compareTo(E o) - compare two constants based on their ordinal value.

Equals(object other) - returns true if specified object is equal to enum constant.

Chapter's Bullet Points

I) Introduction to the Objects

1) Abstraction
2) Encapsulation
3) Composition and Aggregation
4) UML notations
5) Upcasting and downcasting(Ex)
6) Early Binding and Late Binding(Ex)

2) Everything is a Object

1) Object reference is separate than object itself
2) Where storage lives

3) Static Keyword

3.1. Only methods, variables and top level nested classes are marked static

3.2. Static Variables(Ex)

i. Belongs to class not objects
ii. Can be called using classname
iii. Initialized only once

3.3. Static Methods(Ex)

i. Belongs to class not objects
ii. Can access only static data
iii. Can call only other static methods
iv. Can be called using classname
v. Object creation inside static method or getting object reference through method arguments enables it calling non-static methods.

3.4. Static Block(Ex)

```
class Test {
static{
//code
}
}
```

1) Primitives are automatically initialized and placed on the stack hence they are more faster.

2) Autoboxing with Wrapper Classes.

3) High precision numbers

4) Scoping

5) return keyword

i. return type of a method is must
ii. a method with void type can have return;
iii. Any code after return won't compile.

3) Operators

1) Aliasing

2) Aliasing in methods

4) Controlling Execution

1) do while(Ex)

2) for loop(initialization; conditn; step)(Ex)

3) foreach(Ex)

4) unconditional branching (return, bread and continue)(Ex with label)

5) Initialization and cleanup

1) Constructors

i. Default and parameterized(Ex)
ii. Constructors dont have return type
iii. Private constructors : Such classes only extended if non-private constructor is present(Ex).

2) 'this' keyword

i. Provides reference to the current object.
ii. Used for calling one constructor from another when many constructors are presents and provide code reuse(Ex)
iii. Cannot be used inside static method(Ex)


```
GOLDENDEL costs 9 cents
REDDEL costs 12 centsdd
```

3.1 From above example we can understand many things like

- Enum types cannot be instantiate with 'new' as constructor to enum is private.

- Instance of enum is created when enum constants are first called or referenced in code.

- In above example when variable 'ap' is declared constructor for Apple is called once for each constant.

- All Enum constants are implicitly static and final and cannot be changed once created.

- Each enum constant is object of its enumeration type.

- Each enum constant holds copy of instance variable each defined by enum type(ex price in above example).

- enum can have more than one constructor and when no arguments are present with constants(JONATHAN(10)) then default constructor will be called.

- Enums have only two restrictions compare to classes first they cannot extends any class or they couldn't be a super class.

4. values() and valuesOf() methods

- All enum contains these two methods.

- values() method returns an array that contains list of enumeration constants in the order they declared.

- valuesOf(String str) returns corresponding value of 'str'.

5. java.lang.Enum Class

- All enums automatically inherit java.lang.Enum class.

- This class defines methods like
ordinal() - returns positions of constants starting from 0.

```
Days day;
day=day.MONDAY;
s.o.p(day);
day=day.FRIDAY;
s.o.p(day);
//other code

output:
MONDAY
FRIDAY
```

2. Names of enum type fields are in upper cases.

3. Enum type as Class type

- Enum types are class types in java. They can have constructors, instance variables and methods and can even implement interfaces.

Ex:

```
enum Apple
{
   JONATHAN(10), GOLDENDEL(9), REDDEL(12);

   private int price;
   //constructor
   Apple(int p) {       price = p; }

   int getPrice() { return price; }
}

public class EnumAsClassType
{
public static void main(String args[])
{
   {
      Apple ap;
      System.out.print("Goldendel costs
   "+Apple.GOLDENDEL.getPrice()+" cents");
      System.out.println();

      //Display all apple prices
      for(Apple a : Apple.values())
      S.O.P(a+" costs "+a.getPrice()+" cents");
}
} }
Output:
Goldendel costs 9 cents

JONATHAN costs 10 cents
```


3) 'super' keyword

i. Used for getting fields using super.field(Ex).
ii. Used for calling superconstructors using super(int a)(Ex).
iii. Used for calling super class methods using super.fn()(Ex).

