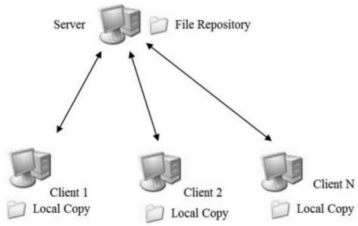
# Lecture 1 - Version Control



## Two types of Version control:

1. Centralized — Code is stored online and maintained/kept at one place
2. Decentralized — Stored at several places

## 1. Centralized Version control :




Kept at centralized location (Repositories)

This copy is called master Copy

To make changes we have to get a copy on your machine(Client machine) — Called working copy

When changes are made on local copy, "Commit " the change the repository

**Question 1 - what is every client makes a commit which one is sent to repositories?**

> That is called conflict. (Version control can resolve the conflict to some extend)

### Terms to remember

**Repository/ Repo**

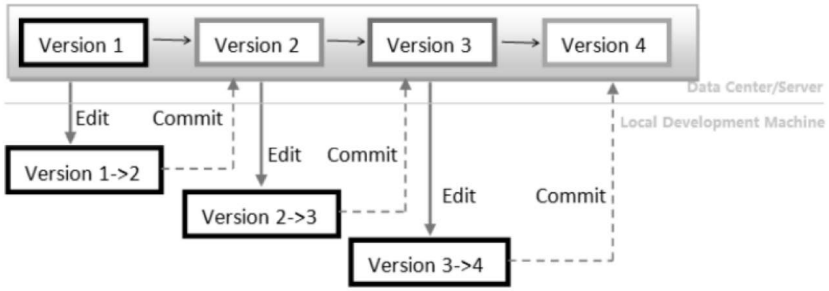
**Client program** (Clients working)

**Working Copy** (Copy you get of the latest repo)

**Checkout**

**Commit** (Sending the changes to repo)

### Tracking Changes when working alone



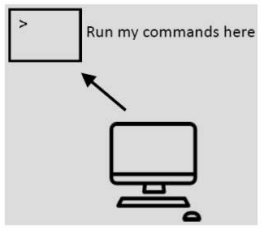

## Sub VersionN (SVN)

- SVN is the successor to Concurrent Versions System (CVS), and was built to help fix many issues in CVS
- Other source control systems include: Git, Mercurial, ClearCase, Perforce, etc

### Not VCS

**Secure Shell (SSH)**

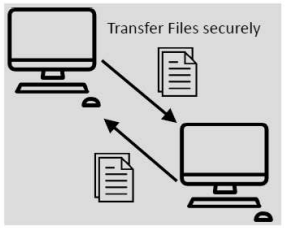
- Connect to remote computer and work in a shell on that computer



Run my commands here

**Secure CoPy (SCP)**

- Way to securely copy files from one computer to another
- Transfers a copy of the files
- Does not version files

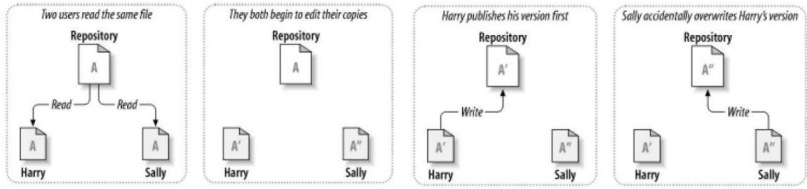

Transfer Files securely

## SVN vs GIT

| SVN | Git |
|---|---|
| It's a Centralized version control system | It's a distributed version control system. |
| It is revision control. | Git is an SCM (source code management). |
| It does not keep a cloned repository. | It has a cloned repository. |
| Branches in SVN are a folder that takes place in the repository. Some special commands are required For merging the branches. | The Git branches are familiar to work. The Git system helps in merging the files quickly and also assists in finding the unmerged ones. |
| It has an internationalized revision number. | It does not have a Global revision number. |
| SVN does not contain any cryptographically hashed contents. | It has cryptographically hashed contents that protect the contents from repository corruption taking place due to network issues or disk failures. |
| SVN stores content as files. | Git stored content as metadata. |
| SVN's content is less secure than Git. | Git has more content protection than SVN |
| SVN's content is less secure than Git. | Git has more content protection than SVN. |
| CollabNet, Inc developed SVN. | Linus Torvalds developed git for Linux kernel. |
| SVN is distributed under the open-source license. | Git is distributed under GNU (General public license). |

### Managing Concurrency

> What if two or more people want to edit the same file at the same time?



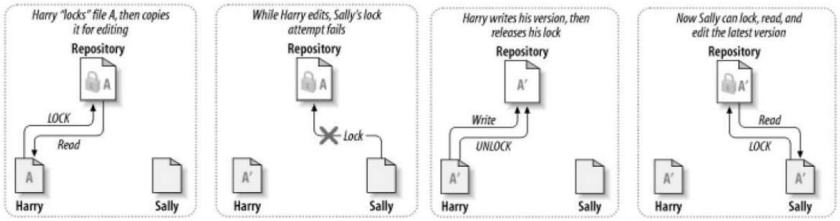

Option 1: Prevent it

• Only allow one writeable copy of each file

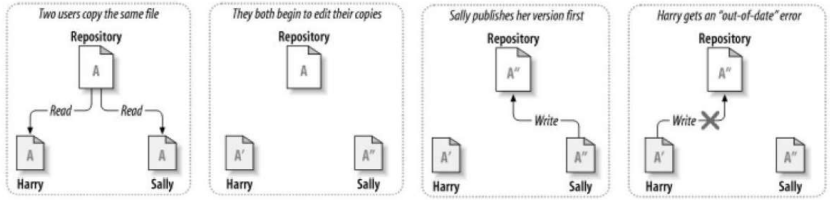
- Known as pessimistic concurrency
- E.g. Microsoft Visual SourceSafe, Rational ClearCas

Option 2: Allow it, fix issues afterwards
- Optimistic concurrency
- E.g. Subversion, CVS, Perforce

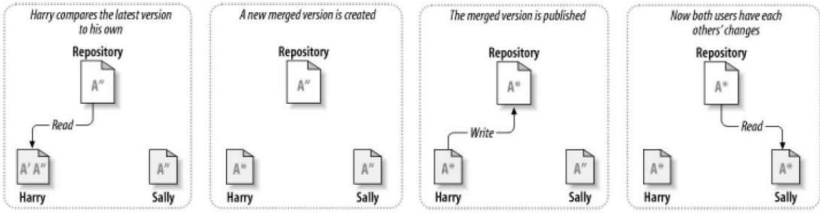
## Pessimistic Concurrency




## Optimistic Concurrency (1/2)




## Optimistic Concurrency (2/2)




### Optimistic Concurrency - Merging

Two Scenarios-

1. SVN is able to merge without help from user
2. Conflict: SVN needs the user to resolve the conflict

Options

```
Select: (p) postpone, (df) diff-full, (e) edit,
      (mc) mine-conflict, (tc) theirs-conflict,
      (s) show all options: s

(e)  edit          - change merged file in an editor
(df) diff-full     - show all changes made to merged file
(r)  resolved      - accept merged version of file

(dc) display-conflict - show all conflicts (ignoring merged version)
(mc) mine-conflict    - accept my version for all conflicts (same)
(tc) theirs-conflict  - accept their version for all conflicts (same)

(mf) mine-full     - accept my version of entire file (even non-conflicts)
(tf) theirs-full   - accept their version of entire file (same)

(p)  postpone      - mark the conflict to be resolved later
(l)  launch        - launch external tool to resolve conflict
(s)  show all      - show this list
```

## Integrating the code

What causes merge conflict:

Communication issue

Complex code bases

Experimental features being built

Two features being built in the same class be different developers

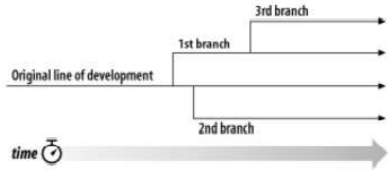
## Branching (VCS)

Branching strategies include

### No branching

### Feature Branching

(Not sure if the code will end up in the main branch, if successful then )




### Release Branching

(A release branching strategy involves creating a branch for a potential release that includes all applicable stories.)

