OOP-JAVA
CST 205

Advanced features of JAVA

MODULE 4

Author – Milan George Mathew

Syllabus

Java Library - String Handling – String Constructors, String Length, Special String Operations - Character Extraction, String Comparison, Searching Strings, Modifying Strings, using valueOf(), Comparison of StringBuffer and String.

Collections framework - Collections overview, Collections Interfaces- Collection Interface, List Interface.

Collections Class – ArrayList class. Accessing a Collection via an Iterator.

Event handling - Event Handling Mechanisms, Delegation Event Model, Event Classes, Sources of Events, Event Listener Interfaces, Using the Delegation Model.

Multithreaded Programming - The Java Thread Model, The Main Thread, Creating Thread, Creating Multiple Threads, Synchronization, Suspending, Resuming and Stopping Threads.

Contents

String Handling

String is Java Class that is used to implement string as objects. The **String** type is used to declare string variables. Java has methods to do different operations on string including string comparison, string concatenation, change case etc... A quoted string can b assigned to a **String** variable. A String variable can also be assigned to another String object

String Constructors

String class supports several constructors

Default Constructor

This is used to create an empty String object.

```
String s = new String();
```

Character Array Constructors

There are two basic constructors for a string that accepts character array to store in String Object.

```
String(char chars[ ])
```

Accepts a character array with character separated by quotation marks.

```
String(char chars[ ], int startIndex, int numChars)
```

Accepts a character array and takes **numChars** number of characters from the array starting from **startIndex** to form the String.

String Object Constructor

We can construct a String object that contains the same character sequence as another String object

```
String(String strObj)
```

strObj is an object of the String class.

Byte Array Constructors

String class provides constructors that initialize a string when given a byte array. There are two basic constructors for the same:

```
String(byte asciiChars[ ])
```

Accepts a byte array that stores the ascii value of the charachters.

```
String(byte asciiChars [ ], int startIndex, int numChars)
```

Accepts a byte array and takes **numChars** number of bytes from the array starting from **startIndex** to form the String.

Here asciiChars specifies the array of bytes. In each of these constructors, the **byte-to-character** conversion is done by using the default character encoding of the platform.

String Buffer Constructor

The String Object can be created using a StringBuffer by using:

```
String(StringBuffer strBufObj)
```

A **StringBuffer** in Java is a class that represents a **mutable sequence** of characters. It is part of the **java.lang** package and is designed to provide a flexible and efficient way to manipulate strings. The key characteristic of **StringBuffer** is that it allows for the modification of the content of a string without creating a new object each time, making it suitable for scenarios where strings are frequently modified.

J2SE5 Added Constructors

Extended Unicode character set is used to create a String object

```
String(int codePoints[ ], int startIndex, int numChars)
```

StringBuilder object is used to create a String object:

```
String(StringBuilder strBuildObj)
```

StringBuilder is a class in Java that belongs to the **java.lang** package and is part of the Java Standard Library. It provides a mutable sequence of characters, allowing you to efficiently construct strings by appending, inserting, or deleting characters. The primary advantage of **StringBuilder** over **String** is that it allows you to modify the content of the string without creating a new object each time.

String Operations

Length

We can use the length() method to find the length of a string , the Java creates a String Object for each string literal in the program. This literal can call the length() method on a string.

String Concatenation

String concatenation is used to join two strings. There are two ways that can be used to concatenate strings

- **Concatenation operator** (+) – any two Strings can be concatenated using the + operator in between the two strings.
- **concat() method** – Java String class provides a method that can be used to one string to another.

They both does the same function. The + operator shows different behaviour at different instances

For Example

```
String s = "four: " + 2 + 2;
```

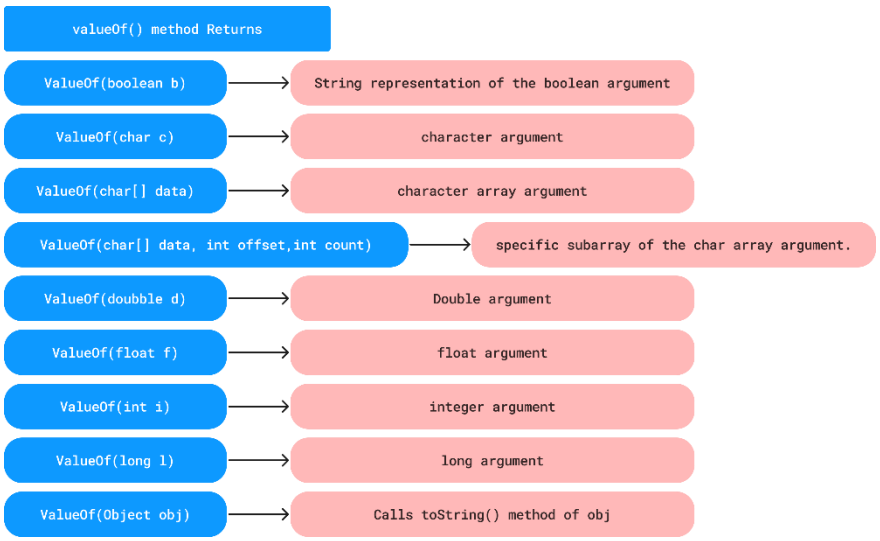
will print **four:22** as the 2 is implicitly converted to a string literal before appending

```
String s = "four: " + (2 + 2);
```

will print **four:4** as there is a parenthesis to denote that the second + is an arithmetic operation

StringConversion and toString()

When Java converts data into its string representation during concatenation, it calls one of the overloaded versions of the string conversion method **valueOf( )** by class **String**. For the simple types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, valueOf( ) calls the **toString( )** method on the Object. The valueOf() returns the string representation of the corresponding argument. Different overloaded form of valueOf() in String class.



The toString() method is overridden in a class then while we ask the valueOf() to print the object will call the overridden method, if it is not overridden it prints **classname@location**

The location is the memory location where the object is store in the memory and is represented as hexadecimal address.

Character extraction

The String class provides the following methods through which characters can be extracted from a String object.

- chatAt()
- getChars()
- getBytes()
- toCharArray()

charAt()

It is used to extract a **single** character from a string located at the specified index.

```
char charAt(int where)
```

where is the index of the character to be extracted

getChars()

It is used to extract **more than one character** to a character array at a time.

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

The sourceStart and sourceEnd are the index value range from which the extraction should be done, they character get extracted from sourceStart to sourceEnd-1. The target is the char array to which the characters will get stored from the starting index of targetStart.

getBytes()

Used to extract the characters in an array of bytes. It uses the default character-to-byte conversions. Most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

```
byte[ ] getBytes( )
```

toCharArray()

Used to convert **all the characters** in a String object into a character array. It returns an array of characters for the entire string.

```
char[ ] toCharArray( )
```

String Comparison

The String class includes several methods that compare strings or substrings within strings.