4) Finalization and Garbage Collection

i. finalized() method(Ex).
ii. System.gc().

5) How Garbage Collector works

I. Reference counter
ii. Stop-copy
iii. Mark-Sweep(flag)

6) Order Of Initialization

i. Static primitive variables, Static reference variables and Static Block whichever comes first.
ii. Non-static primitive variables, Non-static reference variables whichever comes first.
Iii. Constructors.

A) With Separate classes (i) to (iii) take place class wise(Ex).
B) with Inheritance (i) first executed from base to derive class and then (i) and (ii) together executed from class to class first from base then to derived(Ex).

7) Ellipses or VarArgs

i. VarArgs as argument to method(Ex).
ii. Method with two arguments with VarArgs being one. VarArgs should always be the last argument.
iii. A method cannot have two VarArgs as arguments.

6) Overloaded and Overridden methods

1) Rules for Overloading(Ex).
2) Rules for Overridding(Ex).
3) Access Modifiers(Ex).

7) Reusing classes

1) Same topics from early chapter repeated

8) Polymorphism

1) Polymorphism doesn't work for fields, private methods and static methods(Ex).

2) After assigning derive class reference to base class
i. Always overidden methods are called.
ii. Calling unique methods of base class executes them.
iii. To call unique methods of derive class, class-cast is needed.

3) Calling method from constructors. Always reference is used while calling method. i.e. if derive class construction is taking place and base class constructor calls a method of base class and if same method is overridden in derive class then overridden method is called. This proves even method calling from constructors gives preference to overridden methods(Ex).

4) Covariants(Ex).

9) Interfaces

1) Abstract Classes
i. Object of abstract classes can't be created.
ii. Can have both abstract and non-abstract methods
iii. A class cant be both abstract and final as both are opposite of each other.
iv. Somewhere in inheritance tree all abstract methods of an abstract class should be implemented.
v. Only abstract methods of an interface should be marked public during implementation, its not mandatory of abstract classes. Rules for overriding are applied to abstract methods during implementation.
vi. Abstract class can implement an Interface in such case all abstract methods of both should be implmented.
vii. If one or more methods of a class are abstract(dont have body) then that class must be marked abstract else compiler gives error.
viii. If someone don't want object creation of a class then it can be marked abstract.

2) Interfaces
i. They are fully abstract classes with none of the methods have body.
ii. Object of interface cannot be created.
iii. A class can implement any number of interfaces.

iv. Interfaces have
a. Interface : public & abstract

- Ex

```
Car [] cars;
Honda [] cuteCars = new Honda[];

Beer [] beers = new Beer[9];

cars = cuteCars; //Legal Honda is a type of car
cars = beers; //Illegal beer is not type of car
```

C. For Interfaces

- Only those classes type array reference could be assigned to interface type array reference which implements the interface.
- Ex

```
Foldable [] foldingThings;

Box [] boxThings = new Box[3];

foldingThings = boxThings; //Legal , Box implements Foldable, so
Box is a foldable
```

D. For Multi-dimensional array

- References of array types of different array dimensions could not be assigned to each other.

- Ex

```
int [] blots
int [][] squeeges = new int [3][];

blots = squeeges; //Illegal
```

17) Enum Types

1. Enum type is special data type that enables for a variable to be set of predefined constants.
Ex. Days-of-the-week.

```
enum Days
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY
}
```

The above is accessed through

10. Anonymous array initialization

```
int [] testScores = new int[] {4,7,2};
```

it is very useful when array is passed as argument to a method-name

```
amp.ArrMeth(new int[]{1,2,3});
```

11. A super class array type can hold objects of its sub-classes.

Ex :

```
class Car {}
class Subaru extends Car{}
class Ferrari extends Car{}

Car[] myCars = {new Subaru(), new Car(), new Ferrari()};
```

12. Similarly there could be array of interfaces too which holds references to classes which implements the interface.

13. int[] array type can hold short, byte, char etc as elements but not vice versa.

