Generics

Background and Concepts

- Generics came in Java from version 1.5, provide for defining classes, interfaces and methods that work in type-safe manner with any data type.
- Generics provide for defining algorithm once and work for various data types.
- Provide for higher level abstraction.
- Extensively used with Collection classes.
- General Syntax :

```
class class-name <type-arg-list>
  { //......}
class-name <type-arg-list> obj-name =
                      new class-name<type-arg-list>(constr-arg-list) ;
```

contd..

- Generics means parameterized types.
- Parameter is the data type upon which a class, an interface or a method operates.
- The single class/interface/method operates on various data types, hence called as generic.
- With generics all casts are automatic and implicit. Hence generics expand the ability to reuse the code.
- Generics work only with Objects. When declaring an instance of generic type, the type argument passes to the type parameter must be of class type.
- Generics types differ based on their type arguments.
- A reference of one specific version of a generic type is not type compatible with another version of same generic type.

contd..

- **Bounded types**
- Java provides bounded types wherein a user can create an upper bound that declares the super-class from which all type arguments must be derived.
  <T extends super-class>
- When specifying a bound that has a class, interface or multiple interfaces, & operator is used to connect them.
- **class** class–name <**T extends class A & class B**>
  { //....
  }

- **Wildcard Arguments**
- The wild card argument is specified by the ? and it represents an unknown type.
- The wild card matches any valid object.
- Wild card arguments can be bounded in much the same way that a type parameter can be bounded.

contd..

- **Generics and Inheritance**
  - A generic class can be a superclass or be a subclass, in other words generic classes can be part of a class hierarchy.
  - Any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses, similar to the manner of constructor arguments.

    ```
    class class-name<T> extends Gen<T>
    ```
  - A non-generic class can be the superclass of a generic subclass.
  - Casting of one instance of a generic class into another is possible only if the two are compatible and their type arguments are the same.
  - A method in a generic class can be overridden.
- Erasure
  - Process of removing the generics.

Collections

- Collections is group of objects.
- One of the Java's most powerful subsystem contained in java.util package; providing a framework to handling group of objects.
- The goals of collections are –
  - high performance;
  - framework allows different types of collections to work in similar manner and with high degree of interoperability;
  - extending/adapting a collection had to be easy.
- Algorithms operate on collections. Algorithms are defined as static methods within Collections class.
- Iterator interface – provides a general-purpose standardized way of accessing the elements one at a time.

contd..

- **Collection Interfaces** – The standard interfaces of Collection framework are –

1. **Collection** – enables user to work with groups of objects; is at top of the collections hierarchy; declares core methods that all collections will have.
2. **List** – extends Collection to handle sequences.
3. **Set** – extends Collection to handle sets, which must contain unique elements.
4. **SortedSet** – extends Set to handle sorted sets.
5. **NavigableSet** – extends SortedSet, handles retrieval of elements.

- Additionally collections use the following interfaces –

1. **Comparator** – defines how to objects are compared;
2. **Iterator** – enumerate the objects within a collection;
3. **ListIterator** – enumerate the objects within a collection;
4. **RandomAccess** – supports random access to its elements;

contd..

- Collections define two exceptions –
  - <u>UnsupportedOperationException</u> extends RuntimeException and thrown when requested operation is not supported.
  - <u>ClassCastException</u> extends RuntimeException and thrown when code has attempted to cast an object to a type of which it is not an object.
  - the ClassCastException can be prevented by using Generics, because Generics provide compile time checks and can be used to develop type-safe applications.
  - both the above exception classes are defined in java.lang

contd..

- Methods of **Collection** interface
  - boolean add(Object *obj*)
  - boolean addAll(Collection *c*)
  - void clear( )
  - boolean contains(Object *obj*)
  - boolean containsAll(Collection *c*)
  - boolean equals(Object *obj*)
  - int hashCode( )
  - boolean isEmpty( )
  - Iterator iterator( )
  - boolean remove(Object *obj*)
  - boolean removeAll(Collection *c*)
  - boolean retainAll(Collection *c*)
  - int size( )
  - Object[ ] toArray( )

contd..

