

Programming with Java for Beginners

Bineet Sharma

Input/Output & Collection

Input/Output & Collections Series

Assumptions & Expectations

Input/Output & Collection

- **Assumptions**
 - Advanced OOP Series
- **Expectations**
 - Understand I/O, Exceptions and Collections

Objectives

Input/Output & Collection

- **Exception Handling (advanced)**
- **Java Input/Output**
- **Collections & Generics**

Exception Handling:

Input/Output & Collection Series I

```
public class StudyException {
    public static void main(String[] args) {
        Scanner readInput = new Scanner(System.in);
        readInput.nextInt();
        readInput.nextInt();
    }
}
```

```
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2
```

Exception Handling:

Input/Output & Collection Series I

Exception in java is such condition like previous slide happening at run time. Java run time generates such exception when abnormal thing happens.

Java bundles the information regarding that exception in a class which are derived from **Throwable**, it then creates that object and **throws** to the offending methods which caused it.

Java allows ways to **catch** such exception object in your code to handle in graceful way or throw back to the method which called your method.

Even you can generate your own exception if you know a bad thing has happened.

Exception Handling:

Input/Output & Collection Series I

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Why Exception Handling? (take care of the un-handled situation described in previous code and much more)

Check out the method above (a pseudo code). Looks simple to implement, however, it ignores all these errors:

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

From Oracle Java Docs

Exception Handling:

Input/Output & Collection Series I

It will look something like this without exception handling

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
    }
    ...
} else {
    errorCode = -3;
}
    close the file;
    if (theFileDontClose && errorCode == 0) {
        errorCode = -4;
    } else {
        errorCode = errorCode and -4;
    }
} else {
    errorCode = -5;
}
    return errorCode;
}
```

From Oracle Java Docs

Exception Handling:

Input/Output & Collection Series I

It will look something like this with exception handling

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

From Oracle Java Docs

Exception Handling:

Input/Output & Collection Series I

All Java exceptions extends the class **Throwable**.
Error and Exception are two subclasses of Throwable.

Error: You rarely do anything with Error as they are internal Java resource errors i.e. stack overflow

Exception: You generally deal with Exception and its subclasses **RunTimeException** and **IOException**

```

◦ java.lang.Throwable (implements java.io.Serializable)
  ◦ java.lang.Error
    ◦ java.lang.annotation.AnnotationFormatError
    ◦ java.lang.AssertionError
    ◦ java.awt.AWTError
    ◦ java.nio.charset.CoderMalfunctionError
    ◦ javax.xml.parsers.FactoryConfigurationError
    ◦ javax.xml.stream.FactoryConfigurationError
    ◦ java.io.IOException
    ◦ java.lang.LinkageError
      ◦ java.lang.BootstrapMethodError
      ◦ java.lang.ClassCircularityError
      ◦ java.lang.ClassFormatError
  ◦ java.lang.Exception
    ◦ java.security.AccessControlException
    ◦ java.rmi.activation.ActivationException
      ◦ java.rmi.activation.UnknownObjectException
    ◦ java.rmi.activation.UnknownObjectException
    ◦ org.omg.CORBA.portable.ApplicationException
    ◦ java.rmi.RemoteException
    ◦ java.rmi.RemoteException
    ◦ java.management.BadAttributeValueTypeException
    ◦ java.management.BadBinaryOpValueException
    ◦ java.management.BadStringOperationException
    ◦ java.util.concurrent.BrokenBarrierException
    ◦ java.security.cert.CertificateException
  
```

Exception Handling:

Input/Output & Collection Series I

RunTimeException: Happens due to error in **your** code – array out of bounds, type mismatch etc.

IOException: Out of your code. Bad URL, or a File read

All exceptions derived from **Error** and **RunTimeException** are called unchecked exceptions. Java does not enforce it, it is up to you to deal with.

All other exceptions like **IOException** are called checked exceptions and Java enforces you that you handle them in some way

Exception Handling:

Input/Output & Collection Series I

```

readFile {
  try {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
  } catch (FileNotFoundException) {
    doSomething;
  } catch (SizeDeterminationFailed) {
    doSomething;
  } catch (MemoryAllocationFailed) {
    doSomething;
  } catch (ReadFailed) {
    doSomething;
  }
}
  
```

There are five keywords used in Java for exception:

try: block for offending code

catch: block to catch exception

throw: to throw exception manually. Java throws system generated exception automatically

throws: use to describe a method which throws an exception out

finally: block executed after try with or without catch

Exception Handling:

Input/Output & Collection Series I

Checked Exception: Strictly enforced by java. If a method throws checked exception then, you could:

1. Either catch the exception and handle in some way, or
2. You can simply declare it using **throws** and let it pass through your method, or
3. You can map it to your own exception class

Exception Handling:

Input/Output & Collection Series I

You saw examples already how to catch and handle exception. Let us look at examples on how to **throw** exception in your own code.

You can write methods/constructors which can handle **checked** exceptions (unlike unchecked exceptions: Error and RuntimeException)

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

```
{
    .....
}
```

```
void method1() {
    method2();
} catch (Exception e) {
}
```

```
void method2() throws Exception {
    method3();
}
```

```
void method3() throws Exception {
    method4();
}
```

```
void method4() throws Exception {
    throw new Exception();
}
```

Exception Handling:

Input/Output & Collection Series I

Example:

```
public static int getInput(Scanner in) throws NullPointerException {
    int userChoice = 0;
    if (in == null) {
        throw new NullPointerException("Null Scanner");
    }
    System.out.println("Please enter a value between 1 to 5");
    userChoice = Integer.parseInt(in.next());
    // all good
    return userChoice;
}

public static void main(String[] args) {
    int choice;
    Scanner inputReader = null; // new Scanner(System.in);
    try {
        choice = getInput(inputReader);
    } catch (NullPointerException e) {
        System.out.println("Message: " + e.getMessage());
        System.out.println("Stack Trace: " + e.getStackTrace());
    }
}
```

Exception Handling:

Input/Output & Collection Series I

```
public static int getInput(Scanner in) throws NullPointerException {
    int userChoice = 0;
    if (in == null) {
        throw new NullPointerException("Null Scanner");
    }
}
```

- You can throw only those exceptions which you have declared in the **'throws'** clause of method declaration or the subclass of that exception.

- If you can't find a suitable exception class in the library, you can create your own exception

Exception Handling:

Input/Output & Collection Series I

If you can't find a suitable exception class in the library, you can create your own exception

```
//Write your own
class MyOutOfRangeException extends Exception {
    MyOutOfRangeException(String message){
        super(message);
    }
}
```

Exception Handling:

Input/Output & Collection Series I

```
public static int getInput(Scanner in) throws
    NullPointerException, MyOutOfRangeException {
    int userChoice = 0;
    if (in == null) { throw new
        NullPointerException("Null Scanner");
    }
    System.out.println("Please enter a value between 1 to 5");
    userChoice = Integer.parseInt(in.next());
    if (userChoice < 1 || userChoice > 5) {
        throw new MyOutOfRangeException("Please
            enter value between 1 to 5");
    }
    return userChoice;
}
```

Exception Handling:

Input/Output & Collection Series I

```
try {
    userChoice = getInput(inputReader);
}
// order of catch is important
catch (NullPointerException e) {
    System.out.println("Message: " + e.getMessage());
    System.out.println("Stack Trace: " + e.getStackTrace());
} catch (MyOutOfRangeException e) {
    System.out.println("Message: " + e.getMessage());
}
catch (Exception e) {
    System.out.println("Catch all Exception");
}
finally { //always executed with our without catch
    if (userChoice == 0)
        System.out.println("Please try again ...");
    else
        System.out.println("Your choice is: " + userChoice);
}
```

Can you improve this? What is missing?

Please enter a value between 1 to 5: 9
Message: Only choice of 1 to 5 is allowed

Exception Handling:

Input/Output & Collection Series I

Java Exception Class Hierarchy: Find in latest javadocs.
For example: <http://docs.oracle.com/javase/7/docs/api/>

- java.lang.Throwable (implements java.io.Serializable)
 - java.lang.Error
 - java.util.ServiceConfigurationError
 - java.lang.Exception
 - java.io.IOException
 - java.util.InvalidPropertiesFormatException
 - java.lang.RuntimeException
 - java.util.ConcurrentModificationException
 - java.util.EmptyStackException
 - java.lang.IllegalArgumentException
 - java.util.IllegalFormatConversionException
 - java.util.DuplicateFormatFlagsException
 - java.util.FormatterFlagsConversionMismatchException
 - java.util.IllegalFormatCodePointException
 - java.util.IllegalFormatConversionException
 - java.util.IllegalFormatArgumentException
 - java.util.IllegalFormatRecursionException
 - java.util.IllegalFormatWidthException
 - java.util.MissingFormatArgumentException
 - java.util.MissingFormatWidthException
 - java.util.UnknownFormatConversionException
 - java.util.UnknownFormatFlagsException
 - java.lang.IllegalStateException
 - java.util.FormatterClosedException
 - java.util.IllegalAccessOperationException
 - java.util.MissingResourceException
 - java.util.NoSuchElementException
 - java.util.InputMismatchException
 - java.util.TimeoutException

Exception Handling: Input/Output & Collection Series I

• Demo

Exception Handling:

```
Please enter a value between 1 to 5: 9
Message: Only choice of 1 to 5 is allowed
Please try again ...
Please enter a value between 1 to 5: 4
Your choice is: 4
```

Input/Output (Advanced): Input/Output & Collection Series II

So far we have used **Scanner** for all of our input and output. We did not have to do much, other than importing `java.util.Scanner` class.

Scanner is flexible and very easy to use. It can read/write, not only from the **console** or keyboard, but from/to a **file** also.

Behind the scene, a lot is done for us by the API: `java.lang` package. It contains three predefined *public static stream* (available every part a program) variables, **in**, **out**, and **err**.

System.out, and **System.in** refers to standard output (screen/console,) and standard input (keyboard).

```
java.util.ResourceBundle.Control
java.util.Scanner (implements java.io.Closeable, java.util.Iterator<E>)
java.util.ServiceLoader<C> (implements java.lang.Iterable<C>)
java.util.StringTokenizer (implements java.util.Enumeration<E>)
java.util.Throwable (implements java.io.Serializable)
```

File Class: import it from `java.io`

Use this to work with the disk file in operating system. It has many useful methods which will allow you to work on a file.

- Find out if it exists using `exists` method
- Find out if you can read or write using `canRead`, `canWrite`
- Delete the file using `delete`
- Find the size using `length`
- Find it's full path with `getPath`

You typically use **File** class in conjunction with other classes

