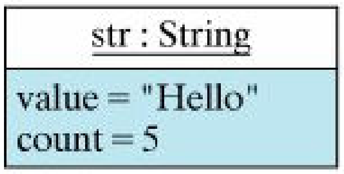
STRINGS

STRINGS API

- Strings are considered to be objects (instances of the string class)
- A string object is a sequence of the characters that make up the string including the methods used to manipulate the strings
- Strings are constants and once they are created their values cannot be changed
- The java.lang.String class is a direct subclass of Object (superclass)



- A string object (str) stores the value of the string (Hello) and the number of letters (count = 5)



CREATING STRINGS

Strings can be created in various ways (constructors):

1. System.out.println(“This is a string object”);
2. str = new String(“Hello”);

3. String s0 = “Hello”;
4. String s1 = new String(); //empty
5. String s2 = new String(charArray);**
6. String s3 = new String(charArray, 6, 4); //prints the 4 characters after the 6th one

** s2 and s3 are derived from an array, charArray which would have been declared before:

char charArray[ ] = {‘h’, ’e’, ’l’, ’l’, ’o’,’ ‘,’w’, ’o’, ‘r, ’l’, ‘d’}

STRING METHODS

LENGTH()

The length of a string is the number of letters, and spaces that make up a particular string. A number is assigned to each letter. This number is known as the index (zero indexed – starts from 0).

String s1 = “Hello World”;

s1.length() would return 11

Index	0	1	2	3	4	5	6	7	8	9	10
Letter	H	e	l	l	o	(space)	W	o	r	l	d

CONCAT() & ‘+’ SIGN

String lastName = "Onassis";
String jackie = new String("Jacqueline " + "Kennedy " + lastName);
System.out.println("Jacqueline".concat(lastName));

Note: Using the ‘+’ operator for concatenation is an example of Operator overloading

What is the output of the following?

System.out.println("The sum of 5 and 5 = "+ (5 + 5));

System.out.println("The sum of 5 and 5 = "+ 5 + 5);

CHARAT()

This method is used to retrieve the particular character stored at the required index, e.g.

String anotherExample = "Niagara Falls!";
char aChar = anotherExample.charAt(8); // this will return F

N	i	a	g	a	r	a	(space)	F	a	l	l	s	!
0	1	2	3	4	5	6	7	8	9	10	11	12	13

GETCHARS()

This method copies a part of a string and places it in an array at a particular index:

String s1 = "Niagara Falls!";
char charArray[ ] = new char[5];
s1.getChars(0, 5, charArray, 0); //0 is first index, 5 is last index (one more than required)

N	i	a	g	a	r	a	(space)	F	a	l	l	s	!
0	1	2	3	4	5	6	7	8	9	10	11	12	13

charArray

N	i	a	g	a
0	1	2	3	4

COMPARETO()

The method compareTo() compares one string with another. The comparison is based on the ASCII/Unicode value of the characters since all the letters of the alphabet are represented according to a particular number. This method returns 0 if the strings are equal, -1 if the string that invokes method is less than the argument and +1 if it is greater.

String s1 = new String("hello");
String s2 = "goodbye";

s1.compareTo(s2) would return -1

STARTSWITH() & ENDSWITH()

The method startsWith() takes a string as an argument and checks whether the string starts with the defined argument. endWith() works just the same but it checks the ending part of a string:

string.startsWith(“st”); // will check if the string starts with “st”
string.endsWith(“op”); // checks if string ends with “op”

Alternatively one can set the index at which the method will check for a particular string:

string.startsWith(“art”, 2); //mathches the string “art” starting from index 2

REPLACE()

This method will traverse the string and replace any character with the required character:

String s1 = new String (“hello”);
s1.replace(‘l’, ‘L’) would return “heLLo”

TRIM()

The method trim() removes any blank spaces from a string:

String s1 = new String (“   hello   ”);
s1.trim() would return “hello”

TOLOWERCASE() & TOUPPERCASE()

These two methods change the case of a string of characters:

