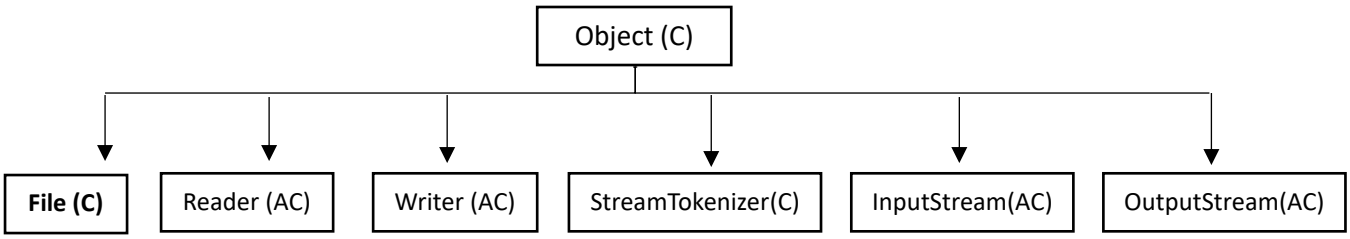
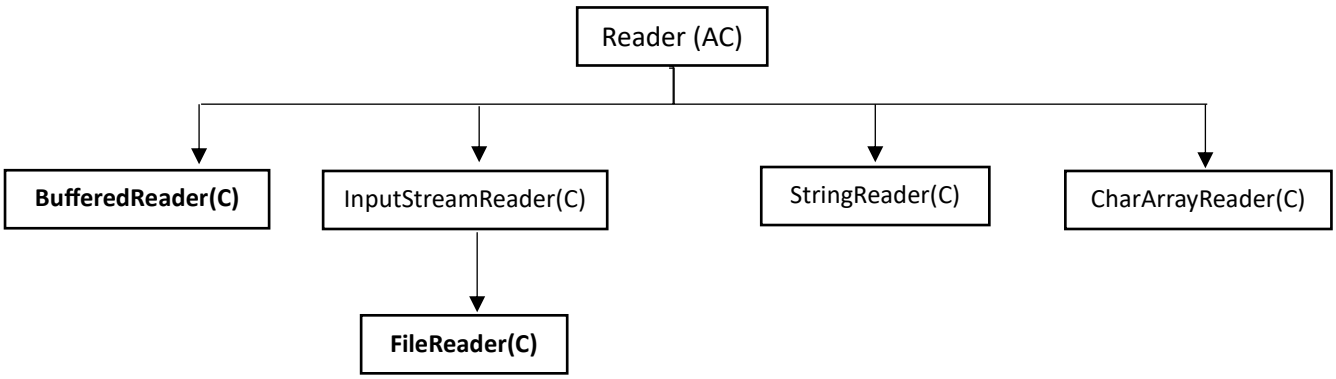
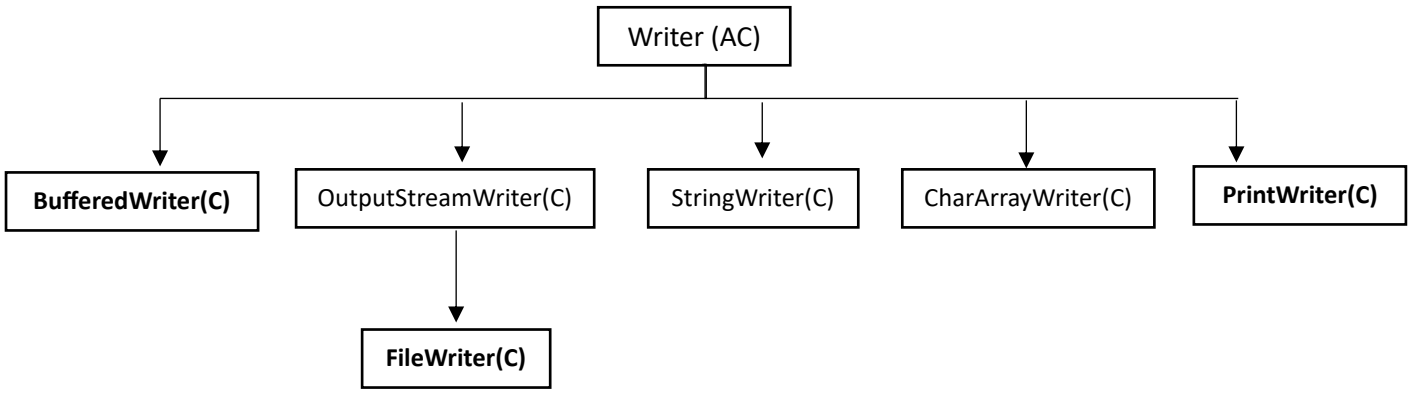
File Input Output, Inner classes, Generics, Garbage Collection

No.	Topics
1.	File class, constructors, methods
2.	FileWriter, constructors, methods
3.	FileReader, constructors, methods
4.	BufferedWriter, constructors, methods
5.	BufferedReader, constructors, methods
6.	PrintWriter, constructors, methods
1.	Inner classes intro, types of inner classes
2.	Normal or Regular Inner classes
3.	Method local inner classes
4.	Anonymous Inner classes
5.	Static Nested classes
1.	Generics Intro & classes
2.	Bounded types
3.	Generics methods & wild – card character
4.	Communication with non – generic code
1.	Garbage collection (GC) intro
2.	Ways to make an object eligible for GC
3.	Methods for requesting JVM to run GC
4.	Finalization







java.io.**InputStream** (implements java.io.Closeable)

- java.io.ByteArrayInputStream
- java.io.FileInputStream
- java.io.FilterInputStream
  - java.io.BufferedInputStream
  - java.io.DataInputStream (implements java.io.DataInput)
- java.io.ObjectInputStream (implements java.io.ObjectInput, java.io.ObjectStreamConstants)
- java.io.StringBufferInputStream

java.io.**OutputStream** (implements java.io.Closeable, java.io.Flushable)