## Storage Scheme

• Storing every copy of every file we generated over the course  of a project is not practical

• **Version control systems store incremental differences in  files/folder structures**

- **These differences store enough information to re-construct previous versions**, without storing every single copy ever made of the file.

### What is Stored where

Server Side: This is out of the scope of this course
• Your **local copy contains a special directory .svn**
• It **stores (locally) the information subversion** needs **to keep track of your files, version numbers, where the repository** is, etc.
• Needless to say, you **should not mess with the contents of this directory.** Let subversion do its job.

### General Rules

1. **Update and commit** frequently
2. **Never break main branch**
3. **always comment changes** made by you
4. **Test all code** before accepting merge
5. **Communicate with your team**

### Commands

1. Create a Repository (Get the URL path)
2. Checkout

```
svn checkout <url path>
```

or

```
svm co <url path>
```

3. Add the files in the repository for the svn.

Create new file send it to working copy

```
svn add <filename>
```

4. Commit Command

Used to save changes made on the repository

Whenever we made changes on our working copy and want to reflect it on the SVN server. In such a case, we have to make a commit operation.

```
svn commit -m "Comment message"
```

Log in to the username, or create one.

A. SVN Delete Command

The svn delete command is used to remove the files from the repository. When we perform a delete operation, it removes the file from the working. To delete it from the repository, run a commit command after the delete command.

```
svn delete <filename>
```

To remove it from the repository, run the commit command as follows:

```
svn commit -m "Removing a file"
```

B. SVN Diff Command

The svn diff command is used to display the differences between two versions of files. We can find the differences between the working copy and the remote (SVN) copy. We can also find the two revisions, paths, and more.

```
svn diff filename

svn diff -r R1:R2 filename
```

C. SVN Status Command

The svn status command displays the status of the working copy. It shows the status whether the repository is updated, added/deleted, or file is not under revision control and more.

```
svn status (path)
```

D. SVN Log Command

Stores history of the project. Display all commits made on repo or file

```
svn log (Path)
```

E. SVN Move Command

Used to move the files from the working directory. However, these files can be sent to the SVN server by commit operation.

```
svn move src dest
```

F. SVN Rename Command

```
svn rename CURR_PATH NEW_PATH
```

G. SVN List Command

Used to display the content of the repository. It is useful in the case; you want to view the details of the repository without making a working copy.

```
svn list
```

H. SVN Update Command

The update command is used to update the working copy of the project. It brings the changes from the working copy to the repository. It matches the working copy with the HEAD by default.

```
svn update (path)
```

I. SVN Info Command

Quick look at the working copy

```
svn info
```

# Lecture 2 - Introduction to JAVA

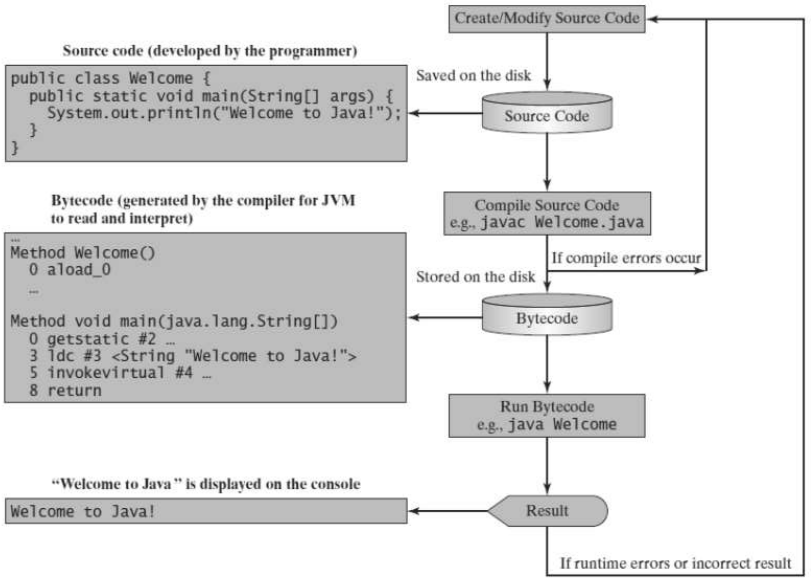


## Basics :

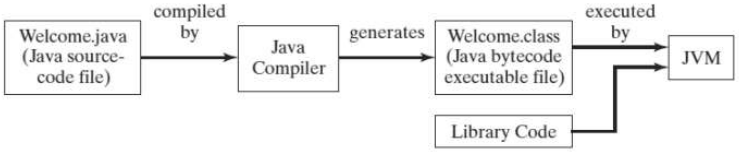
- Invented by James Gosling, 1994
- **Portability/WORA**(write once run anywhere) language, Can run anywhere in any device where we can run JVM (Java virtual Machine)
- Widely used in industry
- High level lang
- Develop software running:
  - Desktop computer
  - Server
  - Mobile devices

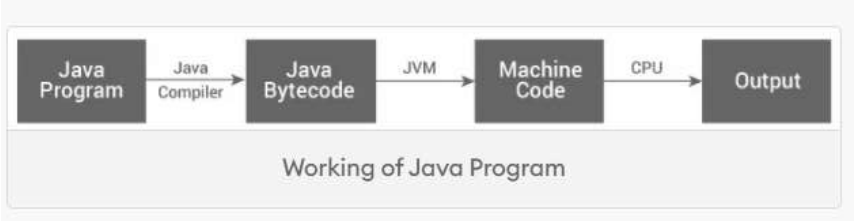
### Three main steps in java:

1. **Writing the source code** using text editor
2. Translating **source code into JAVA bytecode** using compiler
   a. *Bytecode is similar to machine instructions*(Low level) but is architectures neutral and can run on any platform that has JVM
3. **Executing Bytecode**
   a. JVM is interpreter: it translates *bytecode into target machine language* code one at a time rather than whole program as a single unit
   b. Each step is execute immediately after it is translated

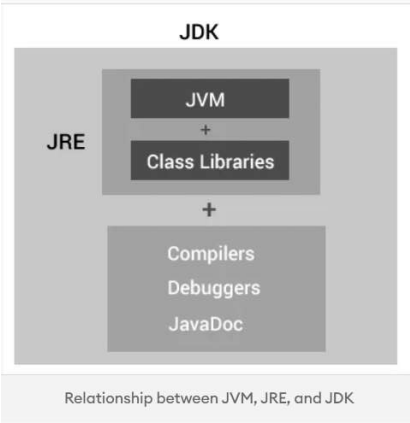



Compiler converts Java file to bytecode






## Class includes:

1. **Data fields** to represent state of objects
2. **Methods** to represent the behavior if objects
3. Special type of methods - **Constructor**
    a. **not return type**
    b. 0 or more arguments
    c. Should have the **same name of class**
    d. invoked using the **new** operator