- Methods of List interface
  - void add(int *index*, Object *obj*)
  - boolean addAll(int *index*, Collection *c*)
  - Object get(int *index*)
  - int indexOf(Object *obj*)
  - int lastIndexOf(Object *obj*)
  - ListIterator listIterator( )
  - ListIterator listIterator(int *index*)
  - Object remove(int *index*)
  - Object set(int *index*, Object *obj*)
  - List subList(int *start*, int *end*)

Collection Classes

- Collection classes – Standard set of classes which implement collection interfaces.
  - Some classes provide full implementations that can be used as it is.
  - Some are abstract who provide starting points for creating concrete collections.
- Standard collection classes –
  - AbstractList
  - LinkedList
  - ArrayList
  - AbstractSet
  - HashSet
  - LinkedHashedSet
  - TreeSet

contd..

- **<u>ArrayList</u>**
  - can be thought of as <u>variable-length</u> array of object references.
  - supports dynamic arrays that can grow or shrink as needed.
- class extends the AbstractList and implements the List interface and has the following constructors –
  - ArrayList() – builds an empty array list.
  - ArrayList(Collection c) – builds an array list that is initialized with elements of collection c.
  - ArrayList(int capacity) – builds an array list that has specified initial capacity.
  - Methods of ArrayList – void ensurecapacity(int cap) to explicitly increase the capacity of the ArrayList.

contd..

- **LinkedList** –
  - Linkedlist provides for linked list data structure.
  - the class extends AbstractSequentialList and implements List, Queue and Dequeue interfaces.
- The class defines two constructors
  - LinkedList() – builds an empty linked list
  - LinkedList(Collection c) – builds a linked list out of the Collection object.
- Methods of LinkedList class – all methods of Collection, List, Queue and Dequeue interfaces.
  - addFirst()
  - getFirst()
  - getLast()
  - removeFirst()
  - removeLast()
  - add(int x, Element e)

Iterator

- ➢ An interface that provides for access/display of each element in the collection.
- ➢ **Methods** –
- ✓ **boolean hasNext( )** – Returns true if there are more elements else returns false.
- ✓ **Object next( )** Returns the next element. The method throws NoSuchElementException if there is no successive element.
- ✓ **void remove( )** Removes the current element. The method throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

contd..

- ListIterator an interface that extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
- void add(Object 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 is a next element. Otherwise, returns false.
- boolean hasPrevious( ) – Returns true if there is a previous element. Otherwise, returns false.
- Object next( ) Returns the next element. The method throws NoSuchElementException is thrown if there is not a next element.
- int nextIndex( ) Returns the index of the next element. If there is not a next element, returns the size of the list.
- Object previous( ) Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.

contd..

- <u>int previousIndex( )</u> Returns the index of the previous element. If there is not a previous element, returns1.
- <u>void remove( )</u> Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
- <u>void set(Object obj)</u> Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Note : The Collection classes provides iterator() method that returns the iterator to the start of the collection, which is then used to access each element in the collection, one at a time.

contd..

- **HashSet**
  - HashSet extends AbstractSet and implements the Set interface.
  - It creates a collection that uses a hash table for storage.
  - Note - A hash table stores information by using a mechanism called hashing. In *hashing*, the informational content of a key is used to determine a unique value, called its *hash code*.
  - The hash code is then used as the index at which the data associated with the key is stored.
  - Hashing allows the execution time of basic operations, such as add( ), contains( ), remove( ), and size( ), to remain constant even for large sets.

contd..

- **Constructors of HashSet class**
  - HashSet( )
  - HashSet(Collection *c*)
  - HashSet(int *capacity*)
  - HashSet(int *capacity*, float *fillRatio*)
    - ✓ The fill ratio determines how full the hash set can be before it is resized upward, default is 0.75
    - ✓ The hash set is expanded when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio.
  - HashSet does not define any additional methods beyond those provided by its super classes and interfaces.

contd..