- java.io.ByteArrayOutputStream
- java.io.FileOutputStream
- java.io.FilterOutputStream
  - java.io.BufferedOutputStream
  - java.io.DataOutputStream (implements java.io.DataOutput)
  - java.io.PrintStream (implements java.lang.Appendable, java.io.Closeable)
- java.io.ObjectOutputStream (implements java.io.ObjectOutput, java.io.ObjectStreamConstants)

1. File class

```
File f = new File ("abc.txt");
```

- This line won't create any physical file. First it will check is there any physical file named with "abc.txt" is available or not. If it's available, then f simply refers to that file. If not available, then **we're just creating java file object to represent the name "abc.txt"**

```
class FileDemo {
   public static void main (String [] args) throws IOException {
      File f = new File("abc.txt");
      System.out.println(f.exists());
      f.createNewFile ();
      System.out.println(f.exists());
   }
}
```

1st Run Output
false
true

2nd Run Output
true
true

- **We can use Java file object to represent directory** also (because java file i/o concept is implemented based on UNIX OS i.e., In unix everything is treated as file)

```
class FileDemo {
   public static void main (String [] args) throws IOException {
      File f = new File("abc.txt");
      System.out.println(f.exists());
      f.mkdir ();
      System.out.println(f.exists());
   }
}
```

1st Run Output
false
true

2nd Run Output
true
true

File class Constructors

1. File f = new File (String name)
   - creates a java file object to represent name of the file or directory in current working directory.
2. File f = new File (String subdirName, String name)
   - Creates a java file object to represent name of the file or directory present in specified subdirectory.
3. File f = new File (File subdir, String name)

Imp. Methods of File class

No.	Methods	Description
1.	boolean exists ()	Returns true if the specified file or directory is available.
2.	boolean createNewFile (String fileName)	If file is already available returns false else create a new file & returns true.
3.	boolean mkdir (String dirName)	If directory is already available returns false else create a new directory & returns true
4.	boolean isFile ()	Returns true if the specified file object pointing to the physical file.
5.	boolean isDirectory ()	Returns true if the specified file object pointing to the directory.
6.	String [] list ()	Returns the names of all files & sub directories present in specified directory.
7.	long length ()	Returns no. of characters present in specified file.
8.	boolean delete ()	Deletes specified file or directory.

2. FileWriter

public class FileWriter **extends** OutputStreamWriter public class OutputStreamWriter **extends** Writer

- We can use FileWriter to write character data to the File.

Constructors

1. FileWriter fw = new FileWriter (String name);
2. FileWriter fw = new FileWriter (File f);
   - Above both FileWriter constructors **meant for overriding of existing data.**
3. FileWriter fw = new FileWriter (String name, boolean append);
4. FileWriter fw = new FileWriter (File f, boolean append);
   - Instead of overriding, if we want **append operation** by using the above constructors.

- If the specified file is not already available then all the above constructors will create that file.

Imp. Methods of FileWriter class

No.	Methods	Description
1.	write (int i)	To write a single character
2.	write (char [] ch)	To write an array of characters
3.	write (String s)	To write string to the file.
4.	flush ()	To give the guarantee that total data including last character will be written to the file.
5.	close ()	To close the writer

Note: The main problem with FileWriter is we have to insert line separator (\n) manually which is varied from system to system. We can solve this problem by using BufferedWriter & PrintWriter classes.

3. BufferedWriter

public class BufferedWriter **extends** Writer

- We can use BufferedWriter to write character data to the file.

Constructors

1. BufferedWriter bw = new BufferedWriter (Writer w);
2. BufferedWriter bw = new BufferedWriter (Writer w, int buffersize);

- BufferedWriter can't communicate directly with the File. It can communicate via some writer object like FileWriter etc.

  E.g., BufferedWriter bw = new BufferedWriter (new FileWriter("abc.txt"));

Methods: same as FileWriter including

newLine ()	To insert a line separator

- Whenever we're closing BufferedWriter automatically internal FileWriter will be closed & we're not required to close explicitly.

4. PrintWriter

```
public class PrintWriter extends Writer
```

- It is the most enhanced writer to write character data to the file.
- The main advantage of PrintWriter over FileWriter & BufferedWriter is we can write any type of Primitive data directly to the file.

Constructors

1. PrintWriter pw = new PrintWriter (String name);
2. PrintWriter pw = new PrintWriter (File f);
3. PrintWriter pw = new PrintWriter (Writer w);

Note: PrintWriter can communicate directly with the file & can communicate via some writer object also.

Imp. Methods of PrintWriter (all methods of Writer class)

- Below methods adds specific type of datatype to the file.

No.	Methods
1.	print (char ch), println (char ch)
2.	print (int i), println (int i)
3.	print (double d), println (double d)
4.	print (String s), println (String s)
5.	print (boolean b), println (boolean b)

5. FileReader

```
public class FileReader extends InputStreamReader

public class InputStreamReader extends Reader
```

- We can use FileReader to read character data from the file.

Constructors

1. FileReader fr = new FileReader (String name);
2. FileReader fr = new FileReader (File f);

Methods

No.	Methods	Description
1.	int read()	It attempts to read next character from the file & returns its unicode value. If the next character not available, then it returns -1. As this method returns unicode value, at the time of printing we have to perform typecasting.
2.	int read (char [] ch)	It attempts to read enough characters from the file into char [] array & returns no. of characters copied from the file.
3.	void close ()	

Note: By using FileReader, we can read data character by character which is not convenient to the Programmer.

- Usage of FileWriter & FileReader is not recommended because
  1. While writing data by FileWriter, we have to insert line separator (\n) manually which is varied from system to system. It is difficult to the Programmer.
  2. By using FileReader, we can read data character by character, which is not convenient to the programmer.
- To overcome these problems, we should go for BufferedWriter & BufferedReader.

6. BufferedReader

public class BufferedReader **extends** Reader

- We can use BufferedReader to read character data from the file.
- The main advantage of BufferedReader when compared with FileReader is we can read both line by line in addition to character to character.

