Java is Platform Independent

By Dr. Subrat Kumar Nayak

Associate Professor,

Department of CSE, FET, S'O'A (Deemed to be) University.

General Procedure

- What is a Platform?

Platform is combination of processor and OS(operating system). In general we can say the hardware or software component in which programs run.

- How program get executed?

When you write program in C/C++ and when you compile it, it is directly converted into machine readable language(.exe).

- Two types of codes get generated after compilation need to be focused.

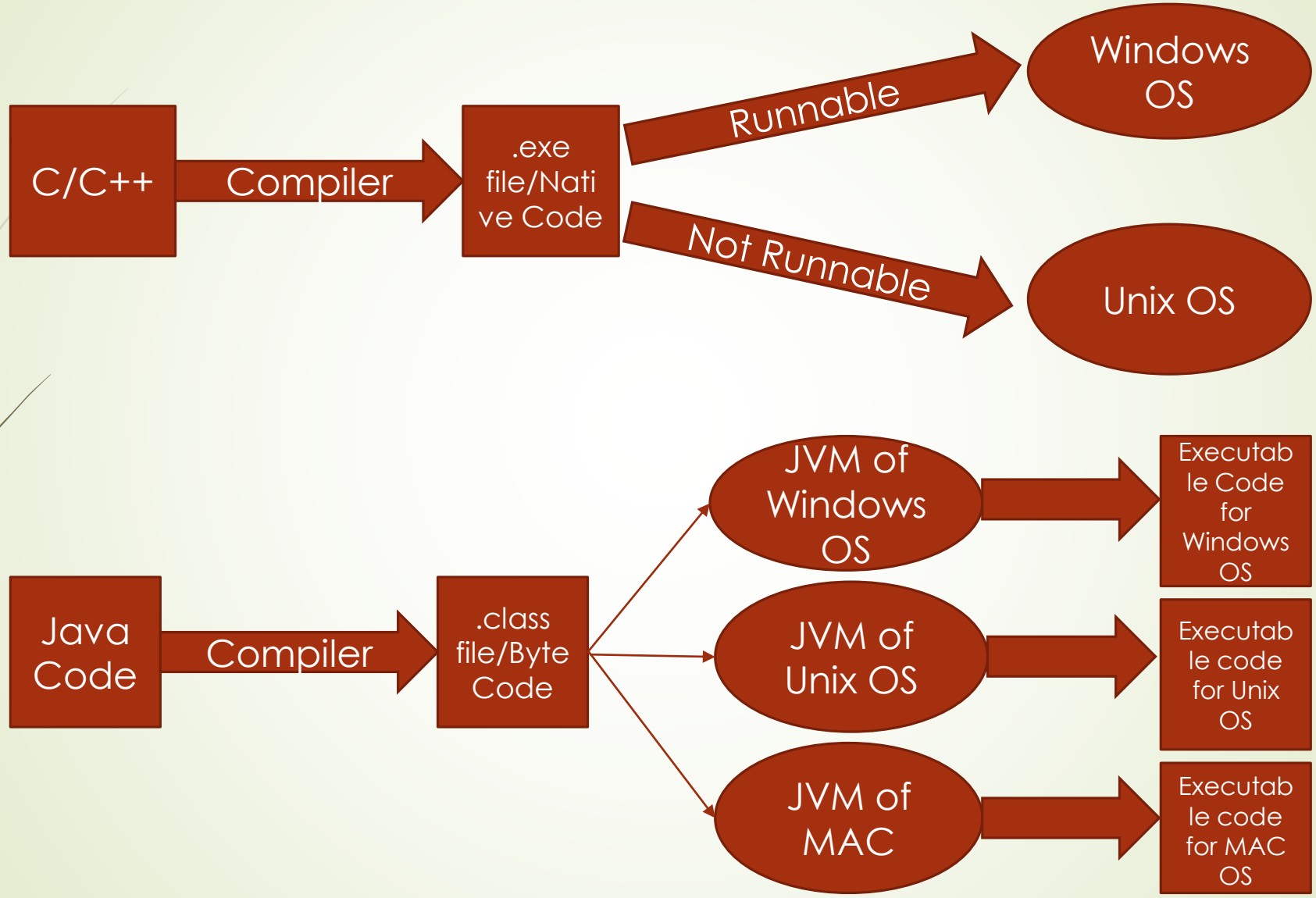
1. Native code

   Native code is similar to machine code i.e codes that is understood by machine. Native codes are specific to platform i.e, Native code generated by program for Windows OS is different from Native code generated for the same program for Unix OS.

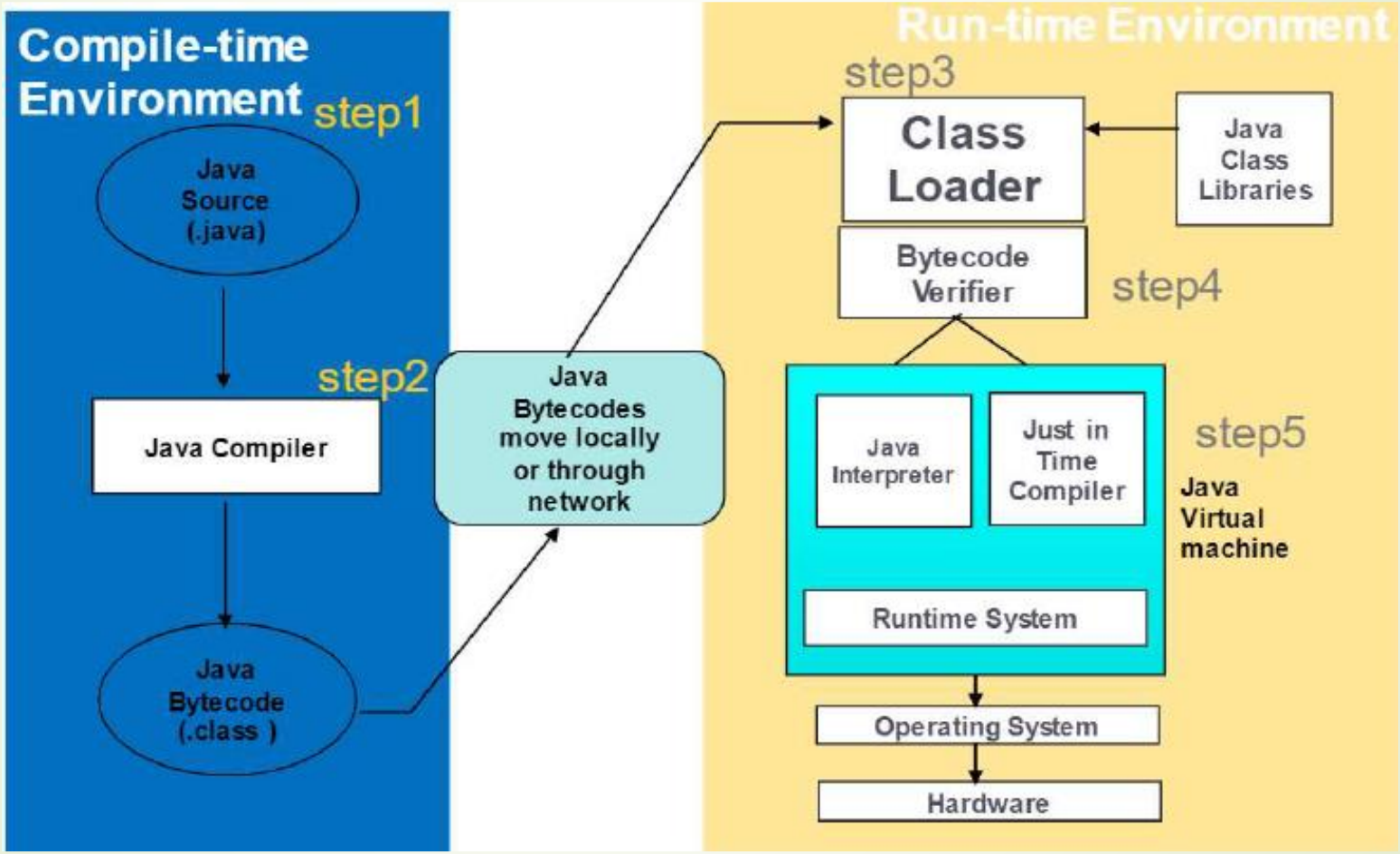
1. Byte code

   Byte codes are nothing but intermediate codes generated after compilation and it is not the executable code like Native code. The Byte code requires a virtual machine to execute in machine. Byte codes generated by one platform can be executed in another platform also.

- Java compiler converts the source code to .class file (byte code).
- Whether JVM is platform independent?


C/C++
Compiler
.exe file/Native Code
Runnable
Windows OS
Not Runnable
Unix OS
Java Code
Compiler
.class file/Byte Code
JVM of Windows OS
JVM of Unix OS
JVM of MAC
Executable Code for Windows OS
Executable code for Unix OS
Executable code for MAC OS

Java Architecture



Java main()

- `public static void main(String[] args)` is the most important java method.

`public`: This is access modifier of `main()`. It has to be public so that java runtime can access this method.

`static`: `main()` function need to be static so that JVM can load the class to memory and call the main method.

`void`: Java `main()` does not return anything.

`String[] args`: Java `main()` accepts a single argument of type String array. This is also called as java command line arguments.

End of Class

Interacting with the Environment, using java.lang.System class

By Dr. Subrat Kumar Nayak

Associate Professor,

Department of CSE, FET, S'O'A (Deemed to be) University.

Introduction

- This chapter describes how your Java program can deal with its immediate surroundings, with what we call the runtime environment.
- In other sense, everything we do in a Java program using almost any Java API involves the environment.
- Here, the focus more narrowly on things that directly surround your program.
- This also talks about the **System** class that will help us to know a lot about the system.
- Many operating systems use *environment variables* to pass configuration information to applications. Like properties in the Java platform, environment variables are **key/value pairs**, where both the key and the value are strings.
- The conventions for setting and using environment variables vary between operating systems, and also between command line interpreters.

Getting Environment Variables

- Here the intention is to get the value of "environment variables" from within your Java program.
- Environment variables are commonly used for customizing an individual computer user's runtime environment.
- How to read environment variable using Java API?
- System class provides two methods
- ✓ `System.getenv(String name)` - which returns specific variable value
- ✓ `System.getenv()` which returns all environment variables values.

Getting Information from System Properties

- Get information from the system properties.
- A property is just a name and value pair stored in a java.util.Properties object.
- In Properties, we examined **the way an application can use Properties objects to maintain its configuration**. The Java platform itself uses a Properties object to maintain its own configuration. The **System** class maintains a **Properties object** that describes the **configuration of the current working environment**.
- System properties include **information about the current user, the current version of the Java runtime, and the character used to separate components of a file path name.**
- The following table describes some of the most important system properties

Key	Meaning
`"file.separator"`	Character that separates components of a file path. This is "/" on UNIX and "\" on Windows.
`"java.class.path"`	Path used to find directories and JAR archives containing class files. Elements of the class path are separated by a platform-specific character specified in the **path.separator** property.
`"java.home"`	Installation directory for Java Runtime Environment (JRE)
`"java.vendor"`	JRE vendor name

Key	Meaning
"java.vendor.url"	JRE vendor URL
"java.version"	JRE version number
"line.separator"	Sequence used by operating system to separate lines in text files
"os.arch"	Operating system architecture
"os.name"	Operating system name
"os.version"	Operating system version
"path.separator"	Path separator character used in java.class.path
"user.dir"	User working directory
"user.home"	User home directory
"user.name"	User account name

Continue…(Reading System Properties)

- The **System** class has two methods used to read system properties:

  `getProperty` and `getProperties`

- To retrieve one system-provided property, use `System.getProperty()`. If you want them all, use `System.getProperties()`.
- Example: `System.getProperty("path.separator");`

Learning About the Current JDK Release

- You need to write code that looks at the current JDK release
- Use `System.getProperty()` with an argument of `java.specification.version`.
- Solution:
  `System.out.println(System.getProperty("java.specification.version"))`

Dealing with Operating System–Dependent Variations

- To write code that adapts to the underlying operating system.
- Though Java is designed to be portable, some things aren't.
- Such as file separetor (/) for unix and (\) for dos or windows.
- You can use System.Properties to find out the operating system, and various features in the File class to find out some platform-dependent features.
- Example:

  use System . getProperty (" file . separator ")

  or

  use String ret = java .io. File . separator ;

End of Chapter

String and things

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

String and things

- In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

  char[] ch={'I','T','E','R'};

  String s=**new** String(ch);

- The above one can also be written as follows.

  String s="ITER";

- Java **String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
- The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.
- The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.
- The Java String is **immutable** which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

String ?

- Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to Create a String object?

- There are two ways to create String object:
  - By string literal
  - By new keyword

By String literal

- Java String **literal** is created by using double quotes. For Example:

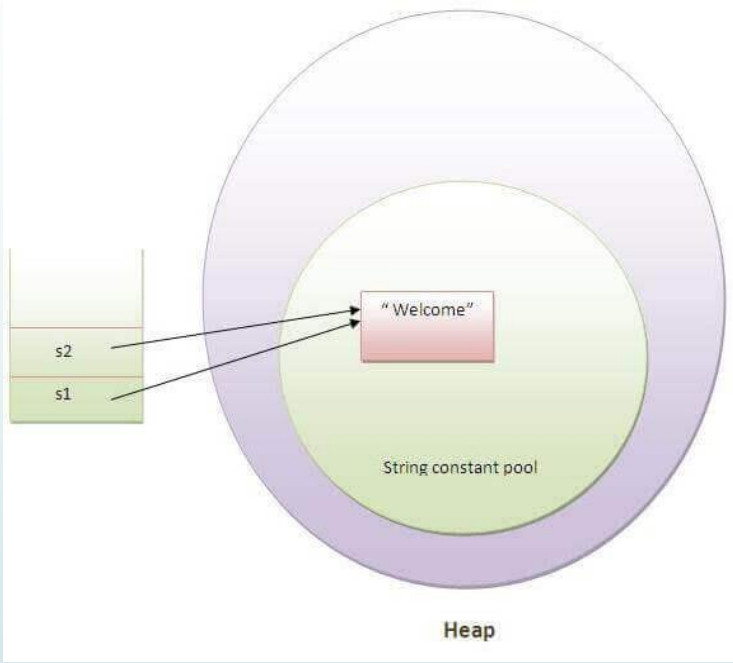
```
String s="welcome";
```

- Each time you create a string literal, the JVM checks the **"string constant pool"** first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

String…

- In the previous example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.



Why Java uses the concept of String literal?

- To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

By new keyword

- String s=**new** String("Welcome");//creates two objects and one reference variable
- In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

String...

```
public class StringExample{
public static void main(String args[]){
    String s1="java";//creating string by java string literal
    char ch[]={'s','t','r','i','n','g','s'};
    String s2=new String(ch);//converting char array to string
    String s3=new String("example");//creating java string by new keyword
    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s3);
}
}
```

Output:

```
java
strings
example
```

String is Immutable

- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.
- Let's try to understand the immutability concept by the example given below:

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Siksha";
   s.concat(" O Anusandhan");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects

 }
}
```

Output: Siksha

String is Immutable

- Here Siksha is not changed but a new object is created with Siksha O Anusndhan. That is why string is known as immutable.
- But if we explicitely assign it to the reference variable, it will refer to " Siksha o Anusandhan" object.For example:

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Siksha";
   s=s.concat(" O Anusandhan");
   System.out.println(s);
 }
}
```

```
Output: Siksha O Anusandhan
```

- In such case, s points to the "Siksha o Anusandhan". Please notice that still Siksha object is not modified.

