Java Arrays

Normally, an array is a collection of similar type of elements which have a contiguous memory location.

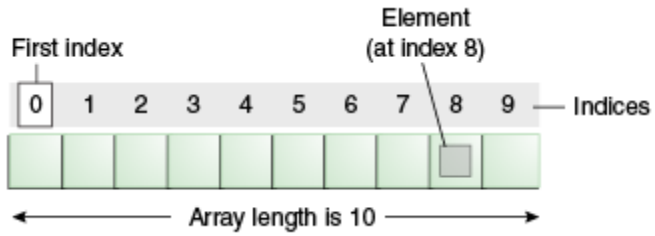
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.
- Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array
- Single/One Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. class Testarray{
4. public static void main(String args[]){
5. int a[]=new int[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. for(int i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
1. int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. class Testarray1{
4. public static void main(String args[]){
5. int a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
```

```
7. for(int i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9. }}
10. Output:
11. 33
12. 3
13. 4
14. 5
```

Multidimensional Arrays in Java

Array-Basics in Java

Multidimensional Arrays **can be defined in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).**

Syntax:

data_type*[1st dimension][2nd dimension][]..[Nth dimension]* ***array_name*** *=* ***new data_type****[size1][size2]….[sizeN];*

where:

- **data_type**: Type of data to be stored in the array. For example: int, char, etc.
- **dimension**: The dimension of the array created. For example: 1D, 2D, etc.
- **array_name**: Name of the array
- **size1, size2, …, sizeN**: Sizes of the dimensions respectively.

Examples:

Two dimensional array:

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

Three dimensional array:

```
int[][][] threeD_arr = new int[10][20][30];
```

Size of multidimensional arrays: The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

The array **int[][] x = new int[10][20]** can store a total of (10*20) = 200 elements.
Similarly, array **int[][][] x = new int[5][10][20]** can store a total of (5*10*20) = 1000 elements.

Two – dimensional Array (2D-Array)

Two – dimensional array is the simplest form of a multidimensional array. A two – dimensional array can be seen as an array of one – dimensional array for easier understanding.

Indirect Method of Declaration:

- **Declaration – Syntax:**
- **data_type[][] array_name = new data_type[x][y];**
- For example: int[][] arr = new int[10][20];

- **Initialization – Syntax:**
- **array_name[row_index][column_index] = value;**
- For example: arr[0][0] = 1;

Example:
filter_none

edit
play_arrow
brightness_4

```
class GFG {

    public static void main(String[] args)

    {

        int[][] arr = new int[10][20];

        arr[0][0] = 1;



        System.out.println("arr[0][0] = " + arr[0][0]);

    }

}
```

Output:

```
arr[0][0] = 1
```

Direct Method of Declaration:
Syntax:

```
data_type[][] array_name = {
                        {valueR1C1, valueR1C2, ....},
                        {valueR2C1, valueR2C2, ....}
                       };
```

For example: int[][] arr = {{1, 2}, {3, 4}};
Example:
filter_none

edit
play_arrow
brightness_4

```
class GFG {

    public static void main(String[] args)

    {
```

```
        int[][] arr = { { 1, 2 }, { 3, 4 } };
            for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
                System.out.println("arr[" + i + "][" + j + "] = "
                                   + arr[i][j]);
    }
}
```

Output:

```
arr[0][0] = 1
arr[0][1] = 2
arr[1][0] = 3
arr[1][1] = 4
```

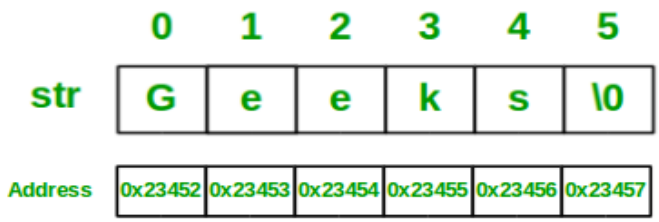
Strings in Java

Strings in Java are Objects that are backed internally by a char array. Since arrays are immutable(cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created.

Below is the basic syntax for declaring a string in **Java programming** language.
Syntax:
<String_Type> <string_variable> = "<sequence_of_string>";

Example:

```
String str = "Geeks";
```

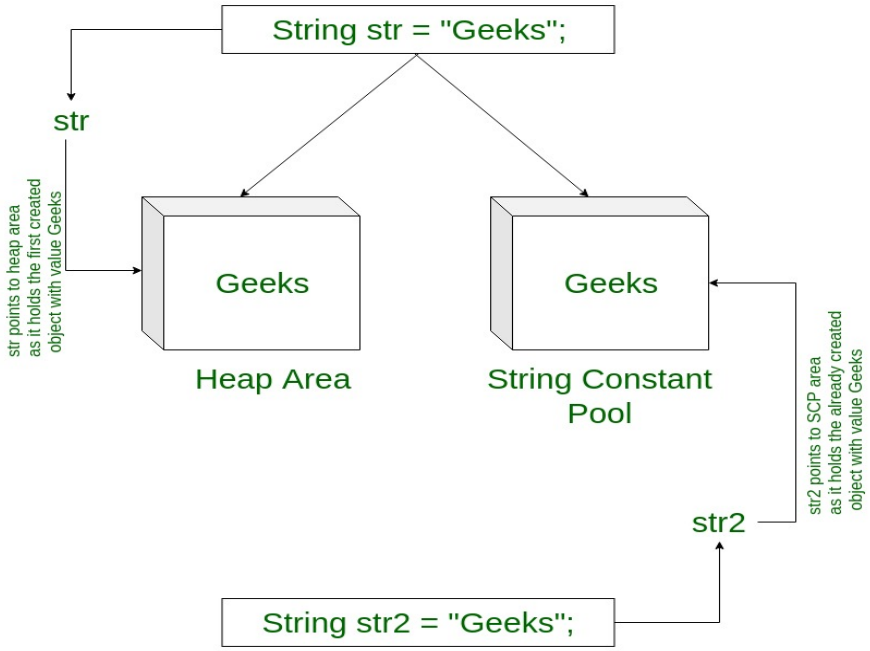


Memory allotment of String

Whenever a String Object is created, two objects will be created- one in the Heap Area and one in the String constant pool and the String object reference always points to heap area object.

For example:

```
String str = "Geeks";
```



An Example that shows how to declare String

```
// Java code to illustrate String

import java.io.*;
import java.lang.*;

class Test {
   public static void main(String[] args)
   {
      // Declare String without using new operator
      String s = "GeeksforGeeks";

      // Prints the String.
      System.out.println("String s = " + s);

      // Declare String using new operator
      String s1 = new String("GeeksforGeeks");

      // Prints the String.
      System.out.println("String s1 = " + s1);
   }
}
```

Output:

```
String s = GeeksforGeeks

String s1 = GeeksforGeeks
```

Java - The Vector Class

Vector implements a dynamic array. It is similar to ArrayList, but with two differences −

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

Sr.No.	Constructor & Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10.
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
4	**Vector(Collection c)** This constructor creates a vector that contains the elements of collection c.

Apart from the methods inherited from its parent classes, Vector defines the following methods −

Sr.No.	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position in this Vector.
2	**boolean add(Object o)** Appends the specified element to the end of this Vector.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.

4	**boolean addAll(int index, Collection c)** Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5	**void addElement(Object obj)** Adds the specified component to the end of this vector, increasing its size by one.
6	**int capacity()** Returns the current capacity of this vector.
7	**void clear()** Removes all of the elements from this vector.
8	**Object clone()** Returns a clone of this vector.
9	**boolean contains(Object elem)** Tests if the specified object is a component in this vector.
10	**boolean containsAll(Collection c)** Returns true if this vector contains all of the elements in the specified Collection.
11	**void copyInto(Object[] anArray)** Copies the components of this vector into the specified array.
12	**Object elementAt(int index)** Returns the component at the specified index.
13	**Enumeration elements()** Returns an enumeration of the components of this vector.
14	**void ensureCapacity(int minCapacity)** Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
15	**boolean equals(Object o)** Compares the specified Object with this vector for equality.

16	**Object firstElement()** Returns the first component (the item at index 0) of this vector.
17	**Object get(int index)** Returns the element at the specified position in this vector.
18	**int hashCode()** Returns the hash code value for this vector.
19	**int indexOf(Object elem)** Searches for the first occurence of the given argument, testing for equality using the equals method.
20	**int indexOf(Object elem, int index)** Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
21	**void insertElementAt(Object obj, int index)** Inserts the specified object as a component in this vector at the specified index.
22	**boolean isEmpty()** Tests if this vector has no components.
23	**Object lastElement()** Returns the last component of the vector.
24	**int lastIndexOf(Object elem)** Returns the index of the last occurrence of the specified object in this vector.
25	**int lastIndexOf(Object elem, int index)** Searches backwards for the specified object, starting from the specified index, and returns an index to it.
26	**Object remove(int index)** Removes the element at the specified position in this vector.
27	**boolean remove(Object o)** Removes the first occurrence of the specified element in this vector, If the vector

	does not contain the element, it is unchanged.
28	**boolean removeAll(Collection c)** Removes from this vector all of its elements that are contained in the specified Collection.
29	**void removeAllElements()** Removes all components from this vector and sets its size to zero.
30	**boolean removeElement(Object obj)** Removes the first (lowest-indexed) occurrence of the argument from this vector.
31	**void removeElementAt(int index)** removeElementAt(int index).
32	**protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
33	**boolean retainAll(Collection c)** Retains only the elements in this vector that are contained in the specified Collection.
34	**Object set(int index, Object element)** Replaces the element at the specified position in this vector with the specified element.
35	**void setElementAt(Object obj, int index)** Sets the component at the specified index of this vector to be the specified object.
36	**void setSize(int newSize)** Sets the size of this vector.
37	**int size()** Returns the number of components in this vector.
38	**List subList(int fromIndex, int toIndex)** Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.

39	**Object[] toArray()** Returns an array containing all of the elements in this vector in the correct order.
40	**Object[] toArray(Object[] a)** Returns an array containing all of the elements in this vector in the correct order; the runtime type of the returned array is that of the specified array.
41	**String toString()** Returns a string representation of this vector, containing the String representation of each element.
42	**void trimToSize()** Trims the capacity of this vector to be the vector's current size.

Example

The following program illustrates several of the methods supported by this collection −