14. Array Reference Assignments

A. For Primitives

- A byte or char can be put as elements inside an int array but an int array reference cannot be assigned to any other type like byte or char array..

- Ex.

```
int [] intArr1 = {1,2,3};
int [] intArr2 = {4,5,6};

byte [] byteArr = {11,12,13};

intArr1 = intArr2;//legal
intArr1 = byteArr;//Illegal
```

B. For Objects

- If two object passes IS-A test then sub class array reference could be assigned to super class array reference.

b. method : public & abstract
c. variables : public static final

v. An abstract method of an interface must be marked public if implemented or subsituted.
vi. An Interface can extends many other interfaces.
Vii. Its legal to mark interfaces and their methods as abstract but its not needed as they are implicitly abstract.

10) Generics

1. Generics are basically invented for collections. As collections accepts any objects and while retrieving one needs to casts them all. Hence Generics provides homogenous collections.

2. Homogenous collecctions like LinkedList<String> or LinkedList<Integer> accepts only String and Integer elements resply.

3. Type mismatch found during compile time as error rather than run time as exceptions.

4. Generics Notations

a. T – Type
b. E – Elements
c. K – Keys
d. V – Values
e. N – Numbers
S,U,V etc – 2nd, 3rd, 4th types

Allowed Generic Types(Any non-primitive type)
- Class type
- Interface type
- Array type
- or any other type variable

Not allowed Generic type
- Enums
- Anonymous inner classes
- Exception classes

5. Generics is useful

i. Type mismatch can be found at compile time.
ii. No need of casts.
iii. Enables to implement generics algorithms.

6. Example

```
public class Box<T>
{
private T t;

public void set(T t) { this.t = t;}
public T get() {return t;}
}
```

7. Invoking and Instantiating a Generic Type

```
Box<Integer> integerBox;
```

It is similar to method invocation, instead of passing an argument here we pass type arguments-integer in this case.

8. Here T is 'type parameter' and Integer is 'type argument'.

9. From Java 7, diamond has been introduced
Box<Integer> integerBox = new Box<Integer>;
is same as
Box<Integer> integerBox = new Box<>;

10. Multiple type parameters(Ex)
MultipleTypeClass<K, V>

11. For a parameterized type
public class Box<T>
{
}
one can create raw type as
Box rawBox = new Box();

12. Raw types permitted to facilitate interfacing with (non-generic) legacy code.

13. For backward compatibility, one can assign `parameterized type to a raw type(Ex)`

```
Box<String> strBox = new Box<>;
Box rawBox = strBox;    //OK
```

But

```
Box rawBox = new Box();
```



```
GameCharacter elf = (GameCharacter)one;
GameCharacter troll = GameCharacter)two;
GameCharacter magician= (GameCharacter)three;
```

v. Close the object input stream

```
os.close();
```

16) Array Basics

1. `int [] testScores = new int[5];`
LHS is creating an array
RHS is constructing an array
int [] - Array type(int in this case)
testScores – Identifier.

2. Array must be given size during construction so that JVM can provide memory space. Arrays are of fixed size, they cannot change dyncamically like collections.

3. Initializating an array is nothing but putting things into it.

4. Array elements could be primitives or object references.

5. Array elements are accessed through index numbers from 0 to ArrayLength – 1.

6. **NullPointerExeception –** Is thrown when object reference has been accessed which is not pointing to any object.
ArrayIndexOutOfBoundsException – Is thrown when out of the array range has been accessed.

7. When Array construction take place its elements are set to default values. Objects to null and primitives to their default values.

8. `int [] testScores = {1,2,3};`
Here declaration, construction and intialization taking place in one go.

9. `Dog puppy = new Dog("frido");`
`Dog [] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};`

Four objects are created here
1 Dog object referenced by puppy.
1 Dog [] arrayy object referenced by myDogs.
2 Dog objects referenced by myDog[1], myDog[2].