End of Session

String and things

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

String and things

Taking Strings Apart with Substrings

- Problem: You want to break a string apart into substrings by position.
- Solution: Use the String object's **substring()** method.

substring()

- A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

Syntax 1:

```
public String substring(int begIndex)
```

Parameters :

begIndex : the begin index, inclusive.

Return Value : The specified substring.

This method returns new String object containing the substring of the given string from specified startIndex (inclusive).

String and things

Syntax 2:

public String substring(int begIndex, int endIndex)

Parameters :

beginIndex : the begin index, inclusive.

endIndex : the end index, exclusive.

Return Value : The specified substring.

This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

Q: Write a program to find sub string of a string from a start index (eg. 5).

Q: Write a program to find sub string of a string from a start index to end index (eg. 2,5).

Q: Write a program that read two strings from user and concat the sub string from 0 to 5 index of first string with the 0 to 7 index of the second string and print the resultant string.

String and things

indexof()

- The **java string indexOf()** method returns index of given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

Syntax for Signature:

There are 4 types of Signatures.

- int indexOf(int ch)

returns **index position** for the given char value

- int indexOf(int ch, int fromIndex)

returns **index position** for the given char value and from **index**

- int indexOf(String substring)

returns **index position** for the given **substring**

- int indexOf(String substring, int fromIndex)

returns **index position** for the given **substring** and **from index**

String and things

```
public ublic class IndexOfExample{
public static void main(String args[]){
    String s1="this is index of example";
    //passing substring
    int index1=s1.indexOf("is");//returns the index of is substring
    int index2=s1.indexOf("index");//returns the index of index substring
    System.out.println(index1+"  "+index2);//2 8
    //passing substring with from index
    int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index
    System.out.println(index3);//5 i.e. the index of another is
    //passing char value
    int index4=s1.indexOf('s');//returns the index of s char value
    System.out.println(index4);//3
}}
```

Output: 2 8

5

3

String and things

lastIndexOf()

The **java string lastIndexOf()** method returns last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

Syntax for Signature

- 4 types

(1) int lastIndexOf(int ch)

returns last index position for the given char value

(2) int lastIndexOf(int ch, int fromIndex)

returns last index position for the given char value and from index

(3) int lastIndexOf(String substring)

returns last index position for the given substring

(4) int lastIndexOf(String substring, int fromIndex)

returns last index position for the given substring and from index

String and things

```
public class LastIndexOfExample{
  public static void main(String args[]){
    String s1="this is index of example";//there are 2 's' characters in this sentence

    int index1=s1.lastIndexOf('s');//returns last index of 's' char value
    System.out.println(index1);//6
}}
```

Output: 6

```
public class LastIndexOfExample2 {
    public static void main(String[] args) {
        String str = "This is index of example";
        int index = str.lastIndexOf('s',5);
        System.out.println(index);
    }
}
```

Output: 3

String and things

```
public class LastIndexOfExample3 {
    public static void main(String[] args) {
        String str = "This is last index of example";
        int index = str.lastIndexOf("of");
        System.out.println(index);
    }
}
```

Output:19

```
public class LastIndexOfExample4 {
    public static void main(String[] args) {
        String str = "This is last index of example";
        int index = str.lastIndexOf("of", 25);
        System.out.println(index);
        index = str.lastIndexOf("of", 10);
        System.out.println(index); // -1, if not found
    }
}
```

Output: 19

-1

End of Session

String and things

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

String and things

length()

The **java string length()** method length of the string. It returns count of total number of characters.

Signature:

- `public int length()`

Example:

```
public class LengthExample{
public static void main(String args[]){
String s1=“java";
String s2="python";
System.out.println("string length is: "+s1.length());//4 is the length of java s
tring
System.out.println("string length is: "+s2.length());//6 is the length of python
 string
}}
```

Output:

```
string length is: 4
string length is: 6
```

String and things

equals()

- The **java string equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.
- The String equals() method overrides the equals() method of Object class.

Signature:

- **public boolean** equals(Object anotherObject)

Example:

```
public class EqualsExample{
public static void main(String args[]){
        String s1="java";
        String s2="java";
        String s3="JAVA";
        String s4="python";
        System.out.println(s1.equals(s2));//true because content and case is same
        System.out.println(s1.equals(s3));//false because case is not same
        System.out.println(s1.equals(s4));//false because content is not same
}}
```

String and things

Q write a program to differentiate between == and equals() method in java.

compareTo()

- The **java string compareTo()** method compares the given string with current string lexicographically. It returns positive number, negative number or 0.
- It compares strings on the basis of Unicode value of each character in the strings.
- If first string is lexicographically greater than second string, it returns positive number (difference of character value). If first string is less than second string lexicographically, it returns negative number and if first string is lexicographically equal to second string, it returns 0.
- **if** s1 > s2, it returns positive number
- **if** s1 < s2, it returns negative number
- **if** s1 == s2, it returns 0

Signature:

- **public int** compareTo(String anotherString)

Q: Write a program to demonstrate the functionality of compareTo() function.

String and things

trim()

- The **java string trim()** method eliminates leading and trailing spaces. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.
- The string trim() method doesn't omits middle spaces

Signature

- **public** String trim()

Example:

```
public class StringTrimExample{
public static void main(String args[]){
    String s1="  hello string   ";
    System.out.println(s1+"java");//without trim()
    System.out.println(s1.trim()+"java");//with trim()
}}
```

Output:

```
   hello string    java
hello stringjava
```

End of Session

String and things

(Breaking Strings into Words)

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Breaking Strings into Word

Problem: You need to take a string apart into words or tokens.

Solutions: To accomplish this, construct a StringTokenizer around your string and call its methods hasMoreTokens() and nextToken(). Another way is to use the String `split()` function for this purpose.

split()

- The java string split() method splits this string against given regular expression and returns an array of string.

Signature

- There are two types of signature for this function

(1) Public String[] split(String regex)

(2) Public String[] split(String regex, int limit)

- Regex: regular expression to be applied on string.
- Limit: limit for the number of stings in the array. If it is zero, it will return all the strings matching regex.

Breaking Strings into Word

Example:

```
public class SplitExample{
public static void main(String args[]){
    String s1="java string split method by ITER";
    String[] words=s1.split("\\s");//splits the string based on whitespace
    //using java foreach loop to print elements of string array
    for(String w:words){
        System.out.println(w);
    }
}}
```

Output:

java
string
split
method
by
ITER

Breaking Strings into Word

```
public class SplitExample2{
    public static void main(String args[]){
        String s1="welcome to split world";
        System.out.println("returning words:");
        for(String w:s1.split("\\s",0)){
            System.out.println(w);
        }
        System.out.println("returning words:");
        for(String w:s1.split("\\s",1)){
            System.out.println(w);
        }
        System.out.println("returning words:");
        for(String w:s1.split("\\s",2)){
            System.out.println(w);
        }
}}
```

Output:
returning words:
welcome
to
split
world
returning words:
welcome to split world
returning words:
welcome
to split world

Breaking Strings into Word

StringTokenizer in Java

- The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

- **StringTokenizer(String str)**

It creates StringTokenizer with specified string. Considers default delimeters like new line, tab and space etc.

- **StringTokenizer(String str, String delim)**

It creates StringTokenizer with specified string and delimeter.

- **StringTokenizer(String str, String delim, boolean returnValue)**

It creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Breaking Strings into Word

Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

- boolean hasMoreTokens()

It checks if there is more tokens available.

- String nextToken()

It returns the next token from the StringTokenizer object.

- String nextToken(String delim)

It returns the next token based on the delimeter.

- boolean hasMoreElements()

It same as hasMoreTokens() method.

- Object nextElement()

It same as nextToken() but its return type is Object

- int countTokens()

returns the total number of tokens.

Breaking Strings into Word

```
import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        System.out.println("Using Constructor 1 - ");
        StringTokenizer st1 =
             new StringTokenizer("Hello Subrat How are you", " ");
        while (st1.hasMoreTokens())
            System.out.println(st1.nextToken());

        System.out.println("Using Constructor 2 - ");
        StringTokenizer st2 =
             new StringTokenizer("JAVA : Code : String", " :");
        while (st2.hasMoreTokens())
            System.out.println(st2.nextToken());

        System.out.println("Using Constructor 3 - ");
        StringTokenizer st3 =
             new StringTokenizer("JAVA : Code : String", " :",  true);
        while (st3.hasMoreTokens())
            System.out.println(st3.nextToken());
    }
}
```

Output:

```
Using Constructor 1 –
Hello
Subrat
How
are
you
Using Constructor 2 –
JAVA
Code
String
Using Constructor 3 –
JAVA

:

Code

:

String
```

End of Session

String and things

(StringBuffer & StringBuilder)

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Java StringBuffer class

- Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.
- Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Constructors:

- StringBuffer()

creates an empty string buffer with the initial capacity of 16

- StringBuffer(String str)

creates a string buffer with the specified string

- StringBuffer(int capacity)

creates an empty string buffer with the specified capacity as length

Java StringBuffer class

Some Important Functions:

append(String s)

- This is used to append the specified string with this string.
- The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

Signature:

- public StringBuffer append(String s)

Example:

```
class StringBufferExample{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello ");
    sb.append("Java");//now original string is changed
    System.out.println(sb);
}}
output:
Hello Java
```

Java StringBuffer class

insert(int offset, String s)

- This is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

Signature:

- public StringBuffer insert(int offset, String s)

Example:

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);
}}
```

Output:

```
HJavaello
```

Java StringBuffer class

replace()

- This is used to replace the string from specified startIndex and endIndex.

Signature:

- public StringBuffer replace(int startIndex, int endIndex, String str)

Example:

```
class StringBufferExample3{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello");
    sb.replace(1,3,"Java");
    System.out.println(sb);
}}
```

Output:

```
HJavalo
```

Java StringBuffer class

delete()

- This is used to delete the string from specified startIndex and endIndex.

Signature:

- public StringBuffer delete(int startIndex, int endIndex)

Example:

```
class StringBufferExample4{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello");
    sb.delete(1,3);
    System.out.println(sb);
}}
```

Output:

```
Hlo
```

Java StringBuffer class

reverse()

- This is used to reverse the string.

Signature

- public StringBuffer reverse()

Example

```
class StringBufferExample5{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello");
    sb.reverse();
    System.out.println(sb);
}}
```

Output:

```
olleH
```

Java StringBuffer class

capacity()

- The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.
- **Signature:**

public int capacity()

Example:

```
class StringBufferExample6{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer();
    System.out.println(sb.capacity());//default 16
    sb.append("Hello");
    System.out.println(sb.capacity());//now 16
    sb.append("java is my favourite language");
    System.out.println(sb.capacity());
    //now (16*2)+2=34 i.e (oldcapacity*2)+2
}}
```

Java StringBuffer class

Some more functions

- public char charAt(int index)
- public int length()
- public String substring(int beginIndex)
- public String substring(int beginIndex, int endIndex)

Java StringBuffer class

String Vs StringBuffer (Programming Example)

```
public class ConcatTest{
   public static String concatWithString()    {
      String t = "ITER";
      for (int i=0; i<10000; i++){
         t = t + "SOADU";
      }
      return t;
   }
   public static String concatWithStringBuffer(){
      StringBuffer sb = new StringBuffer("ITER");
      for (int i=0; i<10000; i++){
         sb.append("SOADU");
      }
      return sb.toString();
   }
```

```
public static void main(String[] args){
      long startTime = System.currentTimeMillis();
      concatWithString();

      System.out.println("Time taken by Concating with
String: "+(System.currentTimeMillis()-startTime)+"ms");

      startTime = System.currentTimeMillis();

      concatWithStringBuffer();

      System.out.println("Time taken by Concating with
 StringBuffer: "+(System.currentTimeMillis()-
startTime)+"ms");
   }
}
```

Java StringBuilder class

- Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.

Constructors:

- StringBuilder()

creates an empty string Builder with the initial capacity of 16

- StringBuilder(String str)

creates a string Builder with the specified string

- StringBuilder(int length)

StringBuilder vs String

String	StringBuilder
• String class is immutable.	• StringBuilder class is mutable
• String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	• StringBuilder is fast and consumes less memory when you cancat strings
• String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method	• StringBuilder class doesn't override the equals() method of Object class

End of Session

Exception Handling

By Dr. Subrat Kumar Nayak
Associate Professor
Department of CSE
ITER, SOADU

Exception Handling in Java

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is Exception in Java?

- Exception is an abnormal condition.
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

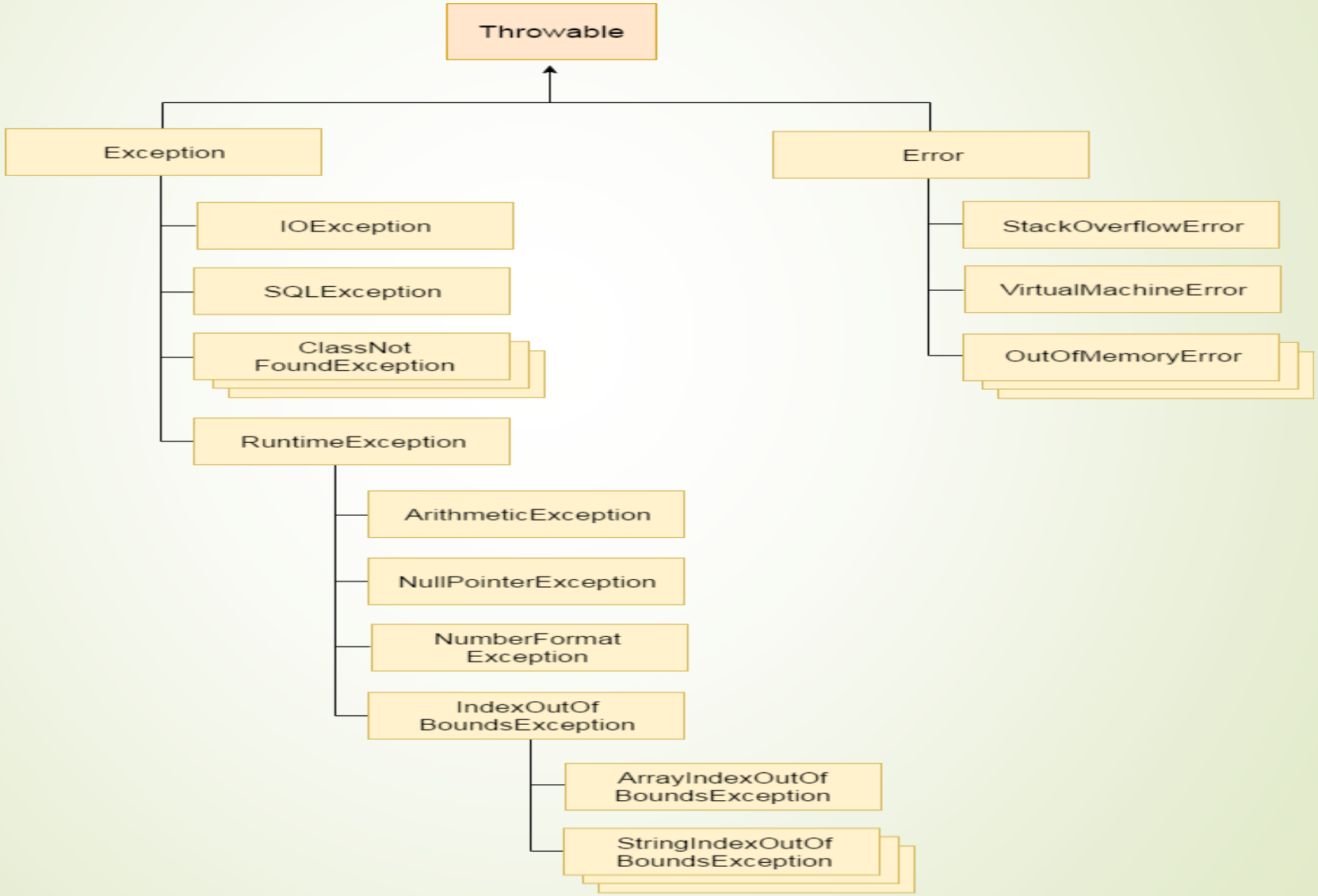
What is Exception Handling?

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling.

Hierarchy of Java Exception classes



Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception

- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc.
- Checked exceptions are checked at **compile-time**.

Unchecked Exception

- The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- Unchecked exceptions are not checked at compile-time, but they are checked at **runtime.**

Error

- Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Exception Handling in Java

Example: (Java Exception Handling using a try-catch statement)

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Common Scenarios of Java Exceptions

- A scenario where ArithmeticException occurs