```
File myFile = new File(fileName);
if (myFile.exists()){
    File name is: input_final.txt
    File is: C:\Users\Space\Java\060\Projects\Java063\input_final.txt
    System.out.println("File name is: " + myFile.getName());
    System.out.println("File is: " + myFile.getAbsolutePath());
}
```

Input/Output (Advanced): Input/Output & Collection Series II

You can use **File** class with **Scanner** just like you used **System.in**

```
Scanner readInput;
try {readInput = new Scanner(new File(fileName));
    while (readInput.hasNextLine())
        {System.out.println(readInput.nextLine());}
}
//start with most specific to most general exception
catch (FileNotFoundException e) {
    System.out.println("File: " + fileName + " not found");
} // end catch
catch (IOException e) {
    System.out.println("Error Reading from file: " + fileName + e.getMessage());
} // end catch
catch (Exception e) {
    System.out.println(e);
} // end catch
```

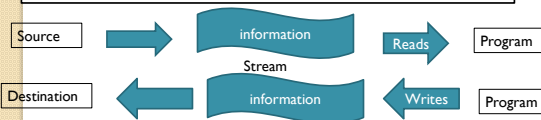
```
John Doe1: 1, 100, 90, 90, 100, 99, 99
John Doe2: 90, 90, 100, 100, 99, 100
John Doe3: 100, 90, 100, 70, 70, 70
John Doe4: 00, 70, 90, 100, 00, 99
John Doe5: 70, 90, 100, 100, 60, 100
John Doe6: 00, 90, 90, 80, 70, 01
```

Input/Output (Advanced):

Input/Output & Collection Series II

There are many choices for I/O in Java.
Java and most other I/O are stream-based. A **stream** is really a connection to a *source* of information (ordered sequence of byte) or to a *destination* of information.

An input stream may be associated with a keyboard or a disk file, and a output stream may be associated with a screen or a disk file



Input/Output (Advanced):

Input/Output & Collection Series II

Stream: In java is an object, which either delivers data to its destination, e.g. screen, disk file or other output medium, or it reads the data from a source, e.g. keyboard, disk file, or other output. It is sort of a buffer between your program and the devices and connects your program to the outside world (devices)

Input Stream: Your program reads information from a input stream, e.g. *System.in* – connects your program with keyboard

Output Stream: Your program writes information into a output stream, eg. *System.out* – connects your program with screen.

Input/Output (Advanced):

Input/Output & Collection Series II

A program can have **multiple streams**: console, disk files, keyboard, sockets, memory, strings

What do you do with stream? You first **open** the stream, use the stream to read data sequentially, write or both, and then finally **close** them.

There are two types of streams in Java:

Byte streams to read/write binary data

Character streams to read/write Unicode characters

Input/Output (Advanced):

Input/Output & Collection Series II

Java I/O Class Hierarchy:

```

java.lang.Object
├── java.io.Console (implements java.io.Flushable)
├── java.io.File (implements java.lang.Comparable<T>, java.io.Serializable)
├── java.io.FilterDescriptor
├── java.io.InputStream (implements java.io.Closeable)
├── java.io.FileInputStream
├── java.io.FilterInputStream
├── java.io.DataInputStream (implements java.io.DataInput)
├── java.io.LittleEndianInputStream
├── java.io.PushbackInputStream
├── java.io.ObjectInputStream (implements java.io.ObjectInput)
├── java.io.SequenceInputStream
├── java.io.StringBufferInputStream
├── java.io.ObjectStreamField
├── java.io.ObjectStreamClass (implements java.io.Serializable)
├── java.io.ObjectStreamField (implements java.io.Serializable)
├── 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.io.Appendable, java.io.Closeable)
├── java.io.ObjectOutputStream (implements java.io.ObjectOutput, java.io.Serializable)
├── java.io.PipedOutputStream
├── java.security.Permission (implements java.security.Guard, java.io.Serializable)
├── java.security.BasicPermission (implements java.io.Serializable)
├── java.io.SerializablePermission
├── java.io.FilePermission (implements java.io.Serializable)
├── java.io.RandomAccessFile (implements java.io.Closeable, java.io.DataInput, java.io.DataOutput)
├── java.io.Reader (implements java.io.Closeable, java.lang.Readable)
├── java.io.BufferedReader
├── java.io.LineNumberReader
├── java.io.CharArrayReader
├── java.io.FilterReader
├── java.io.PushbackReader
├── java.io.InputStreamReader
├── java.io.FileReader
├── java.io.PipedReader
├── java.io.StringReader
  
```

Input/Output (Advanced): Input/Output & Collection Series II

Byte stream: There are two abstract classes on top: **InputStream** and **OutputStream**. There are many concrete classes which implements the behavior to read and write.

System.in, System.out, and System.err are predefined byte stream.

Character stream: There are two abstract classes on top: as well, **Reader** and **Writer**. There are many concrete classes which implement the behavior to read and write.

Input/Output (Advanced): Input/Output & Collection Series II

Byte Stream Example

```
//How big is the file
InputStream inStream;

inStream = new FileInputStream(args[0]);

int total = 0;
while ((inStream.read() != -1))
    total++;