Constructors

1. BufferedReader br = new BufferedReader (Reader r);
2. BufferedReader br = new BufferedReader (Reader r, int bufferSize);

Note: BufferedReader can't communicate directly with the File & it can communicate via some Reader object.

Methods: all methods of Reader including

String **readLine** ()	It attempts to read nextLine form the file & returns it. If the nextLine not available then this method returns null;

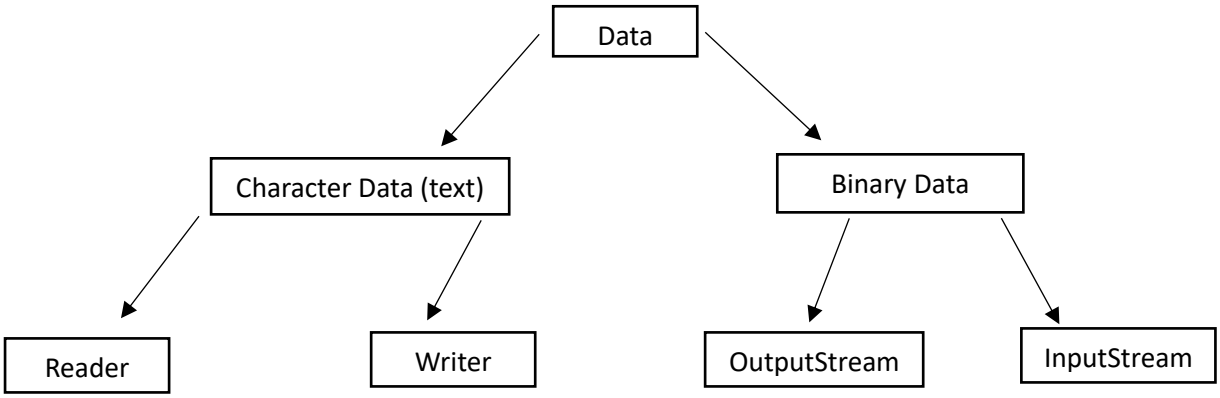
- Whenever we're closing BufferedReader automatically underlying FileReader will be closed & we're not required to close explicitly.
- The most enhanced Reader to read character data from the File is BufferedReader.

Conclusion

1. The most enhanced Writer to write character data to the file is **PrintWriter** whereas the most enhanced Reader to read character data from the file is **BufferedReader**.

2. In general, we can use Readers & Writers to handle character data (text data), whereas we can use streams to handle Binary Data (like images, pdf, video, audio, files etc)

3. We can use **OutputStream** to write binary data to the File. **InputStream** to read binary data from the File.



Inner classes

No.	Topics
1.	Inner classes intro, types of inner classes
2.	Normal or Regular Inner classes
3.	Method local inner classes
4.	Anonymous Inner classes
5.	Static Nested classes

1. Inner classes intro

- Sometimes we can declare a class inside another class, such type of class is called Inner class.
- Inner classes concept introduced in 1.1 version to fix GUI bugs as the part of event handling but because of powerful features & benefits of inner classes, slowly programmers started using in regular coding also.
- Without existing one type of Object, if there is no chance of existing another type of object then we should go for Inner classes.
- Without existing Outer class object, there is no chance of existing Inner class object.
- The Relation b/w Outer class & Inner class is not IS_A_RELATIONSHIP but it's a HAS_A_RELATIONSHIP (Composition or Aggre

E.g.1, **University** consists of several **Departments**. Without existing **University**, there is no chance of existing **Department** hence, we have to declare **Department** class inside **University** class.

```
class University {          // Outer class
    class Department {    // Inner class
        ....
    }
}
```

```
class Car {          // Outer class
    class Engine {    // Inner class
        ....
    }
}
```

E.g.2, **Map** is a group of key – value pairs & each key – value pair is called a **Entry**. Without existing **Map** object, there is no chance of existing **Entry** object, hence interface **Entry** is defined inside **Map** interface

```
interface Map {          // Outer interface
    interface Entry {    // Inner interface
        ....
    }
}
```

- Based on position of declaration & behavior all inner classes are divided into 4 types:
  1. Normal or Regular inner classes
  2. Method local inner classes
  3. Anonymous inner classes
  4. Static Nested classes

1. Normal or Regular inner classes

➢ If we're declaring any named class directly inside a class without static modifier, such type of inner class is called Normal or Regular inner class.

javac Outer.java → 2 class files will be created → Outer.class, Outer$Inner.class

Outer.java

```
class Outer {
    class Inner {

    }
}
```

```
D:\Eclipse Workspace\Practice>java Outer
Error: Main method not found in class Outer, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application

D:\Eclipse Workspace\Practice>java Outer$Inner
Error: Main method not found in class Outer$Inner, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

javac Outer.java → 2 class files will be created → Outer.class, Outer$Inner.class

Outer.java

```
class Outer {
 class Inner {
    public static void main (String [] args) {
      SoPln ("Inner class main method");
    }
 }
 public static void main (String [] args) {
    SoPln ("Outer class main method");
 }
}
```

```
D:\Eclipse Workspace\Practice>javac Outer.java

D:\Eclipse Workspace\Practice>java Outer
Outer class main method

D:\Eclipse Workspace\Practice>java Outer$Inner
Inner class main method
```

javac Outer.java → 2 class files will be created → Outer.class, Outer$Inner.class

Outer.java

```
class Outer {
 class Inner {
    public static void main (String [] args) {
      SoPln ("Inner class main method");
    }
 }
}
```

```
D:\Eclipse Workspace\Practice>java Outer$Inner
Inner class main method

D:\Eclipse Workspace\Practice>javac Outer.java

D:\Eclipse Workspace\Practice>java Outer
Error: Main method not found in class Outer, please define the
main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Ap
plication

D:\Eclipse Workspace\Practice>java Outer$Inner
Inner class main method
```

javac Outer.java → 2 class files will be created → Outer.class, Outer$Inner.class

Outer.java

