Programming with Java for Beginners
Bineet Sharma

Input/Output & Collection

Input/Oput & 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
asfasf
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at utility.StudyException.main(StudyException.java:12)
```

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 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 (theFileDidntClose && 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.**IOError**
    - java.lang.**LinkageError**
      - java.lang.**BootstrapMethodError**
      - java.lang.**ClassCircularityError**
      - java.lang.**ClassFormatError**

- java.lang.**Exception**
  - java.security.acl.**AclNotFoundException**
  - java.rmi.activation.**ActivationException**
    - java.rmi.activation.**UnknownGroupException**
    - java.rmi.activation.**UnknownObjectException**
  - java.rmi.**AlreadyBoundException**
  - org.omg.CORBA.portable.**ApplicationException**
  - java.awt.**AWTException**
  - java.util.prefs.**BackingStoreException**
  - javax.management.**BadAttributeValueExpException**
  - javax.management.**BadBinaryOpValueExpException**
  - javax.swing.text.**BadLocationException**
  - javax.management.**BadStringOperationException**
  - java.util.concurrent.**BrokenBarrierException**
  - javax.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 (fileOpenFailed) {
      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 our without catch

Exception Handling:
Input/Output & Collection Series I

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

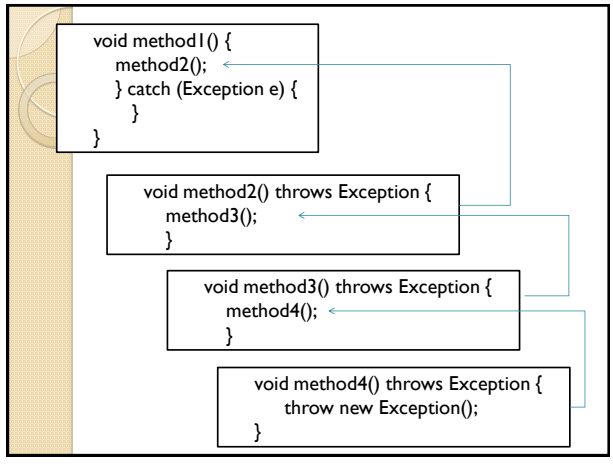
1. Either catch the exception and handle in some way, or
2. You can simply declare it using **throws** and let it pass through your method, or
3. You can map it to your own exception class

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*
{
…..
}



Message: Null Scanner
Stack Trace: [Ljava.lang.StackTraceElement;@321ea24

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;
}
```

```
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

```
try {
            userChoice = getInput(inputReader);
}
// order of catch is important
catch (NullPointerException e) {
            System.out.println("Message: " + e.getMessage());
            System.out.println("StackTrace:" + e.getStackTrace());
} catch (MyOutOfRangeException e) {
            System.out.println("Message: " + e.getMessage());
}

catch (Exception e) {
            System.out.println("Catch all Exception");
}
finally { //always exectued 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?

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/

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).

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()){
    System.out.println("File name is: "+ myFile.getName());
    System.out.println("File  is: " + myFile.getAbsolutePath());
}
```

```
File name is: input_final.txt
File  is: C:\Work_Space\Java_060\Projects\Java063\input_final.txt
```

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

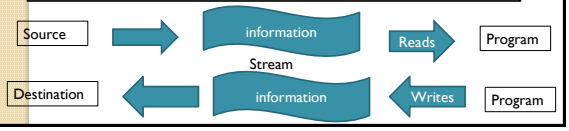
You can use *File* class with *Scanner* just like you used *System.in*

```
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
```

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:

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

- 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.LineNumberInputStream
    - java.io.PushbackInputStream
  - java.io.ObjectInputStream (implements java.io.ObjectInput, java.io.ObjectStreamConstants)
  - java.io.PipedInputStream
  - java.io.SequenceInputStream
  - java.io.StringBufferInputStream

```
//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();
```

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

Text File I/O using Character Stream:

Output: Use **PrintWriter, FileWriter** (FileOuputStream)
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.

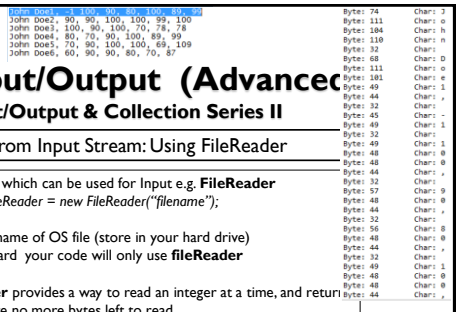
By doing this OS conserves the overhead, as accessing disk is inefficient compare with memory (RAM) access.

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 retur
-1 if there are no more bytes left to read

fint readInt = ileReader.read();

```
FileReader fileReader = new FileReader("input_final.txt");
int readInt;
while ((readInt = fileReader.read()) != -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();
}
```

```
John Doe1, 100, 90, 80, 100, 89, 99
John Doe2, 90, 90, 100, 100, 99, 100
John Doe3, 100, 90, 100, 70, 78, 78
John Doe4, 80, 70, 90, 100, 89, 99
John Doe5, 70, 90, 100, 100, 69, 109
John Doe6, 60, 90, 90, 80, 70, 87
```

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 and read each individual character

InputStreamReader cReader = new InputStreamReader(System.in)

You need a BufferedReader to read like tokens

BufferedReader tokenReader= new BufferedReacer(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 toke in 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, nexttoken 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 delimeters";
//create a tokenizer with multiple delemeters
StringTokenizer parseWords = new StringTokenizer(getLine, " \n.,");
while(parseWords.hasMoreTokens())
{
   System.out.println(parseWords.nextToken());
}
```

```
This
is
a
string
with
four
delimeters
```

Input/Output (Ac
Input/Output & Collection Se

```
I can, type as much, and it will take as . tokens
I
can
type
as
much
and
it
will
take
as
tokens
```

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 delemeters
StringTokenizer parseUserInput =
          new StringTokenizer(getUserInput, " \n.,");

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

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

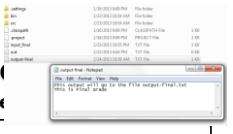
Output Using Stream: Similar to Input Stream.
Here also we have two names: *outputStream* in the program and the physical file name used by OS: *output-final.txt*

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 (A
Input/Output & Collection Se



```
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() -1*
	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:

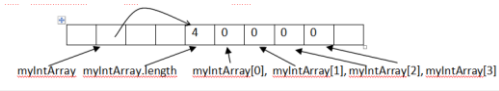
Input/Output & Collection Series III

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

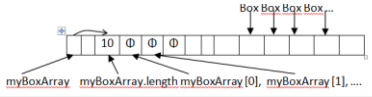


Collections:

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 ArralyList 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.

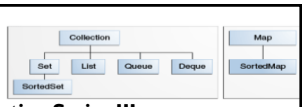
Collections:

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 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();

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

System.out.println(linkedListValues
```

LinkedList: is a doubly lnked 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

```
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
```

Where is ***compareTo*** used? However, you may need better sorting, like alphabetical, then implenent **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 natually - 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());
```

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

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

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());
```

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 S

Extend to another collection:

```
//Continued from previous slide
    //sort natually - 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

In the order of natural sorting:

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

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: 560
```

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);
- Objrect 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 he 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 **Interator** 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*i);
        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**