System.out.println(total + "bytes");
inStream.close();
```

- java.io.InputStream (implements java.io.Closeable)
- java.io.ByteArrayInputStream
- java.io.FilterInputStream
- java.io.FilterOutputStream
- java.io.DataInputStream (implements java.io.DataInput)
- java.io.DataOutputStream (implements java.io.DataOutput)
- java.io.PushbackInputStream
- java.io.ObjectInputStream (implements java.io.ObjectInput, java.io.ObjectStreamConstants)
- java.io.ObjectOutputStream
- java.io.SequenceInputStream
- java.io.StringBufferInputStream

Input/Output (Advanced): Input/Output & Collection Series II

Text File I/O using Character Stream:

Output: Use **PrintWriter**, **FileWriter** (FileOutputStream)

Input: Use **BufferedReader**, **FileReader**

You will use sets two of classes for easier reading and writing

- java.io.Writer (implements java.lang.Appendable, java.io.Closeable, java.io.Flushable)
 - java.io.BufferedWriter
 - java.io.CharArrayWriter
 - java.io.FilterWriter
 - java.io.OutputStreamWriter
 - java.io.FileWriter
 - java.io.PipedWriter
 - java.io.PrintWriter
 - java.io.StringWriter
- java.io.Reader (implements java.io.Closeable, java.lang.Readable)
 - java.io.BufferedReader
 - java.io.LineNumberReader
 - java.io.CharArrayReader
 - java.io.FilterReader
 - java.io.PushbackReader
 - java.io.InputStreamReader
 - java.io.FileReader
 - java.io.PipedReader
 - java.io.StringReader

Input/Output (Advanced): Input/Output & Collection Series II

Buffered Reading/Writing: In most system including Java most of the time, the input/output streams are buffered before it is physically written into the disk file.

By doing this OS conserves the overhead, as accessing disk is inefficient compare with memory (RAM) access.

Stream Names: Java needs to work with the OS while dealing with streams. OS has different view of steam and Java has different. Java API connects these two world together.

Every file has two names: **input_final.txt** is what OS uses, while **InputStream** is what Java uses (meaning your program)

Input/Output (Advanced)

Input/Output & Collection Series II

Reading from Input Stream: Using FileReader

- Use stream which can be used for Input e.g. **FileReader**

```
FileReader fileReader = new FileReader("filename");
```

- "filename" name of OS file (store in your hard drive)
- going forward your code will only use **fileReader**
- **FileReader** provides a way to read an integer at a time, and return
- If there are no more bytes left to read

```
int readInt = fileReader.readInt();
```

```
FileReader fileReader = new FileReader("input_final.txt");
int readInt;
while ((readInt = fileReader.readInt()) != -1)
    System.out.println("Byte: " + readInt + "\tChar: " + (char)readInt);
```

Input/Output (Advanced)

Input/Output & Collection Series II

Reading from Input Stream: Using BufferedReader

Use `.readLine` to read a line into a String (.read for single char)

No methods to read numbers, so, use **Tokenizer** to parse String

```
try {
    FileReader fileReader = new FileReader("input_final.txt");
    BufferedReader finalInStream = new
    BufferedReader(fileReader);
    String s; //readLine - read a line into a string
    while ((s=finalInStream.readLine()) != null) { //end of file returns null
        System.out.println(s);
    }
    finalInStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Input/Output (Advanced):

Input/Output & Collection Series II

Reading from console (System.in): Console provides byte of stream so, reading from Console i.e. `System.in` requires you to read characters. (one at a time).

This will change the byte stream to character stream and read each individual character

```
InputStreamReader cReader = new InputStreamReader(System.in)
```

You need a **BufferedReader** to read like tokens

```
BufferedReader tokenReader = new BufferedReader(cReader)
```

Read a line (token): there are methods for word and char

```
String getLine = tokenReader.readLine();
```

You can parse numbers from these using parse methods of of each type, i.e. `int myInt = Integer.parseInt(getLine);`

Writing to Console (System.out): Good thing is `System.out` already defined to print numbers and strings.

Input/Output (Advanced):

Input/Output & Collection Series II

Tokenizer: Most of the time, reading a line is not what you want, however, you want to extract individual elements from it. For that Java provides **tokenizer** concept.

Use StreamTokenizer to read from a stream and **StringTokenizer** to extract the tokens from a String in your program.

Steps in using StreamTokenizer: You need to first create a tokenizer and connect with **BufferedReader**. Then loop through to get first token, translate the token to your appropriate data and use it, and continue until there is a token.

Input/Output (Advanced):

Input/Output & Collection Series II

How Tokenizer Works? When you call the tokenizer method *nextToken* it returns a flag about the next token:

TT_EOF: indicates that next token is end of file

TT_EOL: indicates that next token is end of line.

TT_WORD: indicates next token is a word

TT_NUMBER: indicates next token is a number

Input/Output (Advanced):

Input/Output & Collection Series II

Using StreamTokenizer:

```
StreamTokenizer myTokenizer = new StreamTokenizer(bufferedReader);
//start getting next token, next token here is type
nextToken = myTokenizer.nextToken();
```

```
while (nextToken != StreamTokenizer.TT_EOF) {
    if (nextToken != StreamTokenizer.TT_EOL && nextToken ==
        StreamTokenizer.TT_WORD){
        strToken = myTokenizer.sval;
        System.out.println("Found a string: " + strToken);
    }
    if (nextToken != StreamTokenizer.TT_EOL &&
        nextToken == StreamTokenizer.TT_NUMBER){
        numberToken = myTokenizer.nval;
        System.out.println("Found a number: " + numberToken);
    }
    nextToken = myTokenizer.nextToken(); //eat up TT_EOL
} // while
```

Input/Output (Advanced):

