Core Java Programming

Core Java concepts : PART I

Installation | System and Java Commands | Java Environment

main() method | Command Line arguments

Data types | Basic I/O operations

Type promotions and conversions | parseInt() method

Java Installation

- Download and Install JRE (Java Runtime Environment)

  It is the Software platform to run java programs

- Download and Install JDK (Java Development Kit)

  These are the Software tools to develop java applications

--Official Websites--

www.oracle.com

www.java.com

System Commands

Recommendation:

Create a separate folder `Java` on `<D:> drive` or any other than <C:> preferably.

- `Start -> cmd` `// Command prompt window of windows OS`
- `set path=%path%;C:\Program Files\Java\jdk1.8.0\bin↵`
  `//Or the relevant path of java jdk as per installation`
- `D:↵` `// To switch to other drive`
- `cd Java↵` `// Change Directory to your folder`

1) **First step is a one time job, next operations / commands to run every time**
2) `set path` **command is not required if system path is already updated in system variables**

Java Commands

- To compile the Java program, use the command

  `javac ProgramName.java`↵

  Note that **`file name`** should match with the **`class name`** used in the program

- To run the java program, use the command

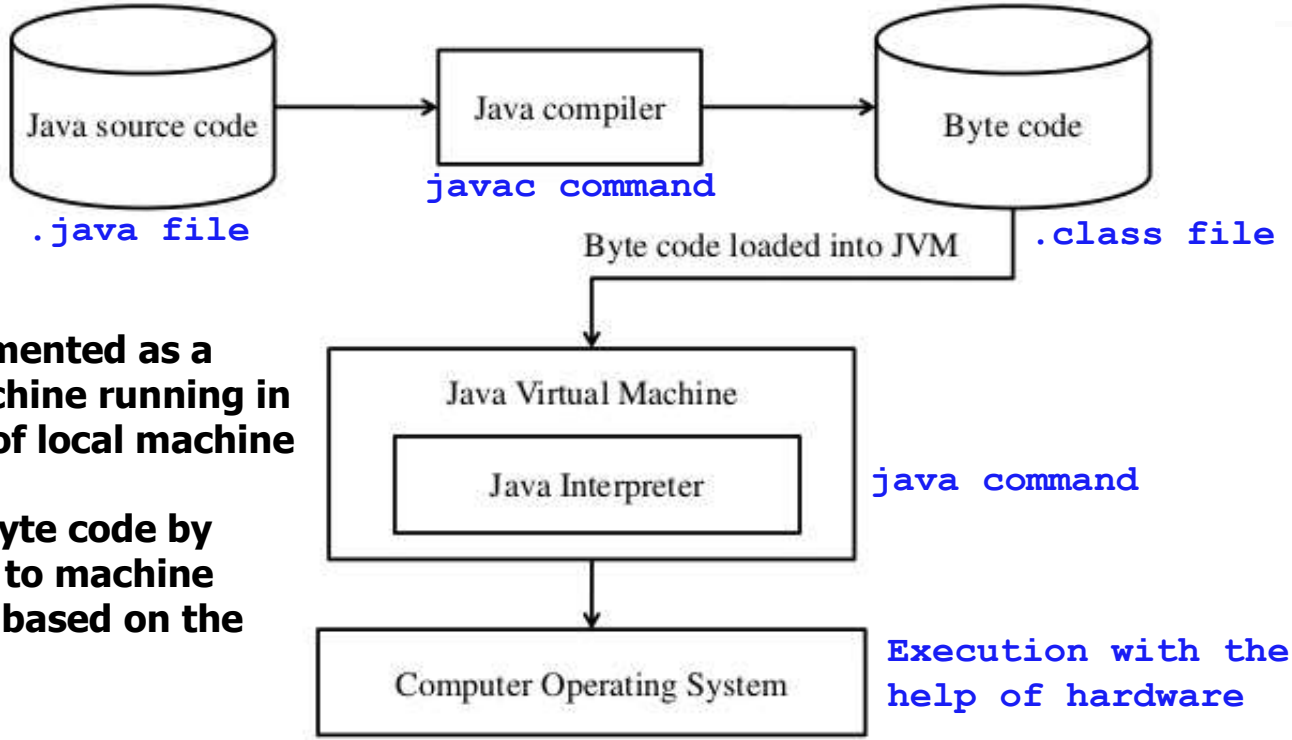
  `java ProgramName <args>`↵

  This command calls the **`main() method`** of the program. The parameters i.e. **`args`** are optional and these are the **`command line arguments`** passed to the **`main() method`** of the program.

Java Environment

- The source code file of java has `.java` extension
  - It is used to edit the source code and compile the program
  - `javac` command uses the source code as an input to the compiler
- The byte code file of java has `.class` extension
  - It is automatically created as an output of compiler
  - It stores the translated program in the form of byte code
  - `java` command use this file to run the program

Concept of JVM



- **JVM is implemented as a software machine running in the memory of local machine**
- **It executes byte code by translating it to machine specific code based on the platform**

Java Program

```
class FirstProgram                    // FirstProgram.java
{
        public static void main(String args[])
        {
                System.out.println("Hello World!");
        }
}
```

main() method of Java

It is always defined with the header

```
public static void main(String args[])
```

- `public` **means it can be accessed from outside the class**
- `static` **means it has a single instance running in the scope of class and JVM can call it as** `ClassName.main()`

```
//no object required for calling main()
```

- `void` **means it returns no value at the end of method**

main() method of Java

`String args[] :`

- **It is an array to store command line arguments**
- **`String` is an built-in `class` from the package `java.lang`**
- **It is not optional i.e. even though every program may not have command line arguments, Java makes it compulsory to mention this array.**
- **Size of array depends on number of arguments given from command line at run-time**

Java Naming Conventions

- Java uses **Camel Case** as a practice for writing names
- `ClassName` should be **like noun**. It starts with CAPITAL letter and first letter of every internal word within name is again CAPITAL

  `IterfaceName` also follow the same rules of classes
- `methodName` should be **like verb**. It starts with lowercase letter and first letter of each internal word as CAPITAL letter

Example,

Class names : `ComplexNumber, MyCity, CollegeInformation`

Method names : `findArea(), searchMax(), getValue(), setValue()`

Java Naming Conventions

- `variableName` and `objectName` should be meaningful to the context and follow the same lexical rules like method names
- CONSTANT is represented with ALL CAPITAL letters
- `packagename` is always written in all lowercase letters.

Example,

Variable and object names : `firstVal, myAccount, totalItems`

constant names : `MAX, MIN, NULL, SIZE`

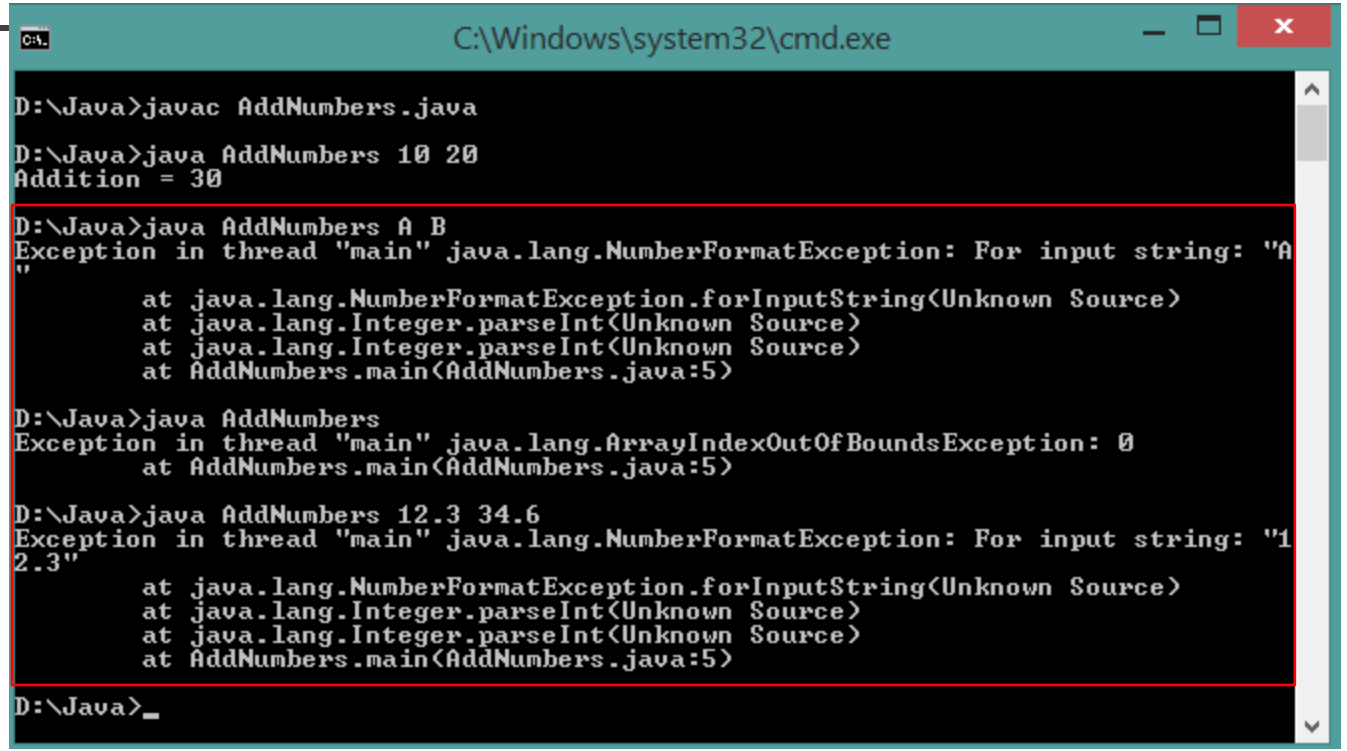
package names : `mypackage, project, course`

Command line arguments

```
class AddNumbers                        // String array to store
{                                          command line arguments
        public static void main(String args[])
        {
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);

                int c = a+b;                          // Accessing the
                                                         arguments
                System.out.println("Addition = "+c);
        }
}
```

Command line arguments



Possible errors at run-time

Command line arguments

Conclusions

- *Input 2 values : result displayed*
- *Input more than 2 values : addition of only first 2 values displayed, as other numbers are ignored.*
- *No or less input given : args[ ] access failed, throws an exception as encountered*
- *Input with wrong types : parse method throws the exception as encountered*
- *Exceptions are different run-time errors, the detailed concept to be explained later*

Command line arguments

Recall :

- **`args`** is a String array. It stores the command line argument values as different strings in the given sequence. Size of **`args`** is automatically set to the total number of arguments.