```
import java.util.*;
public class VectorDemo {

  public static void main(String args[]) {
    // initial size is 3, increment is 2
    Vector v = new Vector(3, 2);
    System.out.println("Initial size: " + v.size());
    System.out.println("Initial capacity: " + v.capacity());

    v.addElement(new Integer(1));
    v.addElement(new Integer(2));
    v.addElement(new Integer(3));
    v.addElement(new Integer(4));
    System.out.println("Capacity after four additions: " + v.capacity());

    v.addElement(new Double(5.45));
    System.out.println("Current capacity: " + v.capacity());

    v.addElement(new Double(6.08));
    v.addElement(new Integer(7));
    System.out.println("Current capacity: " + v.capacity());

    v.addElement(new Float(9.4));
    v.addElement(new Integer(10));
    System.out.println("Current capacity: " + v.capacity());

    v.addElement(new Integer(11));
    v.addElement(new Integer(12));
    System.out.println("First element: " + (Integer)v.firstElement());
```

```
    System.out.println("Last element: " + (Integer)v.lastElement());

    if(v.contains(new Integer(3)))
      System.out.println("Vector contains 3.");

    // enumerate the elements in the vector.
    Enumeration vEnum = v.elements();
    System.out.println("\nElements in vector:");

    while(vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```

This will produce the following result −

Output

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.

- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing
The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
1. //Java program to convert primitive into objects
2. //Autoboxing example of int to Integer
```

```
3. public class WrapperExample1{
4. public static void main(String args[]){
5. //Converting int into Integer
6. int a=20;
7. Integer i=Integer.valueOf(a);//converting int into Integer explicitly
8. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
9.
10. System.out.println(a+" "+i+" "+j);
11. }}
```

Output:

```
20 20 20
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
1. //Java program to convert object into primitives
2. //Unboxing example of Integer to int
3. public class WrapperExample2{
4. public static void main(String args[]){
5. //Converting Integer to int
6. Integer a=new Integer(3);
7. int i=a.intValue();//converting Integer to int explicitly
8. int j=a;//unboxing, now compiler will write a.intValue() internally
9.
10. System.out.println(a+" "+i+" "+j);
11. }}
```

Output:

```
3 3 3
```

Java Wrapper classes Example

```
1. //Java Program to convert all primitives into its corresponding
2. //wrapper objects and vice-versa
3. public class WrapperExample3{
4. public static void main(String args[]){
5. byte b=10;
6. short s=20;
7. int i=30;
8. long l=40;
9. float f=50.0F;
10. double d=60.0D;
11. char c='a';
```