Input/Output & Collection Series II

Using *StringTokenizer*: *StringTokenizer* allows you to parse a string in different tokens. It has easy way to specify delimiters – compare with *StreamTokenizer*.

```
String getLine = "This is, \n a string. with four delimiters";
//create a tokenizer with multiple delimiters
StringTokenizer parseWords = new StringTokenizer(getLine, " \n.,");
while(parseWords.hasMoreTokens())
{
    System.out.println(parseWords.nextToken());
}
```

This
is
a
string
with
four
delimiters

Input/Output (Advanced):

Input/Output & Collection Series II

StringTokenizer can be used with console as well **System.in**

```
Scanner keyboard = new Scanner(System.in);
String getUserInput = keyboard.nextLine();
//create a tokenizer with multiple delimiters
StringTokenizer parseUserInput =
    new StringTokenizer(getUserInput, " \n.,");

while(parseUserInput.hasMoreTokens())
{
    System.out.println(parseUserInput.nextToken());
}
```

Input/Output (Advanced):

Input/Output & Collection Series II

Output Using Stream: Similar to Input Stream.
Here also we have two names: *outputStream* in the program and the physical file name used by OS: *output-final.txt*

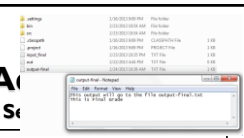
You typically connect a text file to a stream for writing: (there are many ways, however, this will work):

```
FileOutputStream txtStream = new FileOutputStream("f.txt")
//to append use : new FileOutputStream("f.txt",true)
PrintWriter textPrintStream = new PrintWriter(txtStream);
```

Use println, print, format, flush, close of PrintWriter

Input/Output (Advanced):

Input/Output & Collection Series II



```
PrintWriter textPrintStream = null;
String outfileName = "output-final.txt";
try {
    textPrintStream = new PrintWriter(new
        FileOutputStream(outfileName));
    textPrintStream.println("This output will go to the file "
        + outfileName);
    textPrintStream.write("This is Final Grade \n");
} catch (FileNotFoundException e) {
    System.out.println("Error opening the file " + outfileName + "\n"
        + e.getMessage());
}
System.exit(0);
}
textPrintStream.close();
System.out.println(outfileName + " been written and closed");
```

Input/Output (Advanced):

Input/Output & Collection Series II

Putting it all together: It is kind of confusing with all the choices available, what to use. Here are some tips:

Choice A: StringTokenizer + BufferedReader with FileReader
Choice B: Scanner + File

Input/Output (Advanced):

Input/Output & Collection Series II

	Scanner	Buffered Reader
Needs	Scanner+File	BufferedReader+FileReader +Tokenizer
Primitive	nextInt() etc.	read(), readLine() no parsing
EOF	check hasNext	readLine() returns null, read() - I Exception

Since you are familiar with Scanner, you may find it much easier to work with.

Input/Output (Advanced):

Input/Output & Collection Series II

Whats next in File Handling?

Dealing with other files, like:
 Stream sent through internet (sockets)
 Encrypted files
 Compressed files

Input/Output Advanced:

Input/Output & Collection Series II

• Demo

Input/Output (Advanced):

Collections:

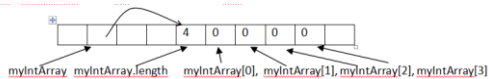
Input/Output & Collection Series III

Java Data Structures: Java provides systematic way of organizing collections of data.

Array: An array is a collection of objects in Java. Array name is reference to the actual object itself.

Array of primitive data types:

`int [] myIntArray = new int[4];` // is array of 4 integer

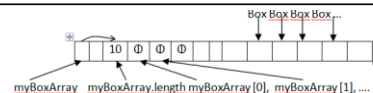


Collections:

Input/Output & Collection Series III

Array of objects:

```
Box [] myBoxArray = new Box[10];
// is array of 10 Box objects – Java objects
myBoxArray[0] = new Box(2, 3);
myBoxArray[1] = new Box(5, 10);
myBoxArray[1] = new Box("I am fancy");
myBoxArray[1] = new Box(20, 30, "I am fat and fancy");
...
```



Collections:

Input/Output & Collection Series III

Java provides an **Array** class which provides static methods to manipulate the array-for both primitive and ref.

- sort(): for sorting
- binarySearch(): for efficient searching

Java Arrays:

- are type safe
- simple implementation
- easy to manipulate
- good to store collection of primitive and references

However, they are not most efficient:

- Size is constant
- Inserting and deleting elements are costly

Collections:

Input/Output & Collection Series III

Java provides even richer set of data structures to store **collection** of primitive data and references.

For example **ArrayList** class: A program written to use ArrayList collection would look like this:

```
ArrayList listOfValues = new ArrayList();
listOfValues.add("John");
listOfValues.add("Jack");
listOfValues.add("Jill");
System.out.println(listOfValues);
System.out.println("3: " + listOfValues.get(2));
```

[John, Jack, Jill]

It looks like array, however, it is much more flexible – as you notice the **size grows dynamically**.

Collections:

Input/Output & Collection Series III

“A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).” – Oracle Documentation

Collections:

Input/Output & Collection Series III

Java provides a unified architecture for representing and manipulating collections through *collections framework*, which contains the following:

- **Interfaces:** Abstract data types that represent collections.
- **Implementations:** Concrete implementations of the collection interfaces.
- **Algorithms:** Methods that perform useful computation like; searching and sorting which work on all implementations of collection interface in a polymorphic way.