- **TreeSet**
  - TreeSet extends AbstractSet class and implements NavigableSet interface and uses a tree structure for storage.
  - Objects are stored in sorted, ascending order.
- **Constructors of TreeSet:**
  - TreeSet( )
  - TreeSet(Collection *c*)
  - TreeSet(Comparator *comp*)
  - TreeSet(SortedSet *ss*)

Map

- A Map is an object that stores associations between keys and values, or key/value pairs.
  - **Map interfaces** –
    - ✓ **Map** – Maps unique keys to values
    - ✓ **Map.Entry** – Describes an element (a key/value pair) in a map. Its an inner class of Map.
    - ✓ **SortedMap** – Extends Map so that keys are maintained in ascending order.
- **Object put(Object k, Object v)** Puts an entry in the invoking map, overwriting any previous value associated with the key. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
- **void putAll(Map m)** Puts all the entries from m into this map.
- **Object remove(Object k)** Removes the entry whose key equals k.
- **int size( )** Returns the number of key/value pairs in the map.

contd..

- <u>Collection values( )</u> Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.
- <u>void clear( )</u> Removes all key/value pairs from the invoking map.
- <u>boolean containsKey(Object k)</u> Returns true if the invoking map contains k as a key. Otherwise, returns false
- <u>boolean containsValue(Object v)</u> Returns true if the map contains v as a value else returns false.
- <u>Set entrySet( )</u> Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.
- <u>boolean equals(Object obj)</u> Returns true if obj is a Map and contains the same entries else it returns false.
- <u>Object get(Object k)</u> Returns the value associated with the key k.
- <u>int hashCode( )</u> Returns the hash code for the invoking map.
- <u>boolean isEmpty( )</u> Returns true if the invoking map is empty else returns false.
- <u>Set keySet( )</u> Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.

SortedMap

- An interface which extends Map. It ensures that the entries are maintained in ascending key order.
- **Methods** –

- <u>Comparator comparator( )</u> Returns the invoking sorted map's comparator. If the natural ordering is used for the invoking map, null is returned.
- <u>Object firstKey( )</u> Returns the first key in the invoking map.
- <u>SortedMap headMap(Object end)</u> Returns a sorted map for those map entries with keys that are less than end.
- <u>Object lastKey( )</u> Returns the last key in the invoking map.
- <u>SortedMap subMap(Object start, Object end)</u> Returns a map containing those entries with keys that are greater than or equal to start and less than end.
- <u>SortedMap tailMap(Object start)</u> Returns a map containing those entries with keys that are greater than or equal to start.

Map Classes

- **HashMap**
- ✓ HashMap implements Map and extends AbstractMap.
- ✓ HashMap class uses a hash table to implement the Map interface. This allows the execution time of basic operations, such as get( ) and put( ), to remain constant even for large sets.
- **Constructors of HashMap class –**
- ✓ HashMap( )
- ✓ HashMap(Map *m*)
- ✓ HashMap(int *capacity*)
- ✓ HashMap(int *capacity*, float *fillRatio*)

contd..

- **TreeMap**
- ✓ The TreeMap class implements the Map interface by using a tree. A TreeMap provides an efficient means of storing key/value pairs in sorted order, and allows rapid retrieval.
- **Constructors of TreeMap class**
- ✓ TreeMap( )
- ✓ TreeMap(Comparator *comp*)
- ✓ TreeMap(Map *m*)
- ✓ TreeMap(SortedMap *sm*)

contd..

- **LinkedHashMap –** the class extends HashMap.
- ✓ LinkedHashMap maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map.
- ✓ When iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted.
- ✓ It is also possible to create a LinkedHashMap that returns its elements in the order in which they were last accessed.
- ✓ LinkedHashMap defines the following constructors.
- ✓ LinkedHashMap( )
- ✓ LinkedHashMap(Map *m*)
- ✓ LinkedHashMap(int *capacity*)
- ✓ LinkedHashMap(int *capacity*, float *fillRatio*)
- ✓ LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)

Comparator

- ➢ The **Comparator** interface defines two methods: **compare( )** and **equals( )**.
- ✓ The **compare( )** method, compares two elements for order :

  int compare(Object *obj1*, Object *obj2*)

  obj1 and *obj2* are the objects to be compared.
  the method returns zero if the objects are equal, positive value if *obj1* is greater than *obj2*, else negative.
- ✓ The method throws a **ClassCastException** if the types of the objects are not compatible for comparison. By overriding **compare( )**, order of the objects can be altered.

- ➢ The **equals( )** method, tests whether an object equals the invoking comparator.
- ✓ boolean equals(Object *obj*)
- ✓ *obj* is the object to be tested for equality.
- ✓ The method returns true if *obj* and the invoking object are both Comparator objects and use the same ordering else it returns false.