```
12. boolean b2=true;
13.
14. //Autoboxing: Converting primitives into objects
15. Byte byteobj=b;
16. Short shortobj=s;
17. Integer intobj=i;
18. Long longobj=l;
19. Float floatobj=f;
20. Double doubleobj=d;
21. Character charobj=c;
22. Boolean boolobj=b2;
23. //Printing objects
24. System.out.println("---Printing object values---");
25. System.out.println("Byte object: "+byteobj);
26. System.out.println("Short object: "+shortobj);
27. System.out.println("Integer object: "+intobj);
28. System.out.println("Long object: "+longobj);
29. System.out.println("Float object: "+floatobj);
30. System.out.println("Double object: "+doubleobj);
31. System.out.println("Character object: "+charobj);
32. System.out.println("Boolean object: "+boolobj);
33. //Unboxing: Converting Objects to Primitives
34. byte bytevalue=byteobj;
35. short shortvalue=shortobj;
36. int intvalue=intobj;
37. long longvalue=longobj;
38. float floatvalue=floatobj;
39. double doublevalue=doubleobj;
40. char charvalue=charobj;
41. boolean boolvalue=boolobj;
42. //Printing primitives
43. System.out.println("---Printing primitive values---");
44. System.out.println("byte value: "+bytevalue);
45. System.out.println("short value: "+shortvalue);
46. System.out.println("int value: "+intvalue);
47. System.out.println("long value: "+longvalue);
48. System.out.println("float value: "+floatvalue);
49. System.out.println("double value: "+doublevalue);
50. System.out.println("char value: "+charvalue);
51. System.out.println("boolean value: "+boolvalue);
52. }}
```

Output:

```
---Printing object values---
```

```
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

2. JAVA APPLETS

Java - Applet Basics

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
  public void paint (Graphics g) {
    g.drawString ("Hello World", 25, 50);
  }
}
```

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet
- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet Class

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

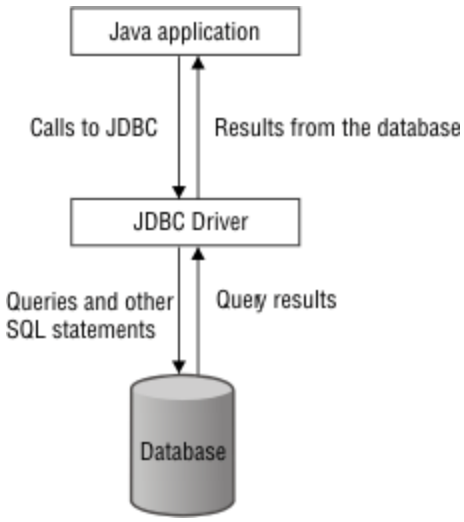
The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

APPLET ARCHITECTURE

When you write a Java application for time series data, you use the JDBC Driver to connect to the database, as shown in the following figure.

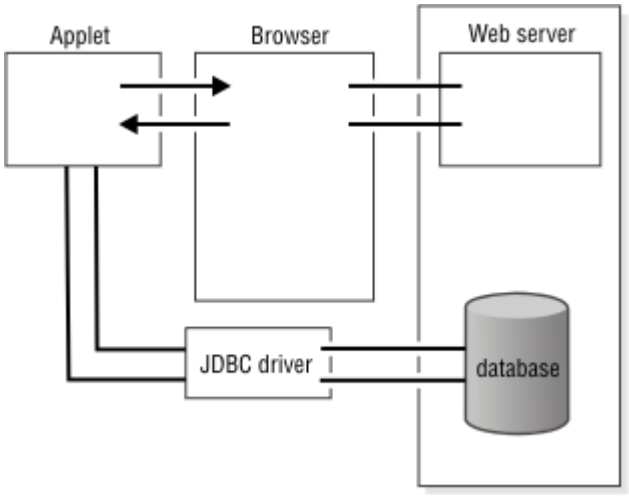
Figure 1. Runtime architecture for Java programs that connect to a database



The Java application makes calls to the JDBC driver, which sends queries and other SQL statements to the database. The database sends query results to the JDBC driver, which sends them on to the Java application.

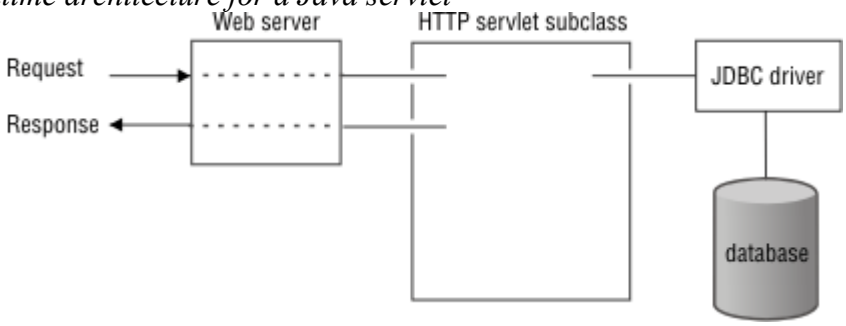
You can also use the time series Java classes in Java applets and servlets, as shown in the following figures.

Figure 2. Runtime architecture for a Java applet



The database server is connected to the JDBC driver, which is connected to the applet. The applet is also connected to a browser, which is connected to a web server that communicates with the database.

Figure 3. Runtime architecture for a Java servlet



A request from an application goes through a web server, an HTTP servlet subclass, and the JDBC driver to the database. The database sends responses back along the same path.

An Applet Skeleton

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.

An Applet Skeleton

APPLET SKELETON

An Applet Skeleton

Most applets override these four methods. These four methods forms Applet lifecycle.

- **init() :** init() is the first method to be called. This is where variable are initialized.

This method is called only once during the runtime of applet.

- **start() :** start() method is called after init(). This method is called to restart an applet

after it has been stopped.

- **stop() :** stop() method is called to suspend thread that does not need to run when applet is not visible.
- **destroy() :** destroy() method is called when your applet needs to be removed completely from memory.

Example of an Applet Skeleton