Collections:

Input/Output & Collection Series III

Benefits of Java Collection Framework:

- Reduces your programming effort
- Increases speed and quality of your program
- Allows interoperability among unrelated APIs
- Shorter learning curve
- Shorter design time for new APIs
- Fosters OOP by software reuse:

Collections:

Input/Output & Collection Series III

Java Collection Interfaces: Primary means by which collections are manipulated

Collection: Just group of objects without any assumptions made about the order of the collection, or whether duplicates are allowed or not.

Set: No duplicate elements are permitted and may not be ordered

List: Ordered collection, duplicates are permitted

Map: Key value pair. Each key can only map to one value, no ordering

SortedSet: Elements are automatically sorted, either in their natural ordering or by a Comparator object

SortedMap: Mappings are automatically sorted by key, either in their natural ordering or by a Comparator object



Collections:

Input/Output & Collection Series III

Methods Implemented with Collection Interface acts on *individual elements*:

```

public void clear()
public int hashCode()
public Iterator iterator();
public Object[] toArray()
public int size()
public boolean isEmpty()
public boolean contains(Object o)
public boolean add(Object o)
public boolean remove(Object elem)
public boolean equals(Object o)

```

Collections:

Input/Output & Collection Series III

Methods Implemented with Collection Interface acts on *bulk of elements*:

```

public boolean containsAll(Collection coll)
public boolean addAll(Collection coll)
public boolean removeAll(Collection coll)
public boolean retainAll(Collection coll)
//Removes from the collection all
//elements that are not elements of coll
public void clear() //Remove all elements from this
//collection

```



Collections:

Input/Output & Collection Series III

Java collection framework provides many Implementation (concrete) Classes, for example:

ArrayList: Resizable-array implementation of the list interface.

LinkedList: Doubly-linked list implementation of the list interface. Better if frequent insertion and deletion is needed.

HashSet: Hash table implementation of the Set Interface.

TreeSet: Tree implementation of SortedSet Interface.

HashMap: Hash map implementation of the Map interface.

TreeMap: Tree implementation of SortedMap Interface



Collections:

Input/Output & Collection Series III

Using Collections: **List**

List is an **ordered Collection** which **allows duplicate** elements. Just like array, element index starts at 0.

List interface adds several methods for an ordered collection

Implementation:

ArrayList: a resizable-array implementation. Simple to use.

Good for frequent scanning, but, not for frequent add/delete

LinkedList: Uses a doubly-linked list for storage. Not good for frequent scanning, however, good for frequent add/delete

```
[John, Jack, Jill]
3: Jill
[John, Jack, Jill, NewBox@5c9aa764, NewBox@2d63c5bb, NewBox@714a8f44]
```

Collections:

Input/Output & Collection Series III

ArrayList: Example:

```
ArrayList listOfValues = new ArrayList();
//or List listOfValues = new ArrayList();
listOfValues.add("John");
listOfValues.add("Jack");
listOfValues.add("Jill");
System.out.println(listOfValues);

System.out.println("3: " + listOfValues.get(2));

listOfValues.add(new NewBox(10));
listOfValues.add(new NewBox(20));
listOfValues.add(new NewBox(30));
System.out.println(listOfValues);
```

```
[Jack, John, Jill, Kerry, Sarah]
```

Collections:

Input/Output & Collection Series III

LinkedList: Example:

```
LinkedList linkedListValues = new
    LinkedList();

linkedListValues.addFirst("John");
linkedListValues.addLast("Jill");
linkedListValues.addFirst("Jack");
linkedListValues.add("Kerry");
linkedListValues.addLast("Sarah");

System.out.println(linkedListValues)
```

LinkedList: is a doubly linked list so, you have more methods available as:
getFirst(), getLast(), removeFirst(), removeLast()

You can implement Queue, or Deque deriving from LinkedList

Collections:

Input/Output & Collection Series III

Set: The Set interface extends the Collection interface.

- does not allow duplicates (contains no new methods)
- Two Set objects are equal, if they contain same elements.
- Null is a valid entry (only one null entry is allowed)
- Objects added to Set, must have **equals()** defined

SortedSet: Extends Set. Elements are ordered in a specific order. *Natural order* implemented by *Comparable* interface. Change ordering by using a *Comparator* object.

Two general purpose implementation:

HashSet: Stores its elements in a hash table and is fast.

TreeSet: Ordered set uses tree for storage. It allows elements to be added, or removed at any location by following an order

Collections:

Input/Output & Collection Series III

TreeSet: Implements Set, provides an ordered set (uses tree for storage). Add/remove from any location, ordering is preserved.

```
int size();
boolean add(Object obj);
boolean remove(Object obj);
boolean contains(Object obj);
Iterator iterator();
Object[] toArray();
```

TreeSet allows you to defined your own sorting through an **Comparator** object passed during creation.

Collections:

Input/Output & Collection Series III

HashSet: Use it for duplicate free set. The objects stored in this set should implement hashCode() method-one is provided by Object may not be optimal. Objects are not physically sorted.

Hash Function: Provide unique integers for each object. Each hash integer must map to the same object, and if two objects are equal (using **equals** method) then they must return same integer.