Understand :

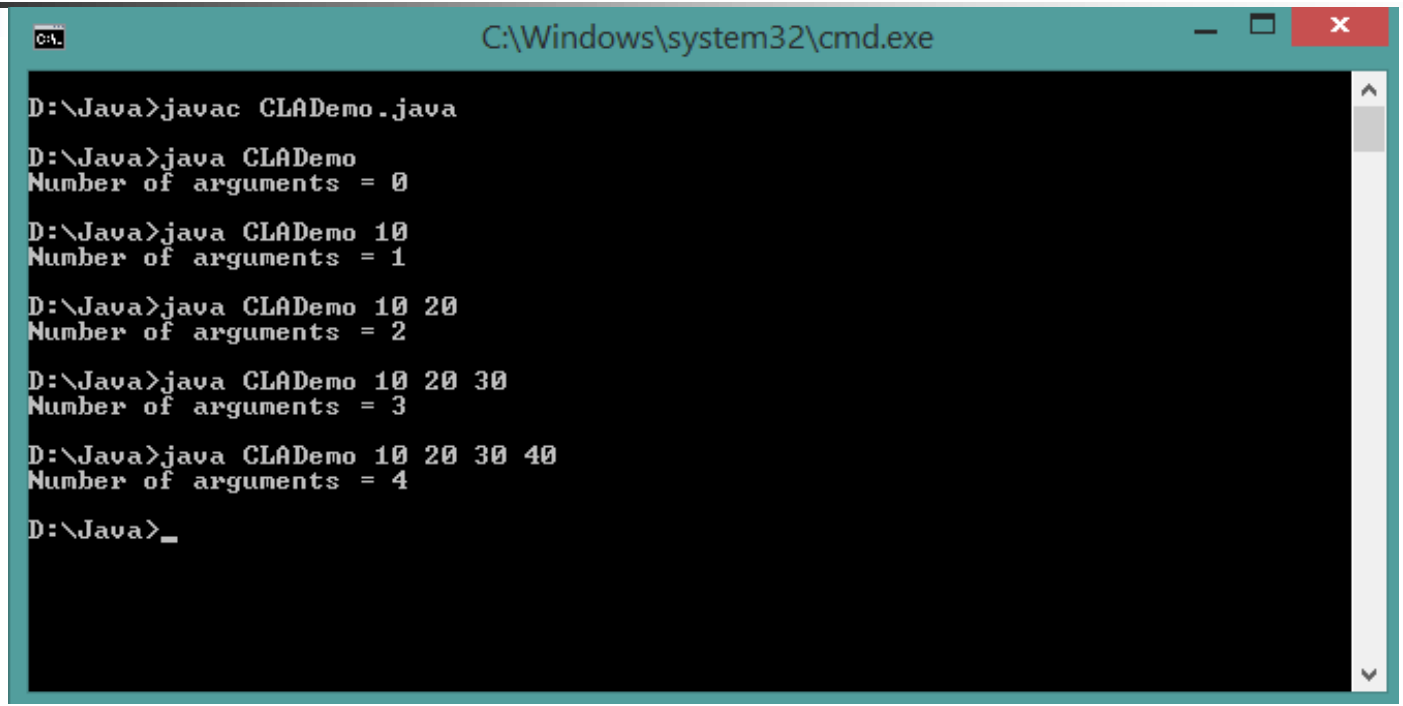
- **`args.length`** is used to find the size of the array during program execution (runtime)
- Values stored in **`args`** can be converted to other data types using corresponding **`parse`** methods.
- For example **`Integer.parseInt()`** method for **`int`** data type.

Command line arguments

```
// Program to demonstrate command line arguments and args.length
class CLADemo
{
    public static void main(String args[])
    {
        System.out.println("Number of arguments = "+args.length);
    }
}

// Remember: There is no sizeof() operator in Java and all primitive types
have fixed memory size across all platforms
```

Command line arguments



Primitive data types in Java

- Integer types
  - `byte` (1 byte)
  - `short` (2 byte)
  - `int` (4 byte)
  - `long` (8 byte)

  Java uses signed value by default

- Character type
  - `char` (2 byte)

  Java uses 16-bit Unicode for characters

- Floating-point types
  - `float` (4 byte)
  - `double` (8 byte)

  Java uses IEEE 754 format

- Boolean type
  - `boolean`

Default data types

- For floating point numbers java considers **`double`** as default data types. That is whenever we mention a floating point number in the program it will be treated as a constant of **`double`** data type
- Consider the statement, float x = 23.45;
- **It gives error** because variable type is float but value type is double
- To make it correct apply **suffix** letter **f** to the value
- float x = 23.45f; *// Now it is correct*

because user has explicitly mentioned that the value should be taken as a float value

boolean data type

- It is a special data type in Java to deal with logical values **`true`** or **`false`**
- We can declare and use **`boolean`** variables to store these values. For example, **`boolean b = true;`**
- We can also use boolean values directly as a result of condition as possible. For example, **`if(Character.isDigit(c)){…}`**
- Here **`isDigit()`** method returns **`true`** or **`false`** depending on value of variable c, and the return value is directly input to **`if`** statement
- **`boolean`** values are directly processed by logical operators

Basic I/O operations

- For inputs, use `Scanner` class from `java.util` package to create an object connected to input stream of Java `System.in`
- `import java.util.*;` //use at the start of program
- `Scanner sc = new Scanner(System.in);` //create object
- Now the object can be used with following methods

```
sc.nextInt()   // to read an int value
sc.nextFloat() // to read a float value
sc.next()      // to read a string
sc.nextLine()  // to read string with blank spaces
```

Basic I/O operations

- For output, use `print()` or `println()` methods as,

```
System.out.print("welcome");      //cursor remains on same
System.out.println("welcome");    //cursor goes to next line
```

- We can insert additional variables and join using + operator

```
System.out.print("Answer = "+var);
System.out.println("Square root of "+n+" is : "+s);
```

- The + operator performs string concatenation
- We can also use escape sequence characters like \n , \t

Type promotions and conversions

- Smaller type can be automatically promoted to larger type in same category.
- For example **`byte`** can be promoted to **`short`** or **`int`**, **`float`** can be promoted to **`double`**.
- Similarly **`char`** can be type-casted to **`int`** and **`int`** can be type-casted to **`float`** automatically.
- Larger type to smaller type gives an error. Still, if required, it can be done using **explicit type casting or conversion**.

Type promotions and conversions

- For example,

```
byte b;
int i = 100;
b = (byte) i;
```

```
float f;
double d = 12.34;
f = (float) d;
```

```
float f = 12.34f;
int i;
i = (int) f;
// explicit type casting
```

```
float f;
int i = 100;
f = i;
// auto type casting
```

**** Refer the program TypeConversionExample.java***

Type promotions and conversions

- For example,

```
char c;
int i = 65;
c = (char)i;
```

```
char c = 'A';
int i;
i = c;
// auto type casting
```

```
float f;
int i = 100;
f = i;
// auto type casting
```

```
double d = 12.34;
float f;
f = (float) d;
```

**** Refer the program TypeConversionExample.java***

parseInt() method

- It is **`static`** method from **`Integer`** class. It is called as a wrapper class.
- it is invoked as **`Integer.parseInt()`**
- It is used to convert a **`String`** value to **`int`** data type
- Since command line arguments are stored as String values it is required to convert those values to appropriate data type before processing
- Similar to **`parseInt()`** method, we have another methods from other wrapper classes for other data types also
  - **`Float.parseFloat()`**
  - **`Double.parseDouble()`**

Core Java Programming

Core Java concepts : PART II

Operators | Control statements | Package, class and method | Access specifiers | Abstraction and Encapsulation | Constructors | creating objects | this reference | wrapper classes | method overloading | Arrays

Operators

- **Arithmetic operators**
  `+ - * / % ++ --`
- **Relational operators**
  `< > <= >= ++ !=`
- **Logical operators**
  `&& || !`
- **Bit-wise operators**
  `& | ~ ^ << >>`
- **Condition operator** (only ternary operator)
  `?:`

Recall:

1. **Unary and binary operators**
2. **Operator precedence and associativity**

Control statements

- **Selection or decision control statement**

  `if…else`

- **Iteration or loop control statements**

  `while, do…while, for`

- **Case control statement**

  `switch…case`

- **Additional control statements**

  `break, continue`

Recall:

1. **Syntax of statements**
2. **Nested statements**

Package, class and method

- In Java we use different packages. It is used to group related classes. It is like a container which contains different classes
- Packages can be of 2 types
  - built-in // packages from Java API

    OR
  - user defined // created by the programmer

Package, class and method

- **Some commonly used packages in Java are,**
  - **java.lang (contains the classes which define the core language)**
  - **java.io (contains the classes required for I/O operations)**
  - **java.util (contains the utility classes for other applications)**
  - **java.net (contains the classes for network programming)**
  - **java.awt (Abstract Window Toolkit, the classes required for GUI)**

And there are many more packages in Java API

Package, class and method

`import` statement

- It is used to import the required class(Or classes) in the program
- Syntax:
  - `import packagename.ClassName;` `//general form`
  - Imports a particular class from the corresponding package

Example,

```
import java.util.Scanner;
       // package name   // ClassName
```

Package, class and method

`import` statement

- **To import multiple classes, as we prefer to import ALL classes from the package**
- **Syntax:**
  - `import packagename.*;  //general form`
  - **Imports ALL classes from the corresponding package**

Example,

```
import java.util.*;
import java.io.*;
```

Package, class and method

`import` statement

- `java.lang` is the called as default package as it defines the core language classes.
- For Example, String, Integer, System, Exception etc.
- All classes from this package are automatically accessible in every java program.
- `import` statement is not needed for the this package

Package, class and method

`class`

- **`class` is like a container which defines attributes of the objects (variables) and a set of methods to perform the operations.**
- **Class decides how the objects will be look like, hence it is like a prototype or blueprint of objects.**
- **Since Java is a pure Object Oriented Programming language, even `main()` method is also inside some class**
- **class can contain two types of methods as `static` OR non-static.**

Package, class and method

`class`

- Methods can be invoked (i.e. called) from the program where `static` methods are invoked with class reference and non-static methods are invoked with object reference
- For example,

```
Integer.parseInt(String s)  // static method, class reference used
sc.nextInt();               // non-static method from Scanner class
                            // object name used on left side as reference
```

Package, class and method

`class`

- **The Java API classes (built-in classes) are stored in different packages as explained earlier.**
- **User defined classes can be stored in user defined packages also.**
- **Class members are in the have different qualifiers like `static, abstract, final` etc. and visibility specifiers**

Package, class and method

`methods`

- The methods are used to accomplish some task and those are defined inside the class.
- Each method must be member of some class.
- As seen earlier, based on the scope of execution, `class` can contain two types of methods as
  - `static`
  - non-static.