- **equals()**
- **equalsIgnoreCase()**
- **regionMatches()**
- **strartsWith()**
- **endsWith()**
- **compareTo()**
- **compareToIgnoreCase()**

equals()

It is used to compare two strings for equality.

```
boolean equals(Object str)
```

Here the String object is compare to the invoking object and returns true is both has the same characters in same order. This comparison is case-sensitive

Note: the == operator cannot be used instead of equals as == compares the object references to find if they refer to the same reference. Thus it can return false even if the strings are same.

equalsIgnoreCase()

It is same as the equals() method apart from the case in-sensitivity. That is it will return return true even if the case of letters are not matching, it takes A and a as the same values.

```
boolean equalsIgnoreCase(String str)
```

regionMatches()

The regionMatches( ) method compares a specific region inside a string with another specific region in another string. It has two forms:

```
boolean regionMatches(int startIndex, String str2, int
                      str2StartIndex, int numChars)
```

```
boolean regionMatches(boolean ignoreCase, int startIndex, String
                      str2, int str2StartIndex, int numChars)
```

startIndex specifies the index from which the region begins to be compared.

str2 is the string to which it is compared and it starts from **str2StartIndex** for **numChars** number of characters.

The **ignoreCase** is to set **true** if the case is to be ignored while comparison.

startsWith() and endsWith()

The **startsWith( )** method determines whether a given String begins with a specified string. Conversely, **endsWith( )** determines whether the String in question ends with a specified string. Both of the methods have a general form of:

```
boolean startsWith(String str)

boolean endsWith(String str)
```

startwith has an additional form to specify a starting point.

```
boolean startsWith(String str, int startIndex)
```

compareTo()

This method is used to compare two strings and find out which is greater that the other ii ifts equal. It returns an integer value:

- **zero** – Equal strings
- **Greater than zero**- invoking string object is greater than passed string
- **Less than zero** – invoking string object is lesser than passed string

The comparison is based on **ASCII** values.

```
int compareTo(String str)
```

compareToIgnoreCase()

it is just like the compareTo() method except for its case insensitivity.

```
int compareToIgnoreCase(String str)
```

Searching Strings

Searching String means to find the occurrence of a character or substring.

The two of the methods are **indexOf()** (finds the **first** occurrence) and **lastIndexOf()** (finds the **last** occurrence) methods which are overloaded by many forms

```
int indexOf(int ch)

int lastIndexOf(int ch)
```

these both accept a character in the integer form and then searches for it, if a character is passed it is implicitly casted to integer type.

```
int indexOf(String str)

int lastIndexOf(String str)
```

these both accepts a substring and finds the occurrence and provide the start index as return.

They also have the declarations to specify the starting point of search by the following methods:

```
int indexOf(int ch, int startIndex)

int lastIndexOf(int ch, int startIndex)

int indexOf(String str, int startIndex)

int lastIndexOf(String str, int startIndex)
```

Here startIndex specifies the index at which point the search begins. For **indexOf( ),** the search runs from **startIndex to the end** of the string. For **lastIndexOf( ),** the search runs from **startIndex to zero.**

Modifying Strings

String objects are immutable, thus modifying a string can only be done by copying it to a StringBuffer or StringBuilder or using the following methods

- substring()
- concat()
- replace()
- trim()

substring()

We can extract a substring using substring( ). It has two forms:

```
String substring(int startIndex)

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

Here the startIndex and endIndex are specified to set bounds of the substring to be exctracted. The endIndex character is not included in the extracted string.

concat()

concat() method is used to concatenate two strings.

```
String concat(String str)
```

This method creates a new object that contains the invoking string with the value of str appended to the end of it.

replace()

Replace method is used to replace a character by another or a character sequence by another character sequence. There are two forms to it:

```
String replace(char original, char replacement)
String replace(CharSequence original, CharSequence replacement)
```

The method changes every occurrence of **original** by **replacement.** A character sequence can be a **String, StringBuffer** or a **StringBuilder.**

trim()

The trim( ) method returns a copy of the invoking string after removing any leading and trailing whitespace.The trim( ) method is quite useful when we process user commands.

```
String trim( )
```

StringBuffer

StringBuffer vs String

The string class differs from the StringBuffer class in many ways:

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- **String** represents **fixed-length, immutable** character sequences.
- **StringBuffer** represents **growable and writeable** character sequences.
- **StringBuffer** may have characters and substrings **inserted** in the middle or **appended** to the end.
- **StringBuffer** will **automatically grow** to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

String	StringBuffer
String is immutable	StringBuffer is mutable.
String represents fixed length, immutable character sequences.	StringBuffer represents growable and writeable character sequences
Concatenation using String is slow.	Concatenation StringBuffer is fast. using
String class can override equals() method.	StringBuffer class doesnot override equals() method.

StringBuffer has four constructors define in it:

```
StringBuffer( )

StringBuffer(int size)

StringBuffer(String str)

StringBuffer(CharSequence chars)
```

The default constructor reserves room for 16 characters without reallocation.

StringBuffer has many methods that can be used to manipulate strings in a convenient way

length()

The current length of a StringBuffer can be found via the **length( )** method. The length is the number of characters stored in it.

```
int length( )
```

capacity()

The total allocated capacity can be found through the **capacity( )** method. String Buffer always adds an extra room for **16 characters** therefore the initial capacity will be **16+number of characters in initial string**. Every time the limit exceeds the capacity is doubled to make room for extra characters.

```
int capacity()
```

ensureCapacity()

ensureCapacity( ) is used to set the **size** of the buffer. This is useful if we know in advance that we will be appending a large number of small strings to a **StringBuffer**.

```
void ensureCapacity(int capacity)
```

setLength()

Used to **set the length** of the buffer within a StringBuffer object. When we **increase the size** of the buffer, **null characters** are added to the end of the existing buffer. If we call **setLength()** with a value **less than the current value** returned by length( ), then the **characters stored beyond** the new length will be **lost**.

```
void setLength(int len)
```

charAt( ) and setCharAt( )

The value of a single character can be obtained from a StringBuffer via the charAt( ) method.

We can set the value of a character within a StringBuffer using setCharAt( ).

```
char charAt(int where)
```

```
void setCharAt(int where, char ch)
```

getChars()

Used to copy a substring of a StringBuffer.

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

append()

The **append( )** method **concatenates** the string representation of any other type of data to the end of the **invoking StringBuffer object**.

The valueOf() method is used to convert any type of data to its string represtenation and is used as an argument which results in a return with the current StringBuffer appended with the new string.

```
StringBuffer append(String str)

StringBuffer append(int num)

StringBuffer append(Object obj)
```

insert()

The **insert( )** method inserts one string into another. The argument passed calls the **valueOf()** method and then inserts it to the invoking **StringBuffer** at the **given index**.

```
StringBuffer insert(int index, String str)

StringBuffer insert(int index, char ch)

StringBuffer insert(int index, Object obj)
```

reverse()

We can reverse the characters within a StringBuffer object using reverse( ).

```
StringBuffer reverse( )
```

delete( ) and deleteCharAt( )

We can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( ).**

```
StringBuffer delete(int startIndex, int endIndex)

StringBuffer deleteCharAt(int loc)
```

delete() deletes from **startIndex to endIndex–1.** The **deleteCharAt( )** method deletes the character at the index specified by **loc**

replace()

We can **replace** one set of characters with another set inside a **StringBuffer** object by calling **replace()**

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

substring()

We can obtain a portion of a **StringBuffer** by calling **substring**( ).