10. Static variables are not encouraged for serialization as they belongs to class.

11. Serialization and Deserialization

A. Seriialization

i. Make a FileOutputStream

```
FileOutputStream fileStream = new
FileOutputStream("MyGame.ser");
```

ii. Make an object output stream

```
ObjectOutputStream os = new ObjectOutputStream(fileStream);
```

iii. Write Objects

```
os.writeObject(characterOne);
os.writeObject(characterTwo);
os.writeObject(characterThree);
```

iv. Close output stream

```
os.close();
```

- Object stream turns object into data(100101 etc) and file stream write data into file.

B. Deserialization

Deserialization takes place in reverse order.

I. Make a File input stream

```
FileInputStream fileStream = new FileInputStream("MyGame.ser");
```

ii. Make an object input stream

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

iii. Read the objects

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

Every time readObject() is called it will get next object in the stream. It will be read back in same order they are written. If one tries to read more objects than what have been written then it will throw an exception.

iv. Cast the Objects


```
Box<Integer> intBox = rawBox; //Warning : unchecked conversions
```

14. Compiler also gives warning when one tries to call generic methods using raw types(Ex).

15. Generic methods are one which introduces their own generic type regardless of class type(Ex).

```
<T> void method(T t)
```

also for static methods

```
static <E> void printArray(E [] inputArray)
```

17. Bounded type Parameters

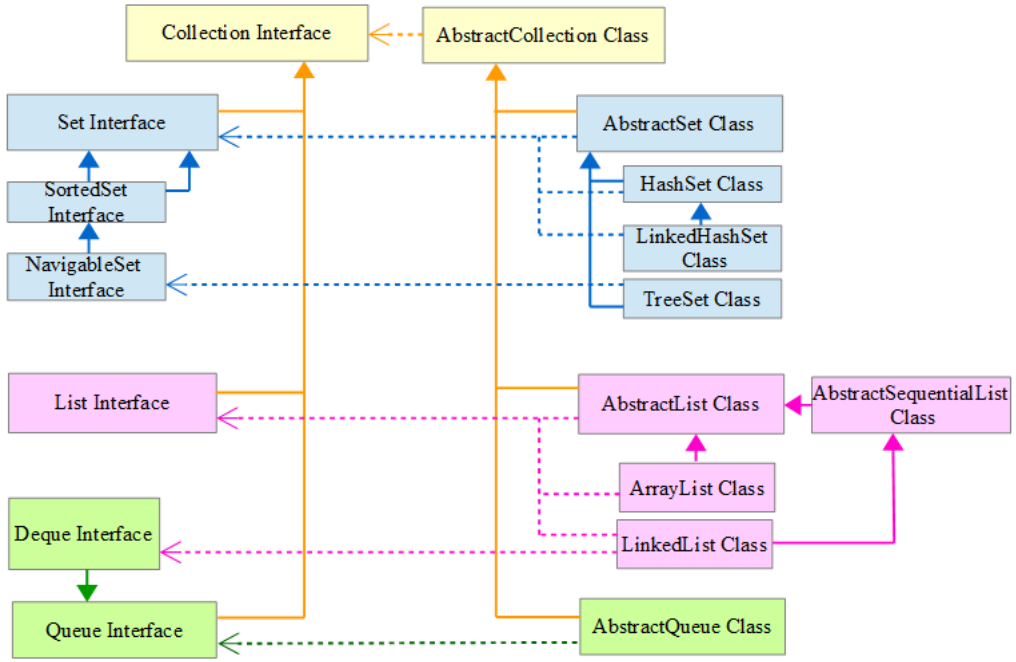
```
BoundedTypeClass<T extends  String>
RedAppleContainer<A extends AppleClass>
```

18. Like raw types generic method could also be called as shown below

```
gm.method(10);
```

19. Multiple bounds

11) Collections



1. Any class defines the collection should implement Collection interface.

2. AbstractCollection is an abstract class which implements Collection interface and defines common methods used by all collections.

3. Optional operations are those whose behavior is depends on corresponding collection and if invoked without implmementation throws UnsupportedOperationException.