Package, class and method

methods

- **Syntax of definition**

```
return_type methodName(parameters)
{

        // TO do: add the code here
}
```

Additional qualifiers like `static, abstract, final` and `visibility specifiers` are mentioned before `return_type`

Package, class and method

`static` methods	Non-static methods
`static` keyword is used	No separate keyword required
Executes in the **scope of class**	Executes in the **scope of object**
To call the methods, use `ClassName.methodName( );`	To call the methods, use `objectName.methodName( );`
No object required to call the method	**At least one object** is required to be created to call the method

Access specifiers

- There are three access specifiers keywords in Java also know as visibility specifiers.
- `private` members can be accessed within the owner class only
- `protected` members can be accessed in the owner class and its subclass as well as other classes in the same package
- `public` members can be access from outside class also.
- The actual visibility levels depend on package also.

Access specifiers

- **The visibility levels are as follows and Security level decreases from top to bottom**

Specifiers	Visibility outside the class
`private`	none
No modifier (default)	Classes in the package
`protected`	Classes in the package and subclasses in same or other packages
`public`	All other classes

Access specifiers

- The following table shows access permissions precisely

specifiers	class	package	sub-class (inheritance)	Other classes / world
`private`	Y	N	N	N
No modifier (default)	Y	Y	N	N
`protected`	Y	Y	Y	N
`public`	Y	Y	Y	Y

https://docs.oracle.com/javase/tutorial/java/javaOO/index.html

Abstraction and Encapsulation

- Access specifiers are used for class members and NOT to local variables inside methods
- A class defines a set of attributes (variables) of objects and the methods (services) that can access the data inside the objects. This is called as Encapsulation (also called as data hiding)
- Member variables (non-static) are instantiated in the scope of objects when the objects are created. This is called as data abstraction
- The `static` member variables have only one instance and it is in the scope of class (NOT in the scope of object)

Abstraction and Encapsulation

- **The method is executed from the class when it called.**
- **The scope of method execution depends on whether it is static OR non-static. This is called as scope binding.**
- **As seen earlier, static methods run in the scope of class hence require no object to call it.**
- ***Recall:* Why `main()` method is `static`?**
- **Non-static methods run in the scope of object and access the variables that reside in the scope of that object only.**
- **This is called as service abstraction.**

Constructors in Java

- It is used to initialize the objects when those are created.
- It is a `public` member method of the class.
- It has no return value specification. (DO NOT mention `void` also)
- Constructor method name is exactly identical to class name.
- It can be overloaded. It means, one class can have multiple constructors.
- During inheritance constructors are executed in the order of derivation. (To be discussed later)
- Constructor can not be `static` or `abstract` method

Constructors in Java

```
class Data
{
        private int x;
        public Data()  //default constructor
        {
                x = 0;
        }
        public Data(int a)     //parameterized constructor
        {
                x = a;
        }
        public Data(Data d)    //copy constructor
        {
                this.x = d.x;
        }
}
```

`this` reference

- Keyword `this` is used as an implicit reference
- Internally it points to the same object on which the method is invoked

- `this.variableName` refers to some attribute of same object
- `this.methodName()` will invoke some non static method internally on the same object.

Constructors in Java

```
class ConstructorDemo
{
        public static void main(String args[])
        {
                Data d = new Data();
                // calls default constructor

                Data d1 = new Data(5);
                //calls parameterized constructor

                Data d2 = new Data(d1);
                // calls copy constructor
        }
}
```

Constructors in Java

- **Three types of constructor**
  - **Default constructor : To assign default initial values inside the object**
  - **Parameterized constructor : To assign user wanted specific values to the instance variables in the object**
  - **Copy constructor : To copy the instance values of one existing object to the other object**

Creating the objects

- **Syntax:**

```
ClassName obj = new ClassName( );
```

- **`new` operator (keyword) is used to perform memory allocation**
- **Constructor is used to perform initialization of data**
- **If required use parameters,**

```
ClassName obj = new ClassName(<args>);
```

Wrapper classes in Java

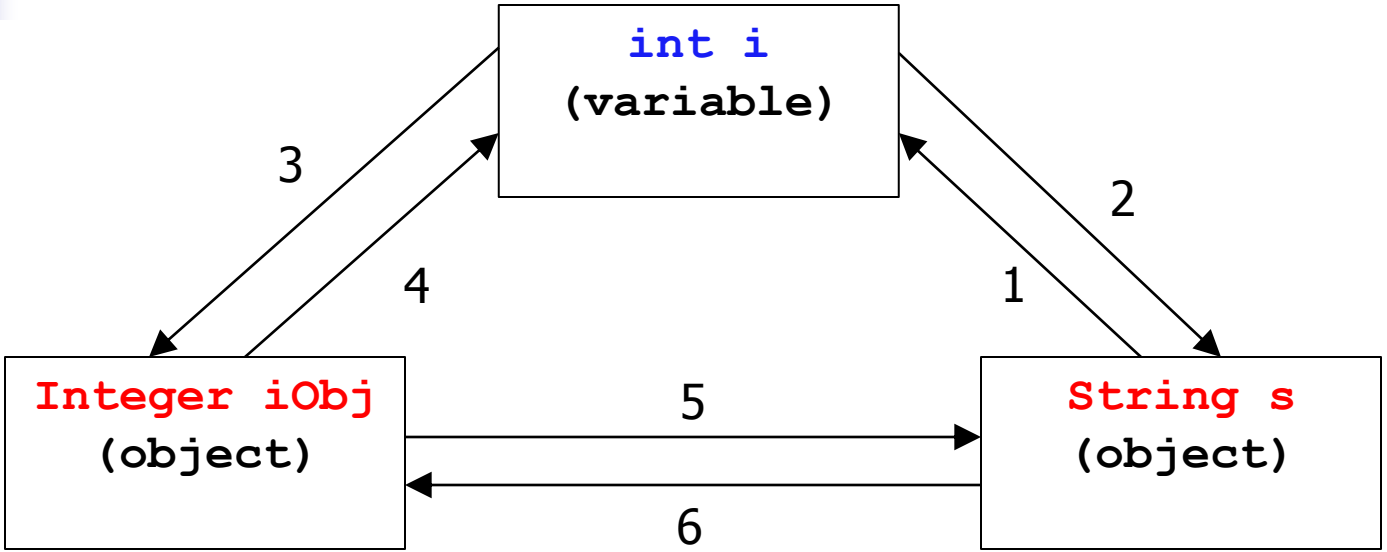
For each primitive type there is equivalent class in Java. These classes are called as wrapper classes. These classes are in the package `java.lang`

Primitive type	Wrapper class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Wrapper classes in Java

- **wrapper classes provide the access to some standard methods to perform data operations of the specific type.**
- **For example, Parse methods are used to convert String to corresponding type as seen earlier.**
- **`valueOf()` method can be used to convert data from its internal form to a String**

Wrapper classes in Java



Wrapper classes in Java

1. Use `parseInt()` method (static) as

```
int i = Integer.parseInt(s);
```

2. Use `toString()` as static method

```
String s = Integer.toString(i);
```

Also `valueOf()` method (static) can be used

```
String s = String.valueOf(i);
```

3. Use constructor to initialize the object

```
Integer iObj = new Integer(i);
```

Wrapper classes in Java

4. Use `intValue()` method (non static) as

```
int i = iObj.intValue(s);
```

5. Use `toString()` as non static method

```
String s = iObj.toString();
```

Also `valueOf()` method (static) can be used

```
String s = String.valueOf(iObj);
```

6. Use `valueOf()` method (static) as

```
Integer iObj = Integer.valueOf(s);
```

Wrapper classes in Java

- `toString()` method is used to convert other data to String type
- It can be used as either `static` or *non static* method *(overloading)*

  For Example,

  ```
  Integer.toString(i) OR iObj.toString()
  ```

- `valueOf()` is a static method used to convert data from specific *object* type into a String

  For example,

  ```
  String.valueOf(i)
  Integer.valueOf(i)
  Float.valueOf(f)
  ```

Method overloading

- When a method is defined more than one time it is called as method overloading (polymorphism)
- In this case `methodName` remains same but parameters are different.

- If different number of parameters are used data types do not cause any conflict.
- If same number of parameters, data types must be clearly distinguishable

Method overloading

- **Example**

```
public static float area(float l, float b)
{
        return(l*b);
}
public static float area(float r)
{
        return(3.142f * r *r);
}
```

Refer the complete program for more understanding

Method overloading

- **Example *continued…***

```
public static void main(String args[])
{
        System.out.println("Rectangle area = "+area(4,5));
        System.out.println("Circle area = "+area(4));
}

// different response from same method (logically)
// This is called as polymorphism
```

***Refer the complete program for more understanding**

Arrays in Java

- **Basic Syntax:**

```
data_type array_name [];            //declaration
array_name = new data_type[size];   //allocation
//Size can be denoted by constant or variable
```

- **Declaration can be also as,**

```
data_type[] array_name;
```

- **Similarly the 2 steps can be merged together as,**

```
data_type array_name[] = new data_type[size];
```

Arrays in Java

Recall:

- `array_name.length` is used to find the **logical size** of array at runtime (the number of elements present in the array)
- **Physical size** depends on the `data_type` also and it is the actual amount of memory (bytes) used by the array
- Array can be initialized in the declaration itself if needed
- Example, `int n[]={10,20,30,40,50};`

Arrays in Java

- **Example,**

```
int a[] = new int [10];
char[] c = new char[20];
float fa[] = new float[n];    // n is another int variable
                                 denotes the required size
String s[] = new String[10]; // objects array can also be
                                created
```

Differentiate between above statement and the next!!!

```
String s = new String("Program");
```

Arrays in Java

- Array elements are auto-indexed `from 0 to size-1`

  Consider, `int a[] = new int [10];`

  Then, `a[0]` is the first element and `a[9]` is the last

  Consider, `float fa[] = new float[n];`

  Then, `fa[0]` is the first element and `fa[n-1]` is the last

- Any access to array location outside such valid boundaries throw `ArrayIndexOutOfBoundsException`

Arrays in Java

- **2-Dimensional arrays are declared as,**

```
data_type array_name [][];                    //declaration
array_name = new data_type[rows][columns]; //creation
```

- **These 2 steps also can be merged in a single statement.**
- **In Java we can initialize 2-D arrays in a dynamic way also.**