***main* Method**

Execution starts here, this is the **entry point of the program**

**Default values**

null - reference type

0 - numeric type

false - Boolean type

'\u0000' - char type

- **Java assigns no default value to local variable inside a method**

```
public double dis(){
  int n;
  System.out.println(n); // !Error cause it is  a local variable
}
```

- **Java Assigns default value to class fields when a new class is created**

## Differences between Variables of Primitive

Types and Reference Types

- Every variable represents a memory location that holds a value
- For a variable of a **primitive type, the value is of the primitive type**
- For a variable of a **reference type, the value is a reference to where an object is located**

# Static Modifier

- Static fields/methods **can be accessed from a reference variable or from their class name**
    - From code below
        System.out.println(Point.s);  // class

        System.out.println(p2.s);  // Ref var
- Non-static (or instance) fields/methods can **only be accessed from a reference variable**

```
public Point{
  int x;
  int y;
  static int s;
  // To keep track of number of objects
  static int counter

  public Point()
    {
      counter++; // Increases
    }
}


// inside the main file -> psvm

  Point p1 = new ....
  Point p2 = new ....
  p2.s = 10;
  p1.s = 20;

  System.out.println(Point.s);   // Result will be 20

  System.out.println(p2.s);      // Result will be 20

  System.out.println(p1.s);      // Result will be 20
```

- Inside **static method** we cannot invoke **non-static method/field**
- We can **invoke static method** by using **instance**(Objects) or by the **class** name
- I can access static method from non static method

## What is a static keyword in Java?

In Java, if we want to access class members, we must first create an instance of the class. But there will be situations where we want to access class members without creating any variables.

In those situations, we can use the `static` keyword in Java. If we want to access class members without creating an instance of the class, we need to declare the class members static.

The `Math` class in Java has almost all of its members static. So, we can access its members without creating instances of the Math class. For example,

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

        // accessing the methods of the Math class
        System.out.println("Absolute value of -12 =  " + Math.abs(-12));
        System.out.println("Value of PI = " + Math.PI);
        System.out.println("Value of E = " + Math.E);
        System.out.println("2^2 = " + Math.pow(2,2));
    }
}
```

**Output**:

```
Absolute value of -12 = 12
Value of PI = 3.141592653589793
Value of E = 2.718281828459045
2^2 = 4.0
```

In the above example, we have not created any instances of the `Math` class. But we are able to access its methods: `abs()` and `pow()` and variables: `PI` and `E`.

It is possible because the methods and variables of the `Math` class are static.

---

## Static Methods

Static methods are also called class methods. It is because a static method belongs to the class rather than the object of a class.

And we can invoke static methods directly using the class name. For example,

```
class Test {
    // static method inside the Test class
    public static void method() {...}
}

class Main {
    // invoking the static method
    Test.method();
}
```

Here, we can see that the static method can be accessed directly from other classes using the class name.

In every Java program, we have declared the `main` method `static`. It is because to run the program the JVM should be able to invoke the main method during the initial phase where no objects exist in the memory.

### Example 1: Java static and non-static Methods

```
class StaticTest {

    // non-static method
    int multiply(int a, int b){
        return a * b;
    }

    // static method
    static int add(int a, int b){
        return a + b;
    }
}

public class Main {

   public static void main( String[] args ) {

        // create an instance of the StaticTest class
        StaticTest st = new StaticTest();

        // call the nonstatic method
        System.out.println(" 2 * 2 = " + st.multiply(2,2));

        // call the static method
        System.out.println(" 2 + 3 = " + StaticTest.add(2,3));
   }
}
```

**Output**:

```
2 * 2 = 4
2 + 3 = 5
```

In the above program, we have declared a non-static method named `multiply()` and a static method named `add()` inside the class `StaticTest`.

Inside the Main class, we can see that we are calling the non-static method using the object of the class (`st.multiply(2, 2)`). However, we are calling the static method by using the class name (`StaticTest.add(2, 3)`).

## Static Variables

In Java, when we create objects of a class, then every object will have its own copy of all the variables of the class. For example,

```
class Test {
    // regular variable
   int age;
}

class Main {
    // create instances of Test
    Test test1 = new Test();
    Test test2 = new Test();
}
```

Here, both the objects test1 and test2 will have separate copies of the variable age. And, they are different from each other.

However, if we declare a variable static, all objects of the class share the same static variable. It is because like static methods, static variables are also associated with the class. And, we don't need to create objects of the class to access the static variables. For example,

```
class Test {
    // static variable
    static int age;
}
class Main {
    // access the static variable
    Test.age = 20;
}
```

Here, we can see that we are accessing the static variable from the other class using the class name.

### Example 2: Java static and non-static Variables

```
class Test {

   // static variable
   static int max = 10;
```

```
    // non-static variable
    int min = 5;
}

public class Main {
    public static void main(String[] args) {
        Test obj = new Test();

        // access the non-static variable
        System.out.println("min + 1 = " + (obj.min + 1));

        // access the static variable
        System.out.println("max + 1 = " + (Test.max + 1));
    }
}
```

**Output**:

```
min + 1 = 6
max + 1 = 11
```

In the above program, we have declared a non-static variable named min and a static variable named max inside the class Test.

Inside the Main class, we can see that we are calling the non-static variable using the object of the class (`obj.min + 1`). However, we are calling the static variable by using the class name (`Test.max + 1`).

**Note**: Static variables are rarely used in Java. Instead, the static constants are used. These static constants are defined by `static final` keyword and represented in uppercase. This is why some people prefer to use uppercase for static variables as well.

---

## Access static Variables and Methods within the Class

We are accessing the static variable from another class. Hence, we have used the class name to access it. However, if we want to access the static member from inside the class, it can be accessed directly. For example,

```
public class Main {

    // static variable
    static int age;
    // static method
    static void display() {
        System.out.println("Static Method");
    }
    public static void main(String[] args) {

        // access the static variable
        age = 30;
        System.out.println("Age is " + age);

        // access the static method
        display();
    }
}
```

**Output**:

```
Age is 30
Static Method
```

Here, we are able to access the static variable and method directly without using the class name. It is because static variables and methods are by default public. And, since we are accessing from the same class, we don't have to specify the class name.

---

## Static Blocks

In Java, static blocks are used to initialize the static variables. For example,

```
class Test {
    // static variable
    static int age;

    // static block
    static {
        age = 23;
    }
}
```

Here we can see that we have used a static block with the syntax:

```
static {
    // variable initialization
}
```

The static block is executed only once when the class is loaded in memory. The class is loaded if either the object of the class is requested in code or the static members are requested in code.

A class can have multiple static blocks and each static block is executed in the same sequence in which they have been written in a program.

### Example 3: Use of static block in java

```
class Main {

   // static variables
   static int a = 23;
   static int b;
   static int max;

   // static blocks
   static {
       System.out.println("First Static block.");
       b = a * 4;
   }
   static {
       System.out.println("Second Static block.");
       max = 30;
   }

   // static method
   static void display() {

       System.out.println("a = " + a);
       System.out.println("b = " + b);
       System.out.println("max = " + max);
   }

   public static void main(String args[]) {
       // calling the static method
       display();
   }
}
```

**Output**:

```
First Static block.
Second Static block.
a = 23
b = 92
max = 30
```

In the above program. as soon as the Main class is loaded,

- The value of  is set to `23`.

  a

- The first static block is executed. Hence, the string `First Static block` is printed and the value of  is set to `a * 4`.

  b

- The second static block is executed. Hence, the string `Second Static block` is printed and the value of  is set to `30`.

  max

- And finally, the print statements inside the method `display()` are executed.

# Arrays

Is a Data structures that represent a collection of the same types of data.