```
class Outer {
  class Inner {
     static int i = 10;
     public static void m1 () {
        SoPln ("Inner static m1() method");
     }
  }
  public static void main (String[] args) {
     SoPln ("static i = " + Inner.i);
     Inner.m1();
  }
}
```

```
D:\Eclipse Workspace\Practice>javac Outer.java

D:\Eclipse Workspace\Practice>java Outer
static i = 10
Inner static m1() method
```

Case 1: Accessing Inner class code from static area of Outer class

```
class Outer {
    class Inner {
      public void m1 () {
          SoPln ("Inner class m1() method");
      }
   }

   public static void main (String [] args) {
       Outer o = new Outer ();
       Outer.Inner i = o.new Inner ();
       i.m1();                                    // Inner class m1() method

       Inner i1 = o.new Inner ();
       i1.m1();                                   // Inner class m1() method

      Inner i2 = new Outer ().new Inner();
      i2.m1();                                   // Inner class m1() method

      new Outer ().new Inner().m1();          //  Inner class m1() method
   }
}
```

Case 2: Accessing inner class code from instance area of Outer class

```
class Outer {
     class Inner {
         public void m1 () {
             SoPln ("Inner class m1() method");
         }
     }
     public void m2 () {
         Inner i = new Inner ();
         i.m1();                    //  Inner class m1() method
     }
     public static void main (String [] args) {
         Outer o = new Outer ();
         o.m2();
     }
}
```

Case 3: From Normal or regular inner class, we can access both static & non – static member of Outer class directly.

```
class Outer {
    int x = 10;
    static int y = 20;
    class Inner {
          public void m1 () {
               System.out.println("x: " + x);       // 10
               System.out.println("y: " + y);       // 20
          }
     }

   public static void main (String [] args) {
        new Outer ().new Inner().m1();
    }
}
```

Case 4: Accessing inner class code from outside the Outer class

```
class Outer {
   class Inner {
      public void m1 () {
          SoPln ("Inner class m1() method");
      }
   }
}

class Test {
    public static void main (String [] args) {
        Outer o = new Outer ();
        Outer.Inner i = o.new Inner ();
         i.m1();                              // Inner class m1() method

        // Inner i1 = o.new Inner ();
        // i1.m1();                            // error: can't find symbol Inner

        // Inner i2 = new Outer().new Inner();
        // i2.m1();                            // error: can't find symbol Inner

        new Outer ().new Inner().m1();   // Inner class m1() method
    }
}
```

Case 5: Within the inner class, this always refers to current Inner class object. If we want to refer current Outer class object, we have to use **Outerclassname.this**

```
class Outer {
     int x = 10;
     class Inner {
          int x = 100;
          public void m1 () {
                int x = 1000;
                System.out.println(x);                  // 1000
                System.out.println(this.x);             // 100
                System.out.println(Outer.this.x);       // 10
          }
     }

     public static void main (String [] args) {
          new Outer().new Inner().m1();
     }
}
```

- The only modifiers applicable for **Outer class** are [public, <default>, final, abstract, strictfp]
- The only modifiers applicable for **Inner class** are [**Outer class modifier** + private, protected, static]

Nesting of Inner class

- Inside Inner class, we can declare another Inner class i.e., nesting of Inner classes is possible.

```
class A {
    class B {
        class C {
            public void m1 () {
                System.out.println("Inner most class m1() method");
            }
        }
    }
}

class Test {
     public static void main (String [] args) {
         A a = new A ();
         A.B b = a.new B ();
         A.B.C c = b.new C ();
         c.m1();
    }
}
```

2. Method Local Inner classes

- Sometimes we can declare a class inside a Method, such types of Inner classes are called Method local inner classes.
- The main purpose of Method local inner class is to define Method specific repeatedly required functionality.
- Method local inner classes are best suitable to meet Nested Method requirements (Nested method is not allowed in Java)
- We can access Method local inner classes only within the method where we declared. Outside of the method we can't access.
- Because of its less scope, Method Local inner classes are most rarely used type of Inner classes.

```
class Test {
    public void m1 () {
        class Inner {
            public void sum (int x, int y) {
                System.out.println("The sum: " + (x + y));
            }
        }
        Inner i = new Inner ();
        i.sum (10, 20);            // 30
        i.sum (100, 200);          // 300
        i.sum (1000, 2000);        // 3000
    }

    public static void main (String [] args) {
         Test t = new Test ();
         t.m1();
     }
}
```

- We can declare Method Local inner class inside both Instance & static methods
- If we declare Inner class inside instance method then from that Method local Inner class, we can access both static & non – static members of Outer class directly.
- If we declare Inner class inside static method then from that Method local Inner class, we can access only static members of Outer class directly.
- From Method local Inner class, we can't access local variable of the methods in which we declare inner class.

```
class Test {
        int x = 1000;
        static int y = 2000;
        public void m1 () {
            int x = 100;
            class Inner {
                public void m2 () {
                    System.out.println(x);              // 100
                    System.out.println(Test.this.x);    // 1000
                    System.out.println(y);              //  2000
                 }

                 public static void m3 () {
                   // System.out.println(x);            // non-static variable x can't be referenced from a static context
                   // System.out.println(Test.this.x);  // non-static variable this can't be referenced from a static context
                    System.out.println(y);              // 2000
                }
            }
            Inner i = new Inner ();
            i.m2();
            Inner.m3();
      }

      public static void main (String [] args) {
          Test t = new Test ();
          t.m1();
      }
}
```

- The only applicable modifiers for Local Method Inner class are [final, abstract, strictfp]

3. Anonymous Inner classes

- Sometimes we can declare Inner class without name, such type of Inner classes are called Anonymous Inner classes.
- The main purpose of Anonymous Inner classes is just for instant use (one time usage)
- Based on declaration & behaviour, these are 3 types of Anonymous Inner classes:
  a) Anonymous Inner class that extends a class
  b) Anonymous Inner class that implements an interface
  c) Anonymous Inner class that defined inside arguments

a) Anonymous Inner class that extends a class

