Chapter: 2

Class, objects & method overloading

Class, objects, methods, Array-String, Bitwise operators, Access modifiers, method-overloading, recursion

2.1 Class in java

The ***methods*** and ***variables*** that constitute a ***class*** are called ***members of the class***. These members are also called ***instance variables***. The **objects** of a class are called ***instances*** of that class. Keyword ***class*** is used to create a class. The simplified form of a class in Java:

Simplest general form	***Example***
class classname { *// declare instance variables* **type** var1; **type** var2; ... **type** varN; *// declare methods* **type** method1(parameters) { */* body of method */* } **type** method2(parameters) { */* body of method */* } ... **type** methodN(parameters) { */* body of method */* }}	*/* defining a class of type Vehicle */* **class** Vehicle { **int** passengers; *// number of passengers* **int** fuelcap; *// fuel capacity in gallons* **int** mpg; *// fuel consumption in miles -gallon* } */* Declaring a Vehicle object called minivan */* **Vehicle** minivan = **new Vehicle**();

☞ Notice that the general form of a ***class*** does not specify a ***main()*** method. A ***main()*** method is required only if that ***class*** is the **starting point** for your program. Also, some types of Java applications, such as ***applets***, don't require a ***main()***.

☞ Notice in the example that ***Vehicle*** is used **twice** to declare an object one as ***type*** fashion ***(object's type)*** and another is as ***method*** fashion **[**Actually ***Vehicle()*** is the ***Vehicle*** class's ***default constructor see 2.5***] with the keyword "***new***". To declare an object of class type "***classname***" we use, **`classname object_name = new classname();`**
After this statement executes, ***object_name*** will be an instance of ***classname***.
[Each time you create an ***instance of a class***, you are creating an ***object*** that contains its own copy of each ***instance variable*** defined by the ***class***. Thus, the ***contents*** of the ***variables*** in one object can differ from the ***contents*** of the ***variables*** in another. There is ***no connection*** between the ***two objects*** except for the fact that they are both objects of the same type.]

⇨ Following two things happened when this line used to declare an object of type Vehicle, **Vehicle** minivan = **new Vehicle()**;

[1] It declares a ***variable*** called ***minivan*** of the class type ***Vehicle*** by " ***Vehicle minivan***". **This *variable* does not define an *object***. Instead, it is simply a **variable** that can ***refer*** to an **object**.

[2] The declaration creates a ***physical copy of the object*** and assigns to ***minivan*** a ***reference*** *to that object* by using ***new***.

⇨ ***In Java, all class objects must be dynamically allocated:*** The ***new*** operator *dynamically allocates* (i.e., allocates at run time) memory for an object and ***returns a reference*** to it. This **reference** is, more or less, the address in memory of the object allocated by ***new*** (reference is kind of **pointer**). This reference is then stored in a variable. ***new*** used here with class's ***default constructor***.

⇨ The two steps combined in the preceding statement can be rewritten like this to show each step individually:

```
Vehicle minivan;          // declare reference to object
minivan = new Vehicle();  // allocate a Vehicle object using class's default constructor Vehicle()
```

▶ The first line declares ***minivan*** as a ***reference*** **to an object** of type ***Vehicle***. Thus, ***minivan*** is a variable that can refer to an object, but it is not an object itself. At this point, ***minivan*** does not refer to an object.

▶ The next line creates a new ***Vehicle*** object and assigns a reference to it to ***minivan***. Now, ***minivan*** is linked with an object.

☞ To access these instance variables of an object, use the dot (**.**) operator. It links the name of an object with the name of a member. The general form, **`object.member`** Example: `minivan.fuelcap = 16;`

❒ ***Example 1:*** complete code that uses the ***Vehicle*** class

➲ The file that contains this program is ***VehicleDemo.java*** because the ***main()*** is in the class called ***VehicleDemo***, not the class called ***Vehicle***.

➲ When you compile this program, two ***.class*** files will be created, one for ***Vehicle*** and one for ***VehicleDemo***. The Java compiler automatically puts each class into its own ***.class*** file.

➲ It is not necessary for both the ***Vehicle*** and the ***VehicleDemo*** class to be in the ***same source file***. Each class can be in its own file, called ***Vehicle.java*** and ***VehicleDemo.java***, respectively.

➲ To run this program, you must execute ***VehicleDemo.class***.

```
class Vehicle{      int passengers;
                    int fuelcap;
                    int mpg; }

class VehicleDemo {          /* main class, starting point of the program */
public static void main(String args[]) {
        Vehicle minivan = new Vehicle();
        int range;

        minivan.passengers = 7;
        minivan.fuelcap = 16;
        minivan.mpg = 21;

        range = minivan.fuelcap * minivan.mpg;
        System.out.println("Minivan can carry " + minivan.passengers +
                                    " with a range of " + range);
        } }
```

2.2 Reference Variables and Assignment

A ***primitive type*** variables (i.e. ***int***, ***double*** etc) and ***object reference variables*** act differently in an assignment operation.

❒ When you assign one ***primitive-type*** variable to another, the situation is straightforward. The variable on the left receives a copy of the value of the variable on the right.

❒ When you assign one ***object reference variable*** to another, the situation is a bit more complicated because you are changing the ***object*** that the reference variable refers to. The effect of this difference can cause some counterintuitive results. For example,

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
```

Looks like ***car1*** and ***car2*** refer to different objects, but this is not the case. Instead, ***car1*** and ***car2*** will both refer to the **same object**. The assignment of ***car1*** to ***car2*** simply makes ***car2*** refer to the ***same object*** as does ***car1***. Thus, the object can be acted upon by either ***car1*** or ***car2***. For example, after the assignment ***`car1.mpg = 26;`*** executes, both of following ***`println()`*** statements display the same value: **26**.

```
System.out.println(car1.mpg);
System.out.println(car2.mpg);
```

☞ Although ***car1*** and ***car2*** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to ***car2*** simply changes the object to which ***car2*** refers. The object referred to by ***car1*** is unchanged.

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
Vehicle car3 = new Vehicle();
car2 = car3; // now car2 and car3 refer to the same object.
```

2.3 Methods and returning from methods

Methods are similar to C/C++'s "***functions***". The general form of a Java method is:

`ret-type name( parameter-list ) {` */* body of method */* ***`}`***

The ***ret-type*** is the return type of the function. If the function returns no value then the ***ret-type*** must be ***`void`***.

♠ NOTE: In C/C++ we can define a ***function*** inside of a class or outside of a class. We used the **scope resolution operator** (**`::`**) to define a ***function*** outside of class. But in Java it is ***un-common***. In Java ***methods*** are usually defined **inside** a class, so it is un-common to use the **scope resolution operator** (**`::`**) to define a method outside of its class.

❒ ***Returning from a Method:*** there are two conditions that cause a method to return
- **[1]** First, when the method's ***closing curly brace*** is encountered.
- **[2]** Second, when a ***return statement*** is executed.

❒ ***Return :*** There are two forms of ***return:***
- **[a]** One for ***void*** methods (those that do not return a value). The form is, **`return;`**
- **[b]** One for methods ***which return values***. The form is, **`return value;`**

☛ Second form of ***return*** can be used only with methods that have a **non-void return type**. Furthermore, a **non-void method** must return a value by using this "**`return value;`**" form of ***return***. Example of The second form " **`return value;`**":

```
int range() { return x * y; }.
```

➲ Notice that ***`range()`*** has a return type of ***`int`***. i.e. it will return an integer value to the caller. The **return type** of a method is important because the type of data returned by a method ***must be compatible*** with the **return type** specified by the method. Thus, if you want a method to return data of type ***double***, its return type must be type ***double***. Eg:

```
double devide(int x, int y) { return (double) (x /y); }
```

Here a ***type cast*** is used to return "***double*** value" from ***int*** values (which are ***int*** parameters of the method)

☛ It is permissible to have multiple ***return statements*** in a method, however, because having too many ***exit points*** (i.e. return statements) in a method can ***destructure*** code. A well-designed method has well-defined ***exit points***.

2.4 Methods with parameters

A value passed to a method is called an argument. Inside the method, the variable that receives the argument is called a parameter. ***Example: `devide(int x, int y)`*** here ***x*** and ***y*** are **parameters** but in ***`devide(2, 3)`*** ***2*** and ***3*** are **arguments.**

☞ A parameter is within the ***scope*** of its ***method***, and aside from its special task of **receiving an argument**, it acts like any other local variable.

☞ A method can have more than one parameter which separatef one from the next with a ***comma***. Eg: ***devide(int x, int y)***

☞ When using **multiple parameters**, each parameter specifies its ***own type***, which can differ from the others. For example, this is perfectly valid:

```
int myMeth(int a, double b, float c){  }
```

2.5 Constructor

A constructor ***initializes*** an ***object*** (i.e. to give ***initial values*** to the ***instance variables*** defined by the class) when it is created or to perform any other **startup procedures** required to create a fully formed ***object***. It has the ***same name as its class*** and is syntactically ***similar to a method***. However, constructors have **no explicit return type**.

❒ ***Java's Default constructor:*** All classes have **constructors**, whether you define one or not, because Java automatically provides a ***default constructor that initializes all member variables to their default values***, which are ***`zero`***, ***`null`***, and ***`false`***, for ***numeric*** types, ***reference*** types, and ***booleans***, respectively. However, once you define your own **constructor**, the **default constructor** is no longer used. ***Example:***

```
class MyClass { int x;
                MyClass(){ x = 10; } }
```

```
class ConsDemo { public static void main(String args[]) {
        MyClass t1 = new MyClass();     /* parameter-less initialization */
        MyClass t2 = new MyClass();     /* parameter-less initialization */
        System.out.println(t1.x + " " + t2.x); }}
```

☞ Constructor is called by ***new*** when an object is created. For example, in the line ***`MyClass t1 = new MyClass();`*** the constructor ***`MyClass()`*** is called on the ***`t1`*** object, giving ***`t1.x`*** the value ***`10`***. The same is true for ***`t2`***. After construction, ***`t2.x`*** has the value ***`10`***. Thus, the output from the program is ***`10 10.`***

⇨ i.e., in **general from** of object in ***2.1***, **`classname object_name = new classname();`** in the part ***"new classname()"*** ***`classname()`*** is the class's ***default constructor***.

❒ ***Parameterized constructor:*** It has the same name as its ***class*** and is ***syntactically similar*** to a **parameterized method. Parameterized constructor** is used to initialize objects on-spot (by setting values of member variables from constructor), for example,

class MyClass { **int** x; MyClass(**int** i){ x = i; } }	**class** PeramConsDemo { **public static void main(String args[])** { MyClass t1 = **new** MyClass(10); */* parameter-less initialization */* MyClass t2 = **new** MyClass(88); */* parameter-less initialization */* **System.out.println**(t1.x + " " + t2.x); }}

☞ The ***MyClass()*** **constructor** defines one parameter called ***i***, which is used to initialize the instance variable, ***x***. Thus, when ***MyClass t1 = new MyClass(10);*** executes, the value ***10*** is passed to ***i***, which is then assigned to ***x***.

Constructor without parameter	***Parameterized constructor***
class Vehicle{ **int** passengers; **int** fuelcap; **int** mpg; } **class** VehicleDemo {**public static void main(String args[])** { **Vehicle** minivan = **new** Vehicle(); minivan.passengers = 7; minivan.fuelcap = 16; minivan.mpg = 21; //. . . . . . }}	**class** Vehicle{ **int** passengers; **int** fuelcap; **int** mpg; Vehicle(int p, int f, int m){ passengers = p; Fuelcap = f; Mpg = m; } **class** VehicleDemo {**public static void main(String args[])** { **Vehicle** minivan = **new Vehicle**(7, 16, 21); //. . . . . . }}

☞ ***Vehicle minivan = new Vehicle(7, 16, 21);*** is much more easier than using "**.**" for each member variable Eg: ***minivan.mpg = 21;***

☞ ***Difference with C++:*** In C++ we would use, **`Vehicle minivan(7, 16, 21);`** to initialize the object minivan instantly.

2.6 General form of "new" and details about object declaration (Recall C/C++'s 10.11)

The new operator has this general form: **`class_variable = new class-name(arg-list);`**

⇨ Here, ***class_variable*** is a variable of the class type being created. The ***class-name*** is the name of the class that is being **instantiated**. The class name followed by a ***parenthesized argument list*** (which can be empty) i.e. ***class-name(arg-list)*** specifies the ***constructor*** for the class. If a class does not define its own **constructor**, ***new*** will use the ***default constructor*** supplied by Java.

⇨ The ***new*** operator returns a ***reference*** to the newly created object, which (in this case) is assigned to ***class-var***.

⇨ A ***run-time exception*** will occur if ***new*** is unable to allocate memory for an object because *insufficient memory space*.

2.7 Garbage collection and `finalize()`

In C++, to free allocated memory space after use of an object we used the keyword "***delete***" and also the ***destructor*** is called when the object goes out of scope. In Java there is **no *destructor*** , the allocated memory cleanup is performed by "**garbage collection**". It is ***unpredictable*** when "***garbage collector***" is called by **Java run-time system,** it is called automatically when it needed.

❒ Java's garbage collection system reclaims objects automatically—occurring transparently, behind the scenes, without any programmer intervention. It works like this: ***When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is released***. This recycled memory can then be used for a subsequent allocation.

❒ **Garbage collection** takes time, so the **Java run-time system** does it only when it is appropriate. It will not occur simply because one or more objects exist that are no longer used. Thus, you can't know precisely when **garbage collection** will take place.

☞ For efficiency, the garbage collector will usually run only when two conditions are met:

▶ There are objects to recycle, and ▶ There is a need to recycle them.

❒ **new *is not used with primitive types (unlike C++) in Java:*** In C++ we used ***new*** for ***int***, ***float*** etc. Recall ***C/C++ 10.11***. In Java you don't need to use ***new*** for ***variables of the primitive types***, such as ***int*** or ***float***.

▶ Java's ***primitive types*** are not implemented as ***objects***, they are implemented as "***normal***" variables. A variable of a ***primitive type*** actually contains the value that you have given it.

▶ But object variables are ***references (pointers)*** to the object. This layer of indirection (and other object features) adds overhead to an object.

O ***finalize(): `finalize()`*** is the Method which can be called just before an object's final ***destruction*** by the ***garbage collector***, it can be used to ensure that an object ***terminates cleanly***. Eg: use ***finalize()*** to make sure that an open file owned by that object is closed.

☛ To add a ***finalizer*** to a class, you simply define the ***finalize()*** and inside ***finalize()*** specify those actions that must be performed before an object is *destroyed*. The Java run-time system calls that *finalizer* whenever it is about to recycle an object of that class.

☛ ***General form*** *of finalize( ):* **`protected void finalize(){ /* finalization code here */ }`**
Here, the keyword ***protected*** is an ***access specifier***.

☛ ***`finalize()`*** is called just before **garbage collection** hence unpredictable (i.e. it is not called when an object goes out of scope). For example, if your program ends before garbage collection occurs, ***`finalize()`*** will not execute. Therefore, it should be used as a "**backup**" procedure to ensure the proper handling of some resource, or for special-use applications.

☛ ***Java does not have "destructors":*** Although it is true that the ***`finalize()`*** approximates the function of a ***destructor***, it is not the same, a C++ destructor is always called just before an object goes out of scope, but you can't know when ***`finalize()`*** will be called for any specific object. Frankly, because of Java's use of ***garbage collection***, there is little need for a ***destructor***.

❒ ***Example:*** To demonstrate garbage collection via ***finalize()***, you often need to create and destroy a large number of objects ,

```
class FDemo {int x;                          /* class that contain finalize() */
             FDemo(int i) { x = i; }         /* constructor */

                    // called when object is recycled
             protected void finalize(){ System.out.println("Finalizing " + x);}

                    // generates an object that is immediately destroyed
             void generator(int i){ FDemo ob = new FDemo(i); }
             }

class Finalize { public static void main(String args[]) {
             int count;
             FDemo ob = new FDemo(0);        /* assigning initial value */

                    /* Now, generate a large number of objects. At some point, garbage collection will occur. */
             for(count=1; count < 100000; count++) ob.generator(count);  }}
```

2.8 The this reference (C/C++ this pointer, Recall C/C++ 10.8)

When a method is called, it is automatically passed an **implicit argument** that is a reference to the invoking object (that is, the object on which the method is called). This reference is called this. To understand this, first consider a program that creates a class called Pwr that computes the result of a number raised to some integer power:

```
class Pwr {double b;
        int e;
        double val;
        Pwr(double base, int exp) { b = base;
                                    e = exp;
                                    val = 1;
                                    if(exp==0) return;
                                    for( ; exp>0; exp--) val = val * base;}
        double get_pwr() { return val; }
        }
```

```
class DemoPwr { public static void main(String args[]) {
                Pwr x = new Pwr(4.0, 2);
                Pwr y = new Pwr(2.5, 1);
                Pwr z = new Pwr(5.7, 0);

System.out.println(x.b + " exp " + x.e + " is " + x.get_pwr());
System.out.println(y.b + " exp " + y.e + " is " + y.get_pwr());
System.out.println(z.b + " exp " + z.e +" is " + z.get_pwr());
}}
```

❒ Within a method, the other members of a class can be accessed directly, without any object or class qualification. Thus, inside **`get_pwr( )`**, the statement **`return val;`** means that the copy of ***val*** associated with the invoking object will be returned.

☞ However, the same statement can also be written like this: **`return this.val;`**

✓ Here, ***this*** refers to the object on which ***get_pwr()*** was called. Thus, ***this.val*** refers to that object's copy of ***val***. Writing the statement without using ***this*** is really just ***shorthand***.

☞ Here is the entire ***Pwr*** class written using the ***this*** reference:

```
class Pwr { double b, val; int e;
        Pwr(double base, int exp) { this.b = base; this.e = exp; this.val = 1;
                                    if(exp==0) return;
                                    for( ; exp>0; exp--) this.val = this.val * base;
                                    }
        double get_pwr() { return this.val; }
        }
```

☞ However, ***this*** has some important uses. For example, the Java syntax permits ***the name of a parameter or a local variable*** to be the same as ***the name of an instance variable***. When this happens, the ***local name*** hides the ***instance variable***. You can gain ***access to the hidden instance variable*** by referring to it through **`this`**.

For Example: Syntactically valid way to write the ***Pwr()*** constructor.

```
Pwr(double b, int e) {  this.b = b;  this.e = e;  val = 1;
                        if(e==0) return;
                        for( ; e>0; e--) val = val * b; }
```

✓ In this version, the names of the parameters are the same as the names of the instance variables, thus hiding them. However, ***this*** is used to "uncover" the instance variables.

2.9 Arrays : One-Dimensional Arrays

Although arrays in Java can be used just like arrays in other programming languages, they have one special attribute: they are implemented as objects. By implementing arrays as objects, several important advantages are gained, not the least of which is that unused arrays can be garbage collected.

❒ To declare a one-dimensional array in Java we the similar object-declaration-form. General form:

`type array-name[ ] = new type[size];`

☞ Here, ***type*** declares the ***element type*** of the array. (The ***element type*** is also commonly referred to as the ***base type***.)

☞ The number of elements that the array will hold is determined by ***size***.

☞ Since **arrays** are implemented as **objects**, they are dynamically allocated using the ***new*** operator. The creation of an **array** is a two-step process.

- ➢ Declare an ***array reference variable***.
- ➢ Allocate memory for the ***array***, assigning a ***reference*** to that memory to the ***array variable***.

☞ To creates an ***int*** array of ***10*** elements and links it to an **array reference variable** named ***sample***:

```
int sample[] = new int[10];
```

➲ This declaration works just like an object declaration. The sample variable holds a reference to the memory allocated by new. Hence we can break down this statement in two parts:

```
int sample[];
sample = new int[10];
```

- ☑ In this case, when ***sample*** is first created, it refers to no physical object.
- ☑ It is only after the second statement executes that sample is linked with an array.

❒ ***Array initialization:*** Similar to C/C++ array initialization. Recall C/C++ 4.4

❒ ***Array Boundaries :*** Array boundaries are strictly enforced in Java; it's a ***run-time error*** to ***overrun*** or ***underrun*** the end of an array.

```
class ArrayErr{ public static void main(String args[]){ int sample[] = new int[10];
                                                         int i;
                                                         for(i = 0; i < 100; i = i+1) sample[i] = i; /* generate an array overrun */ }}
```

☞ As soon as ***i*** reaches ***10***, an ***ArrayIndexOutOfBoundsException*** is generated and the program is terminated.

❒ ***Sorting an Array:*** **Bubble sort** is similar to C/C++ 4.1 sorting example.

★ There are a number of different sorting algorithms. There are the ***quick sort***, the ***shaker sort***, and the ***shell sort***, to name just three. However, the best known, simplest, and easiest to understand is called the ***Bubble sort***.

★ Although the **Bubble sort** is good for small arrays, it is not efficient when used on larger ones. The best general-purpose sorting algorithm is the ***Quicksort***.

2.10 Multidimensional Arrays

❒ ***Two-Dimensional Arrays:*** A two-dimensional array is a list of one-dimensional arrays. Eg: A 2-D integer array ***abs*** of size ***10***, ***20***:

```
int abs[][] = new int[10][20];
```

To access point ***3***, ***5*** of array ***abs***, you would use ***abs[3][5]***.

☞ ***Example:*** A two-dimensional array ***table*** is loaded with the numbers ***1*** through ***12***.

```
int t, i;
table[][] = new int[3][4];  /* declaration of the array */
for(t=0; t < 3; ++t) {      for(i=0; i < 4; ++i){      table[t][i] = (t*4)+i+1;
                                                       System.out.print(table[t][i] + " "); }
                            System.out.println(); }
```

table[0][0] will have the value ***1***, ***table[0][1]*** the value ***2***, ***table[0][2]*** the value ***3***, and so on. The value of ***table[2][3]*** will be ***12***.

❒ ***Irregular Arrays:*** ***Irregular Arrays*** are more like C++'s ***multidimensional-unsized-arrays*** (Recall C/C++ 4.4) but they are ***not same***. In **Irregular** array the ***leftmost-dimension*** **[]** is fixed, not-empty and other **[]** are empty (Eg: rows fixed and columns vary). In C++'s **multidimensional-unsized-array** the ***leftmost-dimension*** **[]** is unfixed, empty while other **[]** are filled with fixed values (Eg: columns fixed and rows vary).

Irregular Arrays	***multidimensional-unsized-arrays***
1 1 2 3 5 1 1 1 3 2 1 3 4 5 5 8 9 1 4 4	1 1 2 3 5 8 9 1 4 . . .

☞ When we allocate memory for a multidimensional array, we usually specify the both dimensions at the same time. For example:

```
int table[][] = new int[3][4]
```

However we can ***specify only the memory for the first (leftmost) dimension and allocate the remaining dimensions separately***. Since multidimensional arrays are implemented as arrays of arrays, the length of each array is under our control. Consider previous statement:

```
int table[][] = new int[3][];
table[0] = new int[4];
table[1] = new int[4];
table[2] = new int[4];
```

☞ But separate specification is not useful when we deal with ***regular array*** (row and columns are fixed). Separate specification of dimension sizes of an array is very helpful for ***irregular array***. For example:

```
int var_ary[][] = new int[4][];
var_ary[0] = new int[10];    /* specifying first row of length  10 */
var_ary[1] = new int[15];    /* specifying 2nd row of length  15 */
var_ary[2] = new int[3];     /* specifying 3rd  row of length  3 */
var_ary[3] = new int[4];     /* specifying 4th  row of length  4 */
```

☞ The use of ***irregular (or ragged) multidimensional arrays*** can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), an irregular array might be a perfect solution.

❒ ***Arrays of Three or More Dimensions:*** Here is the general form of a multidimensional array declaration:

```
type name[ ][ ]...[ ] = new type[size1][size2]...[sizeN];
```

For example, the following declaration creates a 4 × 10 × 3 three-dimensional integer array.

```
int multidim[][][] = new int[4][10][3];
```

❒ ***Initializing Multidimensional Arrays:*** A multidimensional array can be initialized by enclosing each dimension's initializer list within its own set of curly braces **(it is different form C/C++'s array initialization, *extra curly braces not used*. Recall C/C++ 4.4).** For example, the general form of array initialization for a two-dimensional array is shown here:

```
type-specifier array_name[ ] [ ] = {  { val, val, val, ..., val },
                                      { val, val, val, ..., val },
                                         . . . . . .
                                      { val, val, val, ..., val }  };
```

Eg: Array sqrs with the numbers 1 through 4 and their squares:

```
int sqrs[][] = {  { 1, 1 },
                  { 2, 4 },
                  { 3, 9 },
                  { 4, 16 } }
```

❖ Here, ***val*** indicates an initialization value. Each ***inner block*** designates a row. Within each row, the first value will be stored in the first position of the ***subarray***, the second value in the second position, and so on.

❖ Notice that ***commas*** separate the initializer blocks and that a ***semicolon*** follows the closing **}**.

❒ ***Alternative Array Declaration Syntax:*** There is a second form that can be used to declare an array: ***type[] var-name;***
Here, **[]** follow the ***type*** specifier, not the ***name*** of the array variable. For example, following two declarations are equivalent:

```
char table[][] = new char[3][4];
char[][] table = new char[3][4];
```

☞ This alternative declaration form useful when we declare several arrays at the same time. For example, to create three arrays:

```
int[] nums, nums2, nums3;
```
is more easier than
```
int nums[], nums2[], nums3[];
```

☞ The alternative declaration form is also useful when specifying an array as a ***return type*** for a ***method***. For example,

```
int[] someMeth( ) { ... }
```

This declares that ***someMeth()*** returns an array of type ***int***.

2.11 Array (advanced)

❒ ***Assigning Array References (more like assigning pointers of C/C++: copying address):*** As with other objects, when you assign one array reference variable to another, you are simply changing what object that variable refers to. You are ***not causing*** a ***copy of the array*** to be made, ***nor are you causing*** the ***contents of one array to be copied to the other***. (i.e. same object with two different names, "not two different objects with equal values". The object can be modified via both of the names.) For example,

```
int nums1[] = new int[10];
int nums2[] = new int[10];
for(i=0; i < 10; i++) nums1[i] = i;          //putting values to nums1
for(i=0; i < 10; i++) nums2[i] = -i;         //putting values to nums2
```

```
OUTPUT :
Here is nums1: 0 1 2 3 4 5 6 7 8 9
Here is nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Here is nums2 after assignment: 0 1 2 3 4 5 6 7 8 9
nums1 after change through nums2: 0 1 2 99 4 5 6 7 8 9
```

```
System.out.print("Here is nums1: "); for(i=0; i < 10; i++) System.out.print(nums1[i] + " "); System.out.println();
System.out.print("Here is nums2: "); for(i=0; i < 10; i++) System.out.print(nums2[i] + " "); System.out.println();

nums2 = nums1;                               // now nums2 refers to nums1
System.out.print("Here is nums2 after assignment: "); for(i=0; i < 10; i++) System.out.print(nums2[i] + " "); System.out.println();

nums2[3] = 99;                               // operating on nums1 array through nums2
System.out.print("nums1 after change through nums2: "); for(i=0; i < 10; i++) System.out.print(nums1[i] + " "); System.out.println();
```

☞ As the output shows, after the assignment of ***nums1*** to ***nums2***, both array reference variables refer to the same object.

❒ ***Using the length Member (instance member of an array object):*** Because arrays are implemented as ***objects***, each array has associated with it a ***length instance variable*** that contains the **number of elements that the array can hold**. (In other words, ***length*** contains the ***size of the array***.) Here is a program that demonstrates this property:

```
int list[] = new int[10];
int nums[] = { 1, 2, 3 };
int table[][] = {    {1, 2, 3},
                     {4, 5},
                     {6, 7, 8, 9} };        // a variable-length table

System.out.println("length of list is " + list.length);
System.out.println("length of nums is " + nums.length);
System.out.println("length of table is " + table.length);
System.out.println("length of table[0] is " + table[0].length);
System.out.println("length of table[1] is " + table[1].length);
System.out.println("length of table[2] is " + table[2].length);
System.out.println();
```

```
// use length to initialize list
for(int i=0; i < list.length; i++) list[i] = i * i;
System.out.print("Here is list: ");

// now use length to display list
for(int i=0; i < list.length; i++) System.out.print(list[i] + " ");
System.out.println();
```

```
OUTPUT:  length of list is 10
         length of nums is 3
         length of table is 3
         length of table[0] is 3
         length of table[1] is 2
         length of table[2] is 4
         Here is list: 0 1 4 9 16 25 36 49 64 81
```

☛ Pay special attention to the way ***length*** is used with the **two-dimensional array** ***table***. Since a two-dimensional array is an ***array of arrays***.

▶ Thus, when "***table.length***" is used, it obtains the ***number of arrays stored in table***, which is ***3*** in this case.

▶ To obtain the **length of any individual array** in ***table***, you will use an expression such as this, "***table[0].length***" which, in this case, obtains the ***length of the first array***.

☛ Also notice that ***list.length*** is used by the ***for*** loops to govern the number of ***iterations*** that take place. Since each array carries with it its own length, you can use this information rather than manually keeping track of an array's size.

▶ Keep in mind that the **value of length** has nothing to do with the ***number of elements that are actually in use***. It contains the ***number of elements that the array is capable of holding***.

☛ ***length*** simplifies many algorithms by making certain types of array operations easier—and safer—to perform. For example, the following uses ***length*** to copy one array to another while preventing an ***array overrun*** and its attendant ***run-time exception***.

```
class ACopy { public static void main(String args[]) {  int i;
                                                         int nums1[] = new int[10];
                                                         int nums2[] = new int[10];
                                                         for(i=0; i < nums1.length; i++) nums1[i] = i;

// compare array size using length and  copy nums1 to nums2
                                  if(nums2.length >= nums1.length) for(i = 0; i < nums1.length; i++) nums2[i] = nums1[i];
                                  for(i=0; i < nums2.length; i++) System.out.print(nums2[i] + " ");     }}
```

➢ Here, ***length*** helps perform two important functions.

☑ First, it is used to confirm that the target array is *large enough to hold* the contents of the source array.

☑ Second, it provides the *termination condition* of the ***for*** loop that performs the copy.

2.12 Data structure "stack" and "queue" with array

A data structure is a means of organizing data. The simplest data structure is the **array**, which is a linear list that supports random access to its elements. Arrays are often used as the underpinning for ***more sophisticated data structures***, such as ***stacks*** and ***queues***.

⇨ A ***stack*** is a list in which elements can be accessed in ***first-in, last-out (FILO) order*** only. Like a stack of plates on a table

⇨ A ***queue*** is a list in which elements can be accessed in ***first-in, first-out (FIFO) order*** only. Like a line at a bank.

❒ ***stacks and queues are data engines:*** What makes data structures such as ***stacks*** and ***queues*** interesting is that they ***combine storage*** for information with the ***methods*** that access that information. Thus, ***stacks*** and ***queues*** are **data engines** in which storage and retrieval are provided by the **data structure itself**, not manually by your program.

❒ ***Queue class:*** In general, queues support two basic operations: ***put*** and ***get***. Each ***put operation*** places a new element on the ***end*** of the ***queue***. Each ***get operation*** retrieves the next element from the ***front*** of the ***queue***.

☛ There are two basic types of queues—***circular*** and ***noncircular***. A ***circular queue reuses*** locations in the underlying array when elements are removed. A ***noncircular queue*** does ***not reuse*** locations and eventually becomes *exhausted*.

☛ Queue operations are **consumptive**: once an element has been retrieved, it cannot be retrieved again. The queue can also become full, if there is no space available to store an item, and it can become empty, if all of the elements have been removed.

❒ ***Example:*** For the sake of simplicity, we now consider a **noncircular queue**, but it can easily transform it into a **circular queue**.

➲ Although there are other ways to support a **queue**, the method we will use is based upon an **array**. That is, an array will provide the storage for the items put into the queue.

➲ *This array* will be accessed through two indices. The ***put index*** determines where the next element of data will be stored. The ***get index*** indicates at what location the next element of data will be obtained. Keep in mind that the ***get*** operation is consumptive, and it is not possible to retrieve the same element twice.

➲ First create the ***Queue*** class. The **constructor** for the ***Queue*** class creates a queue of a given size (Notice that the put and get indices are initially set to zero).

```
class Queue{
    char q[];                                  // this array holds the queue
    int putloc, getloc;                        // the put and get indices
    Queue(int size) { q = new char[size];  // allocate memory for queue
                      putloc = getloc = 0; }

                        // put a character into the queue
    void put(char ch) {
        if(putloc==q.length){ System.out.println(" – Queue is full.");
                              return; }
        q[putloc++] = ch; }

                        // get a character from the queue
    char get(){
        if(getloc == putloc){System.out.println(" – Queue is empty.");
                             return (char) 0; }
        return q[getloc++]; }
    }
```

```
OUTPUT:   Using bigQ to store the alphabet.
          Contents of bigQ: ABCDEFGHIJKLMNOPQRSTUVWXYZ

          Using smallQ to generate errors.

          Attempting to store Z
          Attempting to store Y
          Attempting to store X
          Attempting to store W
          Attempting to store V – Queue is full.
          Contents of smallQ: ZYXW – Queue is empty.
```

```
                    // Demonstrate the Queue class.
class QDemo { public static void main(String args[]) {
                  Queue bigQ = new Queue(100);
                  Queue smallQ = new Queue(4);
                  char ch;
                  int i;

System.out.println("Using bigQ to store the alphabet.");
for(i=0; i < 26; i++) bigQ.put((char) ('A' + i));       // put numbers

System.out.print("Contents of bigQ: ");      // retrieve and display
for(i=0; i < 26; i++) { ch = bigQ.get();
                        if(ch != (char) 0) System.out.print(ch); }
System.out.println("\n");

System.out.println("Using smallQ to generate errors.");
for(i=0; i < 5; i++) {
      System.out.print("Attempt to store " + (char) ('Z' - i));
      smallQ.put((char) ('Z' - i));
      System.out.println();  }
System.out.println();

                    // more errors on smallQ
System.out.print("Contents of smallQ: ");
for(i=0; i < 5; i++) { ch = smallQ.get();
                       if(ch != (char) 0) System.out.print(ch);}
}}
```

2.13 Enhanced *for* [For-Each Style *for* Loop]

It is so common where each element in an array must be examined, from start to finish. Because such "start to finish" operations are so common, Java defines **Enhanced *for*** loop that streamlines this operation.

❒ **Enhanced *for*** implements a "*for-each*" style loop. A ***for-each*** loop cycles **through a collection of objects**, such as an ***array***, in *strictly sequential fashion, from start to finish.* In early, Java didn't support this **Enhanced *for*** .However, ***JDK 5*** does support.

❒ The ***general form*** of the **for-each style *for*** is, ***for(type itr-var : collection) statement-block***

- Here, ***type*** specifies the type, and ***itr-var*** as iteration variable that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by ***collection***.
- With each iteration of the loop, the ***next element*** in the ***collection*** is retrieved and stored in ***itr-var***. Like a **stream**.
- The loop repeats until **all elements in the collection** have been obtained. For an array of size ***N***, it loops from ***0*** to ***N–1***.
- Because the ***itr-var receives/streams values*** from the collection, ***type*** must be the same as (or compatible with) the elements stored in the collection. (Eg: For arrays, ***type*** must be compatible with the **element type** of the array)

[There are various ***types of collections*** that can be used with the for, but the only type used in here is the ***array***. One of the most important uses of the for-each style for is to cycle through the contents of a ***collection*** defined by the ***Collections Framework***. The ***Collections Framework is a set of classes*** that implement various *data structures*, such as ***lists***, ***vectors***, ***sets***, and ***maps***.]

- ***Example:*** To understand ***enhanced for*** we compare it with **traditional *for***

▶ The entire array is read in strictly sequential order, by manually indexing the ***nums*** array by ***i***, the *loop control variable*.

▶ Furthermore, the ***starting*** and ***ending*** value for the *loop control variable*, and its ***increment***, must be explicitly specified.

❖ The **for-each style *for*** automatically cycles through an array in sequence from the ***lowest index*** to the ***highest***. There is no need of any ***starting*** or ***ending*** value, ***loop counter***, ***manually indexing*** array. Instead of *loop control variable* we use *iterator variable*, which directly access the array data (and ***streamlined***). It obtain one element at a time, in sequence, from beginning to end.

❖ With each pass through the loop, ***x*** is automatically given a value equal to the **next element** in ***nums***. Thus, on the first iteration, ***x*** contains ***19***, on the second iteration, ***x*** contains ***25***, and so on. Not only is the syntax ***streamlined***, it also prevents **boundary errors**.

```
int nums[] = { 19, 25, 33, 42, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++){     system.out.println("value : "+ nums[i]);
                               sum += nums[i];   }
```

```
int nums[] = { 19, 25, 33, 42, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x : nums){  system.out.println("value : "+ x);
                    sum += x;   }
```

❒ ***Enhanced for with break:*** It is possible to terminate the Enhanced **`for`** loop early by using a **`break`** statement. For example,

```
for(int x : nums) { System.out.println("Value is: " + x); sum += x;
                    if(x == 5) break; /* stop the loop when 5 is obtained */ }
```

❒ ***Enhanced for's iteration variable "itr-var" is "read-only":*** **For-each style for** loop's iteration variable is "read-only" as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can't change the contents of the array by assigning the iteration variable a new value. For example,

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for(int x : nums) { System.out.print(x + " ");
                    x = x * 10; /* no effect on nums */ }
```

The ***for*** loop increases the value of the iteration variable by a **factor** of ***1***, which has no effect on the underlying array ***nums***.

❒ ***Enhanced for with Multidimensional Arrays:*** Since in Java, **multidimensional arrays** consist of **arrays of arrays**, each iteration obtains the next array, not an individual element. Furthermore, the ***iteration variable (itr-var)*** in the enhanced for loop must be *compatible with the type of array being obtained.* For example, in the case of a **two-dimensional array**, the iteration variable must be a **reference** to a ***one-dimensional array***. Consider following uses nested for loops to obtain the elements of a two-dimensional array in row order, from first to last.

```
int sum = 0;
int nums[][] = new int[3][5];
      // give nums some values
for(int i = 0; i<3; i++) for(int j=0; j<5; j++) nums[i][j] = (i+1)*(j+1);
      // Use for-each for loop to display and sum the values.
for(int x[] : nums){       /* compatible type of one-dimensional array */
                    for(int y : x) { System.out.println("Value is: " + y);
                                     sum += y; }
                    }
```

☞ Notice how **x** is declared, "**`for(int x[] : nums) {`**" . It is a reference to a one-dimensional array of integers: each iteration of the ***for*** obtains the next array in **nums**, beginning with the array specified by ***nums[0]***.

☞ The **inner** ***for*** loop then cycles through each of these arrays, displaying the values of each element.

❒ ***Searching value*** with Enhanced ***for*** and other usage : Enhanced for can be used to search an unsorted array for a value.

```
int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
int val = 5;
boolean found = false;
```

```
// Use for-each style for to search nums for val.
for(int x : nums) { if(x == val) { found = true; break;}  }
if(found) System.out.println("Value found!");
```

☛ Enhanced for is perfect here because searching an unsorted array involves examining each element in sequence. ***(Of course, if the array were sorted, a binary search could be used, which would require a different style loop.)***

☛ Other usage of for-each style loops include computing an **average**, finding the **minimum** or **maximum** of a set, looking for **duplicates**, and so on.

2.14 Strings

String defines and supports **character strings**. In Java, **strings** are ***objects***. Constructing String is ***similar*** to constructing any other type of object: by using ***new*** and calling the ***`String constructor`***. For example:

```
String str = new String("Hello");
```

➲ You can also construct a String from another String. Eg: Consider previous ***str***, `String str2 = new String(str);`

➲ Another easy way to create a String is: `String str = "Hello";` ***str*** is initialized to the character sequence "***Hello***"

☞ ***String object*** can be used anywhere that a quoted string is allowed. Eg: String object as an argument to ***`println()`***

```
String str1 = new String("Java strings are objects.");
System.out.println(str1);
```

❒ ***6 methods to operate on strings:*** The String class contains several methods that operate on strings. The general forms for a few:

Names of Method	***Description***	***String*** str1 = "Hello."; ***String*** str2 = ***new String***(str1); ***String*** str3 = "You there !!";
boolean equals(str)	Returns ***true*** if the invoking string contains the same character sequence as ***str***.	**if**(str1.***equals***(str2)) **System.out.println**("str1 = str2"); **else return**; **if**(str1.***equals***(str3)) **System.out.println**("str1 = str3"); **else return**; **[str1.equals(str2)** returns ***true***, **str1.equals(str3)** returns ***false*]**
int length( )	Obtains the length of a string.	**System.out.println**("Length of str1: " + str1.***length***());
char charAt(index)	Obtains the character at the index specified by index.	*// display str1, one char at a time.* **for**(**int** i=0; i < str1.length(); i++) **System.out.print**(str1.***charAt***(i));
int compareTo(str)	**-ve** if *invoking string* **< *str***, **+ve** if *invoking string* **> *str***, **0** if *invoking string* **= *str***,	**int** result = str1.***compareTo***(str3); **if**(result == 0) **System.out.println**("str1 = str3 "); **else if**(result < 0) **System.out.println**("str1 < str3"); **else** **System.out.println**("str1 > str3");
int indexOf(str)	Searches the invoking string for the substring specified by ***str***. Returns the index of the first match or ***–1*** on *failure.*	str2 = "*One Two Three One*"; *// assign a new string to str2* idx = str2.***indexOf***("One"); **System.out.println**("Index of first occurrence of One: " + idx); idx = str2.***lastIndexOf***("One"); **System.out.println**("Index of last occurrence of One: " + idx);
int lastIndexOf(str)	Searches the invoking string for the substring specified by ***str***. Returns the index of the last match or ***–1*** on *failure.*	
☛ You can concatenate (join together) two strings using the + operator. For example,		**String** str1 = "One"; **String** str2 = "Two"; **String** str3 = "Three"; **String** str4 = str1 + str2 + str3; *//initializes str4 with the string "OneTwoThree".*

NOTE: ***Why don't use == instead of*** *equals()*: **`equals()`** compares the ***character sequences*** of **two String objects** for ***equality***. Applying the **==** to two **String references** simply determines whether the two ***references*** refer to the same object.

❒ ***Arrays of Strings:*** Like any other data type, strings can be assembled into arrays. For example:

```
String strs[] = { "This", "is", "a", "test." };    System.out.println("Original array: ");
for(String s : strs) System.out.print(s + " ");    System.out.println("\n");
strs[1] = "was";     strs[3] = "test, too!";                    // change a string
System.out.println("Modified array:"); for(String s : strs) System.out.print(s +" ");
```

OUTPUT:
```
Original array:
This is a test.
Modified array:
This was a test, too!
```

2.15 Strings Are Immutable

In Java (C#, python also) the **contents** of a **String object** are ***immutable***. That is, once created, the character sequence that makes up the string cannot be altered.

❒ When you need a **string** that is a variation on one that already exists, simply create a new string that contains the desired changes. Since ***unused String objects are automatically garbage collected***, so it's not a headache.

☞ However, that **String reference variables** may, of course, change the object to which they refer. It is just that the **contents of a specific String object** cannot be changed after it is created.

❒ **`substring()`**: The **`substring()`** method returns a new string that contains a specified portion of the invoking string.

```
String substring(int startIndex, int endIndex)
```

★ Here, startIndex specifies the beginning index, and endIndex specifies the stopping point.

❒ ***Example:*** Now we demonstrate *immutability of strings* "***contents of a specific String object cannot be changed after it is created*** " When we using **`substring()`** a new **String object** is manufactured that contains the ***substring***, the *original string is unaltered*, and the rule of immutability remains intact. Here is the program that demonstrates substring( ) and the principle of immutable strings:

```
String orgstr = "Java makes the Web move.";
                    // construct a substring
String substr = orgstr.substring(5, 18);
System.out.println("orgstr: " + orgstr);
System.out.println("substr: " + substr);
```

OUTPUT:
```
orgstr: Java makes the Web move.
substr: makes the Web
```

As you can see, the original string ***orgstr*** is unchanged, and substr contains the substring.

❒ ***StringBuffer:*** Java offers a **class called *StringBuffer***, which ***creates string objects that can be changed***. For example, in addition to the **`charAt()`** *(which obtains the character at a specific location)*, ***StringBuffer*** defines **`setCharAt()`**, which ***sets a character within the string***. Java also supplies ***StringBuilder***, which is related to ***StringBuffer***, and also supports strings that can be changed.

➲ For general purpose use ***String***, not ***StringBuffer*** or ***StringBuilder***.

NOTES

[1] ***Mutable arrays:*** Once you have created an **array of values**, you can always change any one of the entries. Why? Because **immutability could get costly** as any *change to an immutable array would need to be implemented* as a ***copy*** (garbage collector take care of it).

➢ The most important **non-numeric** type is the ***string***. A string can be viewed as an array of characters so it would not be unreasonable to make it mutable, but strings are also viewed as primitive values (e.g., we don't think of "Daniel" as an array of 6 characters). Consequently, some languages have immutable strings, others have mutable strings.

☑ In ***Java, C#, JavaScript, Python*** and ***Go***, strings are **`immutable`**. Furthermore, **Java**, **C#**, **JavaScript** and **Go** have the notion of a constant: a "variable" that cannot be reassigned.

☑ In ***Ruby*** and ***PHP***, strings are **`mutable`**.

☑ The ***C*** language does not really have **string objects** per se. However, we commonly represent strings as a ***pointer*** **`char *`**. In general, ***C*** strings are **`mutable`**. The C++ language has its own **`string class`**. It is **`mutable`**.

O In both ***C*** and ***C++***, string constants (declared with the **`const`** qualifier) are **`immutable`**, but you can easily "cast away" the const qualifier, so the immutability is weakly enforced.

☑ In ***Swift***, strings are **`mutable`**. However, if you declare a string to be a constant (keyword **`let`**), then it is immutable.

I'm new to C++ coming from a background of C#, and am trying to understand how the string class in C++ works. I've read that strings are mutable in C++, but following doesn't work like that

```
                //Declaration for the string data
std::string strData = "One";
                //Declaration for C++ vector
std:: vector <std::string> str_Vector;

       str_Vector.push_back(strData);
       strData = "Two";
       str_Vector.push_back(strData);
       strData = "Three";
       str_Vector.push_back(strData);
       strData = "Four";
       str_Vector.push_back(strData);
```

I am wondering why ***str_Vector*** does not become ***"Four", "Four", "Four", "Four"***? If strings are mutable in C++ and if ***str_Vector*** stores by ***reference*** (both assumptions I've made which could very well be false), then it seems to me that we just added the pointer to ***strData*** four times, and that modifying ***strData*** should also implicitly modify ***str_Vector***.

That's the problem with Java and C#. The ***differences between object and pointer are muddled beyond all recognition***.

In C++, something doesn't point to something else if it's not declared with ***** or **&**. For the code to behave as you expect it, it would have to look like this:

```
std::string s="One";
std::vector<std::string *> v;
v.push_back(&s);
s="Two";
v.push_back(&s);
s="Three";
v.push_back(&s);
s="Four";
v.push_back(&s);
```

See? Now you've pushed the same pointer into the vector four times, and changes made to any of the elements will be reflected in all the other elements. Or more accurately, changes to the object which any of the elements point to will be reflected in the object which all the other elements point to.

[2] ***Immutable String in Java:*** In java, string objects are **immutable**. Immutable simply means **unmodifiable** or **unchangeable**. Once string object is created its data or state can't be changed but a **new string object** is created. Example given below:

```
String s="Sachin";
s.concat(" Tendulkar");     //concat() method appends the string at the end
System.out.println(s);      //will print Sachin because strings are immutable objects
```

OUTPUT:
Sachin

Here ***Sachin*** is not changed but a new object is created with ***sachintendulkar***. That is why string is known as immutable. That two objects are created but ***s*** reference variable still refers to "***Sachin***" not to "***Sachin Tendulkar***". But if we explicitely assign it to the reference variable, it will refer to "***Sachin Tendulkar***" object. For example:

```
String s="Sachin";
s=s.concat(" Tendulkar");
System.out.println(s);
```

OUTPUT:
Sachin Tendulkar

☛ In such case, s points to the "***Sachin Tendulkar***". Please notice that still ***sachin*** object is not modified.

☛ Why string objects are immutable in java?
Because java uses the concept of string ***literal***. Suppose there are 5 reference variables, all refers to one object "***sachin***". If ***one reference variable changes the value of the object, it will be affected to all the reference variables***. That is why string objects are ***immutable*** in java.

2.16 Strings to control SWITCH and Command-Line arguments

We can use a String to control a switch. For example, using a ***string-based switch*** is an improvement over using the equivalent sequence of ***if/else*** statements.

☒ However, **switching** on ***strings*** can be less efficient than **switching** on ***integers***. Therefore, it is best to switch on strings only in cases in which the controlling data is already in string form. Don't use strings in a switch unnecessarily.

```
String command = "cancel";
switch(command) {   case "connect":     System.out.println("Connecting"); break;
                    case "cancel":      System.out.println("Canceling"); break;
                    case "disconnect":  System.out.println("Disconnecting"); break;
                    default:            System.out.println("Command Error!"); break;   }
```

☛ The string contained in command (which is "cancel" in this program) is tested against the case constants. When a match is found (as it is in the second case), the code sequence associated with that sequence is executed.

❒ ***Command-Line Arguments:*** We noticed ***args[]*** parameter to ***main()*** that has been in every program. Many programs use *command-line arguments*. A *command-line argument* is the information that directly follows the program's name on the *command line* when it is executed.

☞ To access the **command-line arguments** inside a Java program is quite easy—they are stored as strings in the String array passed to ***main()***. For example, the following program displays all of the command-line arguments that it is called with:

```
class CLDemo { public static void main(String args[]) {
System.out.println("There are " + args.length + " command-line arguments.");
System.out.println("They are: ");
for(int i=0; i<args.length; i++) System.out.println("arg[" + i + "]: " + args[i]);
}}
```

If CLDemo is executed like, **java CLDemo one two three**
[passing " one two three " as command line arguments during program execution (not in compilation)]

OUTPUT: There are 3 command-line arguments.
They are:
arg[0]: one
arg[1]: two
arg[2]: three
Notice that the first argument is stored at index 0, the second argument is stored at index 1, and so on.

Another Example: Following takes one command-line argument that specifies a person's name. It then searches through a two-dimensional array of strings for that name. If it finds a match, it displays that person's telephone number.

```
class Phone {
public static void main(String args[]) {
    String numbers[][] = {
            { "Tom", "555-3322" },
            { "Mary", "555-8976" },
            { "Jon", "555-1037" },
            { "Rachel", "555-1400" }  };
    int i;
if(args.length != 1) System.out.println("Usage: java Phone <name>");
else {      //To use the program, one command-line argument must be present.
    for(i=0; i<numbers.length; i++) {
        if(numbers[i][0].equals(args[0])) {
            System.out.println(numbers[i][0] + ": " + numbers[i][1]);
            break; } }
    if(i == numbers.length) System.out.println("Name not found."); }
}}
```

SAMPLE RUN:
java Phone Mary
Mary: 555-8976

2.17 Bitwise Operators (Recall C/C++ 7.7)

Bitwise operators are used to test, set, or shift the ***individual bits*** that make up a value. Bitwise operations are important to a wide variety of **systems-level programming tasks** in which status information from a device must be interrogated or constructed.

☑ The bitwise operators ***can be used*** on values of type ***long***, ***int***, ***short***, ***char***, or ***byte***.

☒ Bitwise operations ***cannot be used*** on ***boolean***, ***float***, or ***double***, or ***class types***.

Operator	**&**	**|**	**^**	**>>**	**>>>**	**<<**	**~**
Result	Bitwise ***AND***	Bitwise ***OR***	Bitwise ***XOR***	Shift right	Unsigned shift right	Shift left	One's complement (unary ***NOT***)

❒ ***Bitwise AND, OR, XOR, and NOT:*** The bitwise operators **&**, **|**, **^**, and **~** perform the same operations as their **Boolean** logical equivalents. The difference is that the bitwise operators work on a ***bit-by-bit basis***.

☞ You can think of the bitwise ***AND*** as a way to ***turn bits off*** (and '**0**' remain off), bitwise ***OR*** as a way to ***turn bits on*** (and '**1**' remain on).

☞ The following program uses **&** to turn lowercase letter into uppercase by resetting the ***6th bit*** to **0**, **|** to turn uppercase letter into lowercase by resetting the ***6th bit*** to **1**.

[***Why 6th bit? :*** By ***Unicode/ASCII*** character set is definition, the lowercase letters are the same as the uppercase ones except that the lowercase ones are greater in value by exactly ***32***. Therefore, to transform a lowercase letter to uppercase, just turn off the ***6th bit***, because in binary ***32*** is ***100 000*** "only 6th digit from right is ***1***/on". That is in binary " $a = A + 100,000$"]

Bitwise AND	***Bitwise OR***
char ch; **for**(**int** i=0; i < 10; i++) { ch = (**char**) ('a' + i); **System.out.print**(ch); *// This statement turns off the 6th bit.* ch = (**char**) ((**int**) ch & 65503); **System.out.print**(ch + " "); }	**char** ch; **for**(**int** i=0; i < 10; i++) {ch = (**char**) ('A' + i); **System.out.print**(ch); // This statement turns on the 6th bit. ch = (**char**) ((**int**) ch | 32); **System.out.print**(ch + " "); }
OUTPUT: aA bB cC dD eE fF gG hH iI jJ	OUTPUT: Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj
65,503 is ***1111 1111 1101 1111*** in binary. Thus, the ***AND*** operation leaves all bits in ***ch*** unchanged except for the **6th one**, which is set to ***0***.	***32*** is ***0000 0000 0010 0000*** in binary. Thus, the ***OR*** operation leaves all bits in ***ch*** unchanged except for the **6th one**, which is set to ***1***.

☞ ***Other usage of bitwise AND:***

➢ The ***AND*** operator is also useful when you want to determine whether a **bit is on or off**. For example, following determines whether bit **4** in status is set:

```
if((status & 8)!= 0) System.out.println("bit 4 is on");
```

The number ***8*** is used because in binary it is "***1000***" has only the ***4th bit on/set***. Therefore, the ***if*** statement can succeed only when bit ***4*** of status is also on.

➢ An interesting use of above concept is to show the ***bits*** of a ***byte*** value in binary format.

```
int t;
byte val;
val = 123;
for(t=128; t>0; t = t/2){ if((val & t) != 0) System.out.print("1 ");
                          else System.out.print("0 "); }
```

OUTPUT: ***0 1 1 1 1 0 1 1***

[The ***for*** loop successively tests each bit in ***val***, using the ***bitwise AND***, to determine whether it is ***on*** or ***off***. If the bit is ***on***, the digit ***1*** is displayed; otherwise, ***0*** is displayed.]

☞ ***XOR*** will set a bit on **iff** the bits being compared are different (i.e. **1^1=1**, **0^0=0**, **1^1=0**):

➢ XOR operator makes it a simple way to encode a message. When some value ***X*** is **XORed** with another value ***Y***, and then that result is **XORed** with ***Y*** again, ***X*** is produced. That is, given the sequence ***R1 = X ^ Y; R2 = R1 ^ Y;*** i.e. **X=(X^Y)^Y**

➢ We can create simple **CIPHER** program in which some ***integer*** is the key that is used to both ***encode*** and ***decode*** a message by ***XORing*** the characters in that message. To ***encode***, the ***XOR*** operation is applied the first time, yielding the ***cipher text***. To ***decode***, the ***XOR*** is applied a second time, yielding the plain text.

[Of course, such a cipher has no practical value, being trivially easy to break. It does, however, provide an interesting way to demonstrate the **XOR**.]

```
String msg = "This is a test";   String encmsg = "";   String decmsg = "";
int key = 88;
                    // encode the message, create string with charAt()
for(int i=0; i < msg.length(); i++) encmsg = encmsg + (char)(msg.charAt(i) ^ key);
                    // decode the message, create string with charAt()
for(int i=0; i < msg.length(); i++) decmsg = decmsg + (char)(encmsg.charAt(i) ^ key);
```

☞ The **unary one's complement** (***NOT***) operator reverses the state of all the bits of the operand. For example, if some integer called **A** has the bit pattern ***1001 0110***, then **~A** produces a result with the bit pattern ***0110 1001***.

❒ ***The Shift Operators:*** Shift operators "**<<**" and "**>>**" are similar to C/C++ but the unsigned right shift "**>>>**" is new in Java.

operators	***Meaning***	***General form***	***Descriptions***
<<	Left shift	***value << num-bits***	***value*** is the value being shifted by the number of bit positions specified by ***num-bits***. Each ***left shift*** causes all bits within the specified value to be ***shifted left one position*** and a ***0*** bit to be brought in on the ***right***. Each ***right shift*** shifts all bits to the ***right one position*** and ***preserves the sign bit***.
>>	Right shift	***value >> num-bits***	
>>>	Unsigned right shift	***value >>> num-bits***	

☛ ***Right shifting*** "**>>**" ***–ve/negative value:*** Negative numbers are usually represented by setting the high-order bit of an integer value to **1**, it is MSB representation for example "**8**" in binary is "**0000 1000**" and "**–8**" in binary is "**1000 1000**"(Here –ve value is represented as ***MSB*** : most significant bit representation, where leftmost bit is reserved for sign, ***0*** for ***+ve*** and ***1*** for ***–ve***). Java ***do not use the MSB Representation***, here ***two's complement*** is used.

▶ Java uses ***two's complement*** to represent negative values. In this approach negative values are stored by first ***reversing*** the bits in the value (one's compliment) and then adding **1**. Thus, the byte value for **–1** in binary is **1111 1111**. Right shifting this value will always produce **–1**! In ***two's complement***, if we want to represent "**–8**" for 8 bit field $(2^8)_2 - 8_2 = 1\,0000\,0000 - 1000 = 1111\,1000$ for detail see ***2.19 Signed binary numbers***), and this is the approach used by Java. If the value being shifted is negative, each right shift brings in a **1** on the left (i.e. **–8>>1** brings "**1111 1100**").

▶ NOTE: In C/C++ also ***two's complement*** is used to represent –ve binary value. (Also in shifting operation)

☛ If the value is positive, each right shift brings in a **0** on the ***left***.

☛ ***Unsigned right shift:*** To remove ***sign bit*** when shifting right, you can use an ***unsigned right shift*** (**>>>**), which always brings in a **0** on the ***left***. For this reason, the **>>>** is also called the ***zero-fill right shift***. (Eg:<<< is used when shifting bit patterns, such as status codes, that do not represent integers.)

☛ ***Lose of bits due to shift:*** For all of the shifts, the ***bits shifted out are lost***. Thus, a shift is not a ***rotate***, and there is no way to retrieve a bit that has been **shifted out**.

Example 1: Here, an integer is given an initial value of ***1***, which means that its **low-order bit** is set. Then, a series of eight shifts are performed on the integer. After each shift, the **lower 8 bits** of the value are shown. The process is then repeated, except that a ***1*** is put in the ***8th bit position***, and ***right shifts*** are performed.

```
class ShiftDemo { public static void main(String args[]) { int val = 1;
    for(int i = 0; i < 8; i++) {    for(int t=128; t > 0; t = t/2) {    if((val & t) != 0) System.out.print("1 ");
                                                                        else System.out.print("0 "); }  System.out.println();
                                    val = val << 1; } System.out.println();          /* left shift */
    val = 128;
    for(int i = 0; i < 8; i++) {    for(int t=128; t > 0; t = t/2) {    if((val & t) != 0) System.out.print("1 ");
                                                                        else System.out.print("0 "); }System.out.println();
                                    val = val >> 1;}    }}                          /* right shift */
```

☛ ***Be careful when shifting*** **`byte`** ***and*** **`short`** ***values*** because Java will automatically **promote** these types to ***`int`*** when evaluating an expression. For example, if you right shift a byte value, it will first be promoted to int and then shifted. The result of the shift will also be of type int. Often this conversion is of no consequence.

☛ ***If you shift a*** **`negative byte`** ***or*** **`short`** ***value***, it will be **sign-extended** when it is promoted to int. Thus, the high-order bits of the resulting integer value ***will be filled with ones***. This is fine when performing a ***normal right shift***. But ***when you perform a zero-fill right shift, there are 24 ones*** to be shifted before the byte value begins to see ***zeros***.

❒ ***Bitwise Shorthand Assignments:*** All of the binary bitwise operators have a shorthand form that combines an assignment with the bitwise operation. For example, the following two statements both assign to x the outcome of an XOR of x with the value 127.

```
x = x ^ 127;          x ^= 127;
```

The bitwise shift operators can be used to perform very fast multiplication or division by "**`2`**":

- A shift ***left*** *doubles* a value.
- A shift ***right*** *halves* it.

2.18 The ? ternary Operator

The **?** is called a ***ternary operator*** because it requires three operands. General form: **`Exp1 ? Exp2 : Exp3;`**

➲ where ***Exp1*** is a **boolean expression**, and ***Exp2*** and ***Exp3*** are expressions of any type other than void. The type of ***Exp2*** and ***Exp3*** must be the same (or compatible), though. Notice the use and placement of the colon. It is similar to C/C++'s "**?**" operator.

➲ The value of a **?** expression is determined like this: **Exp1** is evaluated. If it is true, then **Exp2** is evaluated and becomes the value of the entire **?** expression. If **Exp1** is false, then **Exp3** is evaluated and its value becomes the value of the expression.

❒ Example, Prevent a division by zero using the ?, **if**(i != 0 ? **true** : **false**) **System.out.println**("100 / " + i + " is " + 100 / i);

☞ If i=0, then the outcome of the ***if*** is false, the division by zero is prevented, and no result is displayed. Otherwise, the division performed.

2.19 Signed binary numbers (representations)

In ***digital circuits*** there is no provision made to put a plus or even a minus sign to a number, since digital systems operate with binary numbers that are represented in terms of "0's" and "1's". When used together in microelectronics, these "1's" and "0's", called a bit (being a contraction of BInary digiT), fall into several range sizes of numbers which are referred to by common names, such as a byte or a word. An 8-bit binary number (a byte) can have a value ranging from **0** ($\mathbf{0000\ 0000_2}$) to **255** ($\mathbf{1111\ 1111_2}$), that is $\mathbf{2^8 = 256}$ different combinations of bits forming a single ***8-bit byte***. Eg: $\mathbf{0100\ 1101_2 = 64 + 8 + 4 + 1 = 77_{10}}$ in decimal

➲ ***Most significant bit (MSB) representation:*** We know that binary digits, or bits only have two values, either a "1" or a "0" and conveniently for us, a sign also has only two values, being a "**+**" or a "**–**". Then we can **use a single bit to identify the sign of a signed binary number as being positive or negative in value**. So to represent a positive binary number (+n) and a negative (-n) binary number, we can use them with the addition of a sign.

❒ ***SM (Sign-and-Magnitude) notation:*** For signed binary numbers the **most significant bit** (***MSB***) is used as the sign bit. If the sign bit is "**0**", this means the number is **`positive`** in value. If the sign bit is "**1**", then the number is **`negative`** in value. The remaining bits in the number are used to represent the ***magnitude*** of the binary number in the *usual unsigned binary number format way.*

☞ Then we can see that the ***Sign-and-Magnitude (SM) notation*** stores positive and negative values by dividing the "**n**" total bits (i.e. bit-field length is **n**) into two parts: **1** bit for the sign and **n–1** bits for the value which is a pure binary number. For example, the decimal number ***53*** can be expressed as an ***`8-bit signed binary number`*** as follows.

+ve Signed binary number								
+53	8 bit word							
	0	0	1	1	0	1	0	1
+ve sign bit	Magnitude bits							

-ve signed binary number								
–53	8 bit word							
	1	0	1	1	0	1	0	1
-ve sign bit	Magnitude bits							

☞ ***Disadvantages:***
- ➲ Since bit field is divided into two part, the N length bit field becomes N-1 long, hence the range reduces.
- ➲ We can have a positive result for zero, $+\mathbf{0}$ ***or*** $\mathbf{0000_2}$, and a negative result for zero, $-\mathbf{0}$ ***or*** $\mathbf{1000_2}$. Both are valid but which one is correct.

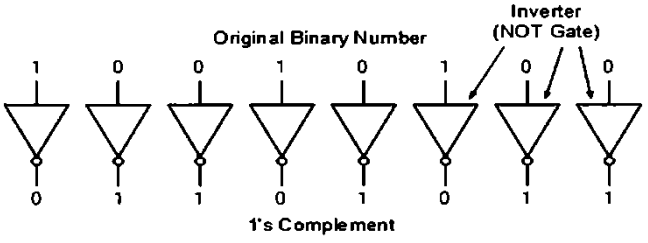
☞ ***Examples:***		
-15 (8 bit)	**1000 1111**	NOTE: for a **4**-bit, **6**-bit, 8-bit, **16**-bit or **32**-bit signed binary number ***all the bits MUST have a value***, therefore "**0**"s" are used to fill the spaces between the leftmost sign bit and the first or highest value "**1**". I,E. $\mathbf{15_{10} = 1111_2}$ must be written as **0000 1111.** ***(For sign bit)***
+23 (8 bit)	**0001 0111**	
-127 (8 bit)	**1111 1111**	

❒ ***One's Complement:*** In one's complement, positive numbers (also known as **non-complements**) remain ***unchanged*** as before with the sign-magnitude numbers.

☞ **Negative numbers** are represented by taking the ***one's complement*** (inversion, negation, ***Bitwise NOT*** "**~**") of the **unsigned positive number**. Since ***+ve*** numbers always start with a "**0**", the complement ***(-ve)*** will always start with a "**1**" to indicate a ***-ve*** number. Eg: the one's complement of **`100101002`** is simply **`011010112`** as all the **1**'s are changed to **0**'s and the **0**'s to **1**'s.

☞ The **easiest way** to find the **one's complement** of a ***signed binary*** number when building **digital arithmetic** or **logic decoder circuits** is to use Inverters (Inverters are **NOT GATE**, which is a complement generator can be used in parallel to find the 1's complement of any binary number).

☞ ***Negative number using One's complement:*** 1's complement can be easily obtained by "***Bitwise NOT ~***".



☛ ***Subtraction of Two Binary Numbers using 1's complement:*** An ***8-bit*** digital system is required to subtract the following two numbers ***115*** and ***27*** from each other using ***one's complement***. So in decimal this would be: **115 − 27 = 88**.

- Here $115_{10} = 0111\ 0011_2$ and $27_{10} = 0001\ 1011_2$
- And "-27_{10}" is $\sim(0001\ 1011_2) = 1110\ 0100_2$
- Then $0111\ 0011_2 - 0001\ 1011_2 = 0111\ 0011_2 + 1110\ 0100_2$
  Which is **1 0101 0111**, the leftmost **1** is the ***overflow***.

```
   01110011
+  11100100
Overflow → 101010111
```

- The 8-bit result from above is: 01010111 (the overflow "1" cancels out) and to convert it back from a **one's complement answer** to the ***real answer*** we now have to add "**1**" to the ***one's complement result***, therefore: **01010111+1=01011000** and $01011000_2 = 88_{10}$

▣ Since the ***digital system*** is to work with ***8-bits***, only the first eight digits are used to provide the answer to the sum, and we simply ***ignore the last bit (bit 9)***. This bit is call an "***overflow***" bit. Overflow occurs when the sum of the most significant (left-most) column produces a carry forward. This overflow or carry bit can be ignored completely or passed to the next digital section for use in its calculations.

- **Overflow** indicates that the answer is **positive**. If there is **no overflow** then the answer is **negative**.

❒ ***Two's Complement of a Signed Binary Number:*** In two's complement, the positive numbers are exactly the same as before for unsigned binary numbers.

☞ ***Negative number using 2's complement:*** A negative number, however, is represented by a binary number, which when added to its corresponding positive equivalent results in zero.

- In two's complement form, a negative number is the 2's complement of its positive number, say B is the +ve number then, 2's complement of B is $(2^N)_2 - B_2$ where N is the "bit-field-length". For example, in 8-bit long field 27 is 0001 1011 then 2's complement of 27 is $(2^8)_2 - 27_2 = (256)_2 - 27_2 = 1\ 0000\ 0000 - 0001\ 1011 = 1110\ 0101$
- When adding non-compliment and compliment we get **1 0000 0000** then leftmost **1** is ignored as overflow.
- ***Two's*** complement is ***1's complement + 1***. Eg: 1's compliment of 27 is $1110\ 0100_2$ then $1110\ 0100 + 1 = 1110\ 0101$.
  Say **B** is the **+ve** number then, 2's complement of **B** can be obtained using $(\sim B + 1)$. "**~**" is the ***Bitwise NOT*** operator.

☞ The main advantage of two's complement over the previous one's complement is that there is no double-zero problem plus it is a lot easier to generate the two's complement of a signed binary number.

☛ ***Subtraction of Two Binary Numbers using 2's complement:*** Reconsider $115 - 27 = 88$. Here $115_{10} = 0111\ 0011_2$ and 2's complement of 27 is $(2^8)_2 - 27_2 = (256)_2 - 27_2 = 1\ 0000\ 0000 - 0001\ 1011 = 1110\ 0101$. Then,
$115_2 - 27_2 = 115_2 + \textit{two's complement of } 27 = 0111\ 0011 + 1110\ 0101 = 1\ 0101\ 0111$, leftmost **1** is the overflow. As previously, the **9th overflow bit** is *disregarded* as we are only interested in the first 8-bits, so the result is: **0101 1000** or ***(64 + 16 + 8) = 88*** in decimal the same as before.

☛ The method of ***2's complement arithmetic is commonly used in computers*** to handle negative numbers.

2.20 Access Modifiers

Difference from C/C++: In Java for any class the **default** access setting is ***public*** (more like C's structure type). But in C/C++ any member of a class is ***private*** in **default**. Also in C/C++ "**:**" is used with Access Modifiers of members of the class but in Java Access Modifiers are used with individual objects.

☑ The meaning of ***private***, ***public*** in Java is similar to C/C++. There is also ***protected*** in Java.

➲ The **default** access setting (in which ***no access modifier*** is used) is the same as ***public*** unless your program is broken down into ***packages***. [A *package* is, essentially, a ***grouping of classes***. Packages are both an organizational and an access control feature]

➲ An ***access modifier precedes the rest of a member's type specification***. That is, it must begin a member's declaration statement.

```
public String errMsg;
private accountBalance bal;
private boolean isError(byte status) { // ... }
```

NOTE: In the real world, restricting access to members—especially ***instance variables***—is an important part of successful object-oriented programming.

2.21 Pass Objects to Methods

Object can be passed to methods as arguments. Which is familiar in C/C++. For example, following uses objects as parameter.

```
class Block {     int a, b, c, volume;
                  Block(int i, int j, int k) { a = i; b = j; c = k; volume = a * b * c; }

                        /* Return true if ob defines same block and object type parameter used */
                  boolean sameBlock(Block ob) {     if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
                                                    else return false; }

                        /* Return true if ob has same volume. And object type parameter used */
                  boolean sameVolume(Block ob) {    if(ob.volume == volume) return true;
                                                    else return false; }
            }

class PassOb { public static void main(String args[]) {Block ob1 = new Block(10, 2, 5);
                                                        Block ob2 = new Block(10, 2, 5);
                                                        Block ob3 = new Block(4, 5, 5);
      System.out.println("ob1 same dimensions as ob2: " + ob1.sameBlock(ob2));      /* passing object as a parameter */
      System.out.println("ob1 same dimensions as ob3: " + ob1.sameBlock(ob3));      /* passing object as a parameter */
      System.out.println("ob1 same volume as ob3: " + ob1.sameVolume(ob3));   }}    /* passing object as a parameter */
```

2.22 Two ways to Pass Arguments

Difference from C/C++: In C/C++ both *primitive types* and *object types* are passed as ***call-by-value*** by default. But in Java *primitive types* and are passed as ***call-by-value*** and *object types* are passed as ***call-by-reference*** by default. [Recall C/C++ ***10.10*** " ***PASSING objects to functions and RETURNING objects from function*** " and ***10.12 References***].

⇨ In Java objects are automatically passed as **reference** but in C/C++ you have to declare **reference**.

❒ ***Call-by-value:*** This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument in the call.

☞ When you pass a **primitive type**, such as ***int*** or ***double***, to a method, it is ***passed by value***. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method.

❒ ***Call-by reference:*** In this approach, a **reference** to an argument (not the **value** of the argument) is passed to the **parameter**. Inside the ***subroutine***, this reference is used to access the actual argument specified in the **call**. This means that changes made to the parameter will affect the argument used to **call** the subroutine.

☞ ***Objects*** are implicitly ***passed by reference***. A **reference to the object** is created when we create a variable of a ***class type***.

☞ It is the ***reference***, not the object itself, that is actually passed to the method. As a result, when you pass this reference to a method, the ***parameter*** that receives it will ***refer*** to the ***same object as that referred to by the argument***. This effectively means that objects are passed to methods by use of ***call-by-reference***. Changes to the object inside the method **do affect the object** used as an argument. Consider the following examples:

Primitive types are passed by value.	Objects are passed through their references.
class Test { */* following method causes no change to the arguments used in the call. */* **void** noChange(**int** i, **int** j) { i = i + j; j = -j; } */* primitive types used */* } **class** CallByValue **{ public static void main(String args[])** { **Test** ob = **new** Test(); **int** a = 15, b = 20; **System.out.println**("a and b before call: " + a + " " + b); ob.noChange(a, b); **System.out.println**("a and b after call: " + a + " " + b); }}	class Test { **int** a, b; Test(**int** i, **int** j) { a = i; b = j; } */* In following method an object is passed. Now, ob.a and ob.b in object used in the call will be changed. */* **void** change(**Test** ob) { ob.a = ob.a + ob.b; ob.b = -ob.b; } } **class** PassObRef { **public static void main(String args[])** { Test ob = new Test(15, 20); **System.out.println**("ob.a and ob.b before call: " + ob.a + " " + ob.b); ob.change(ob); **System.out.println**("ob.a and ob.b after call: " + ob.a + " " + ob.b); }}
OUTPUT: a and b before call: 15 20 a and b after call: 15 20	OUTPUT: ob.a and ob.b before call: 15 20 ob.a and ob.b after call: 35 -20

▶ As you can see, the operations that occur inside ***noChange()*** have no effect on the values of ***a*** and ***b*** used in the call.

▶ On the other hand, the actions inside ***change()*** have affected the object used as an argument.

NOTE

[1] ***To pass a primitive type by reference:*** Java defines a **set of classes** that **wrap** the **primitive types** in *objects*. These are **Double**, **Float**, **Byte**, **Short**, **Integer**, **Long**, and **Character**. In addition to allowing a primitive type to be passed by reference, these ***wrapper classes*** define several methods that enable you to manipulate their values. [For example, the ***numeric type wrappers*** include methods that convert a numeric value from its binary form into its human-readable String form, and vice versa.]

[2] When an object reference is passed to a method, the ***reference itself is passed by use of call-by-value***. However, since the value being passed refers to an **object**, the copy of that value will still refer to the same **object** referred to by its corresponding argument.

2.23 Returning Objects

A method can return any type of data, including class types. For example, ***getErrorMsg()***, returns a String object

```
class msg{      String msgs[] = { "String_1", "STRING_2", "StRiNg_3" };
                String getMsg(int i){ if(i >=0 & i < msgs.length) return msgs[i];  /* Return the error message. */
                else return "Invalid Error Code"; }    }
```

You can, of course, also **return objects** of **classes** that you create. For example, here is a reworked version of the preceding program that creates two **error classes**. One is called ***Err***, and it encapsulates an **error message** along with a severity code. The second is called ***ErrorInfo***, defines a method called ***getErrorInfo(),*** which returns an ***Err*** object.

```
class Err {  String msg;      // error message
        int severity;         // code indicating severity of error
        Err(String m, int s) { msg = m; severity = s; }   /* only string reference declaration */
        }

class ErrorInfo { String msgs[] = { "Output Error", "Input Error", "Disk Full", "Index Out-Of-Bounds" };          /* string initialization */
                  int howbad[] = { 3, 3, 2, 4 };
                  Err getErrorInfo(int i) {      if(i >= 0 & i < msgs.length) return new Err(msgs[i], howbad[i]);  /* returning object type */
                                                 else return new Err("Invalid Error Code", 0); }                   /* returning object type */
                }
class ErrInfo { public static void main(String args[]) { ErrorInfo err = new ErrorInfo();
                                                          Err e; /* no new, constructor is used because ErrorInfo type declaration take care of it */
                                                          e = err.getErrorInfo(2); System.out.println(e.msg + " severity: " + e.severity);
                                                          e = err.getErrorInfo(19); System.out.println(e.msg + " severity: " + e.severity);
                                                        } }
```

☞ Each time ***getErrorInfo()*** is invoked, a new ***Err*** object is created, and a reference to it is returned to the calling routine (that's why no new, constructor and only reference is declared by " **Err e;** "). This object (i.e. ***Err e***) is then used within **main()** to display the error message and severity code.

❒ When an object is returned by a method, it ***remains in existence until there are no more references to it***. At that point, it is subject to garbage collection. Thus, an object won't be destroyed just because the method that created it terminates.

2.24 Method Overloading (Similar to C/C++)

In **Java**, two or more **methods** within the same **class** can share the **same name**, as long as their ***parameter declarations*** are different. When this is the case, the methods are said to be ***overloaded***, and the process is referred to as ***method overloading***.

⇨ Overloaded methods may differ in their **parameters types**, too. When an overloaded method is called, the version of the method whose parameters match the arguments is executed.

❒ ***One important restriction:*** the type and/or number of the parameters of each overloaded method must differ. It is not sufficient for two methods to differ only in their return types.

❒ ***Example 1:***

```
class Overload {   void ovlDemo() { // . . . }                          /* no parameter */
                   void ovlDemo(int a) { // . . . }                     /* one integer parameter. */
                   int ovlDemo(int a, int b) { }                        /* two integer parameter. */
                   double ovlDemo(double a, double b) {// . . . }       /* two double parameter. */
                   }
class OverloadDemo { public static void main(String args[]) {  Overload ob = new Overload();
                                                    // call all versions of ovlDemo()
                                                    ob.ovlDemo();
                                                    ob.ovlDemo(2);
                                                    int resI = ob.ovlDemo(4, 6);
                                                    double resD = ob.ovlDemo(1.1, 2.32);   } }
```

☞ Here ***ovlDemo()*** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes two ***double*** parameters.

☞ Notice that the first two versions of **ovlDemo()** return **void**, and the second two return a value. This is perfectly valid, but as explained, overloading is not affected one way or the other by the **return type** of a method. Thus, attempting to use the following two versions of **ovlDemo()** will cause an error:

```
void ovlDemo(int a) { }     /* One ovlDemo(int) is OK. */
int ovlDemo(int a) { }      /* Error! Two ovlDemo(int)s are not OK even though return types differ. */
```

➢ Return types cannot be used to differentiate overloaded methods.

❒ ***Overloading methods based on parameters type (Allow and prevent Type conversions):*** Since Java provides automatic type conversions. These conversions also apply to parameters of ***overloaded methods***. For example, consider the following:

```
class Overload2 { void f(int x) { /* . . . */ } /* Automatic type conversions affecting overloaded method resolution. */
                  void f(double x) { /* . . . */ }
                }
```

➢ Here one ***f()*** has an ***int*** parameter and another that has a ***double*** parameter.

➢ In the case of ***byte*** and ***short***, (or ***int***) Java automatically converts them to ***int***. Thus, ***f(int)*** is invoked.

➢ In the case of ***float*** (or ***double***), the value is converted to ***double*** and ***f(double)*** is called.

☞ ***Specifying overloaded version for specific type can prevent auto conversion:*** The automatic conversions apply only **if there is no direct match between a *parameter* and an *argument***. For example, here is the preceding program with the addition of a version of ***f()*** that specifies a ***byte*** parameter:

```
class Overload2 {       void f(int x) {  /* . . . */ }     /* add byte parameter version of f() */
                        void f(int x) {  /* . . . */ }
                        void f(double x) { /* . . . */ }
                    }
```

❖ In this version, since there is a version of f( ) that takes a byte argument, when f( ) is called with a byte argument, f(byte) is invoked and the automatic conversion to int does not occur.

NOTE : ***Signature of methods:*** In Java, a signature is the ***name of a* method *plus its* parameter list**. Thus, for the purposes of overloading, ***no two methods within the same class can have the same signature***. Notice that a **signature does not include** the ***return type***, since it is not used by **Java for overload resolution**.

2.25 Overloading Constructors

Like methods, constructors can also be overloaded. Doing so allows you to construct objects in a variety of ways. Eg:

```
class MyClass {   int x;                                    class OverloadConsDemo { public static void main(String args[]) {
                  MyClass() {/* . . . */ }                                  MyClass t1 = new MyClass();
                  MyClass(int i) {/* . . . */ }                             MyClass t2 = new MyClass(88);
                  MyClass(double d) {/*. . . */ }                           MyClass t3 = new MyClass(17.23);
                  MyClass(int i, int j) {/* . . . */}                       MyClass t4 = new MyClass(2, 4);
                  }                                                         }}
```

☞ ***MyClass()*** is overloaded four ways, each constructing an object differently. The proper constructor is called based upon the parameters specified when ***new*** is executed.

❒ One of the most common reasons that **constructors** are **overloaded** is to allow ***one object to initialize another***. For example, consider this program that uses the ***Summation*** class to compute the summation of an integer value:

```
class Summation { int sum;
        Summation(int num) {sum = 0;
                for(int i=1; i <= num; i++) sum += i; }
    /* Construct one object from another. */
        Summation(Summation ob) { sum = ob.sum; }
        }
```

```
class SumDemo {public static void main(String args[]) {
        Summation s1 = new Summation(5);
        Summation s2 = new Summation(s1);
        System.out.println("s1.sum: " + s1.sum);
        System.out.println("s2.sum: " + s2.sum);
        }}
```

☞ In this case, when ***s2*** is constructed, it is not necessary to recompute the summation. Of course, even in cases when efficiency is not an issue, it is often ***useful to provide a constructor*** that makes a ***copy of an object***.

OUTPUT:
```
s1.sum: 15
s2.sum: 15
```

2.26 Recursion(Similar to C/C++ Recall C/C++ 5.2)

Recursion is the process of defining something in terms of itself and is somewhat similar to a circular definition. The key component of a recursive method is ***a statement that executes a call to itself***. For example, following uses recursion to compute factorial:

Recursion is used	***Iteration is used***
int factR(**int** n) { **int** result; **if**(n==1) **return** 1; result = factR(n-1) * n; */* recursion is used */* **return** result; }	**int** factI(**int** n){ **int** t, result; **result** = 1; **for**(t=1; t <= n; t++) result *= t; */* iterative equivalent */* **return** result;

☞ The operation of the non-recursive method ***factI( )*** uses a loop starting at ***1*** and ***progressively multiplies*** each number by the moving product.

☞ When recursive ***factR( )*** is called with an argument of ***1***, the method returns ***1***; otherwise, it returns the product of ***factR(n–1)*n***. To evaluate this expression, ***factR( )*** is called with ***n–1***. This process repeats until ***n*** equals ***1*** and the calls to the method begin returning.

❒ When a **method calls itself**, new local variables and **parameters** are allocated storage on the ***stack***, and the method code is executed with these **new variables** from the start. A

❒ **Recursive call** does not make a **new copy** of the **method**. Only the ***arguments are new***. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to ***"telescope" out and back***.

❒ When writing recursive methods, you must have a **conditional statement**, such as an ***if***, somewhere to ***force the method to return without the recursive call*** being executed. If you don't do this, once you call the method, it will never return.

▣ **Disadvantage:** Recursive versions of many routines may execute a bit more ***slowly*** than their iterative equivalents because of the added overhead of the additional method calls. Too many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the ***Java run-time system*** will cause an ***exception***. However, you probably will not have to worry about this unless a recursive routine runs wild.

▣ **Advantage:** The main advantage to recursion is that some types of algorithms can be implemented more clearly and simply recursively than they can be iteratively. For example, the ***Quicksort*** sorting algorithm is quite difficult to implement in an iterative way. Also, some problems, especially ***AI-related*** ones, seem to lend themselves to recursive solutions.

2.27 Static in Java (Variables, Methods and Blocks)

To define a class member that will be used independently of any object of that class we use ***static***. Normally a class member must be accessed through an object of its class, but it is possible to create a member that can be used by itself, without reference to a specific instance.

➲ To create such a member, **precede its *declaration*** with the keyword ***static***.

❒ When a member is declared ***static***, it can be accessed ***before any objects of its class are created***, and without ***reference*** to any object. You can declare both ***methods*** and ***variables*** to be ***static***.

☞ Therefore, ***main()*** is declared as ***static*** because it must be called by the ***JVM*** when your program begins.

❒ ***Accessing:*** Outside the class, to use a ***static*** **member**, you need only specify the ***name of its class*** followed by the ***dot*** operator. *No object needs to be created.* Eg: if you want to assign the value ***10*** to a **static variable** called ***count*** that is part of the ***Timer*** class,

```
Timer.count = 10;
```

▶ This format is similar to that used to access normal instance variables through an object, except that the class name is used.

☞ A **static method** can be called in the same way using the ***dot*** operator on the ***name of the class***. Eg: ***Timer.calender();***

❒ ***Variables*** declared as ***static*** are, essentially, ***global variables***. When an object is declared, **no copy of a static variable is made**. Instead, all ***instances (i.e. all objects)*** of the class share the same ***static variable***. Eg: A ***static variable*** and an ***instance variable***:

```
class StaticDemo {
        int x;      // a normal instance variable
        static int y;          // a static variable
        int sum() { return x + y; }
                }

class SDemo {
public static void main(String args[]) {
    StaticDemo ob1 = new StaticDemo();
    StaticDemo ob2 = new StaticDemo();

// Each object has its own copy of an instance variable.
    ob1.x = 10;
    ob2.x = 20;
    System.out.println("ob1.x: " + ob1.x +
                "\t ob2.x: " + ob2.x + "\n");
```

```
// Each object shares one copy of a static variable.
StaticDemo.y = 19;          /* Set StaticDemo.y to 19 */
System.out.println("ob1.y: " + ob1.y + "\t ob2.y: " + ob2.y + "\n");
System.out.println("ob1.sum(): " + ob1.sum() + "\t ob2.sum(): " + ob2.sum()+ "\n");

StaticDemo.y = 100;         /* Set StaticDemo.y to 100 */
System.out.println("ob1.y: " + ob1.y + "\t ob2.y: " + ob2.y + "\n");
System.out.println("ob1.sum(): " + ob1.sum() + "\t ob2.sum(): " + ob2.sum()+ "\n");
}}
```

OUTPUT:

```
ob1.x: 10          ob2.x: 20

ob1.y: 19          ob2.y: 19
ob1.sum(): 29      ob2.sum(): 39

ob1.y: 100         ob2.y: 100
ob1.sum(): 110     ob2.sum(): 120
```

☞ Here, the **static variable** ***y*** is shared by both ***ob1*** and ***ob2***. Changing it affects the ***entire class***, not just an ***object/instance***.

❒ ***Method:*** The difference between a ***static method*** and a ***normal method*** is that the ***static method is called through its class name, without any object of that class being created***. Eg: ***sqrt()*** method, which is a ***static method*** within Java's standard ***Math*** class.

```
class StaticMeth {
  static int val = 1024;       // a static variable
  static int valDiv2() { return val/2; }} // static method
```

```
class SDemo2 { public static void main(String args[]) {
        System.out.println("val is " + StaticMeth.val);
        System.out.println("StaticMeth.valDiv2(): " + StaticMeth.valDiv2());

        StaticMeth.val = 4;
        System.out.println("val is " + StaticMeth.val);
        System.out.println("StaticMeth.valDiv2(): " + StaticMeth.valDiv2());
        }}
```

OUTPUT:

```
val is 1024
StaticMeth.valDiv2(): 512
val is 4
StaticMeth.valDiv2(): 2
```

❒ Restrictions on static methods and static variables: Methods declared as static have several restrictions:

☑ They can directly call only other ***static methods***.

☑ They can directly access only ***static data***.

☑ They do not have a ***this reference***.

☞ For example, in the following class, the ***static method*** ***valDivDenom()*** is illegal:

```
class StaticError { int denom = 3;      // a normal instance variable
                    static int val = 1024;      // a static variable
                    static int valDivDenom() { return val/denom; }      /* Causing error */
// won't compile: Error! Can't access a non-static variable from within a static method.
                    }
```

➲ Here, ***denom*** is a **normal instance variable** that cannot be accessed within a ***static method***.

❒ ***Static Blocks:*** A ***static block*** is *executed when the class is first loaded.* Thus, it is ***executed before the class can be used for any other purpose***. Here is an example of a static block:

[Sometimes a class will require *some type of initialization before it is ready to create objects.* **Eg:** it might need to establish a connection to a remote site. It also might *need to initialize certain* ***static variables*** *before any of the class'* ***static methods*** *are used.* To handle these types of situations, Java allows you to declare a ***static block***.]

```
class StaticBlock {
        static double rootOf2;
        static double rootOf3;
/* following block is static. And it is executed when the class is loaded */
        static {  System.out.println("Inside static block.");
                  rootOf2 = Math.sqrt(2.0);
                  rootOf3 = Math.sqrt(3.0); }
                  StaticBlock(String msg) {
                          System.out.println(msg);  }
                  }
```

```
class SDemo3 { public static void main(String args[]) {
        StaticBlock ob = new StaticBlock("Inside Constructor");
        System.out.println("Square root of 2 is " + StaticBlock.rootOf2);
        System.out.println("Square root of 3 is " + StaticBlock.rootOf3);
}}
```

OUTPUT;

```
Inside static block.
Inside Constructor
Square root of 2 is 1.4142135623730951
Square root of 3 is 1.7320508075688772
```

☞ Here, the ***static*** **block** is executed ***before any objects are constructed***.

2.28 QuickSort Algorithm

Among other sorting algorithm such as ***Mergesort, Heapsort, Insertion-sort*** the faster is ***Quicksort. Quicksort*** (sometimes called ***partition-exchange sort***) is a **divide-and-conquer** algorithm. It works by selecting a **'pivot'** element from the array and partitioning the other elements into two **sub-arrays**, according to whether they are ***less than or greater than*** the **pivot**. The **sub-arrays** are then sorted ***recursively***.

Algorithm: It first divides the input array into two smaller sub-arrays: the ***low*** elements and the ***high*** elements. It then recursively sorts the **sub-arrays.** The steps are:

[1] Pick an element, called a ***pivot***, from the array.

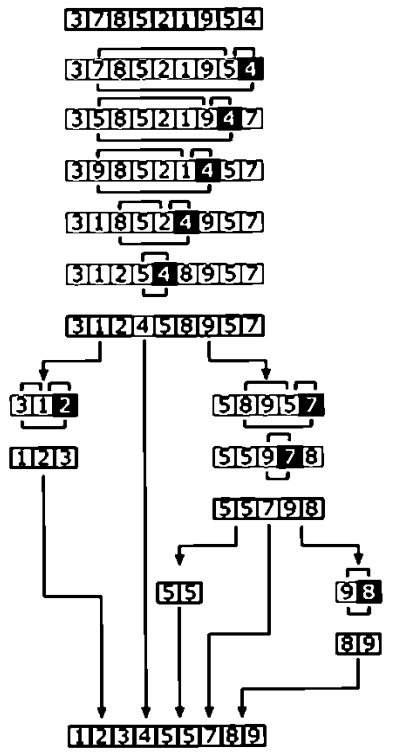
[2] **Partitioning:** ***Reorder*** the array so that all elements with values ***less than the pivot*** come **before** the ***pivot***, while all elements with values ***greater than the pivot*** come **after** it (equal values can go either way). After this ***partitioning***, the ***pivot*** is in its *final* position. This is called the *partition operation*.

[3] ***Recursively*** apply the above steps to the ***sub-array*** of elements with smaller values and separately to the sub-array of elements with greater values.

[The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.]

The **pivot** selection and **partitioning** steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

In the left is a full example of ***quicksort*** on a random set of numbers. The shaded element is the pivot. It is always chosen as the last element of the partition. However, ***always choosing the last element in the partition as the pivot in this way results in poor performance*** ($O(n^2)$) on ***already sorted arrays***, or ***arrays of identical elements***. Since sub-arrays of ***sorted / identical*** elements crop up a lot towards the end of a sorting procedure on a large set, versions of the ***quicksort*** algorithm that choose the ***pivot*** as the **middle element** run *much more quickly* than the algorithm described in this diagram on large sets of numbers.



❒ ***Performance summery:*** **Worst-case** performance: $O(n^2)$
Best-case performance: $O(n \log n)$ (***simple partition***) or $O(n)$ (***three-way partition*** and equal keys)
Average performance: $O(n \log n)$

❒ ***Choice of pivot:*** There are many different versions of quickSort that pick pivot in different ways.

- Always pick **first** element as pivot.
- Always pick **last** element as pivot (implemented below)
- Pick a **random** element as pivot.
- Pick **median** as pivot.

☞ ***Already sorted arrays :*** In the very early versions of **quicksort**, the ***leftmost element*** of the partition would often be chosen as the **pivot** element. Unfortunately, this causes **worst-case** behavior on ***already sorted arrays***. The problem was easily solved by

- ▶ choosing either a ***random index*** for the **pivot**,
- ▶ choosing the ***middle index*** of the partition ***or***
- ▶ (especially for longer partitions) choosing the ***median of the first, middle and last element*** of the partition for the pivot.

[This "median-of-three" rule counters the case of sorted (or reverse-sorted) input, and gives a better estimate of the optimal pivot (the true median) than selecting any single element, when no information about the ordering of the input is known.]

❒ ***Repeated elements:*** **Quicksort** exhibits ***poor performance*** for inputs that contain many ***repeated elements***. The problem is clearly apparent when all the input elements are equal: at each recursion, the left partition is empty (no input values are less than the pivot), and the right partition has only decreased by one element (the pivot is removed).

❒ ***Example:*** Consider the unsorted array `my_list = [1, 12, 33, 14, 52, 16, 71, 18, 94]`. In this case ***bubblesort*** is faster than ***quicksort,*** because:

☞ **Quicksort** gives best performance for *very large quantity* of data, for *small quantity* of data, **bubblesort** is efficient.

☞ **Quicksort** exhibits ***poor performance*** for inputs that contain ***pre-sorted elements.***

❒ ***The cleanest implementation of Quicksort is recursive:*** The ***Quicksort*** is built on the idea of ***partitions***. The general procedure is to select a value, called the ***comparand*** (or ***pivot***), and then to partition the array into two sections. All elements ***greater than or equal*** to the partition value are put on one side, and those less than the value are put on the other. This process is then repeated for each remaining section until the array is sorted.

➢ Eg: given the array **fedacb** and using the value **d** as the ***comparand***, the first pass of the ***Quicksort*** would rearrange the array as:

Initial f e d a c b ***Pass1*** b c a d e f	This process is then repeated for each section—that is, **bca** and **def**. As you can see, the process is essentially recursive in nature, and indeed, the **cleanest implementation of Quicksort is recursive**.

❒ You can select the ***comparand*** value in two ways. You can either choose it at ***random***, or you can select it by averaging a small set of values taken from the array.
For optimal sorting, you should select a value that is precisely in the ***middle*** of the range of values. However, this is not easy to do for most sets of data. In the worst case, the value chosen is at one ***extremity (largest/smallest value)***. Even in this case, however, ***Quicksort*** still performs correctly. In our version, we will selects the middle element of the array as the ***comparand***.

☞ ***Steps:***

[1] First, create the **Quicksort** class.
[2] The **Quicksort** class provides the **qsort()** method, which sets up a call to the actual ***Quicksort method***, **qs()**. This enables the Quicksort to be called with just the name of the array to be sorted, without having to provide an initial partition. Since ***qs()*** is only used internally, it is specified as ***private***.
[3] To use the ***Quicksort***, simply call ***Quicksort.qsort()***. Since ***qsort()*** is specified as ***static***, it can be called through its class rather than on an object. Thus, there is no need to create a Quicksort object. After the call returns, the array will be sorted. Remember, this version works only for character arrays, but you can adapt the logic to sort any type of arrays you want.

```
class Quicksort {                    /* call to the actual Quicksort method. */
static void qsort(char items[]) { qs(items, 0, items.length-1); }

                    /* A recursive version of Quicksort for characters.*/
private static void qs(char items[], int left, int right){  int i, j;
                                                            char x, y;
                                                            i = left; j = right;
                    /* pivot */                             x = items[(left+right)/2];
          do {      while((items[i] < x) && (i < right)) i++; /* skipping partitioned */
                    while((x < items[j]) && (j > left)) j--;  /* skipping partitioned */
                    if(i <= j) {          y = items[i];
                                          items[i] = items[j];        /* swapping */
                                          items[j] = y;
                                          i++; j--;   }
          } while(i <= j);                                  /* Do loop ends */
          if(left < j) qs(items, left, j);                  /* recursive call */
          if(i < right) qs(items, i, right);                /* recursive call */
     }    }
```

```
class QSDemo {
public static void main(String args[]) {
    char a[] = { 'd', 'x', 'a', 'r', 'p', 'j', 'i' };
    int i;
    System.out.print("Original array: ");
    for(i=0; i < a.length; i++)
                System.out.print(a[i]);
    System.out.println();

          // now, sort the array
    Quicksort.qsort(a);
    System.out.print("Sorted array: ");
    for(i=0; i < a.length; i++)
                System.out.print(a[i]);
}}
```

2.29 Nested and Inner Classes

In Java, you can define a ***nested class***. This is a class that is declared within another class. A nested class does not exist independently of its enclosing class.

➲ The scope of a **nested** class is bounded by its **outer** class. A nested class that is *declared directly* within its enclosing **class scope** is a **member** of its enclosing class. It is also possible to declare a nested class that is local to a block.

❒ There are two general types of nested classes: **Static** and **non-Static**

☞ ***Non-Static*** type of nested class is also called an ***inner class***. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

☞ Sometimes an inner class is used to provide a set of services that is used only by its enclosing class. Eg:

```
class Outer{ int nums[];
       Outer(int n[]) { nums = n; }
       void analyze(){Inner inOb = new Inner();    /* using default constructor */
                     System.out.println("Minimum: " + inOb.min());
                     System.out.println("Maximum: " + inOb.max());
                     System.out.println("Average: " + inOb.avg());     }
           // This is an inner class.
       class Inner {int min(){int m = nums[0];
                              for(int i=1; i < nums.length; i++) if(nums[i] < m) m = nums[i];
                              return m;     }
                    int max(){int m = nums[0];
                              for(int i=1; i < nums.length; i++) if(nums[i] > m) m = nums[i];
                              return m;     }
                    int avg(){int a = 0;
                              for(int i=0; i < nums.length; i++) a += nums[i];
                              return a / nums.length;    }
                 }      /* inner closed */
       }   /* outer closed */
class NestedClassDemo { public static void main(String args[]) {
                                              int x[] = { 3, 2, 1, 5, 6, 9, 7, 8 };
                                              Outer outOb = new Outer(x);
                                              outOb.analyze(); }}
```

☞ The inner class ***Inner*** computes various values from the array ***nums***, which is a member of ***Outer***. As explained, an inner class has access to the members of its enclosing class, so it is perfectly acceptable for Inner to access the ***nums*** array directly.

▶ Of course, the opposite is not true. For example, it would not be possible for ***analyze()*** to invoke the ***min()*** method directly, without creating an ***Inner*** object.

❒ It is possible to nest a class within a block scope. Doing so simply creates a localized class that is not known outside its block. Eg:

class LocalClassDemo { **public static void main(String args[])** { **class** ShowBits { */* A local class nested inside a method */* } }}

☞ The **ShowBits** class is not known outside of **main()**, and any attempt to access it by any method other than **main()** will result in an error.

❒ ***Static nested class:*** A **static nested class** is one that has the **static** modifier applied. Because it is ***static***, it can access only other ***static members*** of the enclosing class directly. It must access other members of its outer class through an **object reference**.

❒ ***Annymous inner class:*** You can create ***an inner class that does not have a name***. This is called an **anonymous inner class**. An object of an *anonymous inner class* is ***instantiated*** when the class is declared, using ***new***.

2.30 Varargs: Variable-Length Arguments (Recall C/C++ 5.1.4)

Sometimes you will want to create a method that takes a variable number of arguments, based on its precise usage. For example, a method that opens an Internet connection might take a user name, password, file name, protocol, and so on, but supply defaults if some of this information is not provided.

❒ In the past, methods that required a variable-length argument list could be handled two ways:

➲ First, if the **maximum number** of **arguments** was small and known, then you could create overloaded versions of the method, one for each way the method could be called.

➲ A second approach was used in which the **arguments** were put into an **array**, and then the array was passed to the method.

☞ In JDK5 ***varargs*** is introduced, which is short for ***variable-length arguments***. A method that takes a variable number of arguments is called a ***variable-arity method***, or simply a ***varargs method***. The parameter list for a ***varargs method*** is not fixed, but rather variable in length. Thus, a ***varargs method*** can take a variable number of arguments.

❒ ***Varargs Basics:*** A variable-length argument is specified by three periods (**...**). Eg: variable argument version of **vaTest()** is:

```
class VarArgs {                          /* vaTest() uses a vararg.*/
        static void vaTest(int ... v){ System.out.println("Number of args: " + v.length);
                                       System.out.println("Contents: ");
                                       for(int i=0; i<v.length; i++) System.out.println("arg"+i+":" + v[i]);          }
        public static void main(String args[]) {     vaTest(10);        // 1 arg
                                                      vaTest(1, 2, 3);   // 3 args
                                                      vaTest();          /* no args */   }}
```

➢ Notice that ***v*** is declared as shown here: "***int ... v***". This syntax tells the compiler that ***vaTest()*** can be called with **zero or more arguments**.

➢ Furthermore, it causes ***v*** to be ***implicitly declared as an array*** of type ***int[]***. Thus, inside ***vaTest()***, ***v*** is accessed using the normal array syntax.

➢ Also notice how ***vaTest()*** can be called with a variable number of arguments. In ***main()***, ***vaTest()*** is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to ***v***. In the case of no arguments, the length of the array is ***zero***.

❒ A method can have "**normal**" parameters along with a **variable-length** parameter. However, the variable-length parameter must be the ***last parameter*** declared by the method. Eg: ***int doIt(int a, int b, double c, int ... vals){ }.*** In this case, the **first three** arguments used in a call to ***doIt()*** are matched to the first three parameters. Then, any remaining arguments are assumed to belong to ***vals***.

☞ ***Restriction:*** the **varargs** parameter must be ***last***. For example, the following declaration is **incorrect**:

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error! }
```

Here, there is an attempt to declare a regular parameter after the ***varargs parameter***, which is ***illegal***.

☞ ***Restriction:*** there must be ***only one*** **varargs** parameter. For example, this declaration is also invalid:

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error! }
```

The attempt to declare the second varargs parameter is illegal.

❒ ***Overloading Varargs Methods:*** You can overload a method that takes a variable-length argument. Eg:

```
class VarArgs3 { static void vaTest(int ... v) { /* code-blok*/ }              /* 1st version */
                 static void vaTest(boolean ... v) { /* code-blok*/  }         /* 2nd version */
                 static void vaTest(String msg, int ... v){ /* code-blok*/ } /* 3rd version */

   public static void main(String args[]){ vaTest(1, 2, 3); /* invoke 1st version */
                                           vaTest("Testing: ", 10, 20); /* invoke 3rd version */
                                           vaTest(true, false, false); /* invoke 2nd version */     }}
```

☞ This program illustrates both ways that a varargs method can be overloaded.

➢ First, the types of its ***vararg*** parameter can differ. This is the case for ***vaTest(int ...)*** and ***vaTest(boolean ...)***. *[Therefore, just as you can overload methods by using different types of array parameters, you can overload varargs methods by using different types of varargs. In this case, Java uses the type difference to determine which overloaded method to call.]*

➢ The second way to overload a ***varargs*** method is to add one or more normal parameters. This is what was done with ***vaTest(String, int ...)***. [In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.]

❒ ***Varargs and Ambiguity:*** Somewhat unexpected errors can result when overloading a method that takes a **Varargs**. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded **varargs** method.

☞ ***Case 1:***

```
class VarArgs4 {    static void vaTest(int ... v) { // ... }       // Use an int vararg parameter.
                    static void vaTest(boolean ... v) { // ... }  // Use a boolean vararg parameter.
   public static void main(String args[]) {vaTest(1, 2, 3);                 // OK
                                           vaTest(true, false, false);    // OK
                                           vaTest();     /* Error: Ambiguous!*/    }}
```

➢ Here the overloading of ***vaTest()*** is perfectly correct. However, this program will not compile because of the call:

```
vaTest(); // Error: Ambiguous!
```

Because the vararg parameter can be empty, this call could be translated into a call to ***vaTest(int ...)*** or to ***vaTest(boolean ...)***. Both are equally valid. Thus, the call is inherently ambiguous.

☞ ***Case 2:*** Here is another example of ambiguity. The following overloaded versions of ***vaTest()*** are inherently ambiguous even though one takes a normal parameter:

```
static void vaTest(int ... v) { /* ... */  }
static void vaTest(int n, int ... v) { /* ... */  }
```

➢ Although the parameter lists of ***vaTest()*** differ, there is no way for the compiler to resolve the call: ***vaTest(1)***
Does this translate into a call to ***vaTest(int ...)***, with ***one varargs*** argument, or into a call to ***vaTest(int, int ...)*** with ***no varargs*** arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

☞ Because of **ambiguity errors** like those just shown, sometimes you will need to forego overloading and simply use two different method names.