```
String substring(int startIndex)
```

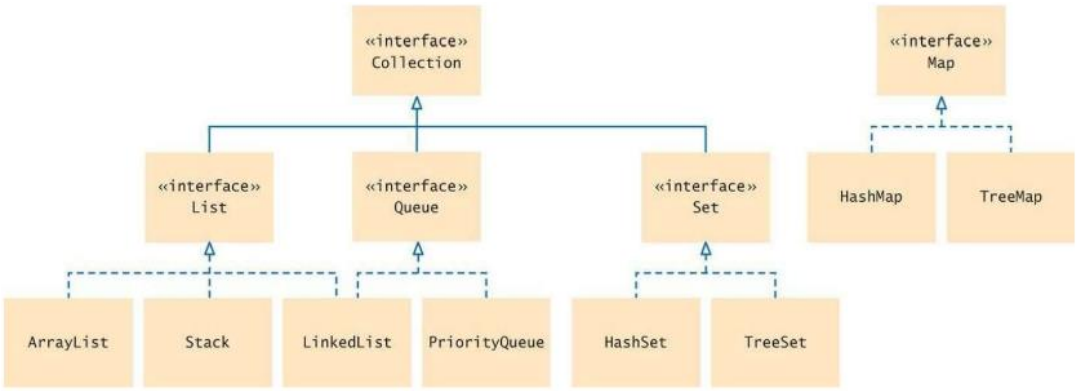
```
String substring(int startIndex, int endIndex)
```

Collections Framework

The **java.util** package contains one of Java's most powerful subsystems: **The Collections Framework**. It is a sophisticated hierarchy of **interfaces and classes** that provide **state-of-the-art technology** for managing **groups of objects**. The Collection in Java is a framework that provides an architecture to **store and manipulate** the group of objects.

Java **Collections framework** provides many

- Intefaces
    - Set
    - List
    - Queue
    - Deque
- Classes
    - ArrayList
    - Vector
    - LinkedList
    - PriorityQueue
    - HashSet
    - LinkedHashSet
    - TreeSet



The Java Collections Framework standardizes the ways in which groups of object are handled by our programs. The entire Collections Framework is built upon a set of standard interfaces. Mechanisms were added that allow the integration of standard arrays into the Collections Framework.

The Collection framework was designed to meet several goals

- **High performance** – the implementations for fundamental collections like dynamic arrays, linked list, trees, hash table etc. are highly efficient
- **Compatibility** - the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- **Extendibility** - extending and/or adapting a collection had to be easy.

Algorithms are an important part of the collection mechanism. Algorithms operate on collections and are defined as **static methods** within the Collections class. The algorithms provide a standard means of manipulating collections.

Java Collections Framework provides algorithm implementations that are commonly used such as

- Sorting
- Searching
- Reversing
- Shuffling
- Minimum
- Maximum
- Counter

Another item closely associated with the **Collections Framework** is the **Iterator interface**. An iterator offers a general-purpose, standardized way of **accessing the elements** within a collection, one at a time. An iterator provides a means of **enumerating the contents** of a collection. Because **each collection implements Iterator**, the elements of **any collection** class can be accessed through the methods defined by **Iterator.**

Maps

The framework defines several map interfaces and classes. They are stored in the form of key-value pairs and they cannot have duplicate keys. Although maps are part of the **Collections Framework**, they are **not** "collections" in the strict use of the term.

Recent changes in the collection framework has significantly increased its power and streamlined its use by the addition of

- **Generics**- With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.
- **Autoboxing or unboxing** - Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. Previously to store primitive values such as an int, in a collection we had to manually box and unbox it with the type-wrapper.
- **For-each** style **for** loop

Collection Interface

Collection interface helps to work with group of objects. The **Collection interface** is at the top of collections **hierarchy**. Collection interface is the **foundation** upon which the Collections Framework is built because it must be **implemented** by any class that defines a collection. **Collection** is a **generic interface** that has this declaration:

```
interface Collection<E>
```

the **E** is the **type of objects** that will be stored in the collection.

Collection extends the **Iterable interface**. This means that all collections can be **cycled through** by use of the **for-each style for loop**.

- **Collection** declares the **core methods** that all collections will have.
- Several of these methods can throw an **UnsupportedOperationException** if a collection **cannot be modified**.
- A **ClassCastException** is generated when one object is **incompatible** with another.
- A **NullPointerException** is thrown if an attempt is made to store a **nullobject** and **null elements are not allowed** in the collection.
- An **IllegalArgumentException** is thrown if an **invalid argument** is used.
- An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length collection that is full.

Some of the standard **Collection interfaces** are:

- Collection
- Deque
- List
- NavigableSet
- Queue
- Set
- SortedSet

add() & addAll()

add( ) is used to add object to a collection and takes an argument of type E, which means that objects added to a collection must be compatible with the type of data expected by the collection.

addAll() is used to add entire contents of a collection to other

remove() , removeAll() , retainAll() & clear()

To remove **an object** call **remove( ).**

To remove **a group of objects**, call **removeAll( ).**

To remove **all elements except** those of a specified group by call **retainAll( ).**

To **empty a collection**, call **clear( ).**

contains() , containsAll() & isEmpty()

To check if a specific object is in a collection use **contains().**

To check whether all member of a collection is in another collection use **containsAll().**

To determine if a collection is empty use **isEmpty().**

size( )

The number of elements currently held in a collection can be determined by calling **size( ).**

toArray()

The **toArray( )** methods return an array that contains the elements stored in the invoking collection. `Object[ ] toArray( )` returns an array of **Object**.

```
Object[ ] toArray( )
```

```
<T>T[ ] toArray(T array[])
```

equals()

Two collections can be compared whether they are equal or not by calling **equals( ).** The precise meaning of "equality" may differ from collection to collection. For example:

- In lists equality means to have same elements in same order
- In sets equality means to have same elements in any order
- equals( ) can be implemented to compare the values of elements stored in the collection.
- equals( ) can be implemented to compare references to those elements.

iterator()

It returns an **iterator** to a collection and helps to loop through a collection like a **for-each style for loop**

List interface

It extends the **Collection interface.** It is a part of the Java Collections Framework and represents an ordered collection of elements. It is the root interface for all list-based collections like **ArrayList, LinkedList**, etc. List declares the behavior of a collection that stores a **sequence of elements**. It stores elements in an ordered collection to store and access them sequentially.

Elements can be inserted or accessed by **their position** in the list, using zero-based index. A list **may** contain **duplicate elements**.

```
interface List<E>
```

List supports methods defined by Collection and has also its own defined methods like:

- **get()** – to get an element by passing its index.
- **set()** – to change the value of an element by accessing it using its index.
- **indexOf() or lastIndexOf()** – to search among the list

- **subList()** – to obtain a specific list by passing start and end indices

Collection Classes

The collection classes **implement collection interfaces**. Some of the collection classes **provide full implementations** that can be used as it is. Some of the collection classes are **abstract**, providing **skeletal implementations** that are used as starting points for creating **concrete collections**. Collection classes are **not synchronized.** Two or more threads can access the methods of collection class at any time.

Standard collection classes are:

- AbstractCollection
- AbstractList
- AbstractQueue
- AbstractSequentialList
- LinkedList
- ArrayList
- ArrayDeque
- AbstractSet
- EnumSet
- HashSet
- LinkedHashSet
- PriorityQueue
- TreeSet

ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List interface**.