```
int a=50/0;//ArithmeticException
```

- A scenario where NullPointerException occurs

```
String s=null;
System.out.println(s.length());//NullPointerException
```

- A scenario where NumberFormatException occurs

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

- A scenario where ArrayIndexOutOfBoundsException occurs

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Exception Handling in Java

Java try-catch block

try block

- Java **try** block is used to enclose the code that might throw an exception.
- If an exception occurs at the particular statement of try block, the rest of the block code will not execute.
- Java try block must be followed by either catch or finally block.

Syntax:

- **try-catch**

```
try{
    //code that may throw an exception
}catch(Exception_class_Name ref){}
```

- **try-finally**

```
try{
    //code that may throw an exception
}finally{}
```

Exception Handling in Java

catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception (i.e., **Exception**) or the **generated exception** type.
- The catch block must be used after the try block only. You can use **multiple catch block** with a single try block.

Problem without exception handling

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

} }
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero

Exception Handling in Java

Solution by exception handling

```
public class TryCatchExample2 {
    public static void main(String[] args) {
        try
        {
              int data=50/0; //may throw exception
        }
        catch(ArithmeticException e)
        {
           System.out.println(e);  / System.out.println("Can't divide by zero");
        }
        System.out.println("rest of the code");
} }
```

Output: java.lang.ArithmeticException: / by zero **Or** Can't divide by zero

rest of the code rest of the code

Exception Handling in Java

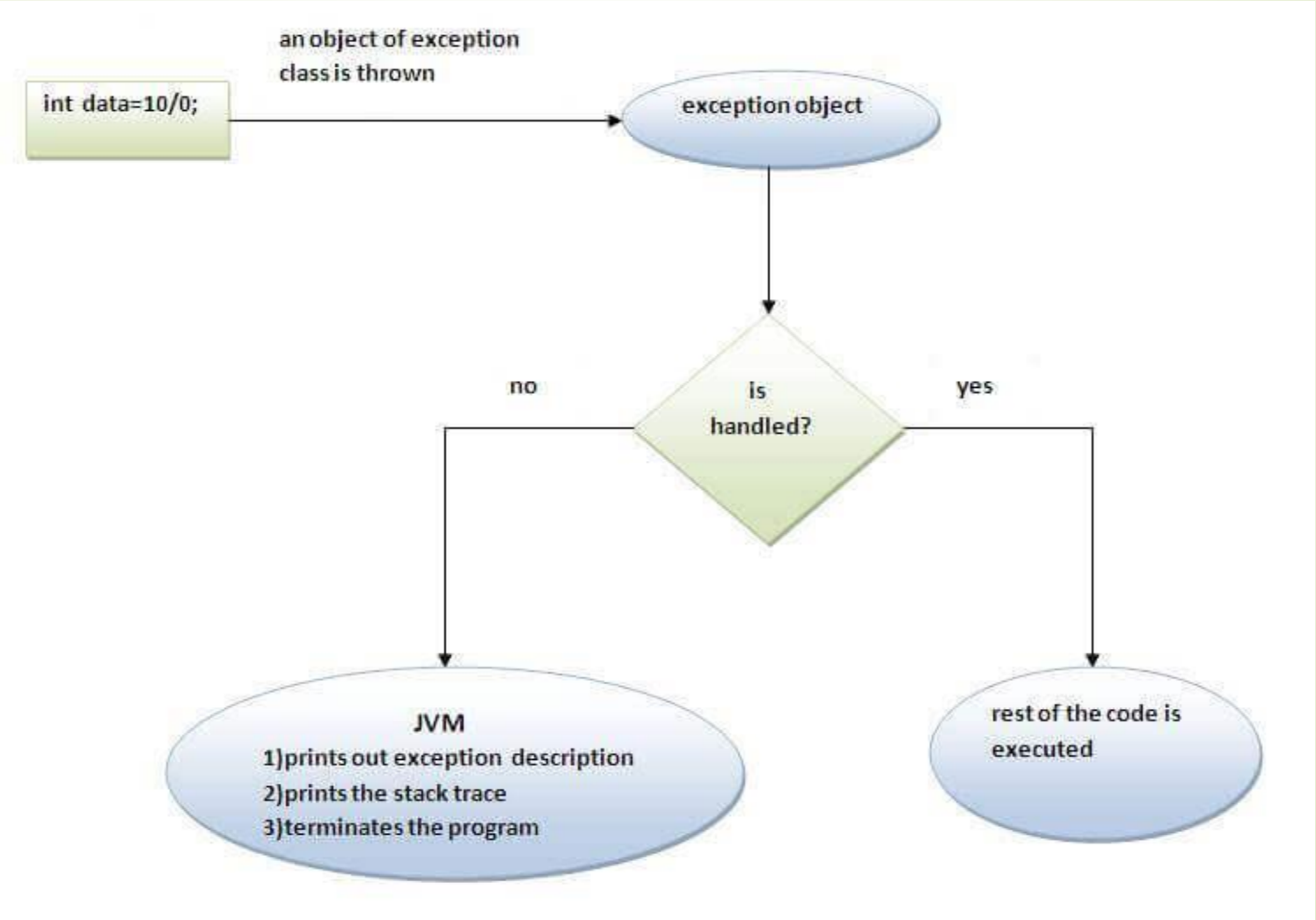
Example: Handling with different type of exception class

```
public class TryCatchExample8 {
    public static void main(String[] args) {
        try
        {
              int data=50/0; //may throw exception
        }
          catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    } }
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero

Exception Handling in Java

Internal working of java try-catch block



Exception Handling in Java

Java Multi-catch block

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
        catch(ArithmeticException e)
            {
            System.out.println("Arithmetic Exception occurs");

           }
```

```
catch(ArrayIndexOutOfBoundsException e)
                {
                 System.out.println("ArrayIndexOutOfBounds Exce
ption occurs");
                }
catch(Exception e)
                {
                 System.out.println("Parent Exception occurs");
                }
               System.out.println("rest of the code");
    }
}
```

Exception Handling in Java

Java finally block

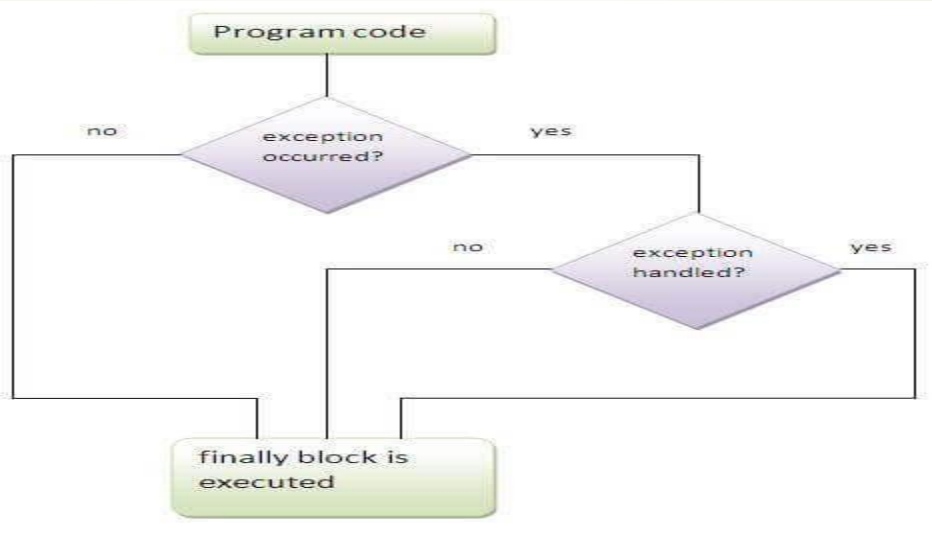
- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
        int data=25/0;
        System.out.println(data);
      }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code..."); //will not execute
  } }
```

Output: finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

Exception Handling in Java



- If you don't handle exception, before terminating the program, JVM executes finally block(if any).
- For each try block there can be zero or more catch blocks, but only one finally block.
- The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Why use java finally?

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Exception Handling in Java

Java throw keyword

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword.
- The throw keyword is **mainly used to throw custom exception**.

Syntax

- **throw** exception;

Example

```
public class TestThrow1{
  static void validate(int age){
   if(age<18)
    throw new ArithmeticException("not valid");
   else
    System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    validate(13);
    System.out.println("rest of the code...");
} }
```

Output: Exception in thread main java.lang.ArithmeticException:not valid

Exception Handling in Java

Java Custom Exception

- If you are creating your own Exception that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.

Example:

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
```

```
class TestCustomException1{

  static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }

  public static void main(String args[]){
    try{
    validate(13);
    }catch(Exception m){System.out.println("Exception occured: "+m);}

    System.out.println("rest of the code...");
  }
}
```

Output: Exception occured: InvalidAgeException: not valid

rest of the code...

End of Session

Pattern Matching with Regular Expressions

Dr. Subrat Kumar Nayak
Associate Professor
Department of CSE
ITER, SOADU

Regular Expression

- Regular expressions, or regexes for short, provide a concise and precise specification of patterns to be matched in text.

Example:

- Suppose you have a bunch of 150000 mail in your drive, And let's further suppose that you remember that somewhere in there is an email message from someone named Angie or Anjie. Or was it Angy? But you don't remember what you called it or where you stored it. Obviously, you have to look for it.
- Simplest way is to write a regular expression to search it:

An[^ dn].*

finding words that begin with "An", while the cryptic [^ dn] requires

The "An" to be followed by a character other than (^ means not in this context) a space (to eliminate the very common English word "an" at the start of a sentence) or "d" (to eliminate the common word "and") or "n" (to eliminate Anne, Announcing, etc.).

Subexpression	Matches	Notes
General		
\^	Start of line/string	
$	End of line/string	
\b	Word boundary	
\B	Not a word boundary	
\A	Beginning of entire string	

Subexpression	Matches	Notes
\p{Punct}	Punctuation characters	One of !"#$%&'()*+,-./:;<=>?@[]\^_`{|}\~
\p{Lower}	Lowercase characters	[a-z]
\p{Upper}	Uppercase characters	[A-Z]
\p{XDigit}	Hexadecimal digit characters	[0-9a-fA-F]

Subexpression	Matches	Notes
\z	End of entire string	
\Z	End of entire string (except allowable final line terminator)	See Recipe 4.9
.	Any one character (except line terminator)	
[…]	"Character class"; any one character from those listed	
[\^…]	Any one character not from those listed	See Recipe 4.2
Alternation and Grouping		
(…)	Grouping (capture groups)	See Recipe 4.3
|	Alternation	
(?:_re_)	Noncapturing parenthesis	
\G	End of the previous match	
\ *n*	Back-reference to capture group number "*n*"	
Normal (greedy) quantifiers		
{ *m*,*n* }	Quantifier for "from *m* to *n* repetitions"	See Recipe 4.4
{ *m* ,}	Quantifier for "*m* or more repetitions"	
{ *m* }	Quantifier for "exactly *m* repetitions"	See Recipe 4.10
{,*n* }	Quantifier for 0 up to *n* repetitions	
*	Quantifier for 0 or more repetitions	Short for {0,}
+	Quantifier for 1 or more repetitions	Short for {1,}; see Recipe 4.2
?	Quantifier for 0 or 1 repetitions (i.e., present exactly once, or not at all)	Short for {0,1}
Reluctant (non-greedy) quantifiers		
{ *m*,*n* }?	Reluctant quantifier for "from *m* to *n* repetitions"	
{ *m* ,}?	Reluctant quantifier for "*m* or more repetitions"	
{,*n* }?	Reluctant quantifier for 0 up to *n* repetitions	
*?	Reluctant quantifier: 0 or more	
+?	Reluctant quantifier: 1 or more	See Recipe 4.10
??	Reluctant quantifier: 0 or 1 times	
Possessive (very greedy) quantifiers		
{ *m*,*n* }+	Possessive quantifier for "from *m* to *n* repetitions"	
{ *m* ,}+	Possessive quantifier for "*m* or more repetitions"	
{,*n* }+	Possessive quantifier for 0 up to *n* repetitions	
*+	Possessive quantifier: 0 or more	
++	Possessive quantifier: 1 or more	

Subexpression	Matches	Notes
?+	Possessive quantifier: 0 or 1 times	
Escapes and shorthands		
\	Escape (quote) character: turns most metacharacters off; turns subsequent alphabetic into metacharacters	
\Q	Escape (quote) all characters up to \E	
\E	Ends quoting begun with \Q	
\t	Tab character	
\r	Return (carriage return) character	
\n	Newline character	See Recipe 4.9
\f	Form feed	
\w	Character in a word	Use \w+ for a word; see Recipe 4.10
\W	A nonword character	
\d	Numeric digit	Use \d+ for an integer; see Recipe 4.2
\D	A nondigit character	
\s	Whitespace	Space, tab, etc., as determined by java.lang.Character.isWhitespace()
\S	A nonwhitespace character	See Recipe 4.10
Unicode blocks (representative samples)		
\p{InGreek}	A character in the Greek block	(Simple block)
\P{InGreek}	Any character not in the Greek block	
\p{Lu}	An uppercase letter	(Simple category)
\p{Sc}	A currency symbol	
POSIX-style character classes (defined only for US-ASCII)		
\p{Alnum}	Alphanumeric characters	[A-Za-z0-9]
\p{Alpha}	Alphabetic characters	[A-Za-z]
\p{ASCII}	Any ASCII character	[\x00-\x7F]
\p{Blank}	Space and tab characters	
\p{Space}	Space characters	[\t\n\x0B\f\r]
\p{Cntrl}	Control characters	[\x00-\x1F\x7F]
\p{Digit}	Numeric digit characters	[0-9]
\p{Graph}	Printable and visible characters (not spaces or control characters)	
\p{Print}	Printable characters	

Regular Expression

Differences Among Greedy, Reluctant, and Possessive Quantifiers

- Greedy quantifiers are considered "greedy" because they force the matcher to **read in**, or *eat*, **the entire input string prior to attempting the first match. If the first match attempt** (the entire input string) **fails**, **the matcher backs off the input string by one character and tries again, repeating the process until a match is found or there are no more characters left to back off from.** Depending on the quantifier used in the expression, the last thing it will try matching against is 1 or 0 characters.
- The reluctant quantifiers, however, take the opposite approach: **They start at the beginning of the input string, then reluctantly eat one character at a time looking for a match.** The last thing they try is the entire input string.
- Finally, the possessive quantifiers always eat the entire input string, trying once (and only once) for a match. Unlike the greedy quantifiers, possessive quantifiers never back off, even if doing so would allow the overall match to succeed.

Regular Expression

Differences Among Greedy, Reluctant, and Possessive Quantifiers

Example:

Enter your regex: .*foo // **greedy quantifier**

Enter input string to search: xfooxxxxxx**foo**

I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.

Enter your regex: .*?foo // **reluctant quantifier**

Enter input string to search: **xfoo**xxxxx**xfoo**

I found the text "xfoo" starting at index 0 and ending at index 4.

I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.

Enter your regex: .*+foo // **possessive quantifier**

Enter input string to search: xfooxxxxxx**foo**

No match found. (Failed to backtrack. Hence, **foo** seems to be missing for possesive)

Regular Expression

Greedy	Reluctant	Possessive	Meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly *n* times
X{n,}	X{n,}?	X{n,}+	X, at least *n* times
X{n,m}	X{n,m}?	X{n,m}+	X, at least *n* but not more than *m* times

Enter your regex: a?
Enter input string to search:
I found the text "" starting at index 0 and ending at index 0.
Enter your regex: a*
Enter input string to search:
I found the text "" starting at index 0 and ending at index 0.
Enter your regex: a+
Enter input string to search:
No match found.

Regular Expression

Q. Write a regular expression to print all the name from n name start with Angie, Anjie or Angy.

Ans: An[^ nd].* //An[^ nd]+ Angelina will not match

Q. Write a regular expression to print the string from bunch of string starting with "A" followed by any number of character.

Ans: A.*/A.+

Q. Write a regular expression to find a match that starts with an alphabate and end with a digit.

Ans: \b\p{Alpha}{1,4}\d{1}\b

Q. Write a regular expression to find a match that starts with a vowel.

Ans: \b[aeiou]\p{Alnum}{1,}

Q. Write a regular expression to find a match that starts with a vowel and end with a vowel.

Ans: \b[aeiou]([0-9] | [a-z]){1,4}[aeiou]\b or

\b[aeiou]\p{Alnum}{1,9}[aeiou]\b or

\b([aeiou] | [AEIOU])\p{Alnum}{1,}[aeiou]\b

Regular Expression

Q. Write a regular expression that matches with a string that starts with a name like Angie/Anjie/Angy.

Ans: ^An[^ dn].*

Q. Write a regular expression that validates a date in MM/DD/YYYY format. Note: ignore leap year

Ans: ^(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}$

Q. Write a regular expression to find a match that starts with an uppercase letter and end with a digit.

Ans: \b\p{Upper}{1}\p{Alpha}{1,}\d{1}\b

Regular Expression

Using regexes in Java: Test for a Pattern

Matching regex using matches() in String class:

- If all you need is to find out whether a given **regex matches a string**, you can use the convenient boolean matches() method of the String class, which accepts a regex pattern in String form as its argument.

Example:

```
if ( inputString . matches ( stringRegexPattern )) {
      // it matched ... do something with it ...
}
```

Regular Expression

Java Regex:

- The **Java Regex** or Regular Expression is an API to *define a pattern for searching or manipulating strings*.

Matching regexes using Pattern and Matcher(s)

- If the regex is going to be used more than once or twice in a program, it is more efficient to construct and use a Pattern and its Matcher (s).
- The normal steps for regex

1. Create a Pattern by calling the static method Pattern.compile().
2. Request a Matcher from the pattern by calling pattern.matcher(CharSequence) for each String (or other CharSequence) you wish to look through.
3. Call (once or more) one of the finder methods (discussed later) in the resulting Matcher .

Regular Expression

java.util.regex package

- The Matcher and Pattern classes provide the facility of Java regular expression.

The Matcher class

It is a *regex engine* which is used to perform match operations on a character sequence.

The Matcher methods

- **boolean matches()**

Used to compare the **entire string** against the pattern; this is the same as the routine in java.lang.String .

- **lookingAt()**

Used to match the pattern **only at the beginning** of the string.

- **boolean find()**

Used to match the pattern in the string (not necessarily at the first character of the string), starting at the beginning of the string or, if the method was previously called and succeeded, at the first character not matched by the previous match.

Regular Expression

Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

Methods

- static Pattern compile(String regex)

compiles the given regex and returns the instance of the Pattern.

- Matcher matcher(CharSequence input)

creates a matcher that matches the given input with the pattern.

…

Regular Expression

matches()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "pqr .*";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. matches ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string "+ input );
        }}
}
```

Output : Patern pqr .* found in string pqr abd pxy

Regular Expression

lookingAt()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "pqr ";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. lookingAt ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string " + input );
        }}
}
```

Output : Patern pqr found in string pqr abd pxy

Regular Expression

find()

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = "abd ";
                String input ="pqr abd pxy ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. find ())
                {
                        System .out. println (" Patern "+ pattern +" found in string "+ input );
                }
                else
                {
                        System .out. println (" Patern "+ pattern +" not found in string "+ input );
        }}
}
```

Output : Patern abd found in string pqr abd pxy

Regular Expression

Finding the Matching Text

- You need to find the text that the regex matched with.

Related functions

start(), end()

- Returns the character position in the string of the starting and ending characters that matched.

groupCount()

- Returns the number of parenthesized capture groups **in the expression/regex**, if any; returns 0 if no groups were used.

group(int i)

- Returns the characters matched by group i of the current match, if i is greater than or equal to zero and less than or equal to the return value of groupCount().
- Group 0 is the entire match, so group(0) (or just group()) returns the entire portion of the input that matched.

Regular Expression

groupCount() example

```
import java . util . regex .*;
      public class RESimple {
            public static void main ( String [] argv ) {
            String pattern = " (.*) (\\d {6}) ";
            //Two groups
            String input =" abdpxy 100000 ";
            Pattern p = Pattern . compile ( pattern );
            Matcher m=p. matcher ( input );
            System .out. println (" Total group = "+m. groupCount ());
      }
}
```

Output : Total group =2

Regular Expression

```
import java . util . regex .*;
public class RESimple {
        public static void main ( String [] argv ) {
                String pattern = " (.*) (\\d {6}) ";
                String input =" abdpxy 100000 ";
                Pattern p = Pattern . compile ( pattern );
                Matcher m=p. matcher ( input );
                if(m. find ()){
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (0) );
                                                        // abdpxy 100000
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (1) );
                                                        // abdpxy
                        System .out. println (" Patern "+ pattern +" found in string "+ input +" with group
                                "+m. group (2) );
                                                        // 100000
                }
}}
```

Regular Expression

Replacing the Matched Text

- **replaceAll(newString)**

Replaces all occurrences that matched with the new string.

Example:

```
import java . util . regex . Pattern ;
import java . util . regex . Matcher ;
public class ReplaceAll {
        public static void main ( String args [])
        {
                String patt = "\\ bfavor \\b"; // A test input .
                String input = "Do me a favor ? Fetch my favorite .favor ";
                System .out. println (" Input : " + input );
                // Run it from a RE instance and see that it works
                Pattern r = Pattern . compile ( patt );
                Matcher m = r. matcher ( input );
        System .out. println (" ReplaceAll : " + m. replaceAll (" favour "));
}}
```

Regular Expression

- **appendReplacement(StringBuffer, newString)**

Copies up to before the first match, plus the given newString .

- **appendTail(StringBuffer)**

Appends text after the last match (normally used after appendReplacement).

```
public class ReplaceAll {
        public static void main ( String args [])
        {
                String patt = "\\ b favor \\b"; // A test input .
                String input = "Do me a favor ? Fetch my favorite (favor) ";
                System .out. println (" Input : " + input );
                Pattern r = Pattern . compile ( patt ); // Run it from a RE instance and see that it works
                Matcher m = r. matcher ( input );
                StringBuffer sb= new StringBuffer ();
                while (m. find ()) {
                m. appendReplacement (sb , " favour ");// Copy to before first match , plus the word " favor "
                }
                m . appendTail (sb); // copy remainder (comment this line to check the importance of appendTail)
                System .out. println (sb. toString ());
}}
```

Regular Expression

Pattern.compile() Flags

- **CASE_INSENSITIVE**

Turns on case-insensitive matching

Ex. Pattern reCaseInsens = Pattern . compile (pattern, **P**attern . CASE_INSENSITIVE)

// check the previous example using "Favor" instead of "favor".

- **COMMENTS**

Causes whitespace and comments (from # to endofline) to be ignored in the pattern.

- **DOTALL**

Allows dot (.) to match any regular character or the newline, not just any regular character other than newline.

- **MULTILINE**

Specifies multiline mode.

Task (Explore)

- **UNICODE_CASE**
- **UNIX_LINES**

Regular Expression

```
import java . util . regex .*;
public class NewLine
{
        public static void main ( String args [])
        {
                String input = "I dream of engines \ nmore
                engines , all day long ";
                System .out. println (" INPUT : " + input );
                System .out. println ("");
                String [] patt = {" engines . More engines "," ines
                \ nmore "," engines$ "};
                for (int i = 0; i < patt . length ; i ++)
                {
                        System .out. println (" PATTERN " + patt [i]);
                        boolean found ;
                        Pattern p1l = Pattern . compile ( patt [i]);
found = p1l. matcher ( input ). find ();
System .out. println (" DEFAULT match " + found );
Pattern pml =
                                Pattern . compile ( patt [i], Pattern .
                                DOTALL | Pattern . MULTILINE );
found = pml. matcher ( input ). find ();
System .out. println (" MultiLine match " + found );
System .out. println ();
}}}
```

Output:

```
INPUT : I dream of engines
more engines , all day long
PATTERN engines . more engines
DEFAULT match false
MultiLine match true
PATTERN ines
more
DEFAULT match true
MultiLine match true
PATTERN engines$
DEFAULT match false
MultiLine match true
```

End of Chapter

Numbers

(Wrapper Classes in Java)

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Wrapper Classes in Java

Wrapper class:

- The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Why Wrapper class?

- Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc.

1. Change the value in Method:

Java supports only call by value. So, if we pass a primitive value, it will not change the original value.

2. Serialization:

We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.

3. Synchronization:

Java synchronization works with objects in Multithreading.

4. java.util package:

The java.util package provides the utility classes to deal with objects.

5. Collection Framework:

Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Wrapper Classes in Java

- The list of primitive types and their corresponding Wrapper classes.

Built-in type	Object wrapper	Size of built-in (bits)	Contents
byte	Byte	8	Signed integer
short	Short	16	Signed integer
int	Integer	32	Signed integer
long	Long	64	Signed integer
float	Float	32	IEEE-754 floating point
double	Double	64	IEEE-754 floating point
char	Character	16	Unsigned Unicode character
n/a	BigInteger	unlimited	Arbitrary-size immutable integer value
n/a	BigDecimal	unlimited	Arbitrary-size-and-precision immutable floating-point value

- Along with these, there is another wrapper class for boolean primitive data type, i.e. Boolean wrapper class
- The **java.lang.Number** class is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.The Subclasses of Number must provide methods to convert the represented numeric value to byte, double, float, int, long, and short.

Wrapper Classes in Java

Constructors:

- Every wrapper class in java has two constructors,

(a) First constructor takes corresponding primitive data as an argument

(b) Second constructor takes string as an argument.

Notes:

- The string passed to second constructor should be parse-able to number , otherwise you will get run time **NumberFormatException**.
- Wrapper Class Character has only one constructor which takes char type as an argument. It doesn't have a constructor which takes String as an argument. Because, String can not be converted into Character.
- Wrapper class Float has three constructors. The third constructor takes double type as an argument.

Examples:

```
Integer I1 = new  Integer(30);
//Constructor which takes int value as an argument

Integer I2 = new  Integer("30");
//Constructor which takes String as an argument
```

Wrapper Classes in Java

Examples…

```
Short S1 = new Short((short) 20);   //Constructor which takes short value as an argument
Short S2 = new Short("10");         //Constructor which takes String as an argument


Float F1 = new Float(12.2f); //Constructor which takes float value as an argument
Float F2 = new Float("15.6");//Constructor which takes String as an argument
Float F3 = new Float(15.6d); //Constructor which takes double value as an argument


Character C1 = new Character('D'); //Constructor which takes char value as an argument
Character C2 = new Character("a");
//Compile time error : String a can not be converted to character
```

- Converting primitive data types into object is called **boxing**, and this is taken care by the compiler.
- Similarly, the Wrapper object can also be converted back to a primitive data type, and this process is called **unboxing**.

- The **automatic conversion** of primitive data type into its corresponding wrapper class is known as **auto-boxing**.
- The **automatic conversion** of wrapper type into its corresponding primitive type is known as **auto-unboxing**.

Wrapper Classes in Java

Examples for Auto-boxing & Auto-unboxing:

```
public class WrapperExample1{
public static void main(String args[]){
int a=20;
Integer j=a;//auto-boxing, now compiler will call Integer.valueOf(a) internally
System.out.println(j);// will print 20
}}
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(10);
int j=a;//auto-unboxing, now compiler will call a.intValue() internally
System.out.println(j);// will print 10
}}
```

Wrapper Classes in Java

Number Methods:

Following is the list of the instance methods that **all the subclasses** of the Number class implements.

xxxValue()

Converts the value of *this* Number object to the **xxx** data type and returns it.

Syntax:

xxx xxxValue()

- xxx stands for different primitive data type.

Example:

```
public class Test {
public static void main(String args[]) {
     Integer x = 5;
     // Returns byte primitive data type
     System.out.println(x.byteValue());
     // Returns double primitive data type
     System.out.println(x.doubleValue());
     // Returns long primitive data type
     System.out.println(x.longValue());
     //will not work x.charValue()for Integer.
} }
```

```
Output:
5
5.0
5
```

Wrapper Classes in Java

compareTo()

- The method compares the Number object that invoked the method to the argument. It is possible to compare Byte, Long, Integer, etc.
- However, two different types cannot be compared, both the argument and the Number object invoking the method should be of the same type.

Syntax:

```
public int compareTo(NumberSubClass referenceName)
```

Return Value:

- If this object equal to the argument then 0 is returned.
- If this object less than the argument then -1 is returned.
- If this object greater than the argument then 1 is returned.

Example:

```
public class Test {
public static void main(String args[]) {
     Integer x = 5;
     System.out.println(x.compareTo(3));
     System.out.println(x.compareTo(5));
     System.out.println(x.compareTo(8));
} }
```

```
Output:
1
0
-1
```

Wrapper Classes in Java

valueOf()

- The valueOf method returns the relevant Number Object holding the value of the argument passed. The argument can be a primitive data type, String, etc.
- This method is a **static** method. The method can take two arguments, where one is a String and the other is a radix.