```
class Popcorn {
    public void taste () {
        System.out.println("Salty");
    }
}
class Test {
    public static void main (String [] args) {
        Popcorn p1 = new Popcorn ();     // just creating Popcorn object
        p1.taste();      // Salty
       // Here we're creating child object with Parent Reference & we're declaring a class
          that extends Popcorn without name & we're overriding taste() method.
        Popcorn p2 = new Popcorn () {
            public void taste () {
                System.out.println("Spicy");
            }
        };
        p2.taste();       // Spicy
        Popcorn p3 = new Popcorn () {
            public void taste () {
                System.out.println("Sweet");
            }
        };
        p3.taste();       //  Sweet
        System.out.println(p1.getClass().getName());     // Popcorn
        System.out.println(p2.getClass().getName());     // Test$1
        System.out.println(p3.getClass().getName());     // Test$2
    }
}
```

```
class MyThread extends Thread {
    public void run () {
        for (int i = 0; i < 10; i++)
            System.out.println("Child Thread");
    }
}

class ThreadDemo {
    public static void main (String [] args) {
        MyThread t = new MyThread ();
        t.start();
        for (int i = 0; i < 10; i++)
            System.out.println("Main Thread");
    }
}
```

Anonymous Inner class (Thread creation)

```
class ThreadDemo {
    public static void main (String [] args) {
        MyThread t = new MyThread () {
            public void run () {
                for (int i = 0; i < 10; i++)
                    System.out.println("Child Thread");
            }
        };
        t.start();
        for (int i = 0; i < 10; i++)
            System.out.println("Main Thread");
    }
}
```

b) Anonymous Inner class that implements an interface

```
class MyRunnable implements Runnable {
    public void run () {
        for (int i = 0; i < 10; i++)
            System.out.println("Child Thread");
    }
}

class ThreadDemo {
     public static void main (String [] args) {
          MyRunnable r = new MyRunnable ();
          Thread t = new Thread(r);
          t.start();
          for (int i = 0; i < 10; i++)
               System.out.println("Main Thread");
     }
}
```

Anonymous Inner class (Implementing Runnable i/f)

```
class ThreadDemo {
     public static void main (String [] args) {
          MyRunnable r = new MyRunnable () {
               public void run () {
                    for (int i = 0; i < 10; i++)
                         System.out.println("Child Thread");
               }
          };
          Thread t = new Thread(r);
          t.start();
          for (int i = 0; i < 10; i++)
               System.out.println("Main Thread");
     }
}
```

c) Anonymous Inner class that defined inside Arguments

```
class MyRunnable implements Runnable {
    public void run () {
        for (int i = 0; i < 10; i++)
            System.out.println("Child Thread");
    }
}

class ThreadDemo {
     public static void main (String [] args) {
          MyRunnable r = new MyRunnable ();
          Thread t = new Thread(r);
          t.start();
          for (int i = 0; i < 10; i++)
               System.out.println("Main Thread");
     }
}
```

Anonymous Inner class (Implementing Runnable i/f)

```
class ThreadDemo {
    public static void main (String [] args) {
          new Thread (new Runnable () {
               public void run () {
                    for (int i = 0; i < 10; i++)
                         System.out.println("Child thread");
               }
          }).start();

          for (int i = 0; i < 10; i++)
               System.out.println("Main thread");
    }
}
```

No.	Normal Java Class	Anonymous Inner class
1.	A normal Java class can implement any no. of interfaces simultaneously.	Anonymous inner class can implement only one interface at a time.
2.	A normal Java class can extend a class & can implement any no. of interfaces simultaneously.	Anonymous inner class can either extend a class or can implement an interface but not simultaneously.
3.	In normal Java class, we can write any no. of constructors.	In Anonymous inner classes, we can't write any constructors (because the name of the class & the name of the constructor must be same but anonymous inner classes not having any name)

- If the requirement is standard & required several times then we should go for Normal Top – level class.
- If the requirement is temporary & required only one (instant use) then we should go for Anonymous inner class.

- We can use Anonymous inner classes frequently in GUI based applications to implement event handling.

4. Static Nested classes

- Sometimes we can declare inner class with static modifier, such types of inner classes are called static nested classes.
- In case of Normal or Regular Inner class, without existing Outer class object, there is no chance of existing inner class object i.e., Inner class object is strongly associated with outer class object.
- But in case of static Nested classes, without existing Outer class object, there may be a chance of existing Nested class object, hence static Nested class object is not strongly associated with Outer class object.

```
class Outer {
    static class Nested {
        public void m1 () {
            SoPln ("static Nested class method");
        }
    }

    public static void main (String [] args) {
        Nested n = new Nested ();
        n.m1();
    }
}
```

```
class Outer {
    static class Nested {
        public void m1 () {
            SoPln ("static Nested class method");
        }
    }
}