```
class ArrayList<E>
```

ArrayList supports **dynamic arrays** that can grow as needed. This is needed because in some cases we may not know how large an array we need precisely **until run time**. Array lists are created with an **initial size**. When this size is **exceeded**, the collection is **automatically enlarged**. When objects are **removed**, the array can be **shrunk**.

There are three main constructors

```
ArrayList( )

ArrayList(Collection <? extends E> c)

ArrayList(int capacity)
```

Some of the methods in the Array list class :

- **toString()** - The contents of a collection are displayed using the default conversion.
- **ensureCapacity()** – it is used to increase the capacity of an ArrayList manually

- **trimToSize()** - to reduce the size of the array that of ArrayList object so that it is precisely as large as the number of items that it is currently holding.

Iterator

To **cycle through** the elements in a collection (e.g. display each element, sum of elements etc.), we can use iterator, which is an object that implements either **Iterator or List Iterator. ListIterator** extends **Iterator** to allow:

- **bidirectional** traversal of list
- **modification** of elements

```
interface Iterator<E>
```

```
interface ListIterator<E>
```

Methods defined by Iterator

`boolean` **`hasNext( )`**

Returns true if there are more elements in the list.

`E` **`next( )`**

Returns the next element and throws **NoSuchElementException** if there is no next element.

`void` **`remove( )`**

Removes the current element and throws **IllegalStateException** if an attempt is made to call remove( ) that is not preceded by a call to next( ).

Methods defined by ListIterator

`void` **`add(E obj)`**

Inserts obj into the list in front of the element that will be returned by the next call to next( ).

`boolean` **`hasNext( )`**

Returns true if there are more elements in the list.

`boolean` **`hasPrevious( )`**

Returns true if there are more elements in the list in backward direction.

`E` **`next( )`**

Returns the next element and throws **NoSuchElementException** if there is no next element.

`E` **`previous( )`**

Returns the previous element and throws **NoSuchElementException** if there is no next element.

`void remove( )`

Removes the current element and throws **IllegalStateException** if an attempt is made to call remove( ) that is not preceded by a call to next( ).

`int nextIndex( )`

Returns the index of the element that would be returned by a subsequent call to next().

`int previousIndex( )`

Returns the index of the element that would be returned by a subsequent call to previous().

`void set(E obj )`

Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Using an Iterator

Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, we can access each element in the collection, one element at a time.

Steps to setup an iterator:

1. Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext() returns true.
3. Within the loop, obtain each element by calling next( ).

For-Each loop

The syntax of Java for-each loop consists of data_type with the variable followed by a colon (:), then array or collection.

```
for(data_type variable : array | collection)
```

The Java for-each loop traverses the array or collection until the last element. For each element, it stores the element in the variable and executes the body of the for-each loop.

If we don't want to modify the contents of a collection or obtaining elements in reverse order, then the for-each version of the for loop is often a more convenient alternative to cycling through a collection than is using an iterator. The for loop is substantially shorter and simpler to use than the iterator based approach.

Event Handling

There are several types of events which can happen with the user interaction or without.

Events can be generated by

- **Mouse** – such as click, move, drag etc.
- **Keyboard** – such a type, press , release etc.
- **GUI components** – such as pressing a button, moving a scroll bar etc.

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.Events are supported by a number of packages including

- **java.util**
- **java.awt**
- **java.awt.event**

Event handling is an integral part in the creation of applets and other types of GUI-based programs. **Applets** are **event-driven** programs that use a **graphical user interface(GUI)** to interact with the user. Any program that uses a graphical user interface is event driven. Thus, we cannot write these types of programs without a solid command of event handling.

There are two event handling mechanisms.

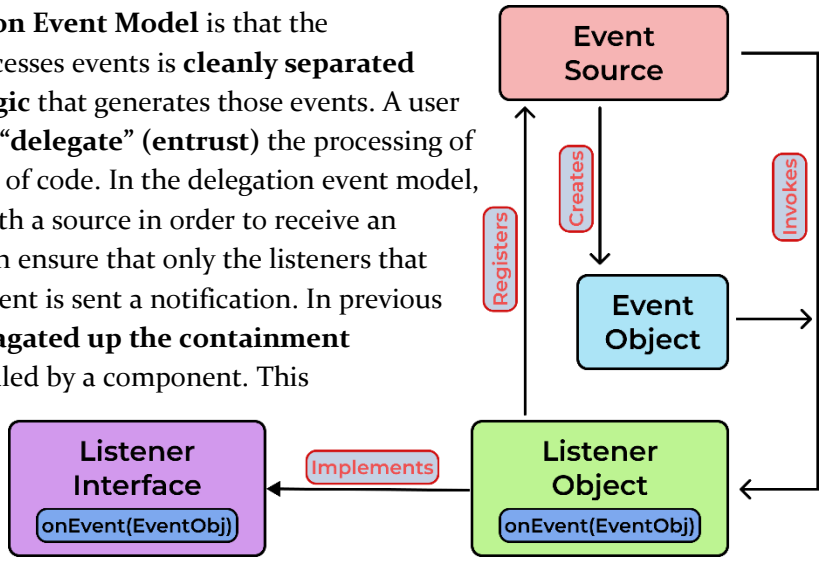
Original version of Java (1.0) event handling is still supported but is not recommended for new programs. Many of the methods that support the old 1.0 event model have been deprecated.

Modern versions of Java (beginning with version 1.1) event handling which is based on the delegation event model. It defines standard and consistent mechanisms to generate and process events.

The concept of Delegation Event Model

- A **source** generates an event and sends it to **one or more listeners**.
- In this scheme, the **listener simply waits** until it receives an event.
- Once an event is received, the **listener processes** the event and then **returns**.

The advantage of **Delegation Event Model** is that the **application logic** that processes events is **cleanly separated** from the **user interface logic** that generates those events. A user interface element is able to "**delegate**" (**entrust**) the processing of an event to a separate piece of code. In the delegation event model, **listeners must register** with a source in order to receive an **event notification**, this can ensure that only the listeners that are meant to handle that event is sent a notification. In previous models, an event was **propagated up the containment hierarchy** until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time.



An event is an object that describes the change of state change in a source.

Some events are caused by the interaction with a user interface such as

- Pressing a button
- Entering a character via keyboard
- Selecting an item in a list
- Clicking the mouse

Events may also occur that are not directly caused by user interactions such as:

- When a timer expires
- A counter exceeds a value
- A software or hardware failure occur
- An operation is completed

The event is generated by the **event source object** and it occurs when the **internal state** of the **object changes** in some way. Sources may generate **more than one type of events**. A **source must register listeners**, so that the listeners get notified about a specific event. Each type of event has its own **registration methods**. General form of a listener registration is

```
public void addTypeListener(TypeListener el)
```

Type is the name of the event, and el is a reference to the event listener. For example:

- **addKeyListener** – registers a keyboard event listener
- **addMouseMotionListener** – registers a mouse motion event listener

Multicasting and Unicasting

When an event occurs, **all registered listeners** are notified and receive a copy of the event object. This is known as **multicasting** the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only **one listener** to register. When such an event occurs, that single registered listener is notified. This is known as **unicasting** the event. If too many listeners are registered then it throws **java.util.TooManyListenersException.**

A source must also provide a method that allows a listener to **unregister** an interest in a specific type of event.

```
public void removeTypeListener(TypeListener el)
```

The methods that **add or remove listeners** are provided by the **source that generates events**. For example, the **Component class** provides methods to **add and remove keyboard and mouse event listeners**.

EventListeners

A listener is an object that is notified when an event occurs. They have two major requirements

- it must be registered with one or more sources to receive notification about specific types of events
- it has to implement methods to receive and process the notifications

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event.**

Event Classes

The classes that represent events (Event classes) are at the core of Java's event handling mechanism. These classes are mostly defined by the **AWT** or the **Swing** packages. At the root of the Java event class hierarchy is **EventObject** that is a in java.util.