Syntax:

```
static Xxx valueOf(xxx i)// works for all Wrapper type

//xxx: primitive type of corresponding wrapper type

static Xxx valueOf(String s)// does not work for Character

static Xxx valueOf(String s, int radix)// not applicable for Character,
Double,Float.

//Xxx represents any of the subclass of Number class(Integer, Float etc.)
```

Return Values:

- **valueOf(int i)** – This returns an Xxx object holding the value of the specified primitive.
- **valueOf(String s)** – This returns an Xxx object holding the value of the specified string representation.
- **valueOf(String s, int radix)** – This returns an Xxx object holding the integer value of the specified string representation, parsed with the value of radix.

Wrapper Classes in Java

Example:

```
public class Test {
public static void main(String args[]) {
    Integer x =Integer.valueOf(9);
    Double c = Double.valueOf(5);
    Float a = Float.valueOf("80");
    Integer b = Integer.valueOf(“0111",2);
    System.out.println(x);
    System.out.println(c);
    System.out.println(a);
    System.out.println(b);
} }
```

```
Output:
9
5.0
80.0
7
```

End of Session

Numbers

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Some More Functions in Wrapper Classes…

equals()

- The method determines whether the Number object that invokes the method is equal to the object that is passed as an argument.

Syntax

```
public boolean equals(Object o)
```

Return Value:

- The method returns True if the argument is not null and is an object of the same type and with the same numeric value.

Example:

```
public class Test {
public static void main(String args[]) {
        Integer x = 5;
        Integer y = 10;
        Integer z =5;
        System.out.println(x.equals(y));
        System.out.println(x.equals(z));
} }
```

Some More Functions in Wrapper Classes…

toString()

- The method is used to get a String object representing the value of the Number Object.
- If the method takes a primitive data type as an argument, then the String object representing the primitive data type value is returned.
- If the method takes two arguments, then a String representation of the first argument in the radix specified by the second argument will be returned.

Syntax:

```
String toString()
static String toString(int i)
Static String toString(int i, int radix)
```

Example :

```
public class Test {
public static void main(String args[]) {
Integer x = 5;
System.out.println(x.toString());
System.out.println(Integer.toString(12));
System.out.println(Integer.toString(12,2));
} }
```

```
Output:
5
12
1100
```

Some More Functions in Wrapper Classes…

parseXxx()

- This method is used to get the primitive data type of a certain String. parseXxx() is a static method and can have one argument or two.
- Does not work for Character.

Note: same as valueOf(). However, it only works on String object.

Syntax:

```
static int parseInt(String s)
static int parseInt(String s, int radix)
```

Example:

```
public class Test { public static void main(String args[]) {
    int x =Integer.parseInt("9");
    double c = Double.parseDouble("5");
    int b = Integer.parseInt("1100",2);
    System.out.println(x);
    System.out.println(c);
    System.out.println(b);
} }
```

```
Output:
9
5
12
```

How much we have proceeded?

- **Introduction**
- **Checking Whether a String Is a Valid Number (parseInt())**
- **Wrapper Class and its Methods**
- **Converting Numbers to Objects and Vice Versa**

Storing a Larger Number in a Smaller Number

```
float f=3.0// won't even compile!
This line will be understood as follows.
double tmp=3.0;
float f=tmp;
```

How to fix?

- Can be fixed in one of the several ways:

1) By making the 3.0 a float (probably the best solution)
2) By making f a double
3) By putting in a cast
4) By assigning an integer value of 3, which will get "promoted"

Example:

```
float f=3.0f;
double f=3.0;
float f=(float)3.0;
float f=3;
```

Ensuring the Accuracy of Floating-Point Numbers

- In java integer division by 0 consider as logical error so it throws an ArithmeticException.
- Floating-point operations, however, do not throw an exception because they are defined over an (almost) infinite range of values.

Java act differently for the following cases.

1) Java signal errors by producing the constant POSITIVE_INFINITY if you divide a positive floating-point number by zero

2) It signal constant NEGATIVE_INFINITY if you divide a negative floating-point value by zero.

3) Produces NaN (Not a Number) if you otherwise generate an invalid result

- Values for these **three public constants** are defined in both the Float and the Double wrapper classes.
- The value NaN has the unusual property that it is not equal to itself (i.e., NaN != NaN).
- x==NaN never be true, instead, the methods Float.isNaN(float) and Double.isNaN(double) must be used.

Ensuring the Accuracy of Floating-Point Numbers

```
public static void main(String[] args){
    double d = 123;
    double e = 0;
    if (d/e == Double.POSITIVE_INFINITY)
        System.out.println("Check for POSITIVE_INFINITY works");
    double s = Math.sqrt(-1);
    if (s == Double.NaN)
        System.out.println("Comparison with NaN incorrectly returns
        true");
    if (Double.isNaN(s))
        System.out.println("Double.isNaN() correctly returns true");
}
```

Output:

```
Check for POSITIVE_INFINITY works
Double.isNaN() correctly returns true
```

Comparing Floating Point Numbers

- The equals() method of Float and Double wrapper class returns true if the two values are the same bit for bit (i.e., if and only if the numbers are the same or are both NaN).
- It returns false otherwise, including if the argument passed in is null, or if one object is +0.0 and the other is –0.0.
- To actually compare floating-point numbers for equality, it is generally desirable to compare them within **some tiny range of allowable differences**; this range is often regarded as a tolerance or as *epsilon*.

Example:

```
public class NumberTest {
    public static void main(String[] args) {
        float x=0.3f*3;
        if(x==0.9)
            System.out.println(x);
    }
}
```

Output:

Comparing Floating Point Numbers

```
public class FloatCmp {
   final static double EPSILON = 0.0000001;
   public static void main(String[] argv) {
      double da = 3 * .3333333333;
      double db = 0.99999992857;
      // Compare two numbers that are expected
      to be close.
      if (da == db) {
        System.out.println("Java considers " +
        da + "==" + db);
       // else compare with our own equals
       overload
      }
      else if (equals(da, db, 0.0000001)) {
        System.out.println("Equal within epsilon
        " + EPSILON);
      }
      else {
        System.out.println(da + " != " + db);
      }
}
```

```
/** Compare two doubles within a given
epsilon */
public static boolean equals(double a, double
b, double eps) {
   if (a==b)
      return true;
   // If the difference is less than epsilon,
   treat as equal.
   return Math.abs(a - b) < eps;
}
/** Compare two doubles, using default
epsilon */
public static boolean equals(double a, double
b) {
   return equals(a, b, EPSILON);
}
```

Rounding Floating-Point Numbers

- To round floating-point numbers properly, use Math.round() .
- It has two overloads:

  if you give it a double , you get a long result;

  if you give it a float , you get an int .

- If the argument is *NaN* (not a number), then the function will return 00.
- If the argument is negative infinity (**Float**) or any value less than or equal to the value of **Integer.MIN_VALUE**, then the function returns **Integer.MIN_VALUE**. (Try for Double.NEGATIVE_INFINITY)
- If the argument is positive infinity or any value greater than or equal to the value of Integer.MAX_VALUE, then the function returns Integer.MAX_VALUE. (Try for Double.POSITIVE_INFINITY)

Example:

```
double d=5.67;
System.out.println(Math.round(d));
float f=9.4255f;
System.out.println(Math.round(f));
```

End of Session

Numbers

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Formatting Numbers

- **NumberFormat** is an **abstract base class** for all number formats. This class provides the interface for formatting and parsing numbers.
- NumberFormat also provides methods for determining which locales (US, India, Italy, etc) have number formats, and what their names are.
- NumberFormat helps you to format and parse numbers for any locale.
- A DecimalFormat object appropriate to the user's locale can be obtained from the factory method NumberFormat.getInstance() and manipulated using set methods.

Package :

`Java.text.NumberFormat`

How to Create a NumberFormat object?

NumberFormat nf=NumberFormat.getInstance()// No constructor

Methods:

setMaximumFractionDigits()

- Sets the maximum number of digits allowed in the fraction portion of a number.

Syntax:

void setMaximumFractionDigits(int newValue)

Formatting Numbers

Example:

```
import java.text.NumberFormat;
public class NumberFormatTest {
    public static void main(String args[])
    {
        NumberFormat nf=NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        double d=123.456;
        System.out.println(nf.format(d));// format the value with format()
    }
}
```

Output:

```
123.46
```

Formatting Numbers

void setMaximumIntegerDigits(int newValue)

- Sets the maximum number of digits allowed in the integer portion of a number.

void setMinimumFractionDigits(int newValue)

- Sets the minimum number of digits allowed in the fraction portion of a number.

void setMinimumIntegerDigits(int newValue)

- Sets the minimum number of digits allowed in the integer portion of a number.

Example:

Q. Write a program to set the minimum integer digit to 3, maximum fraction digit to 4 and minimum fraction digit to 2 of a decimal number.

Formatting Numbers

Changing the pattern dynamically

- You can also construct a DecimalFormat with a particular pattern or change the pattern dynamically using applyPattern().

Common Pattern Characters

Character	Meaning
#	A digit, leading zeroes are omitted.
0	A digit - always displayed, even if number has less digits (then 0 is displayed)
.	Locale-specific decimal separator (decimal point)
,	Locale-specific grouping separator (comma in English)
-	Locale-specific negative indicator (minus sign)
%	Shows the value as a percentage
;	Separates two formats: the first for positive and the second for negative values
'	Escapes one of the above characters so it appears
Anything else	Appears as itself

Formatting Numbers

Examples:

```
import java.text.NumberFormat;
import java.text.DecimalFormat;
public class NumberFormatTest {
     public static void main(String args[])
     {
          NumberFormat ourForm = new DecimalFormat("###.##");
          double d=123.345;
          System.out.println(ourForm.format(d));
     }
}
```

Output:

```
123.34
```

Formatting Numbers

```
import java.text.NumberFormat;
import java.text.DecimalFormat;
public class NumberFormatTest {
    public static void main(String args[])
    {
        NumberFormat ourForm = new DecimalFormat("0000.##");
        double d=12.5678;
        System.out.println(ourForm.format(d));
    }
}
```

Output:

0012.57

Converting Between Binary, Octal, Decimal, and Hexadecimal

- Integer.toString(int input, int radix) to convert from integer to any type.

Example

```
public class ConversionTest {
    public static void main(String args[])
    {
        int i=42;
        String res1=Integer.toString(i,2);
        String res2=Integer.toString(i,8);
        String res3=Integer.toString(i,16);
        String res4=Integer.toString(i,10);
        System.out.println("42 in base 2 is "+res1);
        System.out.println("42 in base 8 is "+res2);
        System.out.println("42 in base 16 is "+res3);
        System.out.println("42 in base 10 is "+res4);
    }
}
```

Output:
42 in base 2 is 101010
42 in base 8 is 52
42 in base 16 is 2a
42 in base 10 is 42

Converting Between Binary, Octal, Decimal, and Hexadecimal

- Integer.parseInt(String input, int radix) to convert from any type of number to an Integer

```
public class ConversionTest {
public static void main(String args[])
{
    String str="1010";
    Integer iObj=Integer.parseInt(str,2);
    System.out.println("1010 in base 2 is "+iObj);
    Integer iObj1=Integer.parseInt(str,8);
    System.out.println("1010 in base 8 is "+iObj1);
    Integer iObj2=Integer.parseInt(str,16);
    System.out.println("1010 in base 16 is "+iObj2);
}
}
```

Output:
1010 in base 2 is 10
1010 in base 8 is 520
1010 in base 16 is 4112

End of Session

Numbers

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Operating on a Series of Integers

- For a contiguous set, use a for loop.

Example:

```
String months[]={
"January", "February", "March", "April",
"May", "June", "July", "August",
"September", "October", "November", "December"
};
for(int i=0;i<months.length;i++)
{
System.out.println("Month " + months[i])
}
```

Operating on a Series of Integers

- For discontinuous ranges of numbers, use a java.util.BitSet .

```
// A discontiguous set of integers, using a BitSet
// Create a BitSet and turn on a couple of bits.
String months[]={"January", "February", "March", "April", "May", "June", "July",
"August", "September", "October", "November", "December‟ };
BitSet b = new BitSet();
b.set(0);
// January
b.set(3);
// April
b.set(8);
// September
// Presumably this would be somewhere else in the code.
for (int i = 0; i<months.length; i++)
{
    if (b.get(i))
    System.out.println("Month " + months[i]);
}
```

Output:
Month January
Month April
Month September

Generating Random Numbers

- Use java.lang.Math.random() to generate random numbers.

Example:

```
//java.lang.Math.random( ) is static, don't need any constructor calls
System.out.println("A random from java.lang.Math is " + Math.random( ));
```

- If you need integers random number, construct a java.util.Random object and call its nextInt() method.

Example:

```
public class RandomInt {
public static void main(String[] a) {
Random r = new Random();
for (int i=0; i<1000; i++)
{
    // nextInt(10) goes from 0-9; add 1 for 1-10;
    System.out.println(1+r.nextInt(10));
}}}
```

Generating Random Numbers

Some other nextXxx()methods from java.util.Random class

- boolean nextBoolean() to find a random boolean .
- byte nextByte() to find a random boolean .
- int nextInt() to find a random int.
- float nextFloat() to find a random float.
- double nextDouble() to find next double.

Handling Very Large Numbers

- **Can you store 100! using any primitive data type?**
- In order to handle very large numbers java provides two classes from java.math package.

(1) BigInteger, to create a large integer number.

(2) BigDecimal, to create a large decimal number(Real number).

BigInteger

- BigInteger class is used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.

Constructors:

- `BigInteger(String val)`

Translates the decimal String representation of a BigInteger into a BigInteger.

- `BigInteger(String val, int radix)`

Translates the String representation of a BigInteger in the specified radix into a BigInteger.

Handling Very Large Numbers

Example:

```
import java.math.BigInteger;
public class BigIntTest {
        public static void main(String args[])
        {
                BigInteger bi=new BigInteger("1234455666666");
                System.out.println(bi);
                BigInteger bi1=new BigInteger("1111",2);
                System.out.println(bi1);
        }
}
```

```
Output:
1234455666666
15
```

Handling Very Large Numbers

Methods:

abs()

- It returns a BigInteger, whose value is the absolute value of this BigInteger.

Example:

```
import java.math.BigInteger;
public class BigIntegerAbsExample {
public static void main(String[] args){
BigInteger big1, big2, big3, big4;  // create 4 BigInteger objects
big1=new BigInteger("345");// assign value to big1
big2=new BigInteger("-345");    // assign value to big2
big3=big1.abs ();         // assign absolute value of big1 to big 3
big4=big2.abs ();        // assign absolute value of big 2 to big 4.
String str1 = "Absolute value of" + big1 + "is" + big3;
String str2 = "Absolute value of" + big2 + "is" + big4;
System.out.println(str1);
System.out.println(str2 );
} }
```

Handling Very Large Numbers

add()

- This method returns a BigInteger by simply computing 'this + val' value.

```
public class BigIntTest {
public static void main(String args[])
{
    BigInteger bi1=new BigInteger("1234455666456");
    BigInteger bi2=new BigInteger("1234455666456");
    BigInteger bi3=bi1.add(bi2);
    System.out.println("Sum of big integer= "+bi3);
}}}
```

```
Output:
Sum of big integer= 2468911332912
```

Handling Very Large Numbers

`BigInteger multiply(BigInteger val)`

- Returns a BigInteger whose value is (this * val).

`BigInteger divide(BigInteger val)`

- Returns a BigInteger whose value is (this / val).