Collection Algorithms

- ➢ The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.
- ❖ Several of these methods throw
- ✓ a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or
- ✓ an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.
- ➢ **Algorithm Methods**
- ✓ <u>static int binarySearch(List *list*, Object *value*, Comparator *c*)</u> Searches for *value* in *list* ordered according to *c*. Returns the position of *value* in *list*, or –1 if *value* is not found.
- ✓ <u>static int binarySearch(List *list*, Object *value*)</u> Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or –1 if *value* is not found.
- ✓ <u>static void copy(List *list1*, List *list2*)</u> Copies the elements of *list2* to *list1*.

contd..

- ✓ static void fill(List *list*, Object *obj*) Assigns *obj* to each element of *list*.
- ✓ static ArrayList list(Enumeration *enum*) Returns an ArrayList that contains the elements of *enum*.
- ✓ static Object max(Collection *c*,Comparator *comp*) Returns the maximum element in *c* as determined by *comp*.
- ✓ static Object max(Collection *c*) Returns the maximum element in *c* as determined by natural ordering.
- ✓ static Object min(Collection *c*, Comparator *comp*) Returns the minimum element in *c* as determined by *comp*. The collection need not be sorted.
- ✓ static Object min(Collection *c*) Returns the minimum element in *c* as determined by natural ordering.
- ✓ static List nCopies(int *num*, Object *obj*) Returns *num* copies of *obj* contained in an immutable list.*num* must be greater than or equal to zero.

contd..

- ✓ static boolean replaceAll(List *list*, Object *old*, Object *new*) Replaces all occurrences of *old* with *new* in *list*. Returns true if at least one replacement occurred. Returns false, otherwise. (Since Java v1.4)
- ✓ static void reverse(List *list*) Reverses the sequence in *list*.
- ✓ static void sort(List *list*, Comparator *comp*) Sorts the elements of *list* as determined by *comp*.
- ✓ static void sort(List *list*) Sorts the elements of *list* as determined by their natural ordering.
- ✓ static void swap(List *list*, int *idx1*, int *idx2*) Exchanges the elements in *list* at the indices specified by *idx1* and *idx2*. (Since Java v1.4)

Stream API

- A stream is a sequence of objects that supports various methods which can be pipelined to produce desired result.
- The Java Stream API provides functional programming approach to iterating and processing a group of objects.
- A Java *Stream* is a component that is capable of *internal iteration* of its elements itself, in contrast to external Iterator object in Collections.
- A stream is not a data structure and takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

contd..

- Stream is an interface in java.util.stream package.
- The JDK's standard implementation of Stream is the internal class java.util.stream.ReferencePipeline, developer cannot instantiate it directly.
- **Collectors** is a final **class** that extends Object **class**.
- It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Operations On Streams

I. **Intermediate or non-terminal Operations:**

1. **map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List number = Arrays.asList(2,3,4,5);
List square =
   number.stream().map(x>x*x ).Collect(Collectors .toList( ));
```

2. **filter:** The filter method is used to select elements as per the predicate passed as argument.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().filter(s>s.startsWith("S")).
              collect(Collectors.toList ());
```

3. **sorted:** The sorted method is used to sort the stream.

```
  List names = Arrays.asList("Reflection","Collection","Stream");
  List result =
  names.stream().sorted().collect(Collectors.toList());
```

contd..

II. Terminal Operations:

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.

```
List number = Arrays.asList(2,3,4,5,3);
Set square =number.stream().map(x>x*x).Collect(Collectors.toSet());
```

2. **forEach:** The forEach method is used to iterate through every element of the stream.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.

The reduce method takes a BinaryOperator as a parameter.

```
List number = Arrays.asList(2,3,4,5);
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)->
ans+i);
```

java.util

- java.util package contains powerful classes and interfaces which provide for extra utility.
- Classes of java.util
  - Arrays
  - StringTokenizer
  - Date
  - Random
  - Scanner
  - Currency
  - Locale
  - EventObject

StringTokenizer

- **StringTokenizer –**
  - StringTokenizer provides the *lexer (or scanner) i.e. the* first step in parsing process.
  - StringTokenizer implements the Enumeration. Hence, an input string can be enumerated as the individual tokens contained in it using **StringTokenizer.**
- **Constructors of StringTokenizer**
  - StringTokenizer(String *str)*
  - StringTokenizer(String *str, String delimiters)*
  - StringTokenizer(String *str, String delimiters, boolean delimAsToken)*

Note : The processing of text often consists of parsing a formatted input string. *Parsing is the division* of text into a set of discrete parts, or *tokens, which in a certain sequence can convey a semantic meaning.*

contd..