```
int[] A = new int[20];
```

Size of Array

```
A.length
```

Array initializer

```
int[] A = {1,2,4,76,6};
```

## For each loop

```
for(int a: A){
    sysout(a);
}
```

# Lecture 3 - I/O and regular Expression



## 1. Input and Output(I/O)

**Input sources include:**

Keyboard

File

Network

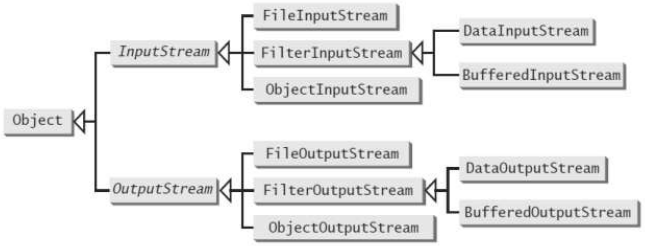
**Output Destinations include:**

Console

File

Network

**Java handles i/o  using streams**




### 1.1 Standard I/O

**System.in**

Object of type InputStream

Typically refers to the keyboard

Reading data could be done using the Scanner class

Methods include:

- String next()
- String nextLine()
- int nextInt()
- double nextDouble()

```
import java.util.Scanner;

class xyz{
  public method(){
    Scanner scan = new  Scanner(System.in);
    sysout("String: ");

      // Reads only uptil a space (delmiter)
    String str = scan.next();
      // reads all but not able to exit through this loop
    while(scan.hasNext())
        String str = scan.next();


    sysout("Int: ");
    String str = scan.nextInt();

    sysout("Double: ");
    String str = scan.nextDouble();

    sysout("Float: ");
    String str = scan.nextFloat();
  }
}
```

**System.out**

Object of type PrintStream (SUBCLASS OF **filteroutputstream**)

Typically refers to console

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

        System.out.println("Java programming is interesting.");
    }
}
```

**Difference between println(), print() and printf()**

- `print()` - It prints string inside the quotes.
- `println()` - It prints string inside the quotes similar like `print()` method. Then the cursor moves to the beginning of the next line.
- `printf()` - It provides string formatting (similar to printf in C/C++ programming).

### 1.1.1 The File class

- Contains Methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory
- Files could be specified using absolute / relative names
- Constructing a **File instance doesnt create a file on the machine**
- Methods include:

- boolean createNewFile()
- boolean delete()
- boolean exists()
- File [] listFiles()

```
class xyz{
  void trial(){
    File a = new File("C:/...../text.txt"); // in the directory of project
    f.createNewFile();
    f.delete();
    boolean i= f.exists();
    boolean b= f.isDirectory();
    File f= new File("C: ..... /abc.txt");

    if(f.exists() && f.isDirectory()){
      File [] files = d.listFiles();

      for (File f:files){
        if (!f.isHidden()) // hidden files are not displayed{
          sysout(f.getName()); // names of the files will be displayed
        }
      }
    }
  }
}
```

### File I/O

Reading could be done using **Scanner** class

```
Scanner input = new Scanner(new File(filename));
```

Writing could be done using the **FileWriter** class

```
FileWriter output = new FileWriter((String) "filename", (boolean) append);

output.write("hello");
output.write("hello"); // will add hello next to previous hello not in the new line
output.close();
```

Extra

```
package lab3;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class HashtagClass {

    private static void findHashtags(File file, File outputDirectory){
        Pattern pattern = Pattern.compile("    Your REGEX    ");
        int hashtag = 0;


        File outputFile = new File(outputDirectory.getAbsolutePath() + '/' + file.getName().substring(0, file.getName().length()-


        try{
          // Writing in file
          FileWriter myWriter = new FileWriter(outputFile.getAbsolutePath());
```

```
          // Scanner doesnt work properly in windows
            //!!!!!!Scanner sc = new Scanner(file);

          BufferedReader sc = new BufferedReader(new FileReader(file));
          String line = "";

          while ((line = sc.readLine()) != null) {
            //Saving all words in line in an array
              String[] words = line.split(" ");
            //Runnin a loop on words
            for (int i =0;  i< words.length; i++){
              Matcher m = pattern.matcher(words[i]);
              if (m.matches()){
                hashtag++;
                // Writing the word matched in the file
                  myWriter.write(words[i] + "\n");
                }
              }
            }

          //closing io
            myWriter.close();
            sc.close();

          System.out.println("File "+ outputFile.getName() +"Written with Total Hashtags: " + String.valueOf(hashtag));
        }

        catch(Exception e){
          System.out.println("Failed to read");
        }
    }


    public static void main(String[] args){
        Scanner reader = new Scanner(System.in);
        System.out.println("Enter the absolute path that conatains the files.");
        String inputPath = reader.nextLine();
        System.out.println("Enter the absolute path that you would like to store the output files.");
        String outputPath = reader.nextLine();


        //Making Directory
          File outputDirectory = new File(outputPath);
          outputDirectory.mkdir();
          reader.close();

        // Opening all files in an array of files
          File path = new File(inputPath);
          File[] files = path.listFiles();


        //Runnin loop on all the files in array
          for (File file: files){
            findHashtags(file, outputDirectory);
          }
        System.out.println("Completed");

    }
}
```

# 2. Regular Expression

Regular expression (regex) is a string that describes a pattern for matching a set of strings

The **Pattern class** can be used to define pattern

- The **compile** method takes a string representing the regular expression as an argument and compiles it into a pattern.

The **Matcher** class can be used to search for the pattern. its method include:

- boolean find()
- boolean matches()

Example:

```
Pattern pattern = Pattern.compile("H.*d");
Matcher matcher = pattern.matcher("Hello World");
System.out.println(matcher.matches());
System.out.println(matcher.find());        //checks the subset of the string
```

regex101: build, test, and debug regex

Regular expression tester with syntax highlighting, explanation, cheat sheet for PHP/PCRE, Python, GO, JavaScript, Java. Features a regex quiz & library.

https://regex101.com/


## Commonly used Regex

| Regular Expression | Matches | Example |
|---|---|---|
| . | any single character | Java matches J..a |
| (ab\|cd) | ab or cd | ten matches t(en\|im) |
| [abc] | a, b, or c | Java matches Ja[uvwx]a |
| [^abc] | any character except a, b, or c | Java matches Ja[^ars]a |
| [a-z] | a through z | Java matches [A-M]av[a-d] |
| [^a-z] | any character except a through z | Java matches Jav[^b-d] |
| [a-e[m-p]] | a through e or m through p | Java matches [A-G[I-M]]av[a-d] |

| Regular Expression | Matches | Example |
|---|---|---|
| [a-e&&[c-p]] | intersection of a-e with c-p | Java matches [A-P&&[I-M]]av[a-d] |
| \d | a digit, same as [0-9] | Java2 matches "Java[\\d]" |
| \D | a non-digit | $Java matches "[\\D][\\D]ava" |
| \w | a word character | Java1 matches "[\\w]ava[\\w]" |
| \W | a non-word character | $Java matches "[\\W][\\w]ava" |
| \s | a whitespace character | "Java 2" matches "Java\\s2" |
| \S | a non-whitespace char | Java matches "[\\S]ava" |