`int compareTo(BigInteger val)`

- Compares this BigInteger with the specified BigInteger.

`boolean equals(Object x)`

- Compares this BigInteger with the specified Object for equality.

`float floatValue()`

- Converts this BigInteger to a float.

`int intValue()`

- Converts this BigInteger to a int.

Handling Very Large Numbers

BigDecimal()

BigDecimal class is used for mathematical operation which involves very big real number calculations that are outside the limit of all available primitive data types.

Constructors:

- `BigDecimal(String val)`

Translates the string representation of a BigDecimal into a BigDecimal object.

- `BigDecimal(BigInteger val)`

Translates a BigInteger into a BigDecimal.

Handling Very Large Numbers

Methods of BigDecimal

`BigDecimal abs()`

- Returns a BigDecimal whose value is the absolute value of this BigDecimal.

`BigDecimal add(BigDecimal augend)`

- Returns a BigDecimal whose value is (this + augend).

`BigDecimal divide(BigDecimal divisor)`

- Returns a BigDecimal whose value is (this / divisor).

`BigDecimal multiply(BigDecimal multiplicand)`

- Returns a BigDecimal whose value is (this * multiplicand). + multiplicand.scale()).

`int compareTo(BigDecimal val)`

- Compares this BigDecimal with the specified BigDecimal.

`boolean equals(Object x)`

- Compares this BigDecimal with the specified Object for equality.

End of Session

Structuring Data with Java

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Structuring Data with Java

- Except very few application, all needs to keep track of the structured data.
- Hence, a well-defined data structure is present in different sections in order to access data systematically.
- There are data structures
  - **in the memory of a running program**
  - in the data in a file on disk
  - in the information stored in a database

Using Arrays for Data Structuring

- To keep track of a fixed amount of information and retrieve it (usually) sequentially, we need an array.
- Arrays can be used to hold any linear collection of data. The items in an array must all be of the same type.
- An array can be formed by using either any **primitive type** or **any object type**.

Using Arrays for Data Structuring

Examples of creating and initializing a one dimensional array.

```
(1) int[] monthLen1; // declare a reference
    monthLen1 = new int[12]; // construct it
(2) int[] monthLen2 = new int[12]; // short form
(3) // even shorter is this initializer form:
    int[] monthLen3 = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,};
```

Examples of creating a two dimensional array.

```
// Two-Dimensional Arrays
// Want a 10-by-24 array
int[][] me = new int[10][];
for (int i=0; i<10; i++)
me[i] = new int[24];
```

Using Arrays for Data Structuring

How to find and print the length we can do the following.

- `System.out.println(me.length);`
- `System.out.println(me[0].length);`

Implement the following programs.

- Q1. Write a program to create an array of integer and display it.
- Q2. Write a program to create an array of Strings object and display it.

Hint: Create an array of Strings and display individual string from that array.

Resizing an Array

- We can not add more elements unless until the array is allocated with a reasonable size.
- Or we can take the help of ArrayList collection class that dynamically changes its size.

The collection Frameworks

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- **Java Collection** means a single unit of objects, i.e., a group
- Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.

What is Collection framework

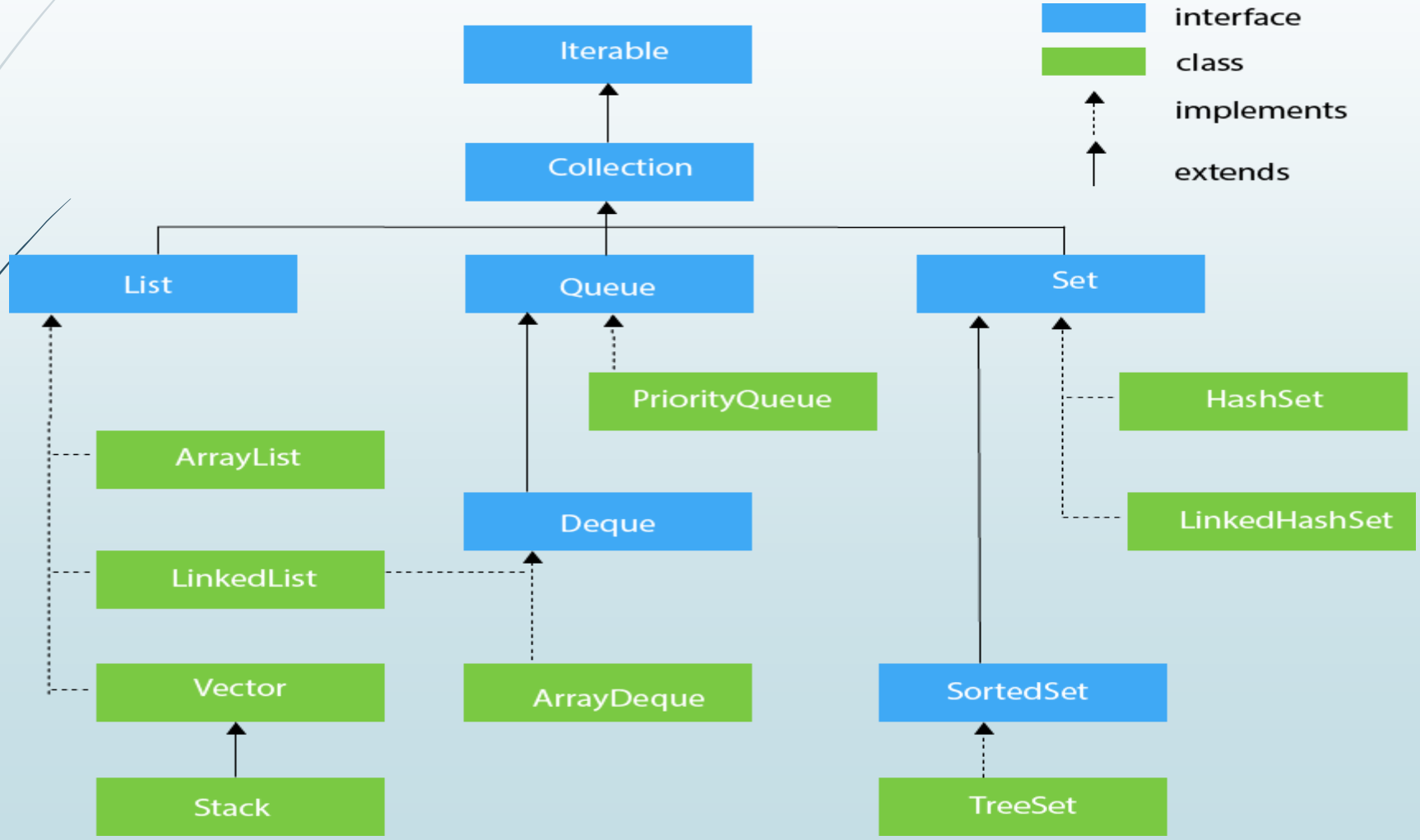
- The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1) Interfaces and its implementations, i.e., classes

1) Algorithm

Hierarchy of Collection Framework

- The **java.util** package contains all the classes and interfaces for the Collection framework.



Like an Array, but More Dynamic

- Java ArrayList class uses a **dynamic array** for storing the elements. It implements List interface. The important points about Java ArrayList class are:
- ✓ Java ArrayList class can contain **duplicate** elements.
- ✓ Java ArrayList class maintains insertion order.
- ✓ Java ArrayList class is non synchronized. (Multiple thread can access ArrayList at a time)
- ✓ Java ArrayList allows random access because array works at the index basis.
- ✓ In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.
- The List interface extends the Collection and Iterable interfaces in hierarchical order.
- **ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases .**

Consructor:

- *ArrayList()*

This constructor is used to build an empty array list

Like an Array, but More Dynamic

Example:

```
import java.util.ArrayList;
public class TestArrayList
{
    public static void main(String[] arg){
        ArrayList al=new ArrayList();
        //or
        List al=new ArrayList();
        //Because ArrayList implements List interface
    }
}
```

Like an Array, but More Dynamic

Methods:

- `add(Object o)` Add the given element at the end
- `add(int i, Object o)` Insert the given element at the specified position
- `clear( )` Remove all element references from the Collection
- `contains(Object o)` True if the List contains the given Object
- `get(int i)` Return the object reference at the specified position
- `indexOf(Object o)` Return the index where the given object is found, or -1
- `remove(Object o)` or `remove(int i)`

Remove an object by reference or by position

- toArray( ) Return an array containing the objects in the Collection

Like an Array, but More Dynamic

- Q. Write a program to add 3 string object to an ArrayList and display it.

```
import java.util.ArrayList;
public class TestArrayList
{
    public static void main(String[] arg)
    {
        ArrayList al=new ArrayList();
        al.add("Subrat");
        al.add("Kumar");
        al.add("Nayak");
        for(int i=0;i<as.size();i++)
        {
            System.out.println(as.get(i));
        }
    }
}
```

Like an Array, but More Dynamic

Q. Write a program to create an ArrayList and insert 5 integer in it and find its sum.

The issue can be resolved by:

- Casting
- By using generics

Like an Array, but More Dynamic
(Avoid Casting by Using Generics)

Java Non-generic Vs. Generic Collection

- Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.
- Java new generic collection **allows you to have only one type of object** in a collection. Now it is type safe so typecasting is not required at runtime.
- The old non-generic example of creating java collection.

```
ArrayList al=new ArrayList();
//creating old non-generic arraylist
```

- The new generic example of creating java collection.

```
ArrayList<object_type> al=new ArrayList<Object_type>();
//creating new generic arraylist
```

Advantages of Generic:

- **Type-safety:** We can hold only a ***single type*** of objects in generics. It does not allow to store other objects. Without Generics, we can store any type of objects.
- **Type casting is not required:** There is no need to typecast the object.
- **Compile-Time Checking:** It is ***checked at compile time*** so problem will not occur at runtime.

End of Session

Structuring Data with Java

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

Structuring Data with Java
(How Shall We Iterate Thee?)

Using Iterator object

- Iterator interface : Iterator is an **interface** that iterates the elements. It is used to traverse the list and modify the elements. Iterator interface has three methods which are mentioned below.

✓ **public boolean hasNext()** - This method returns true if the iterator has more elements.

✓ **public object next()** -It returns the element and moves the cursor pointer to the next element.

✓ **public void remove()** -This method removes the last elements returned by the iterator.

```
import java.util.Iterator;

public static void main(String[]arg)

{

     ArrayList<Integer> as=new ArrayList<Integer>();

     as.add(10);

     as.add(20);

     as.add(30);

     Iterator<Integer> it=as.iterator();

     while(it.hasNext())

     {

          System.out.println(it.next());

     }

}
```

```
void remove()
while(it.hasNext())
{
     if(it.next().equals(20))
        it.remove();
}
```

Structuring Data with Java

(How Shall We Iterate Thee?)

- **Using foreach loop**

```
import java.util.ArrayList;
public class TestArrayList {
public static void main(String[]arg)
{
    ArrayList<Integer> as=new ArrayList<Integer>();
    as.add(10);
    as.add(20);
    as.add(30);
    for(Integer i:as)
    {
        System.out.println(i);
    }
}}
```

- **Using three part for loop**
- **Using while loop**

Structuring Data with Java
(Eschewing Duplicates with a Set)

- Set is an interface which extends Collection. It is an unordered collection of objects in which **duplicate values cannot be stored**.
- Basically, Set is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).
- Set has various methods to add, remove clear, size, etc to enhance the usage of this interface.
- If you add the "same" item (as considered by its equals() method) twice or more, it will only be present once in the set. For this reason, the index-based methods such as add(int, Object) and get(int), are missing from the Set implementation.

Structuring Data with Java
(Eschewing Duplicates with a Set)

HashSet

- Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
- The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements.

- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.

Structuring Data with Java
(Eschewing Duplicates with a Set)

Constructors

- `HashSet()`

It is used to construct a default HashSet

Example

```
Set<Integer> x=new HashSet<Integer>();
x.add(5);
x.add(10);
x.add(20);
x.add(10);
Iterator<Integer> it=x.iterator();
while(it.hasNext())
{
    if(it.next().equals(20))
    it.remove();
}
System.out.println(x);
System.out.println(x.remove(25));
System.out.println(x.size());
System.out.println(x.isEmpty());
System.out.println(x.contains(10));
x.clear();
System.out.println(x);
```

Output:

```
[1, 4, 10]
false
3
false
true
[]
```

Structuring Data with Java
(Using Iterators or Enumerations for Data- Independent Access)

```
Iterator it = l.iterator();
// Process the data structure using an iterator.
// This part of the code does not know or care
// if the data is an an array, a List, a Vector, or whatever.
while (it.hasNext()) {
    Object o = it.next();
    System.out.println("Element " + i++ + " = " + o);
}
```

Enumeration

- The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.
- This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. It is used by several methods defined by the legacy classes such as **Vector and Properties**.

Structuring Data with Java
(Structuring Data in a Linked List)

- Java LinkedList class uses a **doubly linked list** to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.
- The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Constructors:

- **LinkedList()**

It is used to construct an empty list.

Structuring Data with Java

(Structuring Data in a Linked List)

Methods

`add()`

- `Syntax: boolean add(E e)`

It is used to append the specified element to the end of a list

- `Syntax: void add(int index, E element)`

It is used to insert the specified element at the **specified position** index in a list.

`addAll()`

- `Syntax: boolean addAll(Collection<? extends E> c)`

It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

- `Syntax: boolean addAll(int index, Collection<? extends E > c)`

It is used to append all the elements in the specified collection, starting at the specified position of the list

`addFirst()`

- `Syntax: void addFirst(E e)`

It is used to insert the given element at the beginning of a list.

Structuring Data with Java
(Structuring Data in a Linked List)

addLast()

- `Syntax: void addLast(E e)`

It is used to append the given element to the end of a list.

getFirst()

- `Syntax: E getFirst()`

It is used to return the first element in a list.

getLast()

- `Syntax: E getLast()`

It is used to return the last element in a list.

get()

- `Syntax: E get(int index)`

It is used to return the element at the specified position in a list.

indexOf()

- `Syntax: int indexOf(Object o)`

It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.

Structuring Data with Java
(Structuring Data in a Linked List)

peek()

- Syntax: `E peek()`

It retrieves the first element of a list

peekFirst()

- Syntax: `E peekFirst()`

It retrieves the first element of a list or returns null if a list is empty.

peekLast()

- Syntax: `E peekLast()`

It retrieves the last element of a list or returns null if a list is empty.

remove()

- Syntax: `E remove()`

It is used to retrieve and removes the first element of a list

- Syntax: `E remove(int index)`

It is used to remove the element at the specified position in a list.

- Syntax: `boolean remove(Object o)`

It is used to remove the first occurrence of the specified element in a list

Structuring Data with Java
(Structuring Data in a Linked List)

removeFirst()

- `Syntax: E removeFirst()`

It removes and returns the first element from a list.

removeLast()

- `Syntax: E removeLast()`

It removes and returns the last element from a list.

set()

- `Syntax: E set(int index, E element)`

It replaces the element at the specified position in a list with the specified element.

push()

- `Syntax: void push(E e)`

It pushes an element onto the stack represented by a list.

pop()

- `Syntax: E pop()`
- It pops an element from the stack represented by a list.

size()

- `Syntax: int size()`

It is used to return the number of elements in a list.