```
import java.awt.*;
import java.applet.*;
public class AppletTest extends Applet
{
 public void init()
 {
  //initialization
 }
 public void start ()
 {
  //start or resume execution
 }
 public void stop()
 {
  //suspend execution
 {
 public void destroy()
 {
  //perform shutdown activity
 }
 public void paint (Graphics g)
 {
  //display the content of window
 }
}
```

Example of an Applet

```
import java.applet.*;
import java.awt.*;
public class MyApplet extends Applet
{
 int height, width;
 public void init()
 {
  height = getSize().height;
  width = getSize().width;
  setName("MyApplet");
 }
 public void paint(Graphics g)
 {
  g.drawRoundRect(10, 30, 120, 120, 2, 3);
 }
}
```

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

init( )

start( )

paint( )

When an applet is terminated, the following sequence of method calls takes place:

stop( )

destroy( )

Let's look more closely at these methods.

init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )

The **paint( )** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

Overriding update( )

In some situations, an AWT-based applet may need to override another method defined by the AWT, called **update( )**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update( )** simply calls **paint( )**.

However, you can override the **update( )** method so that it performs more subtle repainting. In general, overriding **update( )** is a specialized technique that is not applicable to all applets, and the examples in this chapter do not override **update( )**.

Simple Applet Display Methods: Requesting repainting

AWT-based applets (such as those discussed in this chapter) will also often override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when the applet's output must be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*; import java.applet.*; /*
<applet code="AppletSkel" width=300 height=100> </applet>
*/
public class AppletSkel extends Applet { // Called first.
public void init() { // initialization
}
/* Called second, after init(). Also called whenever the applet is restarted. */
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
/* Called when applet is terminated. This is the last method executed. */
public void destroy() {
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following empty window when viewed with **appletviewer**. Of course, in this and all subsequent examples, the precise look of the **appletviewer** frame may differ based on your execution environment. To help illustrate this fact, a variety of environments were used to generate the screen captures shown throughout this book.

Requesting Repainting

As a general rule, an applet writes to its window only when its **paint( )** method is called by the AWT. This raises an interesting question: How can the applet itself cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? Remember, one of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the run-time system. It cannot create a loop inside **paint( )** that repeatedly scrolls the banner, for example. This would prevent control from passing back to the AWT. Given this constraint, it may seem that output to your applet's window will be difficult at best. Fortunately, this is not the case. Whenever your applet needs to update the information displayed in its window, it simply calls **repaint( )**.

The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**. Thus, for another part of your applet to output to its window, simply store the output and then call **repaint( )**. The AWT will then execute a call to **paint( )**, which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a **String** variable and then call **repaint( )**. Inside **paint( )**, you will output the string using **drawString( )**.

The **repaint( )** method has four forms. Let's look at each one, in turn. The simplest version of **repaint( )** is shown here:

void repaint( )

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

void repaint(int *left*, int *top*, int *width*, int *height*)

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top,* and the width and height of the region are passed in *width* and *height.* These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling **repaint( )** is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately.

Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:

void repaint(long *maxDelay*)

void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called. Beware, though. If the time elapses before **update( )** can be called, it isn't called. There's no return value or exception thrown, so you must be careful.

A Simple Banner Applet

To demonstrate **repaint( )**, a simple banner applet is developed. This applet scrolls a message, from right to left, across the applet's window. Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. The banner applet is shown here:

/* A simple banner applet.

This applet creates a thread that scrolls the message contained in msg right to left across the applet's window.

*/

import java.awt.*; import java.applet.*; /*

<applet code="SimpleBanner" width=300 height=50> </applet>

*/

public class SimpleBanner extends Applet implements Runnable { String msg = " A Simple Moving Banner.";

Thread t = null; int state;

volatile boolean stopFlag;

// Set colors and initialize thread.

public void init() {

setBackground(Color.cyan);

setForeground(Color.red);

}

Start thread public void start() {

t = new Thread(this); stopFlag = false; t.start();