Collections:

Input/Output & Collection Series III

Implement **Comparable** Interface: If you want to provide a *natural ordering* for your object. You must implement **compareTo(Object o)** method. This method should return a +ve, zero, -ve number if this object is less, equal or greater than the Object passed.

```
class NewBox implements Comparable{
    public int compareTo(Object o) { //compares area
        int area1 = ((NewBox)o).height * ((NewBox)o).width;
        int area2 = height * width;

        if ( area1 < area2 ) return 1;
        else if ( area1 > area2 ) return -1;
        else return 0;
    }
}
```


Collections:

Input/Output & Collection Series I

Where is **compareTo** used? However, you may need better sorting, like alphabetical, then implement **Comparator**

```
NewBox box1 = new NewBox("New Box 3", 10, 20);
NewBox box2 = new NewBox("New Box 4", 20, 28);
NewBox box3 = new NewBox("New Box 2", 20, 20);
NewBox box4 = new NewBox("New Box 1", 25, 20);
```

```
NewBox[] lotsOfBoxes = new NewBox[] {box1, box2, box3, box4};
for (int i = 0; i < lotsOfBoxes.length; i++)
    System.out.println(lotsOfBoxes[i].getBoxName());
```

```
Arrays.sort(lotsOfBoxes); //sort naturally - provided by the object
for (int i = 0; i < lotsOfBoxes.length; i++)
    System.out.println(lotsOfBoxes[i].getBoxName() + " Area: " +
        lotsOfBoxes[i].getHeight() * lotsOfBoxes[i].getWidth());
```

```
New Box 3
New Box 4
New Box 2
New Box 1
New Box 3 Area: 200
New Box 2 Area: 400
New Box 1 Area: 500
New Box 4 Area: 560
```

Collections:

Input/Output & Collection Series III

Using **Comparator** Interface: Implement this interface if you want to provide your own comparison.

Comparator requires you to implement method: **compare()** and optionally **equals()**

```
class CompareBoxNames implements Comparator {
    public int compare(Object s1, Object s2) { //required
        String str1 = ((NewBox)s1).getBoxName();
        String str2 = ((NewBox)s2).getBoxName();
        return (str1.compareTo(str2));
    }
    public boolean equals(Object s1, Object s2) { //optional
        String str1 = ((NewBox)s1).getBoxName();
        String str2 = ((NewBox)s2).getBoxName();
        return (str1.equalsIgnoreCase(str2));
    }
}
```

Collections:

Input/Output & Collection Series III

Where will it be used?

```
NewBox box1 = new NewBox("New Box 3", 10, 20);
NewBox box2 = new NewBox("New Box 4", 20, 28);
NewBox box3 = new NewBox("New Box 2", 20, 20);
NewBox box4 = new NewBox("New Box 1", 25, 20);
```

```
NewBox[] lotsOfBoxes = new NewBox[] {box1, box2, box3, box4};
for (int i = 0; i < lotsOfBoxes.length; i++)
    System.out.println(lotsOfBoxes[i].getBoxName());
```

```
Arrays.sort(lotsOfBoxes, new CompareBoxNames());
for (int i = 0; i < lotsOfBoxes.length; i++)
    System.out.println(lotsOfBoxes[i].getBoxName());
```

```
New Box 3
New Box 4
New Box 2
New Box 1
New Box 1
New Box 2
New Box 3
New Box 4
```

Collections:

Input/Output & Collection Series III

This can easily be used by other **Collection** objects as well:

```
//Use with ArrayList
NewBox box1 = new NewBox("New Box 3", 10, 20);
NewBox box2 = new NewBox("New Box 4", 20, 28);
NewBox box3 = new NewBox("New Box 2", 20, 20);
NewBox box4 = new NewBox("New Box 1", 25, 20);
```

```
ArrayList lotsOfBoxes = new ArrayList();
lotsOfBoxes.add(box1);
lotsOfBoxes.add(box2);
lotsOfBoxes.add(box3);
lotsOfBoxes.add(box4);
System.out.println("In the order of creation:\n");
for (int i = 0; i < lotsOfBoxes.size(); i++) {
    NewBox nBox = (NewBox)lotsOfBoxes.get(i);
    System.out.println(nBox.getBoxName());
}
```

Collections:

Input/Output & Collection Series I

Extend to another collection:

```
//Continued from previous slide
//sort naturally - provided by the object
System.out.println("\nIn the order of natural sorting:\n");
Collections.sort(lotsOfBoxes);
for (int i = 0; i < lotsOfBoxes.size(); i++) {
    NewBox nBox = (NewBox)lotsOfBoxes.get(i);
    System.out.println(nBox.getBoxName() + " Area: " +
        nBox.getHeight() * nBox.getWidth());
}

System.out.println("\nIn the order of Comparator sorting Implementation:\n");
Collections.sort(lotsOfBoxes, new CompareBoxNames());
for (int i = 0; i < lotsOfBoxes.size(); i++) {
    NewBox nBox = (NewBox)lotsOfBoxes.get(i);
    System.out.println(nBox.getBoxName() + " Area: " +
        nBox.getHeight() * nBox.getWidth());
}
```

In the order of creation:
 New Box 3
 New Box 4
 New Box 2
 New Box 1
 New Box 3 Area: 200
 New Box 2 Area: 400
 New Box 1 Area: 500
 New Box 4 Area: 500
 In the order of natural sorting:
 New Box 1 Area: 500
 New Box 2 Area: 400
 New Box 3 Area: 200
 New Box 4 Area: 500
 In the order of Comparator sorting Implementation:
 New Box 1 Area: 500
 New Box 2 Area: 400
 New Box 3 Area: 200
 New Box 4 Area: 500

Collections:

Input/Output & Collection Series III

Map Interface: Maps a key to the elements – instead of index
HashMap: Implements Map interface, uses hash to get unique key value.

SortedMap interface extends Map and maintains its keys in sorted order.

TreeMap: Implements SortedMap, uses tree for storage and traversing efficiently.