- **Methods of StringTokenizer –**
  - **int countTokens()** – determines the number of tokens left to be parsed and returns that number.
  - **boolean hasMoreElements()** – Returns true if one or more tokens remain in String and false otherwise
  - **boolean hasMoreTokens()** – Returns true if one or more tokens remain in String and false otherwise
  - **Object nextElement()** – Returns next token as an Object.
  - **String nextToken()** – Returns the next token as a String.
  - **String nextToken(String delimiters)** – Returns the next token as a String and sets the delimiters string to that specified by delimiters.

Date

Java 8 Date Time API is designed to overcome all the flaws in the legacy date time implementations. The design principles are:

1. **Immutability:** All the classes in the new Date Time API are immutable and good for multithreaded environments.
2. **Separation of Concerns:** The new API separates clearly between human readable date time and machine time (unix timestamp). It defines separate classes for Date, Time, DateTime, Timestamp, Timezone etc.
3. **Clarity:** The methods are clearly defined and perform the same action in all the classes, e.g. to get the current instance, use now() method. Additionally there are format() and parse() methods defined in all these classes.
4. **Utility operations:** All the new Date Time API classes come with methods to perform common tasks, such as plus, minus, format, parsing, getting separate part in date/time etc.
5. **Extendable:** The new Date Time API works on ISO-8601 calendar system but can be used it with other non ISO calendars as well.

contd..

1. **java.time** : This is the base package of new Java Date Time API. All the major base classes are part of this package, such as LocalDate, LocalTime, LocalDateTime, Instant, Period, Duration etc. All of these classes are immutable and thread safe. Most of the times, these classes will be sufficient for handling common requirements.
2. **java.time.chrono** : This package defines generic APIs for non ISO calendar systems.
3. **java.time.format**: This package contains classes used for formatting and parsing date time objects.
4. **java.time.temporal**: This package contains temporal objects and we can use it for find out specific date or time related to date/time object. E.g. to find out the first or last day of the month.
5. **java.time.zone** : This package contains classes for supporting different time zones and their rules.

contd..

- **Scanner class**
- Scanner reads formatted input and converts it into its binary form.
- Scanner can be used to read input from the console, a file, a String, or any source that implements the Readable interface or ReadableByteChannel.
- To use Scanner, the procedure is :

1. Determine if a specific type of input is available by calling one of Scanner's hasNext*X* methods, where X is the datatype
2. If input is available, read it by calling one of Scanner's next*X* methods.
3. Repeat the process until input is exhausted.
   - Scanner defines two sets of methods to read input.

contd..

- **Arrays**
- The **Arrays class provides to** help bridge the gap between collections and arrays.
- Methods of Arrays class
- **asList( )** returns a List that is backed by a specified array, both the list and the array refer to the same location.

static <T> List asList(T … *array)*

array is the array that contains the data.