```
}
 //Entry point for the thread that runs the banner.
    public void run() {
 // Redisplay banner
for( ; ; ) {
 try { repaint();
 Thread.sleep(250);
 if(stopFlag)
 break;
 } catch(InterruptedException e) {}
 }
 }
 Pause the banner. public void stop() {
 stopFlag = true; t = null;
 }
 Display the banner.
public void paint(Graphics g) { char ch;
 ch = msg.charAt(0);
 msg = msg.substring(1, msg.length()); msg += ch;
 g.drawString(msg, 50, 30);
 }
 }
```

Following is sample output:

Let's take a close look at how this applet operates. First, notice that **SimpleBanner** extends **Applet**, as expected, but it also implements **Runnable**. This is necessary, since the applet will be creating a second thread of execution that will be used to scroll the banner. Inside **init( )**, the foreground and background colors of the applet are set.

After initialization, the run-time system calls **start( )** to start the applet running. Inside **start( )**, a new thread of execution is created and assigned to the **Thread** variable **t**. Then, the **boolean** variable **stopFlag**, which controls the execution of the applet, is set to **false**.

Next, the thread is started by a call to **t.start( )**. Remember that **t.start( )** calls a method defined by **Thread**, which causes **run( )** to begin executing. It does not cause a call to the version of **start( )** defined by **Applet**. These are two separate methods.

Inside **run( )**, a call to **repaint( )** is made. This eventually causes the **paint( )** method to be called, and the rotated contents of **msg** are displayed. Between each iteration, **run( )** sleeps for a quarter of a second. The net effect is that the contents of **msg** are scrolled right to left in a constantly moving display. The **stopFlag** variable is checked on each iteration. When it is **true**, the **run( )** method terminates.

If a browser is displaying the applet when a new page is viewed, the **stop( )** method is called, which sets **stopFlag** to **true**, causing **run( )** to terminate. This is the mechanism used to stop the thread when its page is no longer in view. When the applet is brought back into view, **start( )** is once again called, which starts a new thread to execute the banner.

Using the Status Window - The Applet Class

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running.

Using the Status Window

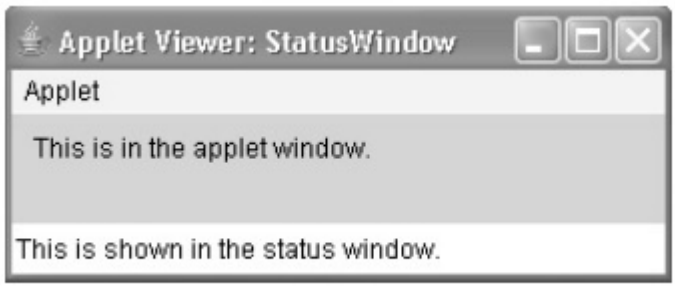
In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **showStatus( )** with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

The following applet demonstrates **showStatus( )**:

```
// Using the Status Window.
import java.awt.*;
import java.applet.*; /*
<applet code="StatusWindow" width=300 height=50> </applet>
*/
```