```
EventObject(Object src)
```

src is the object that generates the event.

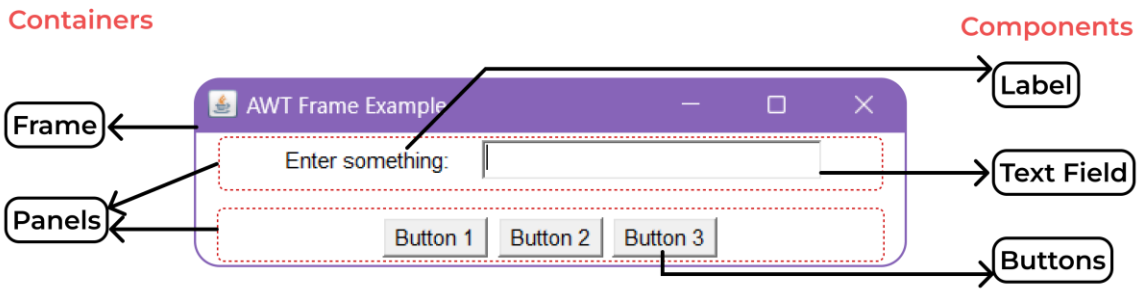
Event object contains two methods:

- **getSource()** – returns the source of the event

- **toString()** – returns the string equivalent of the event

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject.** It is the superclass (either directly or indirectly) of all **AWT-based** events used by the **delegation event model**.

`int` **`getID()`** is a method defined by AWT to determine the **type of the event**



Action Event Class

Action event is generated when

- a button is pressed
- a list item is double-clicked
- menu item is selected

The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event:

- ALT_MASK
- CTRL_MASK
- META_MASK
- SHIFT_MASK

Integer constant **ACTION_ PERFORMED**, can be used to identify action events.

ActionEvent defines three constructors:

```
ActionEvent(Object src, int type, String cmd)
```

```
ActionEvent(Object src, int type, String cmd, int modifiers)
```

```
ActionEvent(Object src, int type, String cmd, long when, int modifiers)
```

- **src** is the reference to the object that generated the event
- **type** is the specification of type of event
- **cmd** is the command passed on to the EventObject
- **modifiers** determine is any modifier keys (CTRL , ALT etc.) were pressed
- **when** parameter specifies when the even occurred

To obtain the command name for the invoking **ActionEvent** object **getActionCommand( )** method is used. For example, when a button is pressed, an action event is generated that has a **command name equal to the label** on that button.

```
String getActionCommand( )
```

The **getModifiers( )** method returns a value that indicates which modifier keys (**ALT, CTRL, META, and/or SHIFT**) were pressed when the event was generated.

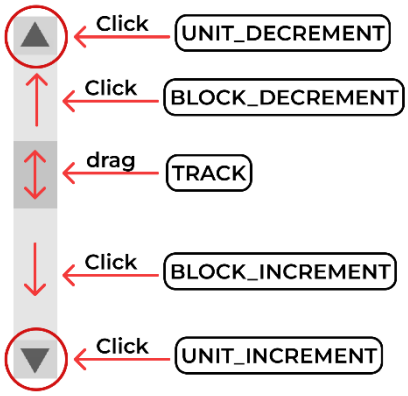
```
int getModifiers( )
```

The method **getWhen( )** returns the **time** at which the event took place. This is called the **event's timestamp.**

```
long getWhen( )
```

AdjustmentEvent Class

An **AdjustmentEvent** is generated by a scrollbar. The **AdjustmentEvent** class defines **integer constants** that can be used to identify among **five** types of adjustment events:



- BLOCK_DECREMENT
- BLOCK_INCREMENT
- TRACK
- UNIT_DECREMENT
- UNIT_INCREMENT

An integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred. It has a single constructor:

```
AdjustmentEvent(Adjustable src, int id, int type, int data)
```

- **src** is the reference to the object that generated the event
- **id** specifies the event
- **type** specifies the type of adjustment
- **data** give the assosciated data

The **getAdjustable( )** method returns the object that generated the event.

```
Adjustable getAdjustable( )
```

The type of the adjustment event may be obtained by the **getAdjustmentType( )** method. It returns one of the constants defined by **AdjustmentEvent.**

```
int getAdjustmentType( )
```

The amount of the adjustment can be obtained from the **getValue( )** method.

```
int getValue( )
```

ComponentEvent Class

A ComponentEvent is generated when the **size, position, or visibility** of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants for this.

- COMPONENT_HIDDEN
- COMPONENT_MOVED
- COMPONENT_RESIZED
- COMPONENT_SHOWN

It has a single constructor:

```
ComponentEvent(Component src, int type)
```

- **src** is the reference to the object that generated the event
- **type** is the type of event that was generated

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent**.

The **getComponent( )** method returns the component that generated the event.

```
Component getComponent( )
```

ContainerEvent Class

A **ContainerEvent** is generated when a component is added to or removed from a container. There are **two types** of container events. The **ContainerEvent** class defines int constants that can be used to identify them.

- COMPONENT_ADDED
- COMPONENT_REMOVED

ContainerEvent is a subclass of **ComponentEvent**. It has a single constructor:

```
ContainerEvent(Component src, int type, Component comp)
```

- **src** is the reference to the object that generated the event
- **type** is the type of event that was generated
- **comp** is the component that was added or removed

A reference to the container that generated this event by using the **getContainer( )** method.

```
Container getContainer( )
```

The **getChild( )** method returns a reference to the **component** that was **added** to or **removed** from the container.

```
Component getChild()
```

FocusEvent Class

A **FocusEvent** is generated when a component **gains or loses input focus**. There are two types of events defined and is identified by integer constants.

- FOCUS_GAINED
- FOCUS_LOST

FocusEvent is a **subclass** of **ComponentEvent.** It has three constructors

```
FocusEvent(Component src, int type)
```

```
FocusEvent(Component src, int type, boolean temporaryFlag)
```

```
FocusEvent(Component src, int type, boolean temporaryFlag, Component other)
```

- **src** is the reference to the object that generated the event
- **type** is the type of event that was generated
- **temporaryFlag** is set to true if the focus event is temporary, like textbox loosing focus while user goes to adjust scroll bar and then comes back to textbox
- **other** is opposite component involved in the focus change

To determine the **getOppositeComponent( )** method is used.

```
Component getOppositeComponent( )
```

The **isTemporary( )** method indicates if this focus change is temporary.

```
boolean isTemporary( )
```

InputEvent Class

The **abstract** class **InputEvent** is a **subclass** of **ComponentEvent** and is the **superclass** for component input events like **KeyEvent** and **MouseEvent**. InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed. There are eight values to represent modifiers.

- ALT_MASK
- ALT_GRAPH_MASK
- BUTTON1_MASK
- BUTTON2_MASK
- BUTTON3_MASK
- CTRL_MASK
- META_MASK
- SHIFT_MASK

The **extended modifier** values to avoid conflict between **keyboard and mouse** event modifiers are:

- ALT_DOWN_MASK
- ALT_GRAPH_DOWN_MASK
- BUTTON1_DOWN_MASK
- BUTTON2_DOWN_MASK
- BUTTON3_DOWN_MASK
- CTRL_MASK
- META_DOWN_MASK
- SHIFT_DOWN_MASK

To test if a modifier was pressed at the time an event is generated, use the **isAltDown( ), isAltGraphDown( ), isControlDown( ), isMetaDown( ),** and **isShiftDown( )** methods.