4. Set interface dont define any new method still it is created just to indicate a separate branch of colllection.

5. Collections in general

Set – Don't allow duplicates
List – Allow duplicates and maintains insertion order.
Queue – Allow duplicates and don't maintain order also. Works mostly in FIFO.
Map – Maps keys to elements. Dont allow duplicate keys.

6. Exceptions

UnsupportedOperationException - When collection can't be modified.
ClassCastException – When incompatible objects are added.
NullPointerException – When attempt is made to store a null object.
IllegalArgumentException – If an invalid argument is used.
IllegalStateException – If attempt is made to add an element to a fixed length collection which is full, also if iterators remove() method called twice continously.
ArrayStoreException – If attempt is made to return an array contains collection elements and type mismatch occurs.

7. Iterator methods
hasNext()
next()
remove()

8. **List** : Maintains insertion order

8.1. ArrayList

- `public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable`

- ArrayList uses dynamic arrays which increases dynamically when full.

- ensureCapacity(int cap) fixes ArrayList's capacity and it only increases when

with the object.

2. If one want to save video game and play it later from the same then he should save its state i.e. object's state.

3. Objects at heap have a state – the values of reference and primitive variables. During serialization these values will be saved and identical object will be brought back during deserialization.

4. If the object have references to other objects then whole object graph will be saved.

5. Serializable Interface

- Every class which implements Serializable interface could be serialized.

- Serializable interface don't have any methods its just a marker or tag interface.

- If a class implements Serializable interface then its sub-classes becomes automatically serializable.

- If a class don't implement Serializable interface then one can create its subclass with implementation of Serializable interface and then can use subclass instance.

6. Trainsient keyword

- If a class1 has object reference for class2 and if class2 don't implement Serializable interface then compiler will throw NotSerializableException.

- To avoid such exception one should mark such references with 'transient' keyword. During serialization such references will be skipped and set to default values during deserialization. i.e. null for objects references and default values for primitives.

7. If two references are holding the same object then it will be recognized in object graph and object's state will be saved only once.

8. During deserialization objects constructor wont run else it will be set to default value.

9. If Object have non-serializable class some where up in its inheritance tree then constructor for that and all classes above that will run and reinitialize their state.

- System class has
'in' as input stream (keyboard by default)
'out' as PrintStream (console by default)
'err' as PrintStream

from above we can understand how System.out.println() statement works.

- PrintStream never throws IOException and it flush automatically.

5. InputStreamReader and OutputStreamWriter

- InputStreamReader is bridge from byte stream to character stream. It reads bytes and decodes them into character using specified charset.

- An OutputStreamWriter is bridge from character stream to byte stream. Character written to it are encoded into byte using specified charset.

- Both InputStreamReader and OutputStreamWriter works best with their corresponding buffered I/O streams.(Ex)

6. LineNumberReader class

- A buffered character input stream that keeps track of line numbers.

- A line considered to be terminated by any of the line feed('\n') or a carriage return('\r') followed immediately by a linefeed etc.

- It read lines from a file through readLine() method and keeps updating value of 'lineNumber'with every line. Also it returns 'lineNumber' through lnr.getLineNumber() method. One can set lineNumber with setLineNumber(int).(Ex).

7. PushbackReader Class

- When we read through the stream control always goes to next byte/char. With PushBackReader its possible to bring control back to previous byte/char.

- This stream has methods to read() to read through the stream and unRead(int c) which unread the previous character and bring back control to it. (Ex).

15) Serialization

1. Objects have both behavior and state. Behavior lives within class but state lives

exceeded. TrimToSize() used to remove extra space when adding of elements is over.

8.2. Difference between ArrayList and Vector

1. All methods of vector are synchronized hence its slow.
2. Both uses dynamic arrays but when size exceeded vector doubles the capacity and ArrayList increases by half.

8.3. LinkedList

- `public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable`

- Implements Deque too.

8.4. Difference between ArrayList and LinkedList

1. ArrayList internally uses dynamic array and Linked list internally uses doubly linked list.
2. ArrayList acts as only list but LinkedList acts as both list and queue.
3. ArrayList is faster and better for storing and accessing. LinkedList is slower but better for manipulation as it provides many functions.