\w mean any letter digit or underscore

| Regular Expression | Matches | Example |
|---|---|---|
| p* | zero or more occurrences of pattern p | aaaabb matches "a*bb"<br>ababab matches "(ab)*" |
| p+ | one or more occurrences of pattern p | a matches "a+b*"<br>able matches "(ab)+.*" |
| p? | zero or one occurrence of pattern p | Java matches "J?Java"<br>Java matches "J?ava" |
| p{n} | exactly n occurrences of pattern p | Java matches "Ja{1}.*"<br>Java does not match ".{2}" |
| p{n,} | at least n occurrences of pattern p | aaaa matches "a{1,}"<br>a does not match "a{2,}" |
| p{n,m} | between n and m occurrences (inclusive) | aaaa matches "a{1,9}"<br>abb does not match "a{2,9}bb" |

https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

| Classes | Matches |
|---|---|
| \p{Lower} | A lowercase character:\p{IsLowercase} |
| \p{Upper} | An uppercase character:\p{IsUppercase} |
| \p{ASCII} | All ASCII:[\x00-\x7F] |
| \p{Alpha} | An alphabetic character:\p{IsAlphabetic} |
| \p{Digit} | A decimal digit character:p{IsDigit} |
| \p{Alnum} | An alphanumeric character:[\p{IsAlphabetic}\p{IsDigit}] |
| \p{Punct} | A punctuation character:p{IsPunctuation} |
| \p{Graph} | A visible character: [^\p{IsWhite_Space}\p{gc=Cc}\p{gc=Cs}\p{gc=Cn}] |
| \p{Print} | A printable character: [\p{Graph}\p{Blank}&&[^\p{Cntrl}]] |
| \p{Blank} | A space or a tab: [\p{IsWhite_Space}&&[^\p{gc=Zl}\p{gc=Zp}\x0a\x0b\x0c\x0d\x85]] |
| \p{Cntrl} | A control character: \p{gc=Cc} |
| \p{XDigit} | A hexadecimal digit: [\p{gc=Nd}\p{IsHex_Digit}] |
| \p{Space} | A whitespace character:\p{IsWhite_Space} |
| \d | A digit: \p{IsDigit} |
| \D | A non-digit: [^\d] |
| \s | A whitespace character: \p{IsWhite_Space} |
| \S | A non-whitespace character: [^\s] |
| \w | A word character: [\p{Alpha}\p{gc=Mn}\p{gc=Me}\p{gc=Mc}\p{Digit}\p{gc=Pc}] |
| \W | A non-word character: [^\w] |

| Characters | |
|---|---|
| *x* | The character *x* |
| \\ | The backslash character |
| \0*n* | The character with octal value 0*n* (0 <= *n* <= 7) |
| \0*nn* | The character with octal value 0*nn* (0 <= *n* <= 7) |
| \0*mnn* | The character with octal value 0*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7) |
| \x*hh* | The character with hexadecimal value 0x*hh* |
| \u*hhhh* | The character with hexadecimal value 0x*hhhh* |
| \x{*h...h*} | The character with hexadecimal value 0x*h...h* (Character.MIN_CODE_POINT <= 0x*h...h* <= Character.MAX_CODE_POINT) |
| \t | The tab character ('\u0009') |
| \n | The newline (line feed) character ('\u000A') |
| \r | The carriage-return character ('\u000D') |
| \f | The form-feed character ('\u000C') |
| \a | The alert (bell) character ('\u0007') |
| \e | The escape character ('\u001B') |
| \c*x* | The control character corresponding to *x* |

### Character classes

| | |
|---|---|
| `[abc]` | `a`, `b`, or `c` (simple class) |
| `[^abc]` | Any character except `a`, `b`, or `c` (negation) |
| `[a-zA-Z]` | `a` through `z` or `A` through `Z`, inclusive (range) |
| `[a-d[m-p]]` | `a` through `d`, or `m` through `p`: `[a-dm-p]` (union) |
| `[a-z&&[def]]` | `d`, `e`, or `f` (intersection) |
| `[a-z&&[^bc]]` | `a` through `z`, except for `b` and `c`: `[ad-z]` (subtraction) |
| `[a-z&&[^m-p]]` | `a` through `z`, and not `m` through `p`: `[a-lq-z]`(subtraction) |

### Predefined character classes

| | |
|---|---|
| `.` | Any character (may or may not match line terminators) |
| `\d` | A digit: `[0-9]` |
| `\D` | A non-digit: `[^0-9]` |
| `\s` | A whitespace character: `[ \t\n\x0B\f\r]` |
| `\S` | A non-whitespace character: `[^\s]` |
| `\w` | A word character: `[a-zA-Z_0-9]` |
| `\W` | A non-word character: `[^\w]` |

### POSIX character classes (US-ASCII only)

| | |
|---|---|
| `\p{Lower}` | A lower-case alphabetic character: `[a-z]` |
| `\p{Upper}` | An upper-case alphabetic character:`[A-Z]` |
| `\p{ASCII}` | All ASCII:`[\x00-\x7F]` |
| `\p{Alpha}` | An alphabetic character:`[\p{Lower}\p{Upper}]` |
| `\p{Digit}` | A decimal digit: `[0-9]` |
| `\p{Alnum}` | An alphanumeric character:`[\p{Alpha}\p{Digit}]` |
| `\p{Punct}` | Punctuation: One of `!"#$%&'()*+,-./:;<=>?@[\]^_`{\|}~` |
| `\p{Graph}` | A visible character: `[\p{Alnum}\p{Punct}]` |
| `\p{Print}` | A printable character: `[\p{Graph}\x20]` |
| `\p{Blank}` | A space or a tab: `[ \t]` |
| `\p{Cntrl}` | A control character: `[\x00-\x1F\x7F]` |
| `\p{XDigit}` | A hexadecimal digit: `[0-9a-fA-F]` |
| `\p{Space}` | A whitespace character: `[ \t\n\x0B\f\r]` |

### java.lang.Character classes (simple java character type)

| | |
|---|---|
| `\p{javaLowerCase}` | Equivalent to java.lang.Character.isLowerCase() |
| `\p{javaUpperCase}` | Equivalent to java.lang.Character.isUpperCase() |
| `\p{javaWhitespace}` | Equivalent to java.lang.Character.isWhitespace() |
| `\p{javaMirrored}` | Equivalent to java.lang.Character.isMirrored() |

### Classes for Unicode scripts, blocks, categories and binary properties

| | |
|---|---|
| `\p{IsLatin}` | A Latin script character (script) |
| `\p{InGreek}` | A character in the Greek block (block) |
| `\p{Lu}` | An uppercase letter (category) |
| `\p{IsAlphabetic}` | An alphabetic character (binary property) |
| `\p{Sc}` | A currency symbol |
| `\P{InGreek}` | Any character except one in the Greek block (negation) |
| `[\p{L}&&[^\p{Lu}]]` | Any letter except an uppercase letter (subtraction) |

### Boundary matchers

| | |
|---|---|
| `^` | The beginning of a line |
| `$` | The end of a line |
| `\b` | A word boundary |
| `\B` | A non-word boundary |
| `\A` | The beginning of the input |
| `\G` | The end of the previous match |
| `\Z` | The end of the input but for the final terminator, if any |
| `\z` | The end of the input |