class Test {
    public static void main (String [] args) {
        Outer.Nested n = new Outer.Nested();
        n.m1();
    }
}
```

For more refer notes

Generics

1. Generics intro

- The main objectives of Generics are:
  - To provide type – safety
  - To resolve type – casting problems

Case 1: Type – safety

No.	Array	Collection
1.	Arrays are type – safe i.e., we can give the guarantee for the type of elements present inside array	Collections are not type – safe i.e., we can't give the guarantee for the type of elements present inside collection.
2.	**E.g.,** If our programming requirement is to hold only String type objects, we can choose String [] (by mistake if we're trying to add any other type of objects, we will get compile time error [incompatible types])	**E.g.,** If our programming requirement is to hold only String type objects & if we choose ArrayList then if by mistake we're trying to add any other type of objects, we will not get any compile time error but the program may fail at Runtime.
3.	String [] s = new String [1000]; s[0] = "sam"; s[1] = "Ravi"; s[2] = 10; // **Compile Time Error: incompatible types** int cannot be converted to String	ArrayList al = new ArrayList (); al.add("sam"); al.add("ravi"); al.add(10); System.out.println(al); // [sam, ravi, 10]
4.	As we have seen above, we can give the guarantee for the type of elements present inside Array. Hence Arrays are safe to use wrt type.	As we have seen above, we can't give the guarantee for the type of elements present inside Collection. Du to this Collections is not safe to use wrt type.

Case 2: Type – casting

No.	Array	Collection
1.	In case of Arrays, at the time of retrieval it's not required to perform type casting because there is guarantee for the type of elements present inside Array.	In case of Collections, at the time of Retrieval compulsory we should perform type casting because there is no guarantee for the type of elements present inside Collection.
2.	String [] s = new String[1000]; s[0] = "sam"; String name1 = s[0]; System.out.println(name1); // sam **// Type Casting is not required**	ArrayList al = new ArrayList (); al.add("sam"); al.add("ravi"); System.out.println(al); // [sam, ravi] //String name1 = al.get(0); **incompatible types: Object cannot be converted to String, TypeCasting required** String name1 = (String)al.get(0); // Type casting System.out.println(name1); // sam
3.	Type casting is not required in case of Array.	Type casting is required in case of Collection.

- Hence, through Generics, we can solve type – safety & type – casting problem.

No.	ArrayList al = new ArrayList ();	ArrayList <String> al = new ArrayList<>();
1.	It is a non – generic version of ArrayList object.	It is the generic version of ArrayList.
2.	For this ArrayList, we can add any type of Object & hence it is not type – safe.	For this ArrayList, we can add only String type of objects & hence it's type – safe.
3.	At the time of Retrieval, compulsory we have to perform type – casting.	At the time of retrieval, compulsory we are not required to perform type – casting.

- Polymorphism concept is applicable only for the Base Type but not for Parameter type. (Polymorphism concept is usage of Parent reference to hold child object)

```
ArrayList <String> al = new ArrayList<> ();                 // Right
ArrayList <Object> al = new ArrayList <String> ();      // Wrong

Here ArrayList is the Base Type & String is Parameter Type.
```

- For the Type Parameter, we can provide any class or interface name but not Primitives. If we're trying to provide Primitive then we will get compile time error.

E.g., **ArrayList <int> x = new ArrayList<>(); // Wrong**

2. Generic classes

- Until 1.4 version, a non – generic version of ArrayList class is declared as follow:
  - The argument to add () method is Object & hence, we can add any type of object to the ArrayList. Due to this we're missing type – safety.
  - The return type of get () method is Object & hence, at the time of retrieval, we have to perform type casting.

```
class ArrayList {
        add (Object o);
        Object get (int index);
}
```

Generic

```
class ArrayList <T> {
      add (T t);
      T get (int index)
}
Where T is Type parameter
```

- But in 1.5 version, a generic version of ArrayList class is declared as above.
- In Generics, we're associating a type parameter to the class. Such type of parameter raised classes are nothing but Generic classes or Template classes.

```
class Gen <T> {
   T obj;
   Gen (T obj) {
      this.obj = obj;
   }
   public void show () {
      SoPln ("The type of obj: " + obj.getClass().getName());
   }
   public T getObj () {
      return obj;
   }
}

public class GenericsDemo {
   public static void main (String [] args) {
      Gen<String> g1 = new Gen<>("sam");
      g1.show();                  // The type of obj: java.lang.String
      SoPln (g1.getObj());     // sam
```

```
        Gen<Integer> g2 = new Gen<>(10);
        g2.show();              // The type of obj: java.lang.Integer
        SoPln (g2.getObj());   // 10

        Gen<Double> g3 = new Gen<>(10.5);
        g3.show();              // The type of obj: java.lang.Double
        SoPln (g3.getObj()); // 10.5
    }
}
```

3. Bounded Types

- We can bound the type parameter for a particular range by using **extends** keyword. Such types are called **Bounded Type.**
- Here, as the type parameter, we can pass any type without any restriction & hence it is Unbounded type.
- **Syntax for Bound Type**

```
class Test <T> {     // Unrestricted type
        ....
}

Test<Integer> t1 = new Test<>();
Test<String> t2 = new Test<>();
```

class **Test** <T **extends** X> { // Here X can be either class or Interface
        ....
}

- If X is a class, then we can pass either X type or its child classes.
- If X is an interface, then we can pass either X type or its implementation classes

E.g.1,

```
class Test <T extends Number> {
        ....
}

Test<Integer> t1 = new Test<>();
Test<String> t2 = new Test<>();     // Wrong; Type parameter String is not within its bound
```

E.g.2,

```
class Test <T extends Runnable> {
        ....
}

Test<Runnable> t1 = new Test<>();
Test<Thread> t2 = new Test<>();
Test<Integer> t3 = new Test<>();     // Wrong; Type parameter String is not within its bound
```

- We can define bounded types even in combination also.

```
class Test <T extends Number & Runnable> {
      ....
}
```

Here as the type – parameter T, we can take anything which should be child class of Number & should implement Runnable interface