9. 'cloneable' interface don't define any method but only a indication for Object.clone() method.

10. Set

10.1. SortedSet Interface

- `public abstract interface SortedSet<E> extends Set<E>`

- It extends Set and declares the behavior of an ascending set.

10.2. NavigableSet Interface

- `public abstract interface NavigableSet<E> extends SortedSet<E>`

- It extends SortedSet and delcares the behavior of a collection that retrieves element based on closest match.

10.1. HashSet

- `public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable`

- Backed by HashMap(which is internally backed by Hash table).

- Each of HashSet method calls corresponding HashMap methods

```
public int size()
{
    return map.size
}
```

- It maintains no orders.

10.2. LinkedHashSet

- `public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable`

- Internally uses LinkedHashMap.

- It is similar to HashSet, only difference is its maintains insertion order.

10.3. TreeSet

- `public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable`

- It maintains natural order of its elements.

11. Map

- `public interface Map<K, V>`

- Map don't implements Collection interface.

- Map stores Key/Value pairs where both key and value are objects.

- It don't implement iterable interface but collection-view can be obtained through entryset(), keySet() and values() methods and can be iterated.

- Map is used as backbone of Set hence we have same SortedMap, NavigableMap and other interfaces with same methods. The implementation of Map is same as Set.

4.5. Filter I/O Streams

Input streams
i. DataInputStream
ii. BufferedInputStream
iii. LineNumberInputStream
iv. PushBackInputStream

Output streams
I. DataOutputStream
ii. BufferedOutputStream
iii. PrintStream

4.5.1. Data I/O streams

- Data streams lets an application read and write primitive data(boolean, char, byte etc) as well as String values. These streams has all the methods to support such operations like readDouble, writeDouble etc.(Ex)

- An application can use DataOutputStream to write data that could be read by DataInputStream.

4.5.2. Buffered I/O streams

- The unbuffered stream calls either read or write will be handled directly by underlying OS. This makes program less effective as each request triggers disk access, network activity n other operations which are relatively expensive. To reduce overhead, we have Buffered I/O streams.

- BufferedInputStream read data from memory area known as buffer and native API is called whenever buffer is empty. Similarly BufferedOutputStream write data to a buffer and native output API is called when buffer is full.

4.5.3. Line number I/O stream

It is deprecated. Only works with writer.

4.5.4. PrintStream

- print() and println() are two important methods of PrintStream. Difference between them is later adds newline when done.

- ByteArrayInputStream and ByteArrayOutputStream
- StringBufferInputStream
- FileInputStream and FileOutputStream
- PipedinputStream and PipedOutputStream
- SequenceInputStream
- FilterInputStream and FilterOutputStream

4. Byte I/O Streams

4.1. Byte Array I/O Streams

- ByteArrayInputStream takes only 'byte array' as argument. Hence it should be first converted to a byte array.

- Ex. For reading from keyboard it should first write to ByteArrayOutputStream using bout.write(System.in.read()) and then converted to a byte array through byte [] b = bout.toByteArray(). Later byte array 'b' is given as argument to new byteArrayInputStream(b) and read through bin.read()(Ex).

- String can directly converted to byte array through byte [ ] b = str.getBytes(). Which can be send directly to input stream(Ex).

4.2. StringBufferInputStream has been deprecated.

4.3. File I/O Streams

- FileInputStream reads from file using fin.read() and write same into another file using FileOutputStream fout.write(). (Ex).

4.4. Sequence I/O streams

- Only input stream is available here with read() method and it has no output stream hence no writer method either.

- One of its constructor is SequenceInputStream(fin1, fin2) which reads fin1 and fin2 in order. Ex. sin(fin1, fin2).

- sin.read() and fout.writer(). (Ex)
  fin1 = hi there!!
  fin2 = how r u?
  Fout = hi there!!how r u?

12) Exceptions Handling

1. When exception condition arises in a method, an object representing that exception created and thrown in the method. Method may chose to handle the exception or pass it on, but somewhere it should be caught and process.