  Example,

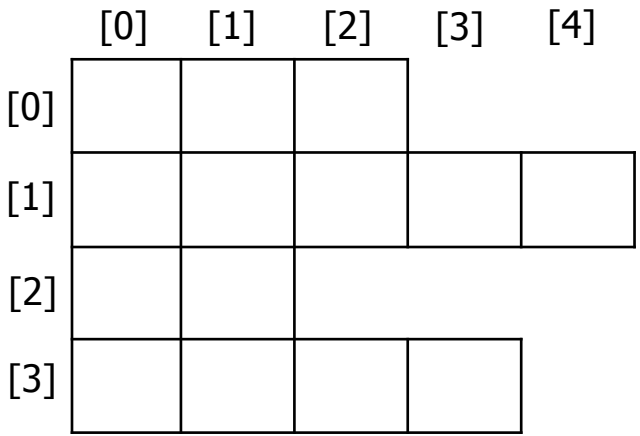
```
int n[][] = {{1,2,3},{1,2,3,4,5},{1,2}};
// 3 rows, each containing different number of values
```

Arrays in Java

- **In Java we can create 2-D arrays in a dynamic way also.**

  Example,

```
int n[][] = new int [4][]; //only number of rows defined
n[0] = new int[3];
n[1] = new int[5];
n[2] = new int[2];
n[3] = new int[4];
```



Core Java Programming

Core Java concepts : PART III

Strings | Vectors | Inheritance | method overriding | abstract methods and classes | interfaces | final methods and final classes | finalize() method | Constructors in Inheritance

Automatic garbage collection

Strings

- **String is a class in Java defined in `java.lang`**
- **String object can be created as,**

```
String s = new String("Java");
```

- **There is one more class called as StringBuffer in Java**
- **String is immutable where as StringBuffer allows to change the contents.**
- **Both class provide methods for different string operations**

Strings

```
chatAt()
compareTo()
compareToIgnoreCase()
concat()
endsWith()
equals()
equalsIgnoreCase()
indexOf()
lastIndexOf()
length()
regionMathces()
replace()
split()
startsWith()
substring()
toCharArray()
toLowerCase()
toUpperCase()
trim()
```

Some commonly used String methods

Visit : https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Strings (StringBuffer)

```
append()
capacity()
delete()
deleteCharAt()
insert()
replace()
reverse()
setCharAt()
```

Some additional methods from StringBuffer

Visit : https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html

Strings comparison

`equals()` | `compareTo()` | `==`

1. `equals()` method returns `true` or `false` by comparing the objects.
   Use `equalsIgnoreCase()` if required
2. `compareTo()` returns `int` value and define the ordering between objects
   - Returns 0 if the objects are same else returns +ve or -ve depending on the order between invoking object and parameter object
   - Use `compareToIgnoreCase()` if required.
3. `==` is the operator which returns `true` or f`alse` depending on whether the object references indicate the same object in memory or not

Strings

```
1)      String s1 = new String("Sachin");
        String s2 = s1;          // both reference to single object
```

Then,

```
        s1 == s2                 // returns true
```

```
2)      String s1 = new String("Sachin");
        String s2 = new String(s1);     // now two different objects
```

Then,

```
        s1 == s2                 // returns false
```

Strings

```
String s1 = new String("Sachin");
String s2 = new String("SACHIN");
String s3 = new String("Saurav");
```

Then,

```
s1.compareTo(s2)                // returns 32, difference between 'a' and 'A'
s1.compareToIgnoreCase(s2)      // returns 0
s1.comapreTo(s3);               // returns -18, difference between 'c' and 'u'
s1.equals(s2);                  // returns false
s1.equals(s3);                  // returns false
s1.equalsIgnoreCase(s2)         //returns true
```

Math class

Class Math is defined in the package `java.lang`

It defines a set of `static` methods used for different mathematical operations

```
Math.abs(x)
Math.ceil(x)
Math.cos(a)
Math.floor(x)
Math.log(x)  // natural log
Math.log10(x)  // log to base 10
Math.max(x,y)
Math.min(x,y)
Math.pow(x,y)  // power x raised to y
Math.round(x)
Math.sin(a)
Math.sqrt(x)
Math.tan(a)
Math.toDegrees(r)
Math.toRadians(d)
```

Visit : https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Vectors

- **Vector is a class from the package `java.util`**
- **Vector is a collection of objects of different types**
- **Primitive values should be converted to object using wrapper class before storing to vector**
- **Vector class provides different methods to store, retrieve and manipulate object stored in the vector**
- **Vector is created as follows**

```
Vector v = new Vector();   // OR

Vector v = new Vector(int initialCapacity);
```

Vectors

`add()`	`isEmpty()`
`addElement()`	`lastElelent()`
`capacity()`	`lastIndexOf()`
`clear()`	`remove()`
`elementAt()`	`removeElementAt()`
`equals()`	`setElementAt()`
`indexOf()`	`size()`
`insertElementAt()`	`toArray()`

Some commonly used Vector methods

Visit : https://docs.oracle.com/javase/8/docs/api/java/lang/Vector.html

Inheritance

- **Inheritance is a feature that allow to define sub classes using existing super classes**
- **The subclass acquires the properties of super class**
- **Use the keyword `extends` for defining subclass as follows**
- **Syntax :**

```
class SubClassName extends SuperClassName
{
        // Members of super class are Inherited here
        // Members of sub class defined here
}
```

Inheritance

```
class A{
  private int a; //not for sub class
  protected float b;

  public void setValue(){…}
  public void displayValue(){…}
}

class B extends A //inherit
{
  private int c;
  public void getNextValue(){…}
  public void displayAll(){…}
}
```

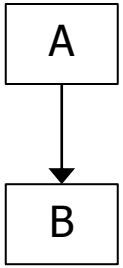
- class B is inherited from A
- Class B objects can access protected and public members of its super class
- Example,

```
B bObj = new B(); //sub class object

bObj.setValue();
bObj.displayValue();
// calling methods of super class
```

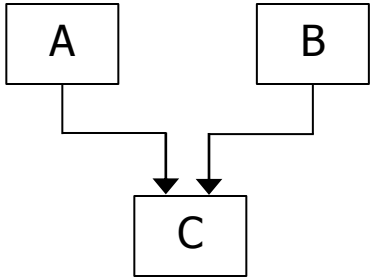
Here,
bObj now has attributes as:
b (inherited) and c (own)

Inheritance



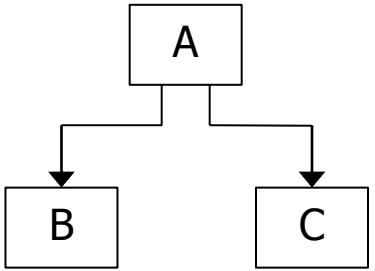
Single

- **One super class and one subclass**



Multiple

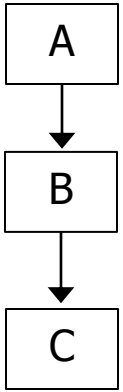
- **More than one super class for one subclass**
- **Not allowed in Java using only classes as super class but can be implemented using interface**



Hierarchical

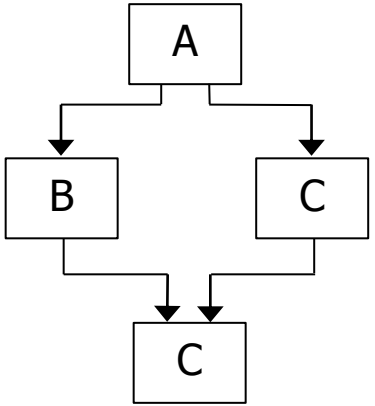
- **One super class for many sub classes**

Inheritance



Multilevel

- Sub class at one level becomes super class for next level



Hybrid

- Combination of other types
- Typically hierarchical and multiple as shown

Inheritance used in Java

- **The top level super class of all java standard classes is `class Object` (`java.lang.Object`) : Note that name of the `class` itself is `Object`**
- **It defines all the generic methods for different java classes**
- **These methods are either accessed by sub class objects via inheritance or the sub classes redefine those methods (overriding)**
- **Similarly different Exception classes are derived from `class Exception` (`java.lang.Exception`)**
- **To implement multithreading we have to define our own sub class using `class Thread` as super class**

Method overriding

- When a method in the super class is **redefined** by the sub class it is called as **method overriding**
- In this case **method name** and **parameters** are **exactly same**
- Compare with overloading where parameters must be different
- **Method overriding** provides **run time polymorphism**
- The binding of methods to method calls is done dynamically at run time which is based on the objet type. **Hence this is called as dynamic binding**

Method overriding

```
class A
{
    public void display()
    {
        S.o.pln("class A");
    }
}

class B extends A  //inherit
{
    public void display()
    {              //override
        S.o.pln("class B");
    }
}
```

```
class OverrideDemo
{
    p.s.void main(String args[])
    {
        A ref;  //only declaration

        ref = new A();
        ref.display();
      //call to super class

        ref = new B();
        ref.display();
      //call to sub class
    }
}
```

Method overriding

- The super class reference can be used to initialized the sub class objects **(but not the reverse)**
- In this case the method is invoked from the sub class first, if it is not defined in the subclass then it executes from the super class
- **For example,**

If Shape is a super class and Circle is a sub class then,

```
Shape s = new Circle();
```

// Super class reference

// Sub class object

Method overriding

- More understanding with standard Java classes

```
Object o1 = new String("Java");
Object o2 = new Integer(10);
Object o3 = new Exception();
Exception e = new NumberFormatException();
```

- In all above examples,
  it is **super class reference** initialized with **sub class objects**

Method overriding

How to call super class method from sub class in spite of overriding?

- Use the **`super`** reference (it is a keyword)
- Inside a method defined to sub class we can use

```
super.someSuperClassMethod()
```

- This invokes the **required method** from **super class** while the sub class method is running

Method overriding

```
class A
{
    public void display()
    {
        S.o.pln("class A");
    }
}

class B extends A  //inherit
{
    public void display()
    {              //override
        super.display();
        S.o.pln("class B");
    }
}
```

```
class OverrideDemo
{
    p.s.void main(String args[])
    {
        A ref;  //only declaration

        ref = new A();
        ref.display();
        //call to super class

        ref = new B();
        ref.display();
        //call to sub class
    }
}
```

Method overriding

Difference between `this` and `super`

```
public void someMethod()
{
    // other statements
       this.someOtherMethod();
}
```

- `someOtherMethod()` is invoked from the same class
- that is **both methods** should be defined in the **same class**

Find where Recursion is possible?

```
public void someMethod()
{
    // other statements
       super.someOtherMethod();
}
```

- `someOtherMethod()` is invoked from the super class
- Here **caller method** is in the **sub class** and **called method** should be defined in the **super class**

Method overriding (Why?)

```
class A
{
   // some other methods
      are defined
   // method display()is
      not here now
}

class B extends A  //inherit
{
   public void display()
   {
        S.o.pln("class B");
   }
}
```

```
class OverrideDemo
{
  p.s.void main(String args[])
  {
        A ref;  //only declaration

      ref = new A();
      ref.display();
// Error: method not found

      ref = new B();
      ref.display();
// Error: method not found
  } // ref is unable to identify
}       sub class method
```

Method overriding

Why overriding ? Can't we define the methods directly in sub class?