```
class Test <T extends Runnable & Comparable> {}
class Test <T extends Number & Runnable & Comparable> {}
// Wrong becoz we have to take class first followed by interface next
class Test <T extends Runnable & Number> {}
// Wrong becoz we can't extend more than one class simultaneously
class Test <T extends Number & Thread> {}
```

- We can define bounded types only by using extends keyword & we can't use implements or super keywords. But we can replace implements keyword purpose with extends keyword.
  E.g., class **Test** <T **extends** Number> {}, class **Test** <T **implements** Runnable> {}, class **Test** <T **super** String> {}

- As the type – parameter 'T', we can take any valid Java identifier but its convention to use 'T'
- Based on our requirement, we can declare any no. of type – parameters & all these type – parameters should be separated by comma (,)

class **Test** <A, B> {} class **Test** <X, Y, Z> {}	class **HashMap** <K, V> {} where K -> key, V -> value e.g., HashMap<Integer, String> h = new HashMap<>()

4. Generic methods & wild – card character (?)

No.	Methods & Description
1.	**m1 (ArrayList<String> al)** • We can call this method by passing ArrayList of only String type. • But within the method, we can add only String type of objects to the list.
2.	**m1 (ArrayList <?> al)** • We can call this method by passing ArrayList of any unknown type or any type. • But within the method, we can't add anything to the list except null. Because we don't know the type exactly (null is allowed becoz it's a valid value for any type) • This type of methods are best suitable for Read – only operations.
3.	**m1 (ArrayList <? Extends X> al)** • X can be either class or interface • If X is a class, then we can call this method by passing ArrayList of either X type or its child classes. • If X is an interface, then we can call this method by passing ArrayList of either X type or its implementation classes. • But within the method, we can't add anything to the list except null. Because we don't know the type exactly (null is allowed becoz it's a valid value for any type) • This type of methods are best suitable for Read – only operations.
4.	**m1 (ArrayList <? Super X> al)** • X can be either class or interface • If X is a class, then we can call this method by passing ArrayList of either X type or its super classes. • If X is an interface, then we can call this method by passing ArrayList of either X type or super class of Implementation class of X. • But within the method, we can add X type of Objects & null to the list.

- We can declare type parameter either at class level or at Method level.

Declaring Type parameter at Class level	Declaring Type parameter at Method level
class **Test** <T> { // We can use 'T' within this class based on our requirement. }	class **Test** { public <T> void **m1**(T ab) { // We can use 'T' anywhere within this method based on our requirement. } } We have to declare Type parameter just before return type.

- We can define bounded types even at method level also.

public <T>void m1()

<T extends Number>
<T extends Runnable>
<T extends Number & Runnable> etc.
Similar to class

5. Communication with Non – Generic Code

- If we send Generic object to Non – generic area then it starts behaving like Non – Generic object.
- Similarly, if we send Non – Generic object to Generic area, then it starts behaving like Generic object i.e., the location in which object present based on that behavior will be defined.

Conclusion

1. The main purpose of Generics is
   - To provide type – safety
   - To resolve type – casting problems
2. Type – safety & type – casting both are applicable at compile time hence Generics concept also applicable only at compile time but not at runtime.
3. At the time of compilation, as last step Generics syntax will be removed & hence for the JVM, Generic syntax won't be available.

```
ArrayList al = new ArrayList <String> ();
ArrayList al = new ArrayList <Integer> ();
ArrayList al = new ArrayList <Double> ();      All are same
ArrayList al = new ArrayList();

ArrayList <String> al = new ArrayList<String>();
ArrayList <String> al = new ArrayList();          // same as above
```

```
class Test {
    public void m1 (ArrayList<String> al) {}    → m1 (ArrayList al)
    public void m1 (ArrayList<Integer> al) {}   → m1 (ArrayList al)
}
CE: name class: Both methods have same erasure
```

4. Within a class, 2 methods with same signature not allowed otherwise we will get compile time error.
5. At compile time
   a) Compile code normally by considering generic syntax
   b) Remove generic code
   c) Recompile the resultant code for erasure checking i.e., checking method signature

Garbage Collection

1.	Garbage collection (GC) intro
2.	Ways to make an object eligible for GC
3.	Methods for requesting JVM to run GC
4.	Finalization

1. Garbage collection intro

- In old languages like C++, Programmer is responsible to create new object & to destroy useless object.
- Usually, Programmers take care while creating objects & neglecting the destruction of useless object. Because of this negligence, at certain point for creation of new object sufficient memory may not be available (because total memory filled with useless objects only) & whole application will be down with memory problem.
- Hence OutofMemoryError is very common problem in C++;

- But in java, Programmer is responsible only for creation of Objects & destruction of useless object is taken care by Garbage collector.
- SUN people provided one assistant to destroy useless objects. This assistant is always running in the background (Daemon Thread) & destroy useless objects. This assistant is called **Garbage Collector.**
- Just because of this assistant, the chance of failing java program with memory problem is very low.

2. Ways to make an object eligible for GC

- Even though programmers are not responsible to destroy useless objects. It is highly recommended to make an object eligible for GC if it's no longer required.
- An object is said to be eligible for GC iff it doesn't have any reference variable.

The following are various ways to make an object eligible for GC:

a) Nullifying the reference variable
b) Reassigning the reference variable
c) Objects created inside a method
d) Island of Isolation

a) Nullifying the reference variable

- If an object no longer required then assign null to all its reference variables, then that object automatically eligible for GC.

```
Student s1 = new Student ();
Student s2 = new Student ();

s1 = null;        //  1st object eligible for GC
s2 = null;        //  2nd object eligible for GC
```

b) Reassigning the reference variable

- If an object no longer required then reassign its reference variable to some other object then old object by default eligible for GC.

```
Student s1 = new Student ();
Student s2 = new Student ();

s1 = new Student ();     //  1st object eligible for GC
s2 = s1                  //  2nd object eligible for GC
```

c) Objects created inside a method

- The objects which are created inside a method are by default eligible for GC once method completes.

```
class Test {
    public static void main (String [] args) {
        m1 ();    // 2 objects eligible for GC
    }
    public static void m1 () {
        Student s1 = new Student ();   // GC
        Student s2 = new Student ();   // GC
    }
}
```

```
class Test {
    public static void main (String [] args) {
        Student s = m1 ();   // 1 object eligible for GC
    }
    public static Student m1 () {
        Student s1 = new Student ();
        Student s2 = new Student ();      // GC
        return s1;
    }
}
```

```
class Test {
    public static void main (String [] args) {
        m1 ();    // 2 objects eligible for GC
    }
    public static Student m1 () {
        Student s1 = new Student ();  // GC
        Student s2 = new Student ();  // GC
        return s1;
    }
}
```