2. Throwable

'java.lang' package defines many exceptions. All exception types are subclass of Throwable. Below Throwable there are two main branches. One is Exception and other is Error.

2.1. Exception

- `public class Exception extends Throwable`

- Exception class is made for exceptional conditions which user program should catch.

- This class has a important subclass called RumTimeeException. Exceptions of these types are automatically defined ex. ArithmeticException.

2.2. Error

- `public class Error extends Throwable`

- They are used by Java Runtime System and not to be caught inside one's programme. Ex: Stack Overlow.

3. Java Exception handling is managed by 5 keywords
i. try-catch : Pgm statements which needed to be monitor are put inside try. If exception occurs will be handled by catch.
ii. throw : To manually throw an exception.
iii. throws : Any method that can be thrown by method should be specified using throws.
iv. finally : Comes after try-catch and executes regardless of exception occurred or not.

4. try-catch

- Example

```
Class A
{
    try
    {
    void method()
    {
    //code
    }
    catch(Exception e)
    {
    }
}
```

- Multiple catch : If method may throw more than one type of exception then multiple catch can be used. When exception occurs all catch blocks are checked one by one and first match gets executed, rest will be bypassed.

- Nested try statements.

5. throw

- It throws exception which are subclass of Throwable or its subclasses.

- No statement will be executed after throw statement.

- Ex : `throw new NullPointerException("demo");`

- Many of java's built in exception has two constructors one is with string and one is without string.

6. throws

- It lists all the exception separated by commas, that a method can throw except Error or RuntimeException or its subclasses.

- Caller of such method can guard himself against those exceptions.

- type method-name(parameter list) throws exception-list

7. finally

- When an exception occurs program terminates suddenly hence some unfinished work like closing a file can be done inside finally.

- This function of suspension and resume can be achieved through wait() and notify().

13. Thread state

- A thread could exist in different states. State is a enumeration defined by thread and returned through getState() method.

- Different states of thread are

NEW – A thread has not begun execution.
RUNNABLE – Either executing or executes when gain access to CPU.
TERMINATED – Completed execution.
BLOCKED – Has suspended execution because its waiting to acquire a lock.
TIMED-WAITING – Suspends execution for a specified amount of time. Ex: while calling timeout version of sleep(), wait() and join().
WAITING – Suspended execution because its waiting for some action. Ex: Calling non-timeout version of wait() or join().

14 Input Output

1. Byte and Character I/O classes

- Java defines two types of I/O streams. Byte streams and Character Streams.

- Byte Streams are old I/O classes which only supports 8-bit byte streams. Character streams supports 16-bit unicode characters hence internationalization.

- InputStream and OutputStream are top level classes for byte streams. InputReader and InputWriter are top level classes for character stream classes.

- Byte stream and Character streams both have corresponding similar classes.

2. The source of Inputs are
i. An array of bytes
ii. A String object
iii. A file
iv. A pipe
v. Sequence of streams
vi. Other sources such as internet

3. For different input sources there are corresponding I/O classes

- To cope with such scenerio we have wait() and notify() methods which completely suspends one thread when other is working on the same data.

- When wait() method is called on some thread then it will suspend execution until another thread notify it through notify() or notifyAll() method.

- A thread can only resume execution when its get objects lock. Only one thread can hold objects lock at one time.

- wait() method should always be called inside a loop with a condition to be waited up(Ex).

- notify() or notifyAll() methods will be called by thread which owns object lock. When it releases the lock other threads gets hold of it.

12. Deadlock

- Dead lock occurs when two threads have a circular dependency on a pair of synchronized objects(Ex).

- We know with synchronization all synchronized methods gets locked by a thread and if two threads locking all synchronized methods of two objects and cannot complete the current task as its depends on method of the others object(which is locked) then it can go nowhere hence deadlocked.

13. suspend(), resume() and stop() methods

- All of these methods have been deprecated since Java 2.0 but useful to work with legacy classes.

- A thread can be suspended through t.suspend() method and can be resumed again through t.resume() method.