Input-Output

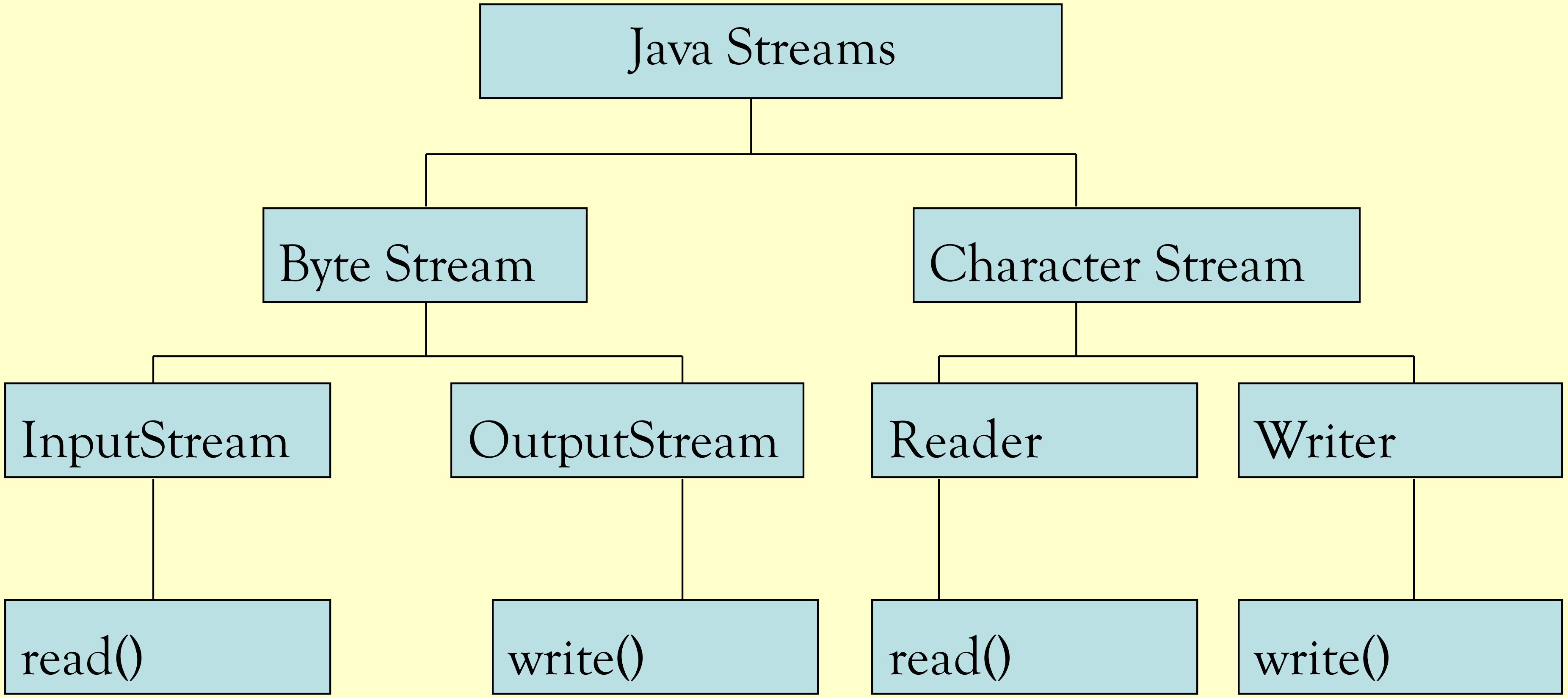
Input-Output in Java

- Java performs I/O through **streams**.
  - Stream is an abstraction that either produces information or consumes information.
  - Streams are connected to physical device by Java I/O system.
  - Input stream can abstract many different kinds of input – keyboard, disk, network socket. Output stream can refer to console, disk, network connection.
  - Java implements streams within class hierarchies defined in the *java.io* package.

contd..

- **Java Streams**
  - Byte Stream
    - Used for writing and reading binary data. At lower level all I-O is in binary form.
  - Character Stream
    - Used for writing and reading characters. Can be inter-nationalized as they use Unicode.

contd..


Java Streams
Byte Stream
Character Stream
InputStream
OutputStream
Reader
Writer
read()
write()
read()
write()

contd..

- **Stream Classes** – The entire Java I/O is based upon four abstract classes viz.. **InputStream, OutputStream,** (byte stream) **Reader, Writer** (character stream).
- Character stream classes are used when working with characters or strings. Byte stream classes are used when working with bytes or other binary objects.
- Byte Stream classes cannot work directly with Unicode characters.
- Java supports direct I/O support for characters keeping in tune with "write once, run anywhere" philosophy.

Byte Stream

- **InputStream** – defines the following methods
  - **int read() –** Returns an integer representation of next available byte of input. At end of file, it returns -1.
  - **int read(byte buffer[]) –** Returns the actual number of bytes that were successfully read. -1is returned when end of file is encountered.
  - **int read(byte buffer[], int offset, int numBytes) –** Returns number of bytes successfully read, -1 is returned when endof file is encountered.

contd..