```
class Test {
    static Student s;
    public static void main (String [] args) {
        m1 ();  // 1 object eligible for GC
    }
    public static void m1 () {
        s = new Student ();
        Student s2 = new Student ();  // GC
    }
}
```

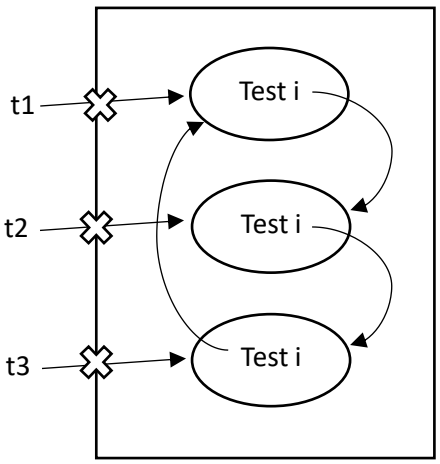
d) Island of Isolation

```
class Test {
    Test i;
    public static void main (String [] args) {
        Test t1 = new Test ();
        Test t2 = new Test ();
        Test t3 = new Test ();
        t1.i = t2;
        t2.i = t3;
        t3.i = t1;

        t1 = null;
        t2 = null;
        t3 = null;
    }
}
```

No object eligible for GC

3 objects are eligible for GC



Island of Isolation
(No external reference)

- Even though objects having references sometimes it's eligible for GC if all references are internal reference.

3. Methods for requesting JVM to run GC

- Once we made an object eligible for GC, it may not be destroyed immediately by Garbage collector.
- Whenever JVM runs GC only then the objects will be destroyed but when exactly JVM runs GC we can't expect. It is varied from JVM to JVM.
- Instead of waiting until JVM runs GC, we can request JVM to run GC programmatically but whether JVM accept our request or not there is no guarantee but most of the times JVM accepts our request.
- 2 ways for requesting JVM to run GC are:

a) By using System class

- System class contains a static method gc () for this purpose

```
System.gc ();
```

b) By using Runtime class

- Java application can communicate with JVM by using Runtime object.
- Runtime class present in java.lang package & it is a singleton class (only one object can be created)
- We can create Runtime object by using **Runtime.getRuntime()** method (Factory method)

```
Runtime r = Runtime.getRuntime ();
```

- Once we get Runtime object, we can call the following methods on that object:
  1) **totalMemory** () – returns no. of bytes of total memory present in the heap (i.e., Heap Size)
  2) **freeMemory** () – returns no. of bytes of free memory present in the heap.
  3) **gc** () – for requesting JVM to run garbage collector.

- **System.gc()** internally call **Runtime.getRuntime.gc().** So, performance wise Runtime gc() is more effective.
- gc () method present in **System class is a static method** whereas gc () method present in **Runtime class is instance method.**

```
import java.util.*;

class GarbageCollectorDemo {
   public static void main (String [] args) {
      Runtime r = Runtime.getRuntime();
      System.out.println(r.totalMemory());
      System.out.println(r.freeMemory());
      for (int i = 0; i < 10000; i++) {
         Date d = new Date ();
         d = null;
      }
      System.out.println(r.freeMemory());
      r.gc();
      System.out.println(r.freeMemory());
   }
}
```

Output:

```
Total Memory – 16252928
Free Memory – 15438272
Free Memory before GC – 15257360
Free Memory after GC – 15773600
```

4. Finalization

- Just before destroying an object, Garbage collector calls finalize () method to perform cleanup activities. Once finalize () method completes, automatically Garbage Collector destroys that object.
- finalize () method present in Object class with following declaration:

```
protected void finalize () throws Throwable
```

- We can override finalize () method in our class to define our own cleanup activities. [recommended]

Case 1: Just before destroying an object, Garbage collector calls **finalize** () method on the object which is eligible for GC then the corresponding classes finalize () method will be executed.

```
class FinalizationDemo {
   public static void main (String [] args) {
      String s = new String ("sam");
      s = null;
      System.gc();
      System.out.println("End of main");
   }

   public void finalize () {
      System.out.println("finalize method called!!");
   }
}
```

Output:
End of main

In the above example, String object eligible for GC & hence String class finalize () method got executed which has empty implementation. Hence the above output.

```
class Test {
   public static void main (String [] args) {
      Test s = new Test ();
      s = null;
      System.gc();
      System.out.println("End of main");
   }

   public void finalize () {
      System.out.println("finalize method called!!");
   }
}
```

Output:

End of main	Finalize method call
Finalize method call	End of main

In above example we have replaced String object with Test object, that's why Test class finalize () method will be executed.

- **System.gc**() starts a daemon thread. That's why the exact order of output can't be expected.

Case 2: Based on our requirement, we can call **finalize** () method explicitly. In that case, **finalize** () method will be executed just like a normal method call & object won't be destroyed. Only if GC calls finalize () method then object will be destroyed.

- If we call finalize () method & while executing the finalize () method if an exception occurs which is uncut then JVM will terminate our program abnormally.
- If GC calls finalize () method & while executing the finalize () method if an exception occurs which is uncut then JVM ignores that exception & rest of the program will continue normally.

Case 3: Even though object eligible for GC multiple times, but GC calls finalize () method only once.

Case 4: We can't expect exact behavior of Garbage collector. It is varied from JVM to JVM. Hence for the following questions we can't provide exact answers

1. When exactly JVM runs Garbage Collector?
2. In which order GC identifies eligible objects?
3. Whether GC destroys all eligible objects or not?
4. What is algorithm followed by GC? Etc.
5. Whenever program runs with low memory then JVM runs GC but we can't expect exactly at what time?

- Most of the GCs follow Standard Algorithm "Mark & Swipe" algorithm. It doesn't mean every GC follows the same algorithm.

Case 5: Memory leaks

- The objects which we're not using in our program & which are not eligible for GC. Such type of useless objects are called Memory leaks.
- In our program, if memory leaks present then the program will be terminated by raising OutofMemoryError.
- Hence, if an object no longer required, it is highly recommended to make that Object eligible for GC.
- The following are various 3rd party memory management tools to identify memory leaks
  - HP OVO, JMeter, JProbe, Patrol etc.