Structuring Data with Java
(Structuring Data in a Linked List)

`listIterator()`

- `Syntax: ListIterator<E> listIterator(int index)`

It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list.

`descendingIterator()`

- `Syntax: Iterator<E> descendingIterator()`

It is used to return an iterator over the elements in a **deque** in reverse sequential order.

End of Session

Structuring Data with Java

By Dr. Subrat Kumar Nayak

Associate Professor

Department of CSE

ITER, SOADU

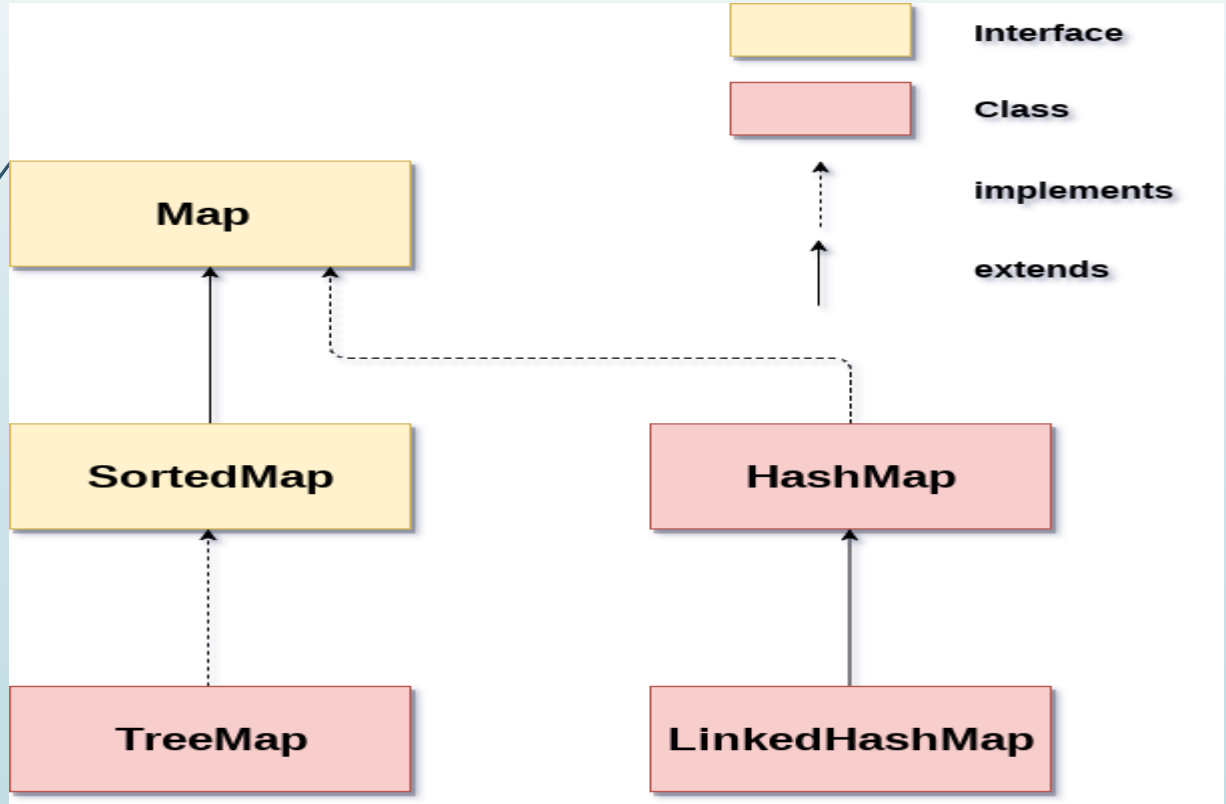
Structuring Data with Java

(Mapping with Hashtable and HashMap)

Java Map Interface

- A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.
- A Map is useful if you have to search, update or delete elements on the basis of a key.

Java Map Hierarchy



- There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap.
- A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow **null keys** and values, but TreeMap **doesn't allow any null key** or value.
- A Map can't be traversed, so you need to **convert it into Set** using *keySet()* or *entrySet()* method.

Structuring Data with Java
(Mapping with Hashtable and HashMap)

Java HashMap class

Java HashMap class implements the map interface by using a hash table. It inherits AbstractMap class and implements Map interface.

- Points to remember

- Java HashMap class contains values based on the key.
- Java HashMap class contains only unique keys.
- Java HashMap class may have one null key and multiple null values.
- Java HashMap class is non synchronized.
- Java HashMap class maintains no order.

Constructor

- `HashMap()`

It is used to construct a default HashMap.

Structuring Data with Java
(Mapping with Hashtable and HashMap)

Methods

`put()`

- `Syntax: V put(Object key, Object value)`

It is used to insert an entry in the map.

`containsValue()`

- `Syntax: boolean containsValue(Object value)`

This method returns true if some value equal to the value exists within the map, else return false.

`containsKey()`

- `Syntax: boolean containsKey(Object key)`

This method returns true if some key equal to the key exists within the map, else return false.

`isEmpty()`

- `Syntax: boolean isEmpty()`

This method returns true if the map is empty; returns false if it contains at least one key.

`replace()`

- `Syntax: V replace(K key, V value)`

It replaces the specified value for a specified key.

- `Syntax:boolean replace(K key, V oldValue, V newValue)`

It replaces the old value with the new value for a specified key.

Structuring Data with Java
(Mapping with Hashtable and HashMap)

Example:

```
public class HashMapTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

    Map<Integer, String> map=new HashMap<Integer, String>();

    map.put(10, "subrat");

    map.put(20, "kumar");

    map.put(30, "nayak");

    System.out.println(map);

    if(map.containsKey(30))

    {

        String v=map.get(30);//get() method returns the object that contains the value
        associated with the key.

        System.out.println("Value of key 30 is: "+v);

    }

    map.clear();

    System.out.println(map);}}
```

Structuring Data with Java
(Mapping with Hashtable and HashMap)

Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

- Points to remember
  - A Hashtable contains values based on the key.
  - Java Hashtable class contains unique elements.
  - Java Hashtable class doesn't allow null key or value.
  - Java Hashtable class is synchronized.

Constructor

- `Hashtable()`

It creates an empty hashtable having the initial default capacity and load factor.

Structuring Data with Java

(Mapping with Hashtable and HashMap)

HashMap vs HashTable

HashMap	HashTable
• HashMap is non synchronized.	• Hashtable is synchronized.
• HashMap allows one null key and multiple null values.	• Hashtable doesn't allow any null key or value.
• HashMap is fast.	• Hashtable is slow.
• We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	• Hashtable is internally synchronized and can't be unsynchronized.
• HashMap is traversed by Iterator.	• Hashtable is traversed by Enumerator and Iterator.
• HashMap inherits AbstractMap class.	• Hashtable inherits Dictionary class.

Structuring Data with Java
(Sorting a Collection)

Problem

- You put your data into a collection in random order that doesn't preserve the order, and now you want it sorted.

Solution

- Use the static method Arrays.sort() or Collections.sort(), optionally providing a Comparator.

Arrays.sort()

```
public class ArraySortTest {
public static void main(String[] args)
{
Double[] arr= {5.5,1.5,3.5,2.5};
Arrays.sort(arr);
for(Double x:arr)
System.out.println(x);
}}
```

Collection.sort()

```
import java.util.Collections;
public class CollectionsSortTest {
public static void main(String[] args) {
List<String> ts=new ArrayList<String>();
ts.add("Subrat");
ts.add("Kumar");
ts.add("Nayak");
Collections.sort(ts);
System.out.println(ts);
}}
```

Structuring Data with Java
(Sorting a Collection)

Problem

- Your data needs to be sorted, but you don't want to stop and sort it periodically.

Solution

- Not everything that requires order requires an explicit *sort* operation. Just keep the data sorted at all times.

Discussion

- You can avoid the overhead and elapsed time of an explicit sorting operation by ensuring that the data is in the correct order at all times, though this may or may not be faster overall, depending on your data and how you choose to keep it sorted. You can keep it sorted either manually or by using a **TreeSet or a TreeMap**.
- One point to note is that if you have a Hashtable or HashMap, you can convert it to a TreeMap, and therefore get it sorted, just by passing it to the TreeMap constructor:

```
TreeMap sorted = new TreeMap(unsortedHashMap);
```

Structuring Data with Java

(Mapping with Hashtable and HashMap)

```
TreeSet<String> theSet = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
theSet.add("Gosling");
theSet.add("da Vinci");
theSet.add("van Gogh");
theSet.add("Java To Go");
theSet.add("Vanguard");
theSet.add("Darwin");
theSet.add("Darwin"); // TreeSet is Set, ignores duplicates.
System.out.printf("Our set contains %d elements", theSet.size());
// Since it is sorted we can easily get various subsets
System.out.println("Lowest (alphabetically) is " + theSet.first());
// Print how many elements are greater than "k"
// Should be 2 - "van Gogh" and "Vanguard"
System.out.println(theSet.tailSet("k").toArray().length +
" elements higher than \"k\"");
// Print the whole list in sorted order
System.out.println("Sorted list:");
theSet.forEach(name -> System.out.println(name));
```

Structuring Data with Java
(Finding an Object in a Collection)

Problem

- You need to see whether a given collection contains a particular value.

Solution

- Ask the collection if it contains an object of the given value.

Discussion

- if the collection is prepared by another part of a large application, or even if you've just been putting objects into it and now need to find out if a given value was found, this recipe's for you. There is quite a variety of methods, depending on which collection class you have.

Table 7-4. Finding objects in a collection

Method(s)	Meaning	Implementing classes
`binarySearch()`	Fairly fast search	`Arrays, Collections`
`contains( )`	Search	`ArrayList, HashSet, Hashtable, LinkList, Properties, Vector`
`containsKey( ), containsValue( )`	Checks if the collection contains the object as a `Key` or as a `Value`	`HashMap, Hashtable, Properties, TreeMap`
`indexOf( )`	Returns location where object is found	`ArrayList, LinkedList, List, Stack, Vector`
`search( )`	Search	`Stack`

Structuring Data with Java
(Converting a Collection to an Array)

Problem

- You have a Collection but you need a Java language array.

Solution

- Use the Collection method toArray().

Example:

```
public static void main(String[] args) {
// TODO Auto-generated method stub
List<String> list = new ArrayList<>();
list.add("Blobbo");
list.add("Cracked");
list.add("Dumbo");
// Convert a collection to Object[], which can store objects // of any type.
Object[] ol = list.toArray();
System.out.println("Array of Object has length " + ol.length);
String[] sl = (String[]) list.toArray(new String[0]);
System.out.println("Array of String has length " + sl.length);
}
```

Structuring Data with Java
(Stack)

Problem

- You need to process data in "last-in, first-out" (LIFO) or "most recently added" order.

Solution

- Write your own code for creating a stack; it's easy. Or, use a java.util.Stack.

Discussion

- This is a common data structuring operation and is often used to reverse the order of objects. The basic operations of any stack are push() (add to stack), pop() (remove from stack), and peek() (examine top element without removing).

```
public class ArrayListTest {

public static void main(String[] args) {

// TODO Auto-generated method stub

Stack<Integer> st=new Stack<Integer>();

st.push(4);

st.push(5);

st.push(6);

System.out.println(st.peek());
System.out.println(st);
while(!st.isEmpty())
System.out.println(st.pop());
System.out.println(st);
}}
```

Structuring Data with Java
(Multidimensional Structures)

Assignment

End of Session

Input and Output

- Java I/O (Input and Output) is used to process the input and produce the output.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- A stream is a sequence of data. In Java, a stream is composed of bytes. It’s called a stream because it is like a stream of water that continues to flow.

- In Java, 3 streams are created for us automatically. All these streams are attached with the console.
- 1. System.out: standard output stream
- 2. System.in: standard input stream
- 3. System.err: standard error stream

- OutputStream
- Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.
- InputStream
- Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

- Byte Streams
- Java byte streams are used to perform input and output of 8-bit bytes.
- Many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream.

- Character Streams
- Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode.
- Java is designed to be used internationally, and eight bits is simply not enough to handle the many different character sets used around the world.
- Script-based languages like Arabic and Indian languages, and pictographic languages like Chinese and Japanese, each have many more than 256 characters, the maximum that can be represented in an eight-bit byte.
- The unification of these many character code sets is called, not surprisingly, Unicode.

- Reading Passwords Without Echoing
- Use the System.console() method to obtain a Console object, and use its methods.
- Printing with Formatter and printf
- Use Formatter for printing values with fine-grained control over the formatting.
- format (" %1 $04d - the year of %2 $f " , 1956 , Math . PI)

- The StreamTokenizer class in java.util provides slightly more capabilities for scanning a file. It reads characters and assembles them into words, or tokens.
- It returns these tokens to you along with a type code describing the kind of token it found.
- Four predefined types (StringTokenizer.TT_WORD , TT_NUMBER, TT_EOF, or TT_EOL for the end-of-line)

- The Scanner class lets you read an input source by tokens.
- StreamTokenizer class bears some resemblance to the C-language scanf() function, but in the Scanner you specify the input token types by calling methods like nextInt() , nextDouble() , and so on.

- To read a text file, you’d create, in order, a FileReader and a BufferedReader.
- To write a file a byte at a time, you’d create a FileOutputStream and probably a BufferedOutputStream for efficiency.

- Lets say you want to redirect all those standard out messages in a File. System class provides some useful API to re-assign “standard” input, output and error streams.
- setErr(PrintStream err): Reassigns the “standard” error output stream.
- setIn(InputStream in): Reassigns the “standard” input stream.
- setOut(PrintStream out): Reassigns the “standard” output stream.

- Thank You.

Directory and Filesystem Operations

- The File class gives you the ability to list directories, obtain file status, rename and delete files on disk, create directories, and perform other filesystem operations.
- Many of these would be considered “system programming” functions on some operating systems; Java makes them all as portable as possible.
- File’s methods return true indicating the operation succeeded, or return false otherwise. They also throw a SecurityException if the user doesn’t have appropriate permission.

- The File class has a number of “informational” methods.
- To use any of these, you must construct a File object containing the name of the file it is to operate upon.
- It should be noted up front that creating a File object has no effect on the permanent filesystem;
- It is only an object in Java’s memory.
- You must call methods on the File object in order to change the filesystem; as we’ll see, there are numerous “change” methods, such as one for creating a new (but empty) file, one for renaming a file, etc., as well as many informational methods.

- You could easily create a new file by constructing a FileOutputStream or FileWriter. But then you'd have to remember to close it as well.
- Sometimes you want a file to exist, but you don't want to bother putting anything into it.
- This might be used, for example, as a simple form of interprogram communication: one program could test for the presence of a file and interpret that to mean that the other program has reached a certain state.

- The renameTo() method requires not the name you want the file renamed to, but another File object referring to the new name.
- So to rename a file you must create two File objects, one for the existing name and another for the new name.
- Then call the renameTo() method of the existing name’s File object, passing in the second File object.

- Use a java.io.File object’s delete() method; it deletes files (subject to permissions) and directories (subject to permissions and to the directory being empty).
- The java.io.File.createTempFile(String prefix, String suffix, File directory) method creates a new empty file in the specified directory.
- deleteOnExit() method is called to delete the file created by this method.