```
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

To obtain a value that contains all of the original modifier flags call **getModifiers( )** method.

```
int getModifiers()
```

We can obtain the extended modifiers **getModifiersEx( ).**

```
int getModifiersEx( )
```

ItemEvent Class

An **ItemEvent** is generated when a **check box** or a **list item** is **clicked** or when a **checkable menu item is selected or deselected**. There are **two** types of item events, which are identified by the integer constants.

- DESELECTED
- SELECTED

ItemEvent defines one integer constant, **ITEM_STATE_CHANGED**, that signifies a change of state. It has a single constructor:

```
ItemEvent(ItemSelectable src, int type, Object entry, int state)
```

- **src** is a reference to the component that generated the event , like a list or choice element
- **type** is the type of event that was generated
- **entry** is the specific item that generated that item event
- **state** gives the current state of the item

The **getItem( )** method can be used to obtain a reference to the item that generated an event.

```
Object getItem( )
```

The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event.

```
ItemSelectable getItemSelectable( )
```

The **getStateChange( )** method returns the state change (that is, **SELECTED** or **DESELECTED**) for the event.

```
int getStateChange( )
```

KeyEvent Class

A **KeyEvent** is generated when **keyboard input** occurs. There are **three** types of events defined and is given by three integer constants

- KEY_PRESSED
- KEY_RELEASED
- KEY_TYPED

The **first two** events are generated when any key is **pressed** or **released**. The **last event** occurs only when a **character** is generated. Some key presse does not result in characters

There are many other integer constants that are defined by KeyEvent.

- **VK_0** to **VK_9** – ASCII values of numbers
- **VK_A** to **VK_Z** – ASCII values of letters
- **VK_ALT**
- **VK_CANCEL**
- **VK_CONTROL**
- **VK_DOWN**
- **VK_ENTER**
- **VK_ESCAPE**
- **VK_LEFT**
- **VK_PAGE_DOWN**
- **VK_PAGE_UP**
- **VK_RIGHT**
- **VK_SHIFT**
- **VK_UP**

The **VK** constants specify **virtual key** codes and are **independent of any modifiers**, such as control, shift, or alt. **KeyEvent** is a **subclass** of **InputEvent**.It has a single constructor:

```
KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
```

- **src** gives the reference to the object that generates the event
- **type** defines the type of event

- **when** returns the event timestamp
- **modifiers** indicates if a modifier was pressed
- **code** is the virtual key code passed by the event object
- **ch** gives the character if a character key was pressed

If **no valid character** exists, then **ch** contains **CHAR_UNDEFINED**. For **KEY_TYPED** events, **code** will contain **VK_UNDEFINED**.

The KeyEvent Class defines several methods including

`char getKeyChar( ),` which returns the character that was entered

`int getKeyCode( ),` which returns the key code

MouseEvent Class

The **MouseEvents** are generated when **mouse input** occurs. There are eight types of mouse events which are defined by integer constants

- MOUSE_CLICKED
- MOUSE_DRAGGED
- MOUSE_ENTERED
- MOUSE_EXITED
- MOUSE_MOVED
- MOUSE_PRESSED
- MOUSE_RELEASED
- MOUSE_WHEEL

MouseEvent is a subclass of **InputEvent**. It has a single constructor:

```
MouseEvent(Component src, int type, long when, int modifiers, int x,
              int y, int clicks, boolean triggersPopup)
```

- **src** gives the reference to the object that generated the event
- **type** gives the type of event that occurred
- **when** returns the event timestamp
- **modifiers** determines any modifiers were added
- **x,y** defines the coordinates of the pointer
- **clicks** keeps a count of clicks made
- **triggersPopup** indicates if the event causes a pop-up menu to appear on the platform

The **getX()** and **getY()** methods can be used to get the X and Y coordinates of the mouse pointer inside the component where the event occurred

```
int getX()

int getY()
```

The **getPoint( )** method is used to obtain the coordinates of the mouse.

```
Point getPoint( )
```

The **translatePoint( )** method changes the location of the event.

```
void translatePoint(int x, int y)
```

The **getClickCount( )** method obtains the **number of mouse clicks** for this event.

```
int getClickCount( )
```

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform.

```
boolean isPopupTrigger( )
```

The **getButton( )** method returns a value that represents the button that caused the event.

```
int getButton( )
```

The return will be one of these constants:

- NOBUTTON
- BUTTON1
- BUTTON2
- BUTTON3

Java SE 6 added three methods to **MouseEvent** that obtain the **coordinates** of the mouse **relative to the screen rather than the component**.

```
Point getLocationOnScreen( )
```

The **getLocationOnScreen( )** method returns a Point object that contains both the **X and Y** coordinate.

```
int getXOnScreen( )
```

returns the **X** coordinate.

```
int getYOnScreen( )
```

returns the **Y** coordinate.

MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a **mouse wheel event**. It is a **subclass** of **MouseEvent**. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. It defines two integer constants as:

- WHEEL_BLOCK_SCROLL
- WHEEL_UNIT_SCROLL

It has a constructor

```
MouseWheelEvent(Component src, int type, long when, int modifiers,
int x, int y, int clicks, boolean triggersPopup, int scrollHow, int
                        amount, int count)
```

- **src** gives the reference to the object that generated the event
- **type** gives the type of event that occurred
- **when** returns the event timestamp
- **modifiers** determines any modifiers were added
- **x,y** defines the coordinates of the pointer
- **clicks** keeps a count of clicks made
- **triggersPopup** indicates if the event causes a pop-up menu to appear on the platform
- **scrollHow** is must be either **WHEEL_BLOCK_SCROLL, WHEEL_UNIT_SCROLL**
- **amount** gives the number of units to scroll
- **count** gives the number of rotational units that wheel moved.

MouseWheelEvent defines methods that give you access to the wheel event.

To obtain the number of rotational units, call **getWheelRotation( ),** :

```
int getWheelRotation( )
```

If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise.

To obtain the type of scroll, call **getScrollType( )**:

```
int getScrollType( )
```

It returns either **WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL.**

If the scroll type is **WHEEL_UNIT_SCROLL**, we can obtain the number of units to scroll by calling **getScrollAmount().**

```
int getScrollAmount( )
```

TextEvent Class

Instances of **TextEvent** class describe **text events**. These are generated by **text fields and text areas** when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED.** The constructor used is

```
TextEvent(Object src, int type)
```

The TextEvent object does not include the characters currently in the text component that generated the event Our program must use other methods associated with the text component to retrieve that information. Text event notification is like as a signal to a listener that it should retrieve information from a specific text component

WindowEvent Class

The WindowEvent Class describes the window based events. There are ten types of window events each defined by integer constants:

- WINDOW_ACTIVATED
- WINDOW_CLOSED
- WINDOW_CLOSING
- WINDOW_DEACTIVATED
- WINDOW_DEICONIFIED
- WINDOW_GAINED_FOCUS
- WINDOW_ICONIFIED
- WINDOW_LOST_FOCUS
- WINDOW_OPENED
- WINDOW_STATE_CHANGED

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors:

```
WindowEvent(Window src, int type)

WindowEvent(Window src, int type, Window other)

WindowEvent(Window src, int type, int fromState, int toState)

WindowEvent(Window src, int type, Window other, int fromState, int
toState)
```

- **src** gives the reference to the object that generated the event
- **type** gives the type of event that occurred
- **other** defines the opposite window involved in the event
- **fromState & toState** defines the state of window before and after event