### Greedy quantifiers

| | |
|---|---|
| *X*`?` | *X*, once or not at all |
| *X*`*` | *X*, zero or more times |
| *X*`+` | *X*, one or more times |
| *X*`{`*n*`}` | *X*, exactly *n* times |
| *X*`{`*n*`,}` | *X*, at least *n* times |
| *X*`{`*n*`,`*m*`}` | *X*, at least *n* but not more than *m* times |

### Reluctant quantifiers

| | |
|---|---|
| *X*`??` | *X*, once or not at all |
| *X*`*?` | *X*, zero or more times |
| *X*`+?` | *X*, one or more times |
| *X*`{`*n*`}?` | *X*, exactly *n* times |
| *X*`{`*n*`,}?` | *X*, at least *n* times |
| *X*`{`*n*`,`*m*`}?` | *X*, at least *n* but not more than *m* times |

### Possessive quantifiers

| | |
|---|---|
| *X*`?+` | *X*, once or not at all |
| *X*`*+` | *X*, zero or more times |
| *X*`++` | *X*, one or more times |
| *X*`{`*n*`}+` | *X*, exactly *n* times |
| *X*`{`*n*`,}+` | *X*, at least *n* times |
| *X*`{`*n*`,`*m*`}+` | *X*, at least *n* but not more than *m* times |

### Logical operators

| | |
|---|---|
| *XY* | *X* followed by *Y* |
| *X*`\|`*Y* | Either *X* or *Y* |
| `(`*X*`)` | X, as a capturing group |

### Back references

| | |
|---|---|
| `\`*n* | Whatever the *n*th capturing group matched |
| `\k<`*name*`>` | Whatever the named-capturing group "name" matched |

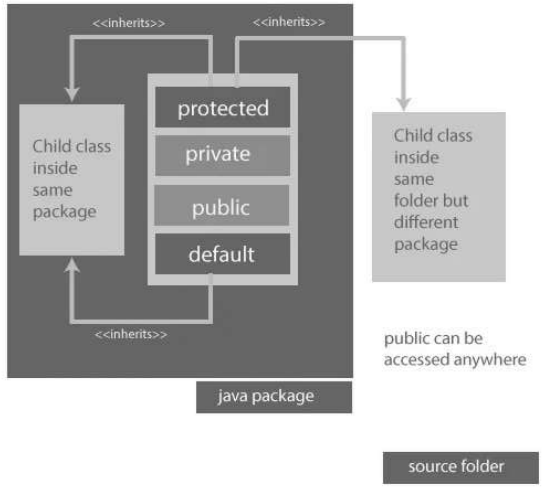
# Lecture 4 - (OOP I)



Java language is a **Procedural programming**.

## Access Modifiers / Visibility Modifiers.

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods. For example,




```
class Animal {
    public void method1() {...}

   private void method2() {...}
}
```

In the above example, we have declared 2 methods: method1() and method2(). Here,

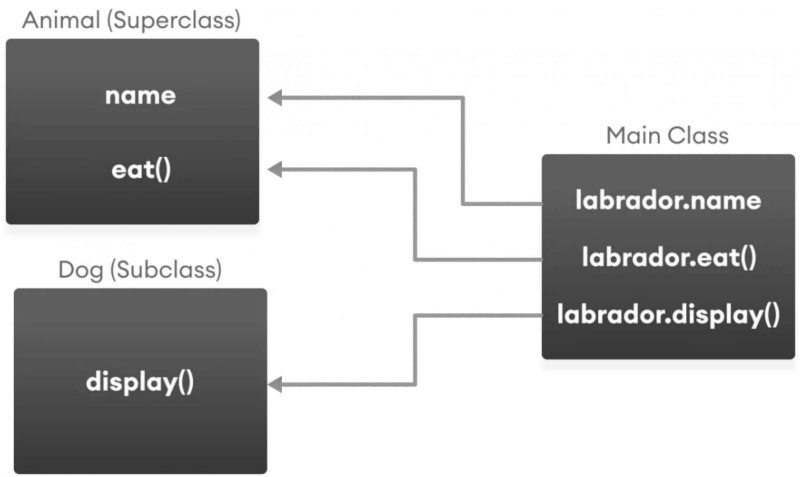
- method1 is `public` - This means it can be accessed by other classes.
- method2 is `private` - This means it can not be accessed by other classes.

### Types

**Types of Modifier**

| Modifier | Description |
|---|---|
| Default | declarations are visible only within the package (package private) |
| Private | declarations are visible within the class only |
| Protected | declarations are visible within the package or all subclasses |
| Public | declarations are visible everywhere |

## Inheritance




**is-a relationship**

In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

---

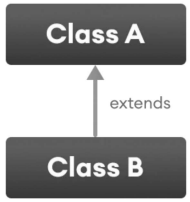
### Why use Inheritance?

Reusability

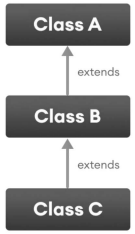
Method Overriding

### Types of Inheritance

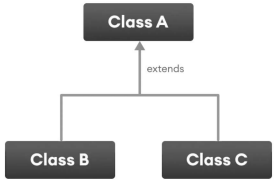
1. Single Inheritance



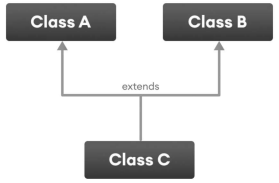

2. Multilevel Inheritance



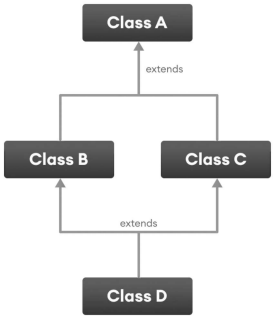

3. Hierarchical Inheritance




4. Multiple Inheritance




5. **Hybrid** Inheritance




# Method Overloading

Defining methods having same name but different signatures

- Signature : Name + types of parameters, return type

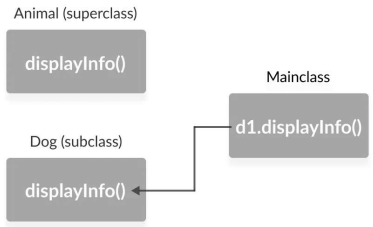
```
class A{
  int a;
  int b;
```

```
  void hey(){
    sysout("Blah blah");
  }

  void hey(int a){
    sysout("Professor is a bit moody today");
  }
  int hey (int a, int b){
    sysout("Blah blah blah");
    if (prof == moody)
      return 1;
    return 0;
  }

}
```

# Method Overriding

```
class Animal{
  public void displayInfo(){
    sysout("I am an Animal");
  }
}
class Dog extends Animal{
  @Override
  public void displayInfo(){
    sysout("I am a Dog");
  }
}

class Main{
  public static void main (String[] args){
  Dog d1 = new Dog();
  d1.displayInfo();
  }
}


Result --
I am a Dog.
```

When we call displayInfo() using the d1 object (object of the subclass), the method inside the subclass Dog is called. The displayInfo() method of the subclass overrides the same method of the superclass.




**Overriding Rules**

- Both superclass and subclass must have the same
  - method name,
  - return type
  - parameter list
- *Can't override the method declared as **final** and **static.***
- We should always override abstract methods of the superclass
- **Constructors in Java are not inherited. Hence, there is no such thing as constructor overriding in Java.**

**Incase of Abstract Method**

In Java, abstract classes are created to be the superclass of other classes. And, if a class contains an abstract method, it is mandatory to override it.