String s1 = new String (“hello”);
String s2 = new String (“WORLD”);

s1.toUpperCase() would return “HELLO”
s2.toLowerCase() would return “world”

TOCHARARRAY()

This method takes a string and transforms it into a new character array:

String s1 = new String (“hello”);
char charArray[] = s1.toCharArray();

charArray

h	e	l	l	o
0	1	2	3	4

SUBSTRING()

This method searches the string for a pre-determined substring:

String anotherExample = "Niagara Falls!";
String anotherExampleTwo = anotherExample.substring(8, 12); // this will return ‘Fall’

N	i	a	g	a	r	a	(space)	F	a	l	l	s	!
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Note that the letter ‘s’ at index 12 is ignored (not returned by method). If only one parameter is given, the substring returned would be till the last index of the string.

EQUALS()

This method returns a Boolean value after checking similarities between two strings, e.g.

```
if(str1.equals(str2))
  System.out.println("str1 equals str2");
else
  System.out.println("str1 does not equal str2");
```

VALUEOF()

This method is used to convert data into strings:
String number = String.valueOf(128); // Creates "128"
String truth = String.valueOf(true); // Creates "true"
String bee = String.valueOf('B');   // Creates "B"
String pi = String.valueOf(Math.PI); // Creates "3.14159"

INDEXOF() & LASTINDEXOF()

These instance methods are used to find the **index** of a particular character in a string:

```
String string2 = "Hello";
String string3 = "World";
String string4 = string2 + " " + string3;
```

string2.indexOf('o') returns 4	string2.lastIndexOf('o') returns 4
string3.indexOf('o') returns 1	string3.lastIndexOf('o') returns 1
string4.indexOf('o') returns 4	string4.lastIndexOf('o') returns 7

indexOf() returns the first occurrence while lastIndexOf() returns the last occurrence. One can also specify the starting point at which the string is traversed, e.g.

string4 = “Hello World”

Index	0	1	2	3	4	5	6	7	8	9	10
Letter	H	e	l	l	o	(space)	W	o	r	l	d

```
string4.indexOf('o', 5)    returns 7 //string traversed from 5 upwards (index increments)
string4.lastIndexOf('o', 5) returns 4 //string is traversed starting from  index 5 downwards (index decrements)
```

FORMAT()

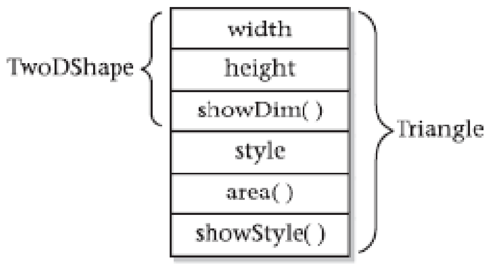
This method allows the creation of formatted string which can be re-used, useful when printing formatted numbers:

```
System.out.printf(“The value of the float variable is … “);
&
String fs;
fs = String.format(“The value of the float variable is … “);
System.out.println(fs);
```

OOP CONCEPTS

INHERITANCE

- One of the foundation principles of OOP
- Allows creation of hierarchical classification
- You can have a generic class which contains particular attributes which can define common traits
- This class(super class) can be inherited by other subclasses which in turn add their own attributes and methods
- Therefore a sub class is a special version of the super class
- In Java we can 'extend' a class by applying inheritance while declaring a particular class:
  e.g. Suppose we have a class TwoDShape,
  class Triangle ***extends*** TwoDShape { …
  class Triangle will inherit all the properties of the TwoDShape (e.g. width and height) and add its own attributes.



MEMBER ACCESS AND INHERITANCE

Member access control is achieved through the following access specifiers:

1. Public: can be accessed by any other code, including methods in other classes
2. Private: can be accessed **only** by members of its class
3. Protected: is accessible within its package (a group of related classes) and to all subclasses, including subclasses in other packages.

Note: *If no access specifier is declared, the default specifier is used and it is the 'Public'.*

When we have a super class which contains attributes which have been declared as 'Private', any subclass will not be able to access them, e.g.

```
class TwoDShape {

private double width; // these are

private double height; // now private

…
```

If we create a subclass Triangle which extends the superclass TwoDShape,

```
class Triangle extends TwoDShape {

String style;

double area() {

return width * height / 2;

}
```

This will return an error as width and height cannot be accessed by the subclass.