- Yes, you can. **But it has no polymorphism now because sub class methods will run only for that specific sub class objects**
  `(which happens without applying inheritance also)`
- Second reason is **super class reference will not be able to identify those methods which are directly defined to sub class and not present in the super class**

Overloading Vs Overriding

```
class A
{
    public void display()
    {
        S.o.pln("class A");
    }
}

class B extends A  //inherit
{
    public void display(int a)
    {                //overload
        super.display();
        S.o.pln("value = "+a);
    }
}
```

```
class OverrideDemo{
    p.s.void main(String args[])
    {
        A refOne;//only declaration
        B refTwo;

        refOne = new A();
        refOne.display();
      //call to super class

        refTwo = new B();
        refTwo.display();
      //call to super class
        refTwo.display(5);
      //call to sub class
    }
}
```

Method overriding

```
class Shape{
    public void area(){…}
}
```

```
class Circle extends Shape{
    public void area(){
      //find area of circle
    }
}
```

```
class Rectangle extends Shape{
    public void area(){
      //find area of rectangle
    }
}
```

```
Shape s = new Circle();
s.area();
```

```
Shape s = new Rectangle();
s.area();
```

- **Example of hierarchical Inheritance**
- **Achieves Run time polymorphism**
- **Can create super class object also if required**

```
Shape s  = new Shape();
s.area(); // will be called from super class
```

Abstract methods and classes

In previous example, Is it necessary to put empty body of method in super class?

Alternative

- **A class can contain a method which is only declared**
- **It is not defined in that class**
- **Such method is called as abstract method**
- **This method will be defined in the sub class (name overriding) such that the sub class can assign the required task to it.**

Abstract methods and classes

- A class that contains an abstract method is called as an abstract class
- Such class can be used for declaration of reference but it can not be used to create any object. It can be also used as super class to define the sub classes though
- This is because it can not allow the abstract method to be invoked since it is undefined for the class
- Earlier class Shape was define as
  Which we can use to create object like

```
Shape s = new Shape();
```

```
class Shape{
    public void area(){…}
}
```

Abstract methods and classes

- **Example:**

```
abstract class Shape{
    abstract public void area();
} // method only declared not defined

Shape s; // allowed to declare
Shape s = new Shape(); // not allowed to create the object
```

- **When the super class is abstract it is compulsory to the sub class to override the method before creating any object and calling the method**
- **If the sub class do not override the method, the sub class also becomes an abstract class automatically and needs to be extended to at least one more next level sub class to define the abstract methods**

Abstract methods and classes

```
abstract class Shape{
   abstract public void area();
}
```

```
class Circle extends Shape{
   public void area(){
     //find area of circle
   }
}
```

```
class Rectangle extends Shape{
   public void area(){
     //find area of rectangle
   }
}
```

```
Shape s = new Circle();
s.area();
```

```
Shape s = new Rectangle();
s.area();
```

- **Example of hierarchical Inheritance**
- **Achieves Run time polymorphism**
- **Can use the superclass reference but cannot create super class object**
- **Method overriding (name overriding) is compulsory for sub class**

Interfaces

- An `interface` is a completely abstract class
- That is, It is used to group related methods with only declarations
- In other words if all methods of an abstract class are also abstract it becomes an `interface`
- **Example:**

```
interface Shape{
        public void area();
        public void readInput();
}
        // method only declared not defined
        // no abstract keyword required for methods (optional)
```

Interfaces

- An `interface` can be extended to another interface (like sub class) or can be implemented in a class
- To create the sub class use `implements` keyword

```
class SubClassName implements InterfaceName
{
        // Members of interface are Inherited here
        // Since those are abstract methods define those here
}
```

Interfaces

Note that

- **Like abstract class, interface can not be used to create any object**
- **All interface methods have only declarations (not allowed to define any method inside the interface)**
- **These methods will be defined by the implemented class further.**
- **If the subclass do not override methods that sub class becomes an abstract class automatically (which should be extended further to define the methods at some next level)**
- **interface attributes (variables declared in an interface) are by default `public static` and `final`**

Interfaces

```
interface Shape{
   public void area();
}
```

```
class Circle implements Shape{
   public void area(){
     //find area of circle
   }
}
```

```
Shape s = new Circle();
s.area();
```

```
class Rectangle implements Shape{
   public void area(){
     //find area of rectangle
   }
}
```

```
Shape s = new Rectangle();
s.area();
```

- **Example of hierarchical Inheritance**
- **Achieves Run time polymorphism**
- **Can use the interface reference but cannot create objects using it.**
- **Method overriding (name overriding) is compulsory for sub class**

Why not multiple Inheritance using classes?

```
class A{
   public void xyz(){…}
}
```

```
class B{
   public void xyz(){…}
}
```

```
class Sub extends A,B
{
   // xyz() not overridden
}
```

```
// Just example, not
   allowed actually
```

```
   Sub s = new Sub();
   s.xyz(); // from which super class?

// ambiguity… can not decide, hence not allowed to use
   more than one class as classes
```

Why not multiple Inheritance using classes?

```
class Super{
    public void xyz(){…}
}
```

```
class A extends Super{…}
// no overriding done
```

```
class B extends Super{…}
// no overriding done
```

// allowed at this level

```
class Sub extends A,B {…}
// xyz() duplicated
```

// Just example, not allowed actually

```
    Sub s = new Sub();
    s.xyz();     // via class A or class B?

// ambiguity… can not decide, hence not allowed to use
```

Then how Java supports multiple Inheritance?

```
class A{
   public void xyz(){…}
}
```

```
interface B{
   public void xyz();
}
```

```
class Sub extends A implements B
{
   // xyz() overriding compulsory
}
```

// now allowed

```
Sub s = new Sub();
s.xyz();
```

```
A a = new Sub();
a.xyz();
```

```
B b = new Sub();
b.xyz();
```

```
// In all case method called from sub class only since overridden
// sub class method can call xyz() from class A using super reference if required
```

Then how Java supports multiple Inheritance?

```
class A{…}
// methods
defined inside
```

```
class B{…}
// methods
defined inside
```

```
interface C{…}
// methods only
declared inside
```

```
interface D{…}
// methods only
declared inside
```

```
class Sub extends A implements C, D    // now allowed
{
  // overriding of all interface
     methods is compulsory
}
```

- **Can NOT use both class A and class B at a time (Use A or B)**
- **Sub class must override (name overriding) all methods from C and D**
- **One sub class can have one class and many interfaces at super level**

More on Interfaces and Inheritance

```
class A
{…}

class B extends A
{…}
```

```
interface A
{…}

class B implements A
{…}
```

```
interface A
{…}

interface B extends A
{…}
```

Super class level → / Sub class level ↓	class	interface
class	extends	implements
interface	N A	extends

class, abstract class and interface

```
class MyClass{
   p.v. methodOne(){…}
   p.v. methodTwo(){…}
   p.v. methodThree(){…}
}
// All methods defined
// complete in nature
// can create the objects
```

```
abstract class MyClass{
   abstract p.v. methodOne();
   p.v. methodTwo(){…}
   p.v. methodThree(){…}
}
// at least one method is abstract
// partially incomplete
// can not create object
// extend to subclass, override and then
create sub class objects
```

```
interface MyInterface {
   p.v. methodOne();
   p.v. methodTwo();
   p.v. methodThree();
}
 // all methods are abstract
 // keyword abstract not required
 // totally incomplete
 // can not create object
 // implement to subclass, override and
 then create sub class objects
```

- When a method is defined in super class it is not compulsory to override the method in sub class as sub class objects can invoke the methods from super class via inheritance.
- But if it is abstract (undefined) in super class then it is compulsory for sub class to define it, before creating the objects

Final methods

- **Sometimes we need to restrict the sub class from overriding particular method from super class**
- **Such method can be declared as final while defining it in the super class**

Example:

```
class A{
    final public void myMethod() {…}
    public void otherMethod() {…}
}
// Now we can extend class A but can not override
   myMethod() in the sub class, the otherMethod() we can
```

Final variables

- **Final variables are like constants**
- **Once we assign a value to a final variable the program is not allowed to change it later**
- **We can use final variable as class member as well as like any local variable inside the method**

Example:

```
final int a; // can be initialized later
final int b = 20;
a = 10;  // allowed first time initialization
// a = 30; Not allowed to re-assign
```

Final classes

- **Similarly sometimes we need to restrict the process of inheritance itself.**
- **That is we do not want to allow to create any sub class further from a particular class.**
- **Such class can be declared as final class**
- **Conclusion : final classes can not be extended further**

Example:

```
final class A {…}
// now we can not use A as super class
```

finalize() method in Java

- **This method is defined in the class `java.lang.Object` (thus auto-inherited in all sub classes)**
- **It work like a destructor**
- **This method is called automatically by garbage collector on an object when garbage collection determines that there are no more references to the object**
- **The sub classes can override this method to release system resources held by the objects at the time when objects are destroyed at the end**

Constructors in Inheritance

```
class SuperClass
{
   protected int x,y;

   public SuperClass(){
     S.o.pln("Super class constructor");
     x = y = 0;
   }

   public SuperClass(int x1, int y1){
      this.x = x1;
      this.y = y1;
   }

   public void displayValues(){
     System.out.println("\nx = "+x);
     System.out.println("y = "+y);
   }
}
```

```
class SubClass extends SuperClass
{
    protected int z;

    public SubClass(){
       S.o.pln("Sub class constructor");
       z = 0;
    } // default constructor of super class is
        called automatically

    public SubClass(int x1, int y1,int z1){
       super(x1,y1); // call from super class
       this.z = z1;
    }

    public void displayValues(){
       super.displayValues();
       System.out.println("z = "+z);
    }
}
```

// Understand the applications of `super` keyword

Constructors in Inheritance

```
class ConstructorInheritanceDemo
{
        public static void main(String args[])
        {
                SuperClass s = new SubClass();

                s.displayValues();

                s = new SubClass(2,3,4);

                s.displayValues();
        }

}
```

Constructors in Inheritance

- **Constructors are called from the sub class to super class, that is sub class constructor will call the super class constructor either in implicit or explicit way.**
- **But constructors are executed in the order of derivation, that is from super class constructor executes first and then that of subclass**
- **Default constructor of sub class calls the super class constructor automatically. (implicit execution)**
- **To call the parameterized constructor use the keyword super. (explicit call to the required constructor)**