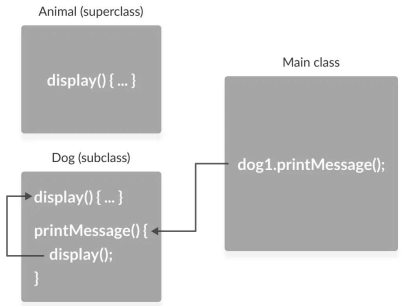
# Super keyword

We can access the methods and fields of the superclass using super keyword

```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   public void displayInfo() {
      super.displayInfo();
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}

Result --
  I am an animal.
  I am a dog.
```

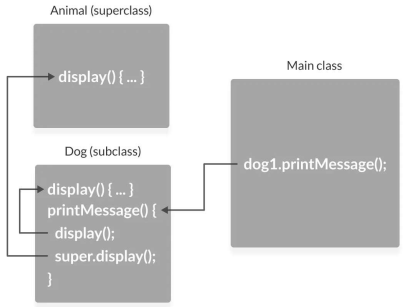
**Uses of super keyword**

- To call methods of superclass (i.e to override)




```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}

Result --
  I am a dog
```

- To access attributes (fields) of superclass




```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){

    // this calls overriding method
    display();

    // this calls overridden method
    super.display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
Result --
  I am a dog
  I am an animal
```

- To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor

```
class Animal {

  // default or no-arg constructor of class Animal
```

```
  Animal() {
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // default or no-arg constructor of class Dog
  Dog() {

    // calling default constructor of the superclass
    // Always the 1st line
    super();

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}


Result --
  I am a dog
  I am an animal
```

# This Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor. For example,

```
class Main {
    int instVar;

    Main(int instVar){
        this.instVar = instVar;
        System.out.println("this reference = " + this);
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("object reference = " + obj);
    }
}
```

## Using this in Constructor Overloading

While working with constructor overloading, we might have to invoke one constructor from another constructor. In such a case, we cannot call the constructor explicitly. Instead, we have to use `this` keyword.

Here, we use a different form of this keyword. That is, `this()`. Let's take an example,

```
class Complex {

    private int a, b;

    // constructor with 2 parameters
    private Complex( int i, int j ){
        this.a = i;
        this.b = j;
    }

    // constructor with single parameter
    private Complex(int i){
        // invokes the constructor with 2 parameters
        this(i, i);
    }

    // constructor with no parameter
    private Complex(){
        // invokes the constructor with single parameter
        this(0);
    }

    @Override
    public String toString(){
        return this.a + " + " + this.b + "i";
    }

    public static void main( String[] args ) {

        // creating object of Complex class
        // calls the constructor with 2 parameters
        Complex c1 = new Complex(2, 3);

        // calls the constructor with a single parameter
        Complex c2 = new Complex(3);

        // calls the constructor with no parameters
        Complex c3 = new Complex();

        // print objects
        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
    }
}
```

**Output**:

```
2 + 3i
3 + 3i
0 + 0i
```

In the above example, we have used `this` keyword,

- to call the constructor `Complex(int i, int j)` from the constructor `Complex(int i)`
- to call the constructor `Complex(int i)` from the constructor `Complex()`

Notice the line,

```
System.out.println(c1);
```

Here, when we print the object c1, the object is converted into a string. In this process, the `toString()` is called. Since we override the `toString()` method inside our class, we get the output according to that method.

One of the huge advantages of `this()` is to reduce the amount of duplicate code. However, we should be always careful while using `this()`.

This is because calling constructor from another constructor adds overhead and it is a slow process. Another huge advantage of using `this()` is to reduce the amount of duplicate code.

## this with Getters and Setters

Another common use of `this` keyword is in *setters and getters methods* of a class. For example:

```
class Main {
   String name;

   // setter method
   void setName( String name ) {
       this.name = name;
   }

   // getter method
   String getName(){
       return this.name;
   }

   public static void main( String[] args ) {
       Main obj = new Main();

       // calling the setter and the getter method
       obj.setName("Toshiba");
       System.out.println("obj.name: "+obj.getName());
   }
}
```

**Output**:

```
obj.name: Toshiba
```

# Final Keyword

**Once any entity (variable, method or class) is declared `final`, it can be assigned only once**. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

## 1. Java final Variable

In Java, **we cannot change the value of a final variable.** For example,

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

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

## 2. Java final Method

Before you learn about final methods and final classes, make sure you know about the Java Inheritance.

In Java, the `final` **method cannot be overridden by the child class**. For example,

```
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

## 3. Java final Class

In Java, the **final class cannot be inherited by another class.** For example,

```
// create a final class
final class FinalClass {
  public void display() {
```

```
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

In the above example, we have created a final class named FinalClass. Here, we have tried to inherit the final class by the Main class.

When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

# Casting

## Casting objects and the *instanceof* operator

- It is always possible to cast an instance of a subclass to a variable of a superclass (known as upcasting)
  - E.g. **Person p = new Employee();**
- When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used
  - E.g. **Person p = new Employee(); Employee e = (Employee)p;**
  - If the superclass object is not an instance of the subclass, a runtime error occurs
  - It is a good practice to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the ***instanceof*** operator
- Casting an object reference does not create a new object

```
Person p = new Person();
Employee e = (Employee)p;

//run time Error
```

```
Result --
    Exception
```

```
Professor f = new Professor();
Person w = new Person();
sysout(f instanceof Professor)  // True

sysout(w instanceof Professor)  // True

sysout(f instanceof Person)  // False
```

# The Object class

Every Java class has **object** as superclass

It has methods that are usually overridden:

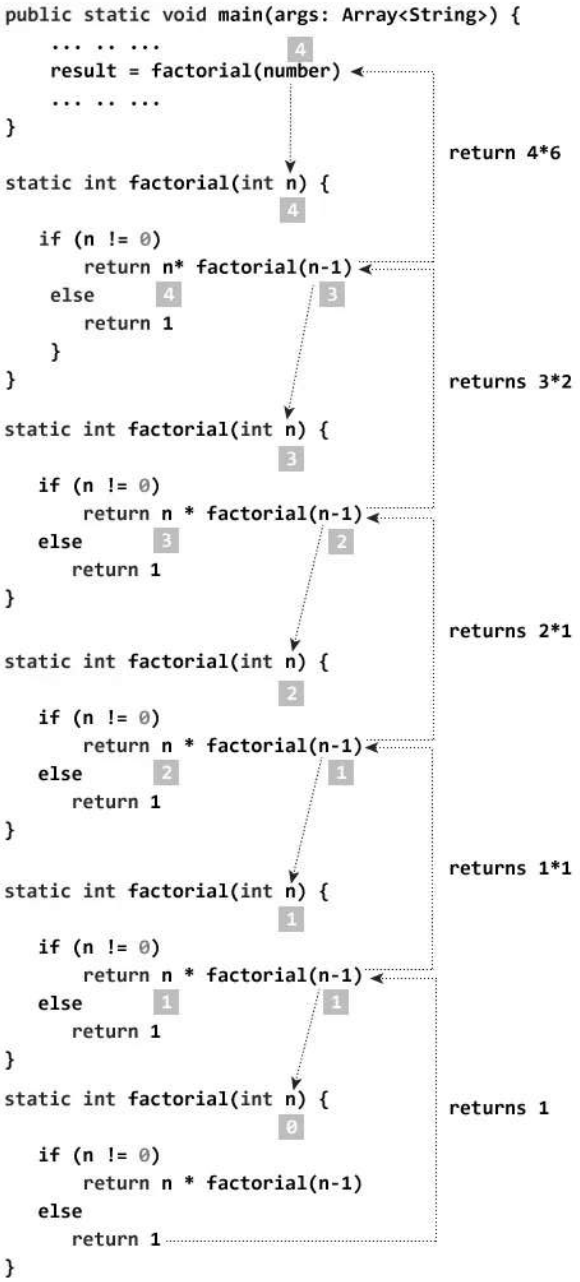
- **equals**

```
Person p1 = new Person('Yashank', 'Toronto');
Person p2 = new Person('Yashank', 'Toronto');
Person p3 = p1;
 // Person is a subclass of object
sysout(p1.equals(p2))
//Returns false, cause their reference is not the same.
// since, their addresses are not the same.
sysout(p1.equals(p3))     // True Cause their ref are same.
```

```
class Person{
String name;
String address;

  public boolean equals(Object obj){
    if (obj == null)
      return false;

    if (getClass != obj.getClass())
      return false;
    Person other = (Person)obj;
    if (name.equals(other.name) && address.equals(other.address))
      return true;
    return false;
    }
}
```