- prefix − The prefix string defines the files name; must be at least three characters long
- suffix − The suffix string defines the file’s extension; if null the suffix ".tmp" will be used
- directory − The directory in which the file is to be created. For default temporary file directory null is to passed
- createTempFile() creates a file with a unique name (in case several users run the same program at the same time on a server) and deleteOnExit() arranges for any file (no matter how it was created) to be deleted when the program exits.

- Create a single directory.
- For example, if /home/ian exists and is a directory, the calls:
- new File("/home/ian/bin").mkdir( );
- new File("/home/ian/src").mkdir( );
- Create multiple directories
- new File("/home/ian/once/twice/again").mkdir( );

- Thank You

Multithreading in Java

- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
- Each part of such program is called a thread. So, threads are light-weight processes within a process.
- Threads can be created by using two mechanisms :

  1. Extending the Thread class
  2. Implementing the Runnable Interface

Thread creation by extending the Thread class

- We create a class that extends the java.lang.Thread class.
- This class overrides the run() method available in the Thread class. A thread begins its life inside run() method.
- We create an object of our new class and call start() method to start the execution of a thread.
- Start() invokes the run() method on the Thread object.

```
// Java code for thread creation by extending
// the Thread class


class Multithread extends Thread
{
public void run()
{
try
{
// Displaying the thread that is running
System.out.println ("Thread " +
Thread.currentThread().getId() +
" is running");

}
catch (Exception e)
{
// Throwing an exception
System.out.println ("Exception is caught");
}
}
}
```

```
public class  MultithreadingDemo

{

public static void main(String[] args)

{

int n = 8; // Number of threads

for (int i=0; i<n; i++)

{

Multithread object = new Multithread();

object.start();

}

}

}
```

Thread creation by implementing the Runnable Interface

- We create a new class which implements java.lang.Runnable interface and override run() method.
- Then we instantiate a Thread object and call start() method on this object.
- // Java code for thread creation by implementing the Runnable Interface

```
import java.lang.*;
class MultithreadingDemo1 implements Runnable
{
public void run()
{
try
{
// Displaying the thread that is running
System.out.println ("Thread " +
Thread.currentThread().getId() +
" is running");

}
catch (Exception e)
{
// Throwing an exception
System.out.println ("Exception is caught");
}
}
```

```
// Main Class
public class Multithread2
{
public static void main(String[] args)
{
int n = 8; // Number of threads
for (int i=0; i<n; i++)
{
Thread object = new Thread(new MultithreadingDemo1());
object.start();
}
}
}
```

Thread Class vs Runnable Interface

- If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
- We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

Graphical User Interfaces

- ava has had windowing capabilities since its earliest days. The first version made public was the Abstract Windowing Toolkit, or AWT. Because it used the native toolkit components, AWT was relatively small and simple.
- The second major implementation was the Swing classes, released in 1998 as part of the Java Foundation Classes. Swing is a full-function, professional-quality GUI toolkit designed to enable almost any kind of client-side GUI-based interaction.

- AWT lives inside, or rather underneath, Swing, and, for this reason, many programs begin by importing both java.awt and javax.swing.
- Containers and Components: There are two types of GUI elements:
- Component: Components are elementary GUI entities, such as Button, Label, and TextField.
- Container: Containers, such as Frame and Panel, are used to hold components in a specific layout (such as FlowLayout or GridLayout). A container can also hold sub-containers.

- Container Classes Each GUI program has a top-level container. The commonly-used top-level containers in AWT are Frame, Dialog and Applet:
- A Frame provides the "main window" for your GUI application. It has a title bar (containing an icon, a title, the minimize, maximize/restore-down and close buttons), an optional menu bar, and the content display area.

- An AWT Dialog is a "pop-up window" used for interacting with the users. A Dialog has a title-bar (containing an icon, a title and a close button) and a content display area, as illustrated.
- The Swing JFrame is more complex—it comes with not one but two containers already constructed inside it.

- The ContentPane is the main container; you should normally use it as your Jframe’s main container.
- The GlassPane has a clear background and sits over the top of the ContentPane ; its primary use is in temporarily painting something over the top of the main ContentPane.

- You can add any number of components (including containers) into this existing container, using the ContentPane add() method.
- The container classes such as Panel have the capability to contain a series of components, but you can arrange components in a window in many ways. Rather than clutter up each container with a variety of different layout computations, the designers of the Java API used a sensible design pattern to divide the labor.

- A layout manager is an object that performs the layout computations for a container. The AWT package has five common layout manager classes, and Swing has a few more.
- If your JFrame is full of well-behaved components, you can set its size to be “just the size of all included components, plus a bit for padding,” just by calling the pack() method, which takes no arguments.

- The pack() method goes around and asks each embedded component for its preferred size.
- The JFrame is then set to the best size to give the components their preferred sizes as much as is possible. If not using pack() , you need to call the setSize() method, which requires either a width and a height, or a Dimension object containing this information.

- The JTabbedPane class acts as a combined container and layout manager. It implements a conventional tab layout.
- The Swing JFrame has a setDefaultCloseOperation() method, which controls the default behavior. You can pass it one of the values defined in the Swing WindowConstants class:
- WindowConstants.DO_NOTHING_ON _CLOSE
- WindowConstants.HIDE_ON_CLOSE
- WindowConstants.DISPOSE_ON_CLOSE
- WindowConstants.EXIT_ON_CLOSE

- Use a JOptionPane method to show a prebuilt dialog.
- The JSpinner class lets the user click up or down to cycle through a set of values. The values can be of any type because they are managed by a helper of type SpinnerModel and displayed by another helper of type SpinnerEditor . A series of predefined Spinner-Models handle Number’s, Date’s, and List’s (which can be arrays or Collections).

- The JFileChooser dialog provides a fairly standard file chooser. It has elements of both a Windows chooser and a Mac chooser, with more resemblance to the former than the latter.
- Each FileFilter subclass instance passed into the JFileChooser’s addChoosableFileFilter() method becomes a selection in the chooser’s Files of Type: choice. The default is All Files (.).

- The JOptionPane class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box. These dialog boxes are used to display information or get input from the user.
- The Swing components that display text, such as JLabel , format the text as HTML—instead of as plain text—if the first six characters are the obvious tag <html> .

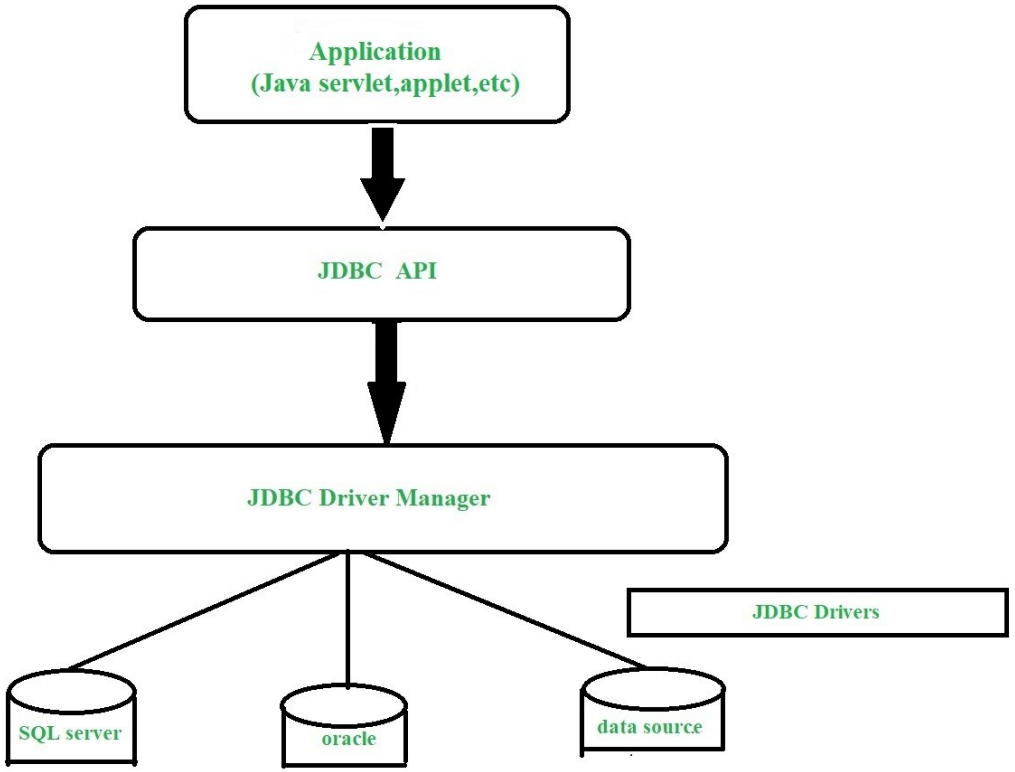
- Thank You

Java Database Access

- JDBC or Java Database Connectivity provides a standard abstraction(that is API or Protocol) for java applications to communicate with various databases.
- It is used to write programs required to access databases.
- JDBC along with the database driver is capable of accessing databases and spreadsheets.
- The enterprise data stored in a relational database(RDB) can be accessed with the help of JDBC APIs.

- JDBC is an API(Application programming interface) which is used in java programming to interact with databases.
- The classes and interfaces of JDBC allows application to send request made by users to the specified database.
- Enterprise applications that are created using the JAVA EE technology need to interact with databases to store application-specific information.

- So, interacting with a database requires efficient database connectivity which can be achieved by using the ODBC(Open database connectivity) driver.
- This driver is used with JDBC to interact or communicate with various kinds of databases such as Oracle, MS Access, Mysql and SQL server database.


Application
(Java servlet,applet,etc)
JDBC API
JDBC Driver Manager
JDBC Drivers
SQL server
oracle
data source

- Application: It is a java applet or a servlet which communicates with a data source.
- The JDBC API: The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows: DriverManager, Driver, Connection, Statement, ResultSet

- DriverManager: It plays an important role in the JDBC architecture.It uses some database-specific drivers to effectively connect enterprise applications to databases.
- JDBC drivers: To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

- JDBC API uses JDBC drivers to connect with the database. There are four
- types of JDBC drivers.
- JDBC-ODBC Bridge Driver
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

- Why Should We Use JDBC?
- Before JDBC, ODBC API was the database API to connect and execute the query with the database.
- ODBC API uses ODBC driver, which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

- Hence, JDBC API can be used to handle database using Java program and can perform the following activities.
- Connect to the database
- Execute queries and update statements to the database
- Retrieve the result received from the database.

JDBC-ODBC bridge driver

- The JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
- The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC
- function calls. This is now discouraged because of thin driver.

Native-API driver

- The Native API driver uses the client-side libraries of the database.
- The driver converts JDBC method calls into native calls of the database API.
- It is not written entirely in java.

• Network Protocol driver

- The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol.
- It is fully written in java.

Thin Driver

- The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.
- Better performance than all other drivers.

JDBC Setup and Connection

- To access a database via JDBC, use Class.forName() and DriverManager.getConnection()
- Prerequisite: Some knowledge about SQL, the universal language used to control relational database.
- Example:

  "SELECT * from table_name";

- To connect any java application with the database using JDBC, we require 5 steps.
- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

Java Network Programming

- There’s a lot of low-level stuff that needs to happen to work but the Java API networking package (java.net) takes care of all of that, making network programming very easy for programmers.

Client Side Programming

- Establish a Socket Connection
- To connect to other machine we need a socket connection
- A socket connection means the two machines have information about each other’s network location (IP Address) and TCP port
- The java.net.Socket class represents a Socket.

- To open a socket:
- Socket socket = new Socket(“127.0.0.1”, 5000)
- First argument – IP address of Server. (127.0.0.1 is the IP address of localhost, Second argument –
- TCP Port. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)where code will run on single stand-alone machine).

- To communicate over a socket connection, streams are used to both input and output the data.
- The socket connection is closed explicitly once the message to server is sent.
- A Java program for a Client

```
import java.net.*;
import java.io.*;

public class Client
{
    // initialize socket and input output streams
    private Socket socket            = null;
    private DataInputStream  input   = null;
    private DataOutputStream out     = null;
```

```
// constructor to put ip address and port
    public Client(String address, int port)
    {
        // establish a connection
        try
        {
            socket = new Socket(address, port);
            System.out.println("Connected");

            // takes input from terminal
            input  = new DataInputStream(System.in);

            // sends output to the socket
            out    = new DataOutputStream(socket.getOutputStream());
        }
```

```
catch(UnknownHostException u)
        {
            System.out.println(u);
        }
        catch(IOException i)
        {
            System.out.println(i);
        }

        // string to read message from input
        String line = "";
```

```
// keep reading until "Over" is input
        while (!line.equals("Over"))
        {
            try
            {
                line = input.readLine();
                out.writeUTF(line);
            }
            catch(IOException i)
            {
                System.out.println(i);
            }
        }
```

```
// close the connection
        try
        {
            input.close();
            out.close();
            socket.close();
        }
        catch(IOException i)
        {
            System.out.println(i);
        }
    }

    public static void main(String args[])
    {
        Client client = new Client("127.0.0.1", 5000);
    }
}
```

Server Programming

- Establish a Socket Connection
- To write a server application two sockets are needed
- A ServerSocket which waits for the client requests (when a client makes a new Socket())
- A plain old Socket socket to use for communication with the client.

- getOutputStream() method is used to send the output through the socket.
- After finishing, it is important to close the connection by closing the socket as well as input/output streams.

- A Java program for a Server

```
import java.net.*;
import java.io.*;

public class Server
{
    //initialize socket and input stream
    private Socket          socket   = null;
    private ServerSocket    server   = null;
    private DataInputStream in       =  null;

```

```
// constructor with port
    public Server(int port)
    {
        // starts server and waits for a connection
        try
        {
            server = new ServerSocket(port);
            System.out.println("Server started");

            System.out.println("Waiting for a client ...");

            socket = server.accept();
            System.out.println("Client accepted");
```

```
// takes input from the client socket
        in = new DataInputStream(
            new BufferedInputStream(socket.getInputStream()));

        String line = "";

        // reads message from client until "Over" is sent
        while (!line.equals("Over"))
        {
            try
            {
                line = in.readUTF();
                System.out.println(line);

            }
            catch(IOException i)
            {
                System.out.println(i);
            }
        }
```

```
System.out.println("Closing connection");

            // close connection
            socket.close();
            in.close();
        }
        catch(IOException i)
        {
            System.out.println(i);
        }
    }

    public static void main(String args[])
    {
        Server server = new Server(5000);
    }
}
```

- Server application makes a ServerSocket on a specific port which is 5000. This starts our Server listening for client requests coming in for port 5000.
- Then Server makes a new Socket to communicate with the client. socket = server.accept()
- The accept() method blocks(just sits there) until a client connects to the server.

- Then we take input from the socket using getInputStream() method. Our Server keeps receiving messages until the Client sends “Over”.
- After we’re done we close the connection by closing the socket and the input stream.
- To run the Client and Server application on your machine, compile both of them. Then first run the server application and then run the Client application.

- To run on Terminal or Command Prompt
- Open two windows one for Server and another for Client
- 1. First run the Server application as , $ java Server

  Server started

  Waiting for a client …
- 2. Then run the Client application on another terminal as, $ java Client
- It will show – Connected and the server accepts the client and shows,

  Client accepted

- 3. Then you can start typing messages in the Client window. Here is a sample input to the Client

  Hello

  I made my first socket connection

  Over
- Which the Server simultaneously receives and shows,

  Hello

  I made my first socket connection

  Over

  Closing connection

- Thank You