- Both of them have been deprecated because when a thread is suspended then it won't release lock and other threads waiting for execution just keep on waiting. Also method resume() can work only with suspend().

- stop() methods stops a thread from execution. It releases lock but when a thread stops in between it does only half work which could bring wrong results like writing a file. Hence stop() is also deprecated.

- finally block comes after try-catch and executes regardless exception occurred or not.

- Each try block must have atleast one catch or finally block.

8. Java's built-in exceptions

- Unchecked Exceptions : Not checked during compile time. Ex. NullPointerException, IndexOutOfBoundsExceptions.

- Checked Exceptions : Needs to be included in method's throws list. Ex. IllegalAccessException.

9. To create customized exception one should extends Exception Class(Ex).

10. Chained Exception

- Sometimes lets say ArithmeticException occurs due to divide by zero but real cause of exception was IOException. In such cases chained exceptions are handy to find the actual cause of exception. It has been added in Java 1.4

- Methods added to the Throwable Class(Ex)

```
Throwable getCause()
Throwable initCause(Throwable causeExc)
```

13) Multithreading

1. There are two types of multithreading one is process based and other is thread based. Process based is heavy weight and thread based is light weight.

2. To create a new thread one should
a. Extend Thread class.
b. Implement Runnable interface.

3. Thread class defines many method like getName(), getPriority(), isAlive(), sleep() etc and Runnable interface has only one method run().

4. Thread Creation

1. When Class A extends Thread, then thread is created through

```
Thread t = new A();
t.start();
```

2. When Class A impements Runnable then thread is created through

```
Thread t = new Thread(new A());
t.start();
```

5. Runnable interface is created because

i. One is not really specializing the Thread behavior rather one is just giving it something to run.

ii. Interface gives cleaner separation between ones code andd implementation of Thread.

Iii. A class implementing Runnable can extend any other class.

6. Thread Priority

- Java assigns priority to threads.
- Priority is an integer.
- Higher priority thread doesn't mean it run faster.
- Priority of an thread decides when to switch one running thread to next. Its called context switch.
- Context switch takes place first if any running thread relinquish its control by yeilding, sleeping or blocking on pending I/O and second when lower priority thread pre-empted by higher priority thread regardless of what it was doing.
- Thread priority could be set through setPriority(int level) method where value of 'level' should be between MIN_PRIORITY(0) and MAX_PRIORITY(10). Also NORMAL_PRIORITY is 5.

7. One can even control main thread using

Thread.currentThread() method(Ex).

8. Thread[main, 5, main] where first main is name of thread, 5 is priority and second main is name of its group.

9. join() method

- Its always expected main() or some other thread to terminate after some other threads have been terminated. In such scenerio method join() comes handy.
- When join() is called on some thread lets say t1 then all other thread will be paused till t1 complete(Ex).

10. Synchronization

- Sometimes we can see two or more thread accessing same method at the same time brings unexpected results. If the method has sequence of print statements then they are printed in wrong order.

Synchronization in java helps in such scenerios.

10.1. There are two ways java uses synchronization

i. Synchronized methods(Ex)

- For methods 'synchronized' keyword should be used infront of method. With it only one thread at a time can access it.
- Each object has a lock also called monitor. When any synchronized method of an object is called then calling thread first acquires its lock. With this now no other thread can call any of it method marked synchronized. That is ones lock is acquired by a thread all synchronized methods of that object gets locked

ii. Synchronized blocks(Ex)

- Insome scenerios when one wants to call a third party method which is not synchronized and also not accessible in such cases synchronized block will be useful.
- Synchronized block must be given the object to synchronized upon.
- Example:

```
public void run()
{
     synchronized(sm1)
     {
      sm1.call(str);
     }
}
```

-(*) One should keep in mind that synchronization only works with same object. Two threads accessing same method of a class using two separate objects of the class don't needs synchronization.

11. wait() and notify()

- There are scenerios when two methods both are synchronized accessing by two different threads too brings unexpected output due to system functioning(Ex). For example one thread is writing a data and other thread reading it at the same time may end up with wrong results even after synchronization.