- **OutputStream** – defines the following methods
  - **int write(int b)** – Writes a single byte to an outputstream.
  - **int write(byte buffer[])** – Writes a complete array of bytes to an output stream.
  - **int write(byte buffer[], int offset, int numBytes)** – Writes a subrange of numbytes from arraybuffer beginning at buffer[offset].

contd..

- PrintStream – the class provides all of the formatting capabilities The two constructors are –
  - PrintStream(OutputStream *outputStream*)
  - PrintStream(OutputStream *outputStream*, boolean *flushOnNewline*)
- where *flushOnNewline* controls whether Java flushes the output stream every time a newline (\n) character is output.
- if *flushOnNewline* is true, flushing automatically takes place. If it is false, flushing is not automatic. The first constructor does not automatically flush.

Console Input-Output

- The System class contains three predefined stream variables *in*, *out* and *err*. These fields are declared as *public and static*.
  - **System.out** refers to standard output stream; out is the object of type PrintStream.
  - **System.in** refers to standard input stream; in is the object of type InputStream;
  - **System.err** refers to standard error stream; its an object of type PrintStream.
- Classes and their purpose :
  - **BufferedReader –** Buffered input character stream.
  - **BufferedWriter** – Buffered output character stream.
  - **InputStreamReader** – Input stream that translates bytes to characters.

contd..

- Console input is done using **System.in**.
- **System.in** is wrapped in a BufferedReader object.
- BufferedReader(Reader *inputreader*);

  where inputreader is the stream linked to instance of BufferedReader ;
- InputStreamReader is one of the concrete subclass of Reader ;
- The following constructor is used to obtain an ISR object linked to System.in

  InputStreamReader(InputStream inputstream)

contd..

- Summarizing the above,
  - ✓ BufferedReader br = new BufferedReader (new InputStreamReader(System.in));
  - ❖ br is a character based stream that is linked to the console through System.in.
  - ❖ console output is achieved through print() and println(), the methods are defined by the class PrintStream.
  - ❖ PrintStream also implements a method write().
  - ✓ write(int byteval)
  - ❖ Usage :
  - ✓ void System.out.write();

Character Streams

- BufferedReader – the class improves performance by buffering input. It defines the following constructors:

- BufferedReader(Reader *inputStream*)
  - creates a buffered character stream using a default buffer size.

- BufferedReader(Reader *inputStream*, int *bufSize*)
  - the size of the buffer is passed in *bufSize*.

contd..

- BufferedWriter – is a writer class that adds a flush( ) method and can be used to ensure that data buffers are physically written to the actual output stream.
- BufferedWriter increases performance by reducing the number of times data is actually physically written to the output stream.
- BufferedWriter has two constructors :
  - BufferedWriter(Writer *outputStream*) it creates a buffered stream using a buffer with a default size.
  - BufferedWriter(Writer *outputStream*, int *bufSize*) in this the size of the buffer is passed in *bufSize*.

contd..

- **PrintWriter -** the class provides for formatted output methods print( ) and println( ). PrintWriter has four constructors.
  - PrintWriter(OutputStream *outputStream*)
  - PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)
  - PrintWriter(Writer *outputStream*)
  - PrintWriter(Writer *outputStream*, boolean *flushOnNewline*)
    - where *flushOnNewline* controls whether Java flushes the output stream every time println( ) is called. If *flushOnNewline* is true, flushing automatically takes place.
    - If false, flushing is not automatic. The first and third constructors do not automatically flush.

File

- **File** – File class directly deals with files and file system, but not through streams.
  - File object is used to obtain or manipulate the information associated with the disk file, such as **permissions, time, date, directory path and navigate sub directory hierarchies**.
  - **Constructors of File class** –
    - File(String directorypath)
    - File(String directorypath, String filename)
    - File(String dirobj, String filename)
    - File(URI uriobj)
  - **Directorypath** – is the path name of the file, filename is the name of the file, dirobj is File object that describes a directory, uriObj is URI object that describes a file.

contd..

- **Methods** –
- **boolean isFile**() – returns true if called on a file and false if called on a directory, device drivers, named pipes.
- **boolean isAbsolute**() – returns true if the file has an absolute path and false if the path is relative.
- **boolean renameTo(File newName)** – renames the file specified by invoking object with newName .
- **boolean delete()** – deletes the disk file represented by the path of the invoking File object. Can also delete directory if its empty.

contd..