Automatic garbage collection

- **Deleting the unused objects from memory is a process called as garbage collection.**
- **The garbage collection is implemented inside JVM itself.**
- **The garbage collector runs in the background and it finds the unused objects and delete them to free the memory.**
- **We say that in Java has automatic garbage collection.**
- **Example,**

```
String s = new String("Java");
s = new  String("Program");

// as soon as the second object is created the first is automatically deleted from
memory i.e. the memory for that object is de-allocated.

// This happens because the same object reference is initialized to other object
```

Core Java Programming

Core Java concepts : PART IV

Exception Handling | Exception Hierarchy | types of exception (checked and unchecked) | try…catch statement | Using multiple catch blocks | user defined exceptions | using throw and throws keywords | using finally

Exception Handling

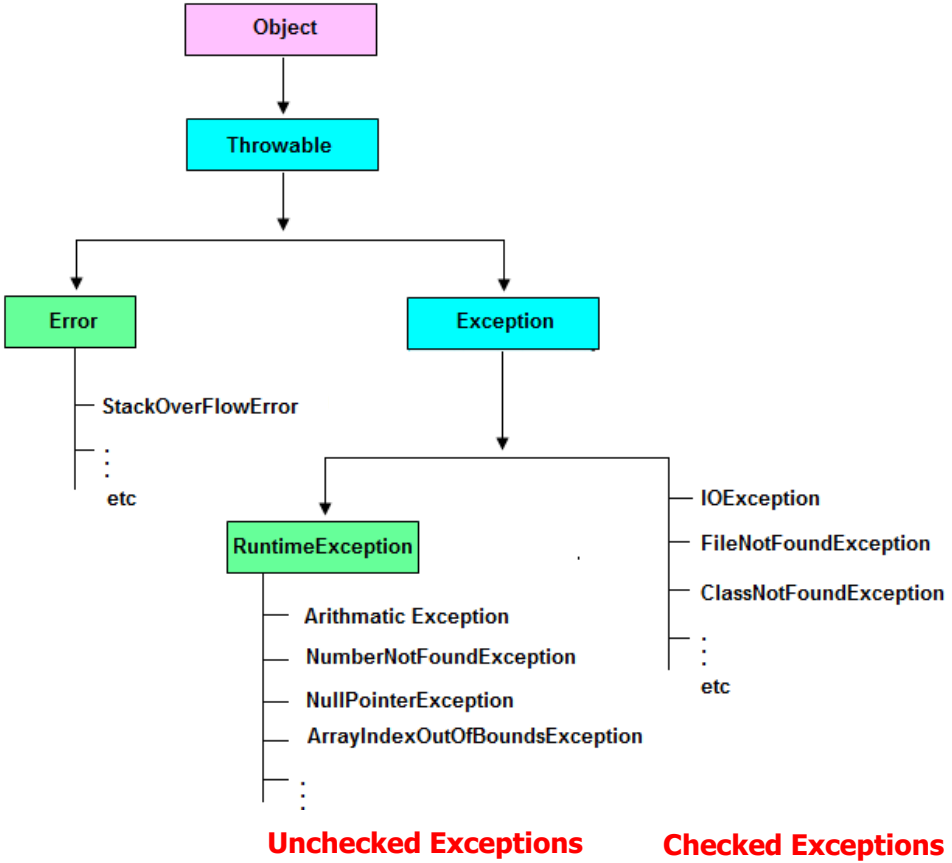
- **Exception is a Run-Time error in Java**
- **There are different Exception classes in Java to specify different exceptions**
- **Whenever Exception occurs, it `throws` an Exception object of particular type which describes that exception**
- **This object is handed over (*said to be thrown*) to exception handling mechanism which will take the necessary actions as required**

Note that, In normal situations we expect that there should not be any exception or interruption to the program execution. That's absolutely fine.

This is now all about what to do if it occurs !!

Exception Handling

- `Object` is the super class and `Throwable` is the sub class
- Further sub classes are class `Error` and class `Exception`
- **Errors** are mainly System related and **Exceptions** are program (application) related
- Class `Exception` is extended to class `RuntimeException` that further define all **unchecked Exceptions**
- Other **sub classes of class** `Exception` are **checked Exceptions**



Exception Handling

Checked Exceptions

- **These are checked at compile time**
- **Every sub class under `Exception` class is a checked exception. If some code within a method throws a checked exception, then either it must be handled by the method, or at-least declare it using `throws` keyword**

Unchecked Exceptions

- **These are not checked at compiled time**
- **In Java exceptions, every sub class under `RuntimeException` class is an unchecked exception, which may not be considered while writing the program**

Note : It is not like Exception must occur every time, but if it is a **checked type of exception** it is compulsory for the program to consider it in the source code even though it may not occur actually. **Unchecked exceptions may not be considered in source code still those can occur during run time.**

Exception Handling

```
import java.io.*;
class CheckedExceptionDemo
{
  public static void main(String args[])
  {
        DataInputStream din = new DataInputStream(System.in);

        String s = din.readLine();
        System.out.println(s);
   }
}
```

```
D:\Java>javac CheckedExceptionDemo.java
CheckedExceptionDemo.java:8: error: unreported exception IOException; must be ca
ught or declared to be thrown
        String s = din.readLine();
                               ^
```

Note that this error is shown by the compiler. The method we have used in the program is `readLine()` method which **can throw** `IOException as it is defined in that way by java API`. But since it is a **checked type of exception**, now it is **compulsory for program** either to **handle it** inside program or **at-least report it**

Exception Handling

```
import java.io.*;
class CheckedExceptionDemo
{
  public static void main(String args[])throws IOException
  {
        DataInputStream din = new DataInputStream(System.in);

        String s = din.readLine();
        System.out.println(s);
   }
}
```

Now we have mentioned something about `IOException` in the header of `main()` method

Now note that:

Actually it can be **thrown by `readLine()`** method and in that case it will be **caught by `main()`** method.

But since main() has no code defined to handle it, **`main()`** will also throw it further to **default exception handler** of the JVM. **So from JVM point of view it is the `main()` that can throw the exception hence it is reported in the header of `main()`**

Exception Handling

```
import java.io.*;
class CheckedExceptionDemo {
   public static void main(String args[]){
         DataInputStream din = new DataInputStream(System.in);

         try{
                  String s = din.readLine();
                  System.out.println(s);
         }
         catch(IOException e){
                  System.out.println("Some error in input operation..");

         }
   }
}
```

- **Now we have handled it inside `main()` method**
- **No need to report or declare it in the header of `main()`**

Exception Handling

- **The different keywords used in Exception handling are `try`, `catch`, `throw`, `throws` and `finally`**
- **`try…catch` are used together to put the code where exception is possible in the `try` block and handle it in the `catch` block**
- **`throw` is used to actually throw an object that represents the exception**
- **`throws` in always used only in the declaration i.e. in the header of method definition, reporting that the method can throw the exception**
- **`finally` is used at the end of exception handling code to ensure that in case of unhandled exception if the program aborts, it will certainly execute some important instructions before aborting**

Exception Handling

try…catch statement:

```
try{
    //statements where some Exception is possible
}
catch(Exception e){
   // what to do if exception occurs
}
```

Example, as seen earlier in case of `IOException`

Exception Handling

```
class ExceptionHandlingDemo{
   public static void main(String[] args){
       int a[] = {10,20,30,40,50};
       int sum=0;
       for(int i=0;i<10;i++){
             sum = sum + a[i];
       }
       System.out.println("Final total = "+sum);
   }
}
```

```
D:\Java>javac ExceptionHandlingDemo.java

D:\Java>java ExceptionHandlingDemo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at ExceptionHandlingDemo.main(ExceptionHandlingDemo.java:6)
```

- **Since the loop iterates beyond array size and attempts to access array element, it will throw ArrayIndexOutOfBoundsException**
- **Since it is an unchecked type of exception, compiler allows to run the program but at run time the exception is going to occur**

Exception Handling

- **Why there is no error while compiling the program?**
  Since the exception is an unchecked exception compiler allows it
- **Where the exception is generated?**
  The expression `a[i]` at index `5` because,
  as `a[0] to a[4]` are valid as per array size, and `a[5]` is invalid
- **What happened to it then?**
  Since this exception is inside `main()`, first it is caught by `main()` itself.
  But since `main()` has not defined exception handling it is thrown outside again
- **Then who caught it?**
  The default exception handler of java which is running inside JVM and the error message we see is displayed by JVM only.

Exception Handling

- **What happened to the program then?**
  - As soon as exception is generated program is aborted at the statement which caused the exception
  - In the example program it stopped at the attempt to access `a[5]`
  - The `for` loop which was supposed to run till index 9, is also aborted
  - Since `main()` is unable to handle the exception, it is also aborted instantly

- **What happened to the sum that the loop has calculated so far before exception?**

  Unfortunately, The data i.e. partially calculated sum is lost.

- **What is the solution then, at-least to avoid data loss?**

  See the next modification in the program !!!

Exception Handling

```
try{
        for(int i=0;i<10;i++){
                sum = sum + a[i];
        }
        System.out.println("Final total = "+sum);
        System.out.println("Program ends successfully..");
}

catch(Exception e){
        System.out.println("Stopping..");
        System.out.println("Partial total = "+sum);
}
```

Exception Handling

- **What we have done now?**

  We put the statements where exception is possible inside the `try {…}` block

- **What will happen now when the exception is generated?**

  As soon as exception occurs the program suspends all the operations from the `try {…}` block and the exception is thrown.

  Note that the `println()` statements from `try{…}` block are now skipped as program is no longer inside that block

- **Alright! Then what happened to that exception?**

  The exception is now caught by the `catch(Exception e){…}` block and it will now execute the statements given inside.

  We consider this as an exception handling

Exception Handling

- **Why we have mentioned `(Exception e)` along-with catch block?**

  This is something like receiving a parameter by a method. The actual exception that we are expecting here is precisely `ArrayIndexOutOfBoundsException`.

  But recall from the concepts of inheritance that sub class object can be assigned to super class reference

- **How come this happens at all?**

  It is as good as,

```
Exception e [ = new ArrayIndexOutOfBoundsException(); ]
```

Super class type of reference Declared in the catch block

Sub class type of object created inside try block at run time automatically and dispatched to the catch block

Exception Handling

- **What if we want to handle different exception types differently?**

Solution: One `try {…}` block can have multiple `catch(…){…}` blocks

```
try{
      //statements where some Exception is possible
}
catch(ExceptionTypeOne e){
      // what to do if type one exception occurs
}
catch(ExceptionTypeTwo e){
      // what to do if type two exception occurs
}

// and so on
```

How to do multiple Exception handling?

```
try{
      //some Exceptions are possible
}
catch(Exception e){
      // what to do if exception occurs
}
catch(IOException e){
      // what to do if exception occurs
}
catch(NumberFormatException e){
      // what to do if exception occurs
}
```

What is wrong here?