A commonly used method in this class is **getWindow( ).**

The **getWindow( ) method** returns the Window object that generated the event.

```
Window getWindow( )
```

The **getOppositeWindow( )** method return the opposite window (when a focus or activation event has occurred),

```
Window getOppositeWindow( )
```

The **getOldState( )** method returns the previous window state,

```
int getOldState( )
```

The **getNewState( )** method returns the current window state.

```
int getNewState( )
```

Sources of Events

The components in user interface that can generate the events are the sources of the event. Some of the event sources are:

- **Button** – generates ActionEvent when it is pressed
- **Check Box** – generates ItemEvents when it is selected or deselected
- **Choice** - generates ItemEvents when choice is changed
- **List** – generates ActionEvent when an item is double-clicked and an ItemEvent when an item is selected or not
- **Menu Item** - generates ActionEvent when a menu-item is selected and an ItemEvent when a checkable menu-item is selected or not
- **Scroll bar** – **g**enerates AdjustmentEvents when scroll bar is manipulated
- **Text Component** – generates TextEvents when a character is entered
- **Window** – generate WindowEvents

Any class derived from Component, such as Applet, can generate events.

Event Listener Interface

Listeners are created by **implementing** one or more of the **interfaces** defined by the **java.awt.event** package, when an event occurs, the **event source** invokes the appropriate **method** defined by the **listener** and provides an **event object as its argument.**

- **ActionEvent** – ActionListener
- **AdjustmentEvent** – AdjustmentListener
- **ComponentEvent** – ComponentListener
- **ContainerEvent** – ContainerListener
- **FocusEvent** – FocusListener
- **InputEvent**
- **ItemEvent** – ItemListener
- **KeyEvent** – KeyListener
- **MouseEvent** – MouseListener , MouseMotionListener
- **MouseWheelEvent** – MouseWheelListener
- **TextEvent** – TextListener
- **WindowEvent** – WindowFocusListener , WindowListener

ActionListener Interface

ActionListener interface defines the **actionPerformed( )** method that is invoked when an action event occurs.

- a button is pressed
- a list item is double-clicked
- menu item is selected

```
void actionPerformed(ActionEvent ae)
```

AdjustmentListener Interface

The AdjustmentListener Interface method defines the **adjustmentValueChanged( )** that is invoked when an adjustment event occurs.

- scroll bar is manipulated

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

ComponentListener Interface

The ComponentListener Interface defines **four** methods that are invoked when a component is resized, moved, shown, or hidden.

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

ContainerListener Interface

The ContainerListener Interface contains two methods that are invoked when a component is added to or removed from a container.

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

FocusListener Interface

The FocusListener Interface contains two methods that are invoked when a component gains or loses focus.

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

ItemListener Interface

The ItemListener Interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes.

```
void itemStateChanged(ItemEvent ie)
```

KeyListener Interface

The KeyListener Interface contains three methods that are invoked when a key is pressed, released or typed.

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

If a user presses and releases a character key , three events are generated in the sequence:

- key pressed
- key typed
- key released

For any non-typed keys it will generate two events:

- key pressed
- key released

MouseListener Interface

The MouseListener Interface contains five methods that are invoked when mouse is clicked, enters a component, exits a component, is pressed, is released .

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouserReleased(MouseEvent me)
```

MouseMotionListener Interface

The MouseMotionListener Interface contains two methods that are invoked when mouse is dragged or moved .

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

MouseWheelListener Interface

The MouseWheelListener Interface contains the **mouseWheelMoved()** method that is invoked when mouse wheel is moved.

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

TextListener Interface

The TextListener Interface contains the **textChamged()** method that is invoked when change occurs in a text area or text field.

```
void textChanged(TextEvent te)
```

WindowFocusListener Interface

The WindowFocusListener Interface contains two methods that are invoked when a window gains or loses focus.

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

WindowListener Interface

The WindowListener Interface contains seven methods.

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

Using Delegation Event Model

To use Delegation Event Model, follow these steps

1- implement appropriate interface in the listener to receive the desired type of event
2- implement the code to register and unregister the listener as a recipient for the event notification

A source may generate **several** types of events. Each event must be **registered separately.** An object may register to **receive several** types of events, but it must **implement all** of the **interfaces** that are **required to receive** these events. For example, to handle mouse events we must implement both MouseListener and MouseMotionListener interfaces and sometimes the MouseWheelListener too.

Multithreaded Programming

Threads are **light weight** processes **within a process**. Using thread we can do **multiple activities** within a single process, just like the way we can scroll on a webpage while playing music on another tab and downloading a file in another. A thread in Java, is the path followed when executing a program. All **Java programs** have **at least one** thread known as the **main thread** which is created by **JVM** at the program's start invoking the **main method**. Java provides built-in support for **multithreaded programming.** A **multithreaded** program contains **two or more parts** that can run concurrently. Each of the part is termed as a Thread and shares the same memory, **no dedicated memory** is allocated for the thread other than the memory allocated for the program. Each thread defines a separate path of execution. Multithreading **maximizes** the **utilization of CPU.**

There are two distinct types of multithreading:

- **Process based**
    - A process based multitasking feature allows your computer to run **two or more program** to run simultaneously
- **Thread based**
    - The thread is the **smallest unit** of dispatchable code
    - A single program can perform **two or more tasks** simultaneously

Multithreading

Multithreading enables to write very efficient programs. It also makes the maximum use of CPU:

- As the idle time can be kept to minimum
- This is of special concern for interactive, networked environment in which Java operates, as idle time is common in these processes
- Threads are independent

In a **single-threaded** environment, one program has to **wait** for other program to finish even though the **CPU is sitting idle** most of the time. **Single threaded** systems use an approach called an **event loop with polling**. In this model, a single thread of control runs in an **infinite loop**, polling a single event queue to decide what to do next. In a **singled-threaded environment**, when a **thread blocks**, as it is **waiting for some resource**, the entire program **stops running**.

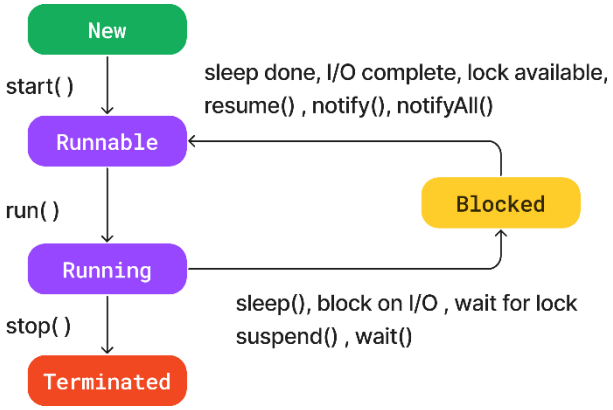
Multithreading helps to effectively make use of this **idle time** as **loop/polling mechanism is eliminated**. One thread can **pause** without **stopping other parts** of your program. When a **thread blocks** in a Java program, only the **single thread** that is **blocked pauses**. All other **threads continue to run**.

The **Java run-time system** depends on **threads** for many things. All the class libraries are designed with **multithreading**. Java uses threads to enable the entire environment to be **asynchronous. Asynchronous threading** means, a thread once **start executing** a task, can **hold it in mid**, **save the current state** and **start executing another task**. This helps reduce inefficiency by **preventing the waste of CPU cycles**.

Java thread model

Threads exist in several states

- A thread can be **running**
- A thread can be **ready-to-run** as soon as it gets CPU time.
- A thread that was running can be **suspended,** to pause the activity.
- A thread that was suspended can be **resumed.**
- A thread can be **blocked** when its waiting for a resource.
- A thread can be **terminated** by completely stopping its execution.



A thread once terminated **cannot be resumed** again.

Thread priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. They are usually given by integers that specifies the **relative priority**. A higher-priority thread **doesn't run any faster** than a lower-priority thread if it is the **only thread running.** A thread's priority is used to **decide when to switch from one running thread** to the next. Switching from one thread to another is called a **context switch.**

There are rules that determine the context switch:

- **A thread can voluntarily relinquish (give up) control** – When a thread goes to blocked state, all other threads are examined and the highest-priority thread in runnable state is given to CPU
- **A thread can be preempted (blocked) by a higher-priority thread** – As soon as a higher priority thread wants to run it suspends other threads and takes control. This is called preemptive multitasking

To set priorities of threads we have the method **setPriority()** given by:

```
final void setPriority(int level)
```

To get the priority of a thread we have the method **getPriority()** given by:

```
final int getPriority( )
```

For threads of equal priority threads the execution flow depends on the operating system

For operating systems like **Windows** – threads are given equal time slices in a **round robin** fashion

For other operating systems – threads must **voluntarily yield control** to their peer, else other threads won't run.

Synchronisation

Multithreading introduces an **asynchronous behavior**. But, when two or more threads need access to a **shared resource**, they need some way to ensure that the **resource will be used by only one thread at a time**. The process by which this is achieved is called **synchronization**. Key to synchronization is the concept of the **monitor** (also called a **semaphore**) and is used as **mutually exclusive lock** or **mutex**.

- Only one thread is allowed to own a monitor at a given time
- It is a control mechanism
- Each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called

Synchronization can be achieved by **Synchronized methods** or **Synchronized statements.**

Messaging

Java provides a **clean, low-cost** way for two or more threads to talk to each other, via **calls to predefined methods** that all objects have. Java's messaging system **allows a thread** to **enter** a **synchronized** method on an **object** and then **wait** there until **other thread explicitly notifies** it to come out.

Thread Class and Runnable Interface

Java's **multithreading** system is built upon the **Thread class**, its methods, its companion **interface-Runnable**. **Thread** class encapsulates a thread of execution. To create a thread our program should do any of the following:

- **extends Thread** class
- **implements Runnable** interface

Thread class has the following methods:

- **getName()** – obtain thread's name
- **getPriority()** – obtain thread's priority
- **isAlive()** – determine if a thread is still running
- **join()** – wait for a thread to terminate
- **run()** – entry point for the thread
- **sleep()** – suspend the thread for a period of time
- **start()** – starts a thread by calling its run method

The Main Thread

When a Java program starts up, **one thread** begins running **immediately**. This is usually called the **main thread** of our program, because it is executed when our program begins. The main thread is important for two functions:

- invoke child threads
- perform shutdown actions

The Main thread can be controlled through a **Thread** object. For that first the reference of the thread is obtained by the **currentThread()** method, a **public static method** of Thread Class.

```
public static Thread currentThread( )
```

This method returns a **reference to the thread** in which it is called. Once we have a reference to the main thread, we can control it just like any other thread.

When the thread object of main is asked to be printed the possible output is

```
[ main , 5 , main ]
```

- the name of the thread is given by first value that is by default main for main method
- the thread priority comes next which is 5 by default
- the thread group name of the thread comes next given as main

A **thread group** is a data structure that controls the **state of a collection of threads** as a whole.

sleep() method

The sleep() method is used to suspend a thread for a specific period defined by milli second. It has two forms which allows the usual one with argument accepted as milliseconds and the other which also allows in nanosecond that are designed for environments where the delay only allows too short periods of time. They are:

```
static void sleep(long milliseconds) throws InterruptedException
```

```
static void sleep(long milliseconds, int nanoseconds) throws
InterruptedException
```

setName() & getName() methods

We can set the name of a thread by using **setName( )** and can obtain the name of a thread by calling **getName( ).**

```
final void setName(String threadName)
```

```
final String getName( )
```

Creating a Thread

Implementing Runnable Interface

The **easiest way** to create a thread is to create a class that implements the **Runnable interface**. To **implement Runnable**, a class need only **implement a single method** called run( )

```
public void run()
```

Inside **run( ),** we will **define the code** that constitutes the new thread. **run( )** method establishes the entry point for another, concurrent thread of execution within our program. **This thread will end when run( ) returns.**

I- Create a class that implements **Runnable interface**
II- Create a run() method implementation with the body of thread
III- Instantiate an object of type Thread from within the class using one of the several constructors : `Thread(Runnable threadOb, String threadName)`
IV- Write the rest of the code body

After creating a thread whenever an object is created in the main method and its start() method is invoked, the thread starts running as **start() invokes run()** method.

Extending Thread Class

Another way to create a thread is to create a new class that extends Thread class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** in its constructor to begin execution of the new thread. Then creating an instance of the class is required to start the thread.

Choosing Thread or Runnable

If we will **not be overriding** any of **Thread's other methods**, it is probably **best** simply to implement **Runnable**.

The classes should be **extended using Thread** only when they are being **enhanced or modified** in some way.

One program can spawn as **many threads as it needs.** This is called **Multiple Threading**

isAlive() & join()

There exist two ways to determine if a thread has finished execution.

The **isAlive()** method returns **true** if the invoking thread is **still running.**

```
final boolean isAlive()
```

The commonly used **join()** method waits until the **invoking thread terminates.**

```
final void join( ) throws InterruptedException
```

Synchronization

Synchronization is easy in Java, because all objects have their **own implicit monitor** associated with them. To enter an object's monitor, just call a method that is **modified with the synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it on the **same instance have to wait**. To exit the monitor and relinquish control of the object to the next waiting thread, **the owner of the monitor** simply **returns** from the synchronized method.

In an unsynchronized thread system, each of the thread can call on a same method on a same object and will try to complete the task as fast as it can as if in a race, this causes a condition called **race condition.**

Using Synchronized Methods

By prefixing **synchronized keyword** in a method, it prevents other threads from entering the method while another thread is using it. Once a thread invokes method other threads wait till the other does the task and releases the lock.

Using Synchronized statements

The **call( )** method is not modified by **synchronized**. Instead, the synchronized statement is used inside Caller's **run( )** method that **synchronizes the object target**. It encloses the statement that calls the function **call() using the object target**.

Suspending, Resuming and Stopping Threads

Before Java 2 the programs used the following functions to suspend, resume and stop a thread:

```
final void suspend()

final void resume()

final void stop()
```

suspend( ), resume( ) and stop( ) methods defined by Thread must not be used for **new Java programs**. These functions are deprecated now because they caused serious failures.

A thread must be designed so that the **run( ) method periodically checks** to determine whether that thread should **suspend, resume, or stop** its own execution. This is accomplished by establishing a **flag variable** that indicates the **execution state** of the thread.

- As long as this flag is set to **running,** the run( ) method must continue to let the thread execute.
- If this variable is set to **suspend**, the thread must pause.
- If it is set to **stop,** the thread must terminate.

wait( ) and **notify( )** methods are inherited from **Object** can be used to control the execution of a thread.

- **wait( )** method is invoked to **suspend the execution** of the thread.
- **notify( )** to wake up the thread