- **<u>Directory</u>** – A directory is a File that contains a list of other files and directories.
  - **<u>isDirectory</u>**() method returns true on a File object when the object in question is Directory.
  - **<u>list()</u>** is invoked to extract the list of files and directories.

File Streams

- **FileOutputStream** – creates an OutputStream used to write bytes to a file. The constructors of FOS are -
    - FileOutputStream(String *filePath*)
    - FileOutputStream(File *fileObj*)
    - FileOutputStream(String *filePath*, boolean *append*)
    - FileOutputStream(File *fileObj*, boolean *append*)
- **FileInputStream** - creates an **InputStream** that is used to read bytes from a file. The constructors of FIS are
    - FileInputStream(String *filepath*)
    - FileInputStream(File *fileObj*)
        - √ The *filepath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.
    - Objects of FIS throw **FileNotFoundException**.

contd..

- **<u>FileReader</u>** – the class creates a Reader that is used to read the contents of a file. It defines following constructors –
  - FileReader(String filePath)
  - FileReader(File fileObj)
  - <u>filePath</u> is the full path name of a file, and <u>fileObj</u> is a File object that describes the file.

- **<u>FileWriter</u>** – the class that creates is used to write to a file. It defines following constructors –
  - FileWriter(String *filePath*)
  - FileWriter(String *filePath*, boolean *append)*
  - FileWriter(File *fileObj*)
  - FileWriter(File *fileObj*, boolean *append*)

contd..

- **RandomAccessFile** the class encapsulates a random-access file.
  - It implements the interfaces DataInput and DataOutput which define the basic I/O methods.
  - It also supports positioning requests — the *file pointer* within the file can be positioned.
  - The class has these two constructors:
    - RandomAccessFile(File *fileObj*, String *access*) throws FileNotFoundException
    - RandomAccessFile(String *filename*, String *access*) throws FileNotFoundException

Note - It is not derived from InputStream or OutputStream.

contd..

- **Methods of RandomAccess** –
- **seek**( ) – used to set the current position of the file pointer within the file:
  - void seek(long *newPos*) throws IOException
  - *newPos* specifies the new position, in bytes, of the file pointer from the beginning of the file.
  - After a call to **seek( )**, the next read or write operation will occur at the new file position.
- **setLength**( ) – used to lengthen or shorten a file.
  - void setLength(long *len*) throws IOException
  - sets the length of the invoking file to that specified by *len*.
  - Note – If the file is lengthened, the added portion is undefined.

contd..

- **Serialization** – it is the process of writing the state of an object to a byte stream.
- Applied in situations to save the state of the object to a persistent storage area, such as a file. At a later time, the state of these objects can be restored by using the process of **de-serialization**.
- Serialization is used in implementing Remote Method Invocation (RMI).
  - √ RMI allows a Java object on one machine to invoke a method of a Java object on a different machine.
  - √ An object may be supplied as an argument to that remote method.
  - √ The sending machine serializes the object and transmits it. The receiving machine de-serializes it.

contd..

- **Serializable** – An interface which provides for implementing serialization.
  - The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized.
  - If a class is serializable, all of its subclasses are also serializable.
  - Variables that are declared as **transient** are not saved by the serialization facilities.
  - Variables that are declared **static** variables are not saved.

contd..

- **<u>ObjectOutput</u>** – The **ObjectOutput** interface extends the **DataOutput** interface and supports object serialization. It defines the **writeObject( ) and other** methods. **writeObject( )** is called to serialize an object. All of these methods will throw an **IOException** on error conditions.

- **<u>ObjectOutputStream</u>** – The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface.
  The constructor ObjectOutputStream(OutputStream *outStream*) throws IOException The argument *outStream* is the output stream to which serialized objects will be written.
  The methods of this class throw an **IOException** on error conditions.

contd..

- **ObjectInput** – The **ObjectInput** interface extends the **DataInput** interface and defines the **readObject( )** and various methods. It supports object serialization. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException.**
- **ObjectInputStream** – The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface. **ObjectInputStream** is responsible for reading objects from a stream. The constructor ObjectInputStream(InputStream *inStream*) throws IOException

The argument *inStream* is the input stream from which serialized objects should be read.Several methods of this class will throw an **IOException** on error conditions.

Thank You