- The first catch block uses Exception reference
- Since it is super class it will catch all exceptions
- Second and third catch blocks are unreachable and compiler detects it and gives error

```
try{
      // some Exception are possible
}
catch(NumberFormatException e){
      // what to do if exception occurs
}
catch(IOException e){
      // what to do if exception occurs
}
catch(Exception e){
      // what to do if exception occurs
}
```

How is it correct now?

- The first and second catch block can catch those precise exceptions only because they are not type compatible to each other
- Last catch block can catch all other exceptions

Exception Handling

```
class MultipleExceptionHandlingDemo
{
    public static void main(String[] args){

      float sum = 0.0f;
      try{
         for(int i=0; i<5; i++){
              sum = sum + Float.parseFloat(args[i]);
         }
         System.out.println("Total ="+sum);
         System.out.println("Program executes successfully..");
      }

      catch(NumberFormatException e){
            System.out.println("Some wrong input..");
            System.out.println("partial sum = "+sum);
      }
      catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Insufficient inputs..");
            System.out.println("partial sum = "+sum);
      }
   }
}
```

Exception Handling

```
D:\Java>javac MultipleExceptionHandlingDemo.java

D:\Java>java MultipleExceptionHandlingDemo 2.1 3.4
Insufficient inputs..
partial sum = 5.5

D:\Java>java MultipleExceptionHandlingDemo 2.1 3.4 abc 4.5
Some wrong input..
partial sum = 5.5

D:\Java>java MultipleExceptionHandlingDemo 2.1 3.4 3.5 2.3 1.2
Total =12.5
Program executes successfully..
```

- **Note that program exits from the try block as soon as first exception occurs**
- **Similarly, if we handle the exception in program we can display the error message that can be understood by users more easily**
- **Error message displayed by default exception handler are more technical that display the exception name, line number from the code etc. those can be understood by programmers but may not understood by non technical users of software further**

Exception handling

```
int a[] = {10,20,30,40};

try{
  System.out.println(a[5]);
}
catch(NumberFormatException e)
{
        // some code here
}
```

```
int a=0;
String s = new String("java");
try{
   a = Integer.parseInt(s);
}
catch(NullPointerException e)
{
        // some code here
}
```

What is wrong here?

- **Conceptually nothing is wrong because those are unchecked exceptions**
- **But technically the actual exceptions that can occur and those considered in catch block do not match**
- ***In both cases, the catch blocks are useless and the corresponding exceptions will be thrown to and handled by default exception handler of java further***
- **Solution: use `catch(Exception e){…}` at the end so it can catch all exceptions if those uncaught by upper specific catch blocks**

Exception Handling

- **Ultimately what can be used in the `catch(…){…}` block to catch the Exception?**

```
catch(AnySubClassException e){…}
    // Allowed, should be in top order of multiple catch blocks

catch(Exception e){…}
    // Allowed, should be in bottom order of multiple catch blocks

catch(Throwable e){…}
    //Allowed, it is super class of Exception and it can catch Errors also.
    Should be the bottom-most catch block as per the hierarchy.

catch(Object e){…}
    //Not allowed anything beyond Throwable Family
```

User defined Exceptions

What is a user defined exception?

This is achieved by defining our own sub class by extending the class Exception

```
class MyException extends Exception
{
        //override required methods
}
```

// Note that by definition MyException is a checked exception here

```
class MyException extends RuntimeException
{
        //override required methods
}
```

// Note that by definition MyException is an unchecked exception here

Exception Handling

- **How to use the throw and throws keyword? What does those mean exactly?**

  `throw` **keyword is used to actually throw an exception type of object**
  `throws` **is used to declare the possibility of exception from some method**

- **Can we have any example?**

```
public static double divide(double n, double d)throws Exception
 {
                if(d == 0.0)
                        throw (new Exception());
                else
                        return (n/d);
 }
```

Note : `throws` is used in the header for declaration purpose only and `throw` is used inside the scope to actually create and throw the object. The object is created using `new` operator and constructor as usual

Exception Handling

What is finally and how to use it?

It is used after try…catch. It is a block of statements that are always executed. It is used to place some crucial operations that we want to execute before program stops. Example, closing files, releasing the system resources to avoid deadlock to system

Syntax:

```
try{
        // where exception can occur
}
catch(Exception e){
      // where exception is handled
}
finally{
      // this executes always
}
```

Exception Handling

```
class FinallyExample{
   public static void main(String[] args){
          int a = 10, b = 5;
          try{
              if(b == 0)
                    throw(new Exception());
              else
                    System.out.println((float)a/b);
          }
          catch(Exception e){
                    System.out.println("Division error...");
          }
          finally{
                    System.out.println("End of program..");
          }
    }
}
```

Note : try block will execute successfully hence catch block is skipped, but the finally block will run before end of the program

Exception Handling

```
class FinallyExample{
   public static void main(String[] args){
          int a = 10, b = 0;
          try{
              if(b == 0)
                    throw(new ArithmeticException());
              else
                    System.out.println((float)a/b);
          }
          catch(ArithmeticException e){
                    System.out.println("Division error...");
          }
          finally{
                    System.out.println("End of program..");
          }
    }
}
```

Note : Here exception is thrown hence try block is not executed completely. The catch block will execute and it is followed by finally block execution

Exception Handling

```
class FinallyExample{
   public static void main(String[] args){
          int a = 10, b = 0;
          try{
              if(b == 0)
                    throw(new ArithmeticException());
              else
                    System.out.println((float)a/b);
          }
          catch(NumberFormatException e){
                    System.out.println("Division error...");
          }
          finally{
                    System.out.println("End of program..");
          }
    }
}
```

Note : The exception actually thrown and the one that is attempted to catch do not match, hence catch block will not execute. The unhandled exception will be thrown to default exception handler. Before that finally block is executed.

Core Java Programming

Core Java concepts : PART V

Multithreading Definition | state transition diagram | Creating user defined threads | Thread class methods | Exceptions in multi threading | Thread synchronization

Multithreading

- **Thread is a active flow of execution within the program**
- **Normally there is always a single flow of execution during run-time hence it is treated as single thread**

In Java when an exception is thrown to default exception handler the error message we observe is as follows

```
D:\Java>javac ExceptionHandlingDemo.java

D:\Java>java ExceptionHandlingDemo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at ExceptionHandlingDemo.main(ExceptionHandlingDemo.java:6)
```

Note that,

Execution of `main()` method is treated as a single thread at run time by default !!!

Multithreading

What are multiple threads?

- **In java we can split the main execution thread in multiple child threads**
- **These multiple threads are then scheduled and executed by java in parallel (concurrency) with required synchronization and other management of threads**
- **The threads are configured in different status by the multi-threading environment and that is changed during run-time.**
- **The thread is in one particular state as per this mechanism**

Multithreading

Multithreading : State transition diagram

New: newly created thread

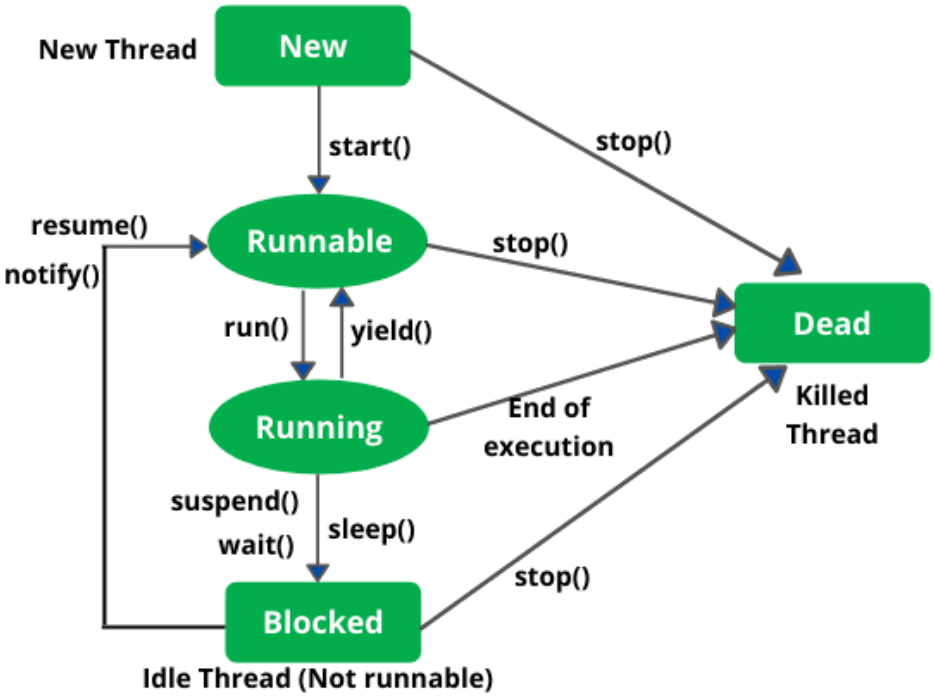
Runnable: Threads which are Ready to run

Running: Actual thread which is running in the CPU *(only one at a time)*

Blocked: Threads which are waiting for I/O or other events to take place

Dead: the terminated threads

The thread scheduler will select the next thread from Runnable state when previous thread is no longer in Running state



Multithreading

- **Creating Threads in Java step 1 : Defining a subclass**
- **Using class Thread** `(java.lang.Thread)`

```
class MyThread extends Thread{
    //override run() method
}
```

- **Using Interface Runnable** `(java.lang.Thread)`

```
class MyThread implements Runnable{
    //define run() method, name overriding
}
```

Multithreading

```
start()
stop()
run()
isAlive()
destroy()
```

```
sleep()
suspend()
resume()
```

```
getPriority()
setPriority()
```

These are the Thread class methods which are commonly used, see the details and other methods at the following link

Visit : https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Multithreading

- The Runnable interface has declared only one method, that is `run() method` which must be defined by the sub class
- It is used in case of multiple inheritance where the sub class is actually extended from some specific super class but needs to implement threading features also.
- Mainly used in GUI and Graphics programming but can be used at simple and basic levels also

Visit : https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

Multithreading

- **Creating Threads in Java step 2 : Creating the object**
- **Once the subclass is defined now we can create the object like,**

```
Thread t = new MyThread();
    //super class reference and subclass object

t.start(); // call the start method
```

- **Since the `start()` method is not overridden it is called from the super class `Thread` which internally calls the `run()` method**
- **`run()` method is overridden hence it is executed from the sub class (because actual object is a sub class object)**

Multithreading

```
class LoopThread extends Thread
{
   public void run(){
      int i;
      System.out.println(this);
      //displays thread description

      for(i=1;i<=10;i++){
          System.out.print(i+"  ");
      }
      System.out.println();
   }
}
```

```
class LoopThreadDemo
{
    p. s. v. main(String[] args){

      Thread t1 = new LoopThread();
      Thread t2 = new LoopThread();

      t1.start();
      t2.start();
   }
}
```

What is done here?