- **hashCode**
- **toString**

# Recursion


public static void main(args: Array<String>) {
... .. ...
result = factorial(number)
... .. ...
}
4
return 4*6
static int factorial(int n) {
4
if (n != 0)
return n* factorial(n-1)
else
return 1
}
4
3
returns 3*2
static int factorial(int n) {
3
if (n != 0)
return n * factorial(n-1)
else
return 1
}
3
2
returns 2*1
static int factorial(int n) {
2
if (n != 0)
return n * factorial(n-1)
else
return 1
}
2
1
returns 1*1
static int factorial(int n) {
1
if (n != 0)
return n * factorial(n-1)
else
return 1
}
1
1
static int factorial(int n) {
returns 1
0
if (n != 0)
return n * factorial(n-1)
else
return 1
}

## Advantages and Disadvantages of Recursion

When a recursive call is made, new storage locations for variables are allocated on the stack. As, each recursive call returns, the old variables and parameters are removed from the stack. Hence, recursion generally uses more memory and is generally slow.

On the other hand, a recursive solution is much simpler and takes less time to write, debug and maintain.

# InstanceOf Operator

The `instanceof` operator in Java is used to check whether an object is an instance of a particular class or not.

Its syntax is

```
objectName instanceOf className;
```

Here, if objectName is an instance of className, the operator returns `true`. Otherwise, it returns `false`.

### Example: Java instanceof

```
class Main {

  public static void main(String[] args) {

    // create a variable of string type
    String name = "Programiz";

    // checks if name is instance of String
      boolean result1 = name instanceof String; // prints true


    // create an object of Main
    Main obj = new Main();

    // checks if obj is an instance of Main
    boolean result2 = obj instanceof Main;  // prints true

  }
}
```

## Java instanceof during Inheritance

We can use the `instanceof` operator to check if objects of the subclass is also an instance of the superclass. For example,

```
// Java Program to check if an object of the subclass
// is also an instance of the superclass

// superclass
class Animal {
}

// subclass
class Dog extends Animal {
}

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

    // create an object of the subclass
    Dog d1 = new Dog();

    // checks if d1 is an instance of the subclass
    System.out.println(d1 instanceof Dog);        // prints true

    // checks if d1 is an instance of the superclass
    System.out.println(d1 instanceof Animal);     // prints true
  }
}
```

# Lecture 5 - (OOP II)



## Polymorphism

Every instance of a subclass is also an instance of its superclass, but not vice versa

**Polymorphism**: An object of a subclass can be used wherever its superclass object is used

```
public class Demo {
  public static void main(String [] args) {
    m(new Point(1,2));
  }
  public static void m(Object x) {
    System.out.println(x);
  }
}
```

We can achieve polymorphism in Java using the following ways:

1. Method Overriding

```
class Language {
  public void displayInfo() {
    System.out.println("Common English Language");
  }
}

class Java extends Language {
  @Override
  public void displayInfo() {
    System.out.println("Java Programming Language");
  }
}
```

2. Method Overloading

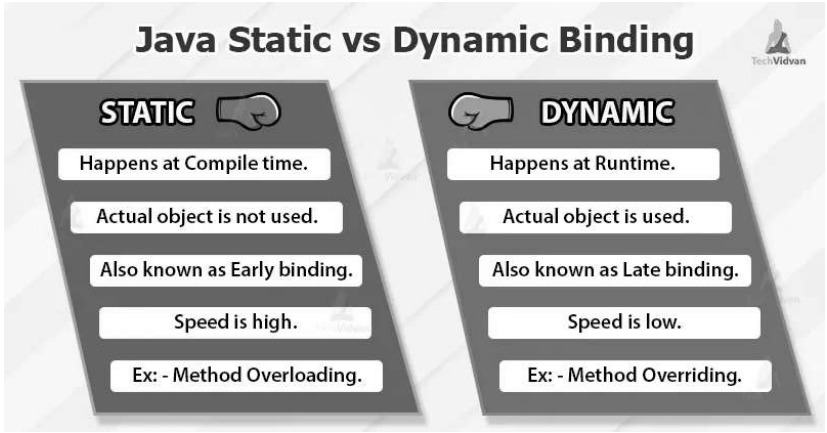
```
void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }
```

3. Operator Overloading

```
1. When + is used with numbers (integers and floating-point numbers), it performs mathematical addition. For example,

int a = 5;
int b = 6;

// + with numbers
int sum = a + b;  // Output = 11

2. When we use the + operator with strings, it will perform string concatenation (join two strings). For example,

String first = "Java ";
String second = "Programming";

// + with strings
name = first + second;  // Output = Java Programming

Here, we can see that the + operator is overloaded in Java to perform two operations: addition and concatenation.
```

## Dynamic Binding

- A method can be implemented in several classes along the inheritance chain




- The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable
  - E.g. Object x = new Point(1,2); //declared type: Object, actual type:Point
  - System.out.println(x); //which toString is invoked?
- Dynamic binding works as follows:
  - Suppose an object x is an instance of classes C1, C2, . . . , Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, . . . , and Cn-1 is a subclass of Cn,
  - If x invokes a method p, the JVM searches for the implementation of the method p in C1, C2, . . . , Cn-1, and Cn, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked

```
class Person
{
  public void speak()
  {
    System.out.println("Person speaks");
  }
}
class Teacher extends Person
{
  @Override
  public void speak()
  {
    System.out.println("Teacher speaks");
  }
}
public class DynamicBinding
{
  public static void main( String args[])
  {
    // Reference and objects are of Person type.
    Person obj2 = new Person();
        obj2.speak();  //Person speaks
    // Reference is of Person type and object is Teacher type
    Person obj = new Teacher();
        obj.speak();   //Teacher speaks
  }
}
```

**From the above code, we got different output because:**

- We have **not declared the methods as static** in the code.
- During compilation, the **compiler has no idea of which method to call**. This happens because the compiler doesn't go according to the type of the object but it checks only according to the reference variable. Therefore the binding gets delayed to runtime, so the respective version of the speak() method will be called on the basis of the type of the object

- **Static Binding or Early Binding** in Java refers to