A solution to this problem would be using Accessor Methods which have to be defined in the Superclass, e.g.

```
double getWidth() {

       return width;

       }

double getHeight() {

       return height;

       }

…
…

class Triangle extends TwoDShape {

String style;

double area() {

return getWidth() * getHeight() / 2;

}

…

…
```

INHERITANCE AND CONSTRUCTORS

Both the Superclass and the subclass can have their own constructors; however the constructors of the superclass are responsible for the objects constructed by the superclasss and similarly for the subclass. A subclass can call a superclass constructor by using the 'super (*parameter list*)', e.g.

```
class TwoDShape {

private double width;

private double height;
```

```
// Parameterized constructor.

TwoDShape(double w, double h) {

width = w;

height = h;

}

…

…

// A subclass of TwoDShape for triangles.

class Triangle extends TwoDShape {

private String style;

Triangle(String s, double w, double h) {

super(w, h);

style = s;

}
```

Calling a constructor located in the Superclass using Super ( )

USING SUPER() AND CONSTRUCTORS

In the following example we will use constructors and each time we are accessing the superclass TwoDShape. Note that the call to the superclass has to be the first statement of the subclass constructor.

```
...
Triangle() {

       super();

       style = "null";

       }

// Constructor

Triangle(String s, double w, double h) {

       super(w, h); // call superclass constructor

       style = s;

       }

// Construct an isosceles triangle.

Triangle(double x) {

       super(x); // call superclass constructor

       style = "isosceles";
}
...
```

MULTI-LEVEL HIERARCHY – MORE LEVELS OF INHERITANCE

We already used the TwoDShape example and Triangle to show inheritance. If we need to create another subclass of triangle, e.g. ColourTriangle, this time we are extending the subclass Triangle.

```
…

class TwoDShape {
       private double width;
       private double height;

…

// Extend TwoDShape.
class Triangle extends TwoDShape {
       private String style;

…

// Extend Triangle.
class ColourTriangle extends Triangle {
       private String color;

…
```

This time if the super() is used, it will be able to call the constructors found in Triangle. The super() can access directly the closest superclass in the hierarchy.

METHOD OVERRIDING

When we have a subclass which contains a method similar to the superclass, then the method of the subclass will override the one defined in the superclass. However one can still access the superclass method by using the super().

```
class A {
       int i, j;

...
...

// display i and j

       void show() {

              System.out.println("i and j: " + i + " " + j);

       }

…

class B extends A {

       int k;
...
...
void show() {
       System.out.println("k: " + k);
       }
```

Method show() is overridden

POLYMORPHISM

At this point we can talk about polymorphism. Imagine we have a superclass Animal, and various subclasses, e.g. bird, frog, dog. Imagine also that the superclass Animal implements a method, move(x, y), where x and y are co-ordinates. Each subclass will implement the same method depending on the object type. This is an example of polymorphism. Points to take into consideration:

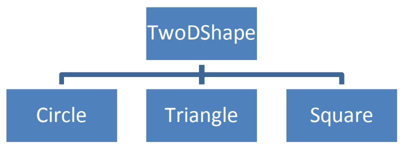
- Polymorphism enables programmers to deal in generalities and let the execution-time environment handle the specifics. Programmers can command objects to behave in manners appropriate to those objects, without knowing the types of the objects (as long as the objects belong to the same inheritance hierarchy).
- Polymorphism promotes extensibility: Software that invokes polymorphic behaviour is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without requiring modification of the base system. Only client code that instantiates new objects must be modified to accommodate new types.