- **Define a user defined thread and override the `run()` method. It defines what task the thread should do.**

What happens here?

- **Create multiple thread objects and call the `start()` method. It will internally call the `run()` method to execute the threads**

Multithreading

```
D:\Java>javac LoopThreadDemo.java

D:\Java>java LoopThreadDemo
Thread[Thread-1,5,main]
Thread[Thread-0,5,main]
1  2  3  1  4  2  3  4  5  6  5  7  6  8  9  10  7
8  9  10

D:\Java>
```

```
D:\Java>javac LoopThreadDemo.java

D:\Java>java LoopThreadDemo
Thread[Thread-1,5,main]
Thread[Thread-0,5,main]
1  2  3  1  4  2  3  4  5  6  5  7  6  8  7  8  9  10  9  10


D:\Java>_
```

This is called as Race Condition. The output is interleaved and uncertain.
The threads are competing with each other for execution.

Multithreading

- `sleep()` **method is used to pause a thread for specified duration. It sets the timer in milliseconds, and the thread is in blocked state until the time elapse**
- **Once the timer is over the thread is configured in runnable state and further scheduled to run**
- **The method throws InterruptedException which is checked exception, hence it is compulsory to declare it or catch it**

Usage:

```
Thread.sleep(int ms)
//static method to pause current thread externally

this.sleep(int ms)   // can be invoked from thread object
```

Multithreading

```
class MainThreadDemo
{
     public static void main(String args[]){
          for(int i=0;i<5;i++){
               System.out.println(i);
               try{
                    Thread.sleep(500);
               } catch(InterruptedException e){}
          }
     }
}
```

```
// calling the static method sleep(), can mention throws in header of
main() instead of try...catch
```

Multithreading

```
class MyThread extends Thread
{
   public void run()
   {
       for(int i=1;i<=5;i++){
           System.out.println(i);
           try{
                this.sleep(500);
           }
          catch(Exception e){}
       }
   }
}
```

```
class SleepMethodDemo
{
   p. s. v. main(String[] args)
   {
       Thread t = new MyThread();
       t.start();
   }
}
```

- **Can not** use `throws` in the header of `run()` method because it is **overriding** and using the keyword `throws` will make a **mismatch with super class method**.
- **Also recall that, super class reference can be used in `catch(…){…}` block to catch all exceptions**

Multithreading

- **Thread priority : In Java it is a number from 1 to 10 which decides the order of execution. 1 means highest priority and 10 is the lowest**
- **The *default priority* of user threads is 5 (which can be changed later)**
- **High priority thread takes over the low priority threads**
  High priority thread will run first and low priority thread will be waiting even though that thread might have arrived early
- **Java uses *pre-emptive scheduling* i.e. at any time high priority thread arrives, the previously running thread with low priority is forced to runnable state**

```
getPriority()   // returns the current priority of the thread

setPriority(int p) // update the priority of the thread to given valid value
```

Multithreading

```
class MyThread extends Thread{
    // just a sublcass
}
class GetThreadPriorityDemo
{
    public static void main(String args[]){
        Thread t1 = new Thread();//super class object
        Thread t2 = new MyThread(); //sub class object

        System.out.println("t1 priority = "+t1.getPriority());
        System.out.println("t2 priority = "+t2.getPriority());
    }
}
```

```
D:\Java>javac GetThreadPriorityDemo.java

D:\Java>java GetThreadPriorityDemo
t1 priority = 5
t2 priority = 5
```

Multithreading

```
class SetThreadPriorityDemo
{
      public static void main(String args[]){
         Thread t = new Thread();

         t.setPriority(12);

         //incorrect, throws an unchecked exception

         System.out.println(t.getPriority());
      }
}
```

```
D:\Java>javac SetThreadPriorityDemo.java

D:\Java>java SetThreadPriorityDemo
Exception in thread "main" java.lang.IllegalArgumentException
        at java.lang.Thread.setPriority(Unknown Source)
        at SetThreadPriorityDemo.main(SetThreadPriorityDemo.java:6)
```

Multithreading

- **Mutual Exclusion : It is one of the solution for Race Condition. It can make sure that when one thread is running a critical section, the another (or all other) threads will wait**
- **We can use `synchronized` block to achieve Mutual Exclusion**

Example:

```
synchronized(this){
        // put the code to run in mutual exclusive mode
        // such as shared data or code
}
```

Multithreading

flag `//initially false, shared by threads`

ThreadOne

```
public void run()
{
   // check flag == false
   // if no, wait till flag == true
   // do the operation
   // set flag = true
}
```

ThreadTwo

```
public void run()
{
   // check flag == true
   // if no, wait till flag == false
   // do the operation
   // reset flag = false
}
```

- **Mutual Exclusion : This is one more solution to assure that threads will run in a *synchronized* and precise *alternate* way**

Core Java Programming

Core Java concepts : PART VI

Miscellaneous : parameter passing | user defined package

User defined package

- **package is a collection of classes and sub-packages**
- **Types of packages:**
  - **built-in package (Java API)**
  - **user defined package.**
- **Advantages of using packages:**
  - **Reusability - we can import the class to other programs**
  - **Better Organization - classes can be stored in a sorted manner**
  - **Avoid Name Conflicts - classes with same names can be stored in different packages to avoid the conflict**

User defined package

- **To define a class inside the package, declare the package name at the top**

```
package mypackage;

class MyClass{
    // members of MyClass
    // class is assumed with no main() method
       but only those methods that object can use
}
```

- **Now `MyClass` is considered as a class from `mypackage`**

User defined package

- **Now it can be imported to other program**

```
import mypackage.MyClass;

class ActualProgram{
    // here we can use MyClass to create
       objects and call methods as required
}
```

- **Note that `MyClass` is stored in other file (.class file) but can be used in `ActualProgram` which is a separate file**

User define package

```
package mypackage;

public class Number{
      int i;

      public Number(int n){
         this.i=n;
      }

      public boolean isPositive()
      {
         if(i>=0)
             return true;
         else
             return false;
         }
}
```

```
import mypackage.Number;

class NumberDemo
{
      p. s. v. main(String args[])
      {
         Number n = new Number(10);
        S.o.pln(n.isPositive());
      }
}
```

What is done here?

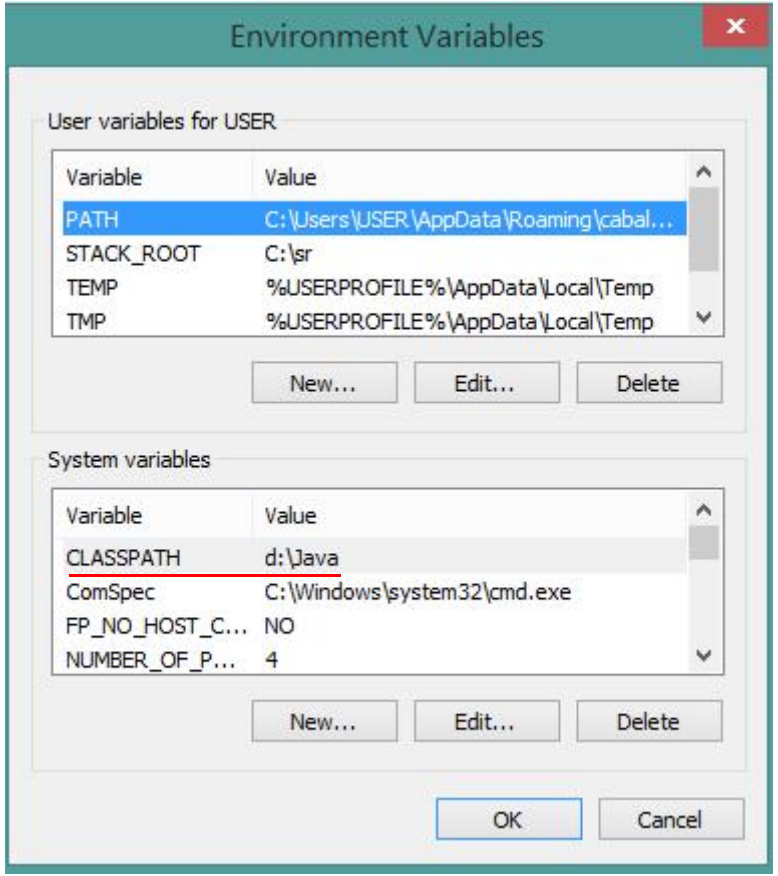
- **Class is defined and declared to be added to the package**

Note that `Number.java` and `NumberDemo.java` must be separately compiled

Only `NumberDemo` to execute

System settings

- **Add CLASSPATH variable to system variables**
- **Set it to the path of folder where java programs and user defined packages are stored**
- **If it already exists, put a semicolon at the end of existing path and then add the next path**
- **This is a one time task only**



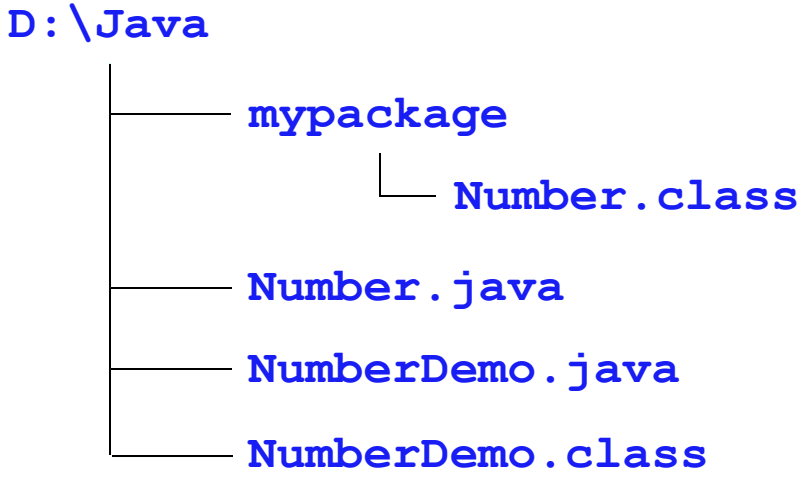
User defined package

- **Steps in user defined package at system level**
  1. **Create a package folder with the name equal to `packagename`**
  2. **Compile the source code and place the class file inside the package folder**
- **This can be done automatically using following command**

```
D:\Java > javac -d . MyCLass.java
```

- **This will create the subfolder and the class file will be placed inside that folder**

User defined package



Note that `source code are placed in the Java folder only`

`mypackage is a sub folder and the class file of package class`

Only `NumberDemo` **to execute**