```
public class StatusWindow extends Applet { public void init() {
 setBackground(Color.cyan);
 }
 // Display msg in applet window.
public void paint(Graphics g) {
 g.drawString("This is in the applet window.", 10, 20);
showStatus("This is shown in the status window.");
 }
 }
```

Sample output from this program is shown here:



HTML | applet Tag

The **<applet>** tag in HTML was used to *embed Java applets into any HTML document*. The **<applet>** tag was deprecated in HTML 4.01, and it's support has been completely discontinued starting from HTML 5. Alternatives available in HTML 5 are the <embed> and the <object> tags.

Syntax:

```
<applet attribute1 attribute2....>
  <param parameter1>
  <param parameter2>
  ....
</applet>
```

Attributes: The <applet> tag takes a number of attributes, with one of the most important being the **code** attribute. This **code** attribute is used to link a Java applet to the concerned HTML document. It specifies the file name of the Java applet.

```
<!DOCTYPE html>

<html>

<applet code="HelloWorld" width=200 height=60>

   </applet>
```

```
</html>
```

Here, **HelloWorld** is the class file, which contains the applet.
The **width** and **height** attributes determine the width and height of the applet in pixels when it is opened in the browser.
Attributes available to be used in conjunction with the **<applet>** tag are as follows:

ATTRIBUTE NAME	VALUES	REMARKS
align	left right top bottom middle baseline	Specifies the alignment of an applet.
alt	text	Specifies an alternate text for an applet
archive	URL	Specifies the location of an archive file
border	pixels	Specifies the border around the applet panel
codebase	URL	Specifies a relative base URL for applets specified in the code attribute
height	pixels	Specifies the height of an applet
hspace	pixels	Defines the horizontal spacing around an applet
mayscript	Any arbitrary value (conventionally	Indicates whether the Java applet is allowed to access the scripting objects

	"mayscript")	of the web page
name	name	Defines the name for an applet (to use in scripts)
vspace	pixels	Defines the vertical spacing around an applet
width	pixels	Specifies the width of an applet

Passing parameters to applet

Parameters: Parameters are quite similar to command-line arguments in the sense that they provide a way to pass information to the applet after it has started. All the information available to the applet before it starts is said to be hard-coded i.e. embedded within it. Parameters make it possible to generate and use data during run-time of the applet.

Syntax:

<param name=*parameter name* value=*parameter value*>

The name assigned to the **name** attribute of the **param** tag is used by the applet code as a variable to access the parameter value specified in the **value** attribute. In this way, the applet is able to interact with the HTML page where it is embedded, and can work on values provided to it by the page during run-time.