Basic operations:

```
Object put(Object key, Object value)
Object get(Object key);
Object remove(Object key)
int size();
```

Collections:

Input/Output & Collection Series III

TreeMap: TreeMap does not contain an iterator method. However, it contains `Set keySet()` method which is used to get the set of keys and then iterate through it. No ordering of the values, the tree is arranged according to the order of keys.

TreeSet: TreeSet is more specific than TreeMap. TreeSet values are compared with each other, so, make sure to only put those which can be compared. Guaranteed to keep the elements in ascending order or through **Comparator**.

Collections:

Input/Output & Collection Series III

Iterator: Collection interface defines an **iterator** method which returns an object implementing the **Iterator** interface

Iterator is used to access elements of a collection, without exposing internal details. Order is not guaranteed

Iterator Interface: Defines these methods

```
public boolean hasNext()
public Object next()
public void remove()
```

Collections:

Input/Output & Collection Series III

Using **Iterator**: You can use iterator with any collection class which implements Iterator interface

```
System.out.println("\nUsing Iterator:\n");
//using
Iterator it = lotsOfBoxes.iterator();
while(it.hasNext()) {
    NewBox box = (NewBox)(it.next());
    System.out.println(box.getBoxName());
}
```

Using Iterator:

```
New Box 1
New Box 2
New Box 3
New Box 4
```

Collections:

Input/Output & Collection Series III

- **Demo**

Collection:

Generics:

Input/Output & Collection Series IV

Enhanced **for** Loop: New syntax is more compact, and it is easier to use with collection:

for (variable-type variable-name: range-of-values)

```
//enhanced for loop
int [] weeklyTemp = {69, 70, 71, 68, 66, 71, 70};
```

//instead of

```
for (int day=0; day < weeklyTemp.length; day++)
    System.out.printf("The temperature on day %d was %d\n", day+1,
        weeklyTemp[day]);
```

System.out.println();

//use this

```
int day=0;
```

for (int dayTemp : weeklyTemp)

```
    System.out.printf("The temperature on day %d was %d\n", ++day, dayTemp);
System.out.println();
```

Generics:

Input/Output & Collection Series IV

Introducing Generics: Generic allows generalized Types. *Generics* abstract over Types and provides readability and type safety during compile time

You can use generics with Methods, Classes and Interfaces as well

Use the "<>" characters to designate the type to be used

Generics:

Input/Output & Collection Series IV

Generics Rationale: Suppose you needed to write a method to find out if an array contains a certain value (integer for example), then you would write like this:

```
public static boolean contains(Integer [] array, Integer intObject {
    for (Integer value : array) {
        if (intObject.equals(value))
            return true;
    }
    return false; //did not find it
}
```

Generics:

Input/Output & Collection Series IV

Generics Rationale: You will write same function again for other objects, for example String

```
public static boolean contains(String [] array, String strObject {
    for (String value : array) {
        if (strObject.equals(value))
            return true;
    }
    return false; //did not find it
}
```

Not very modular, is it?

Generics:

Input/Output & Collection Series IV

Introducing Generics: Now with generics, you need to write only one method, use an abstract type T:

```
public static <T> boolean contains(T[] array, T anyObject) {
    for (T value : array) {
        if (anyObject.equals(value))
            return true;
    }
    return false;
}
```

Generics:

Input/Output & Collection Series IV

Introducing Generics: Now you can test it with different objects.

```
Integer[] array = new Integer[5];
for (int j = 0; j < 5; j++) {
    array[j] = j * j;
}
if (contains(array, new Integer(16))) {
    System.out.println("Found the value");
} else {
    System.out.println("Value not found");
}
```

Generics:

Input/Output & Collection Series IV

Introducing Generics: Now you can test it with different objects.

```
String[] strArray = new String[5];
String strTemp;
for (int i = 0; i < 5; i++) {
    strTemp = String.format("This is string %d", i%4);
    strArray[i] = strTemp;
}
if (contains(strArray, new String("This is string 18"))) {
    System.out.println("Found the value");
} else {
    System.out.println("Value not found");
}
```

Generics:

Input/Output & Collection Series IV

Generics Types: All collection classes are re-written to accommodate Generics

```
interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
interface Iterator<E> {
    E next();
    boolean hasNext();
}
interface Map<K,V> {
    V put(K key,V Value);
}
```

Generics:

Input/Output & Collection Series IV

Generics: Allows compile time type safety. Here is ArrayList example. This is true for rest of the collections as well.

Before:

```
ArrayList lotsOfBoxes = new ArrayList();
lotsOfBoxes.add(box1); //add more boxes....
for (int i = 0; i < lotsOfBoxes.size(); i++) {
    NewBox nBox = (NewBox)lotsOfBoxes.get(i);
    System.out.println(nBox.getBoxName());
}
```

After:

```
ArrayList<NewBox> lotsOfBoxes = new ArrayList<NewBox>();
//add more boxes ..
for (int i = 0; i < lotsOfBoxes.size(); i++) {
    NewBox nBox = lotsOfBoxes.get(i); //no casting
    System.out.println(nBox.getBoxName());
}
```

Generics:

Input/Output & Collection Series IV

More on Generics:

Subtyping of Generic Types
Wildcards
Bounded Type Variables

Read Java Docs on Generics



Generics:

Input/Output & Collection Series IV

- **Demo**

Generics:



Summary:

Input/Output & Collection

- **Exception Handling (advanced)**
- **Java Input/Output**
- **Collections & Generics**