ENCAPSULATION

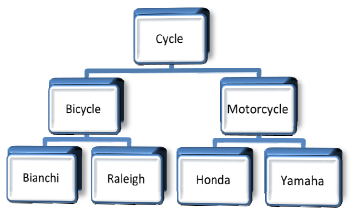
This defines the mechanism by which the code and the data that this code manipulates are bound together forming a sort of protective shell (black box). An Object is created by encapsulation of the code and the data. Remember that the code of an object can be either public or private. Points to consider and remember:

- The basic unit of encapsulation is the class (in Java)
- A class defines the form (template) of an object
- Objects are instances of a class
- The code and data that form a class are called members of the class
- The data defined by the class are referred to as *member variables* or *instance variables*
- The code that operates on that data is referred to as *member methods* (methods/subroutine)

** POLYMORPHISM, ENCAPSULATION AND INHERITANCE – A SUMMARY



Methods
+
Data



When properly applied polymorphism, encapsulation and inheritance combine to produce a programming environment that support the development of far more robust and scalable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes. Polymorphism allows you to create clean, sensible, readable and resilient code.

ABSTRACT CLASSES

An abstract class has the following properties:

1. It cannot be instantiated (no objects – would result in a compile time error)), only subclassed
2. Might contain one or more abstract methods
3. A subclass of an abstract class must implement all the abstract methods, else it has to be declared as abstract too
4. Does not define a complete implementation, a class may be declared abstract even it contains no abstract methods. It could, for example, contain instance variables that are common to all its subclasses

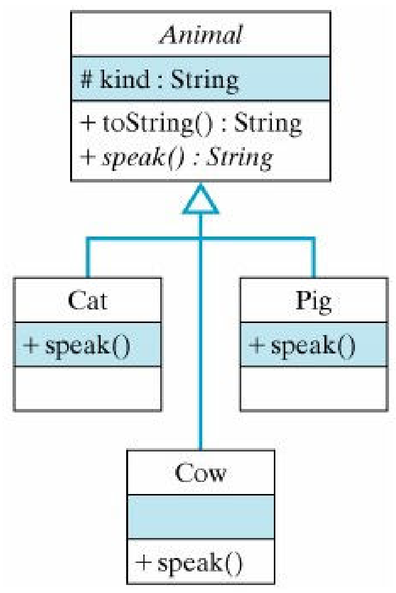
e.g. abstract class TwoDShape {

Declaring an abstract class

... }

abstract double area ( ); // an abstract method in the abstract class

Example 1: Animal Class (abstract)



Sample Code:

```
public abstract class Animal {
    protected String kind;            // Cow, pig, cat, etc.

    public Animal()  {  }
    public String toString() {
        return "I am a " + kind + " and I go " + speak();
    }
    public abstract String speak();   // Abstract method

   }
```

Coding a subclass:

```
public class Cat extends Animal {   // same for cow and pig
    public Cat() {
        kind = "cat";
    }
    public String speak() {
        return "meow";
    }

}
```

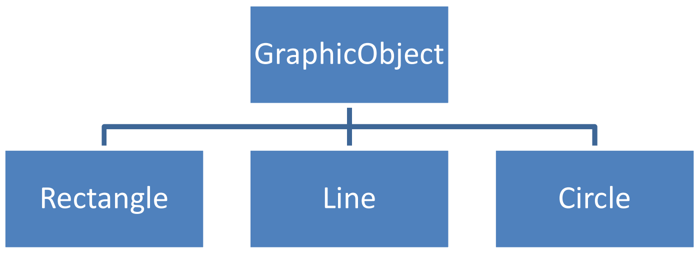
INHERITANCE AND POLYMORPHISM

Consider the following code:

```
   1. Animal animal = new Cow();
   2. System.out.println(animal.toString()); // A cow goes moo
   3. animal = new Cat();
   4. System.out.println(animal.toString()); // A cat goes meow
```

Line 1 creates an object, animal, of type Cow. Line 2 calls an inherited method (toString()). Line 3 creates a new object, Cat and on Line 4 calls the inherited method. This is an example of polymorphism where the same method is called for different types of the same object and at run-time it is decided what to return depending on the particular object on which the method is called.

Example 2: Graphic Object Class



Java Code for abstract class:

```
abstract class GraphicObject {
  int x, y;
  ...
  void moveTo(int newX, int newY) {
    ...
  }
  abstract void draw();   //abstract methods
```