```
<!DOCTYPE html>
<html>
   <applet code="HelloWorld" width=200 height=60>
   <param name="message" value="HelloWorld">
   </applet>
</html>
```

In this piece of code, the applet file HelloWorld can use the variable named **message** to access the value stored in it, which is **"HelloWorld"**.

Note: There is still some browsers that support the <applet> tag with the help of some additional plug-ins/installations to work. Internet Explorer 11 and earlier versions with the help of plug-ins.

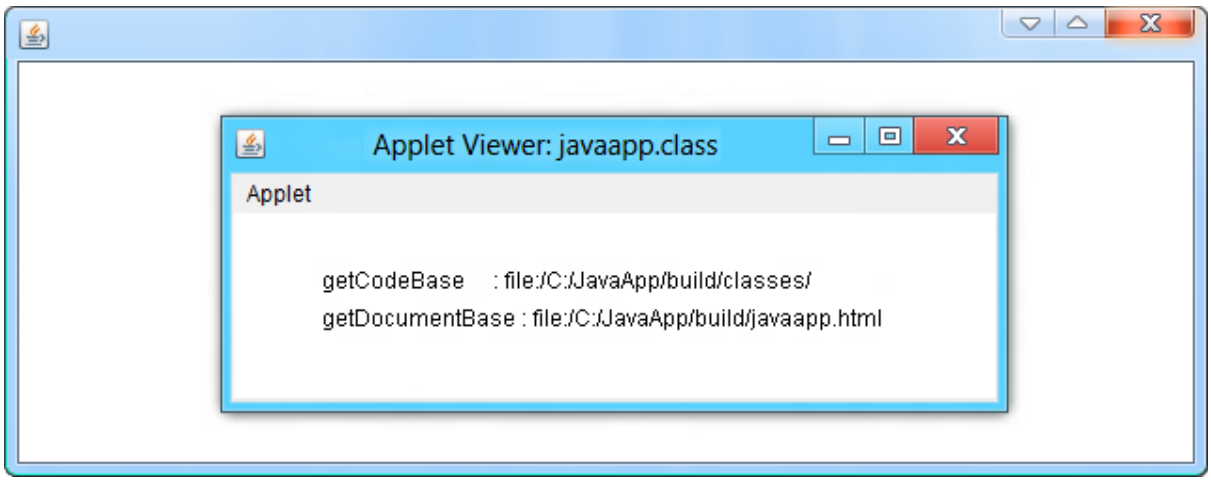
Supported Browsers: The browser supported by <applet> tag are listed below :

- Firefox
- Safari

GetDocumentBase and getCodeBase Example

In most of the applets, it is required to load text and images explicitly. Java enables loading data from two directories. The first one is the directory which contains the HTML file that started the applet (known as the **document base).** The other one is the directory from which the class file of the applet is loaded (known as the **code base).** These directories can be obtained as URL objects by using getDocumentBase () and getCodeBase ()methods

respectively. You can concatenate these URL objects with the string representing the name of the file that is to be loaded.



Java-GetCodeBase and GetDocumentBase

You will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the **html file** that started the applet (the document base) and the directory from which the **applet's class file** was loaded (the code base). These directories are returned by **getDocumentBase**( ) and **getCodeBase**( ).

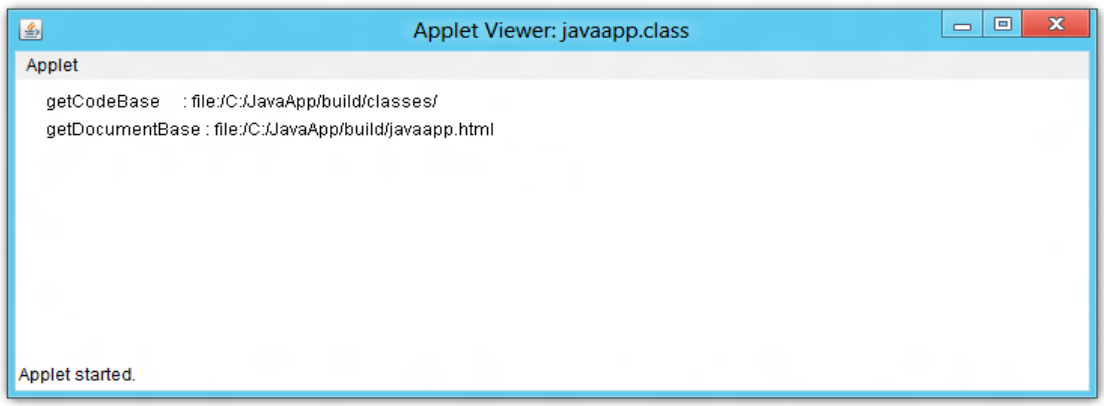
Program

```
import java.applet.Applet;
import java.awt.Graphics;

public class Javaapp extends Applet{

    public void paint(Graphics g)
    {
        g.drawString("getCodeBase     : "+getCodeBase(), 20, 20);
        g.drawString("getDocumentBase : "+getDocumentBase(), 20, 40);
    }
}
```

Program Output



show Document()

Java allows the applet to transfer the control to another URL by using the showDocument () Method defined in the AppletContext interface. For this, first of all, it is needed to obtain the Context of the currently executing applet by calling the getAppletContext () method defined

by the Applet. Once the context of the applet is obtained with in an applet, another document can be brought into view by calling showDocument () method.

There are two showDocument () methods which are as follows:

showDocument(URL url)

showDocument(URL url,string lac)

where,

url is the URL from where the document is to be brought into view.

loc is the location within the browser window where the specified document is to be displayed.

Example An applet code to demonstrate the use of AppletContext and showDocument ().

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.net.*;
/*
<applet code="LoadHTMLFileSample" width="700" height="500"></applet>
*/
public class LoadHTMLFileSample extends Applet
{
   public void start()
   { AppletContext context= getAppletContext();
        //get AppletContext
        URL codeBase = getCodeBase(); //get Applet code base
        try{
              URL url = new URL(codeBase + "Test.html");
              context.showDocument(url,"_blank");
              repaint();
            }catch(MalformedURLException mfe) {
                    mfe.printStackTrace();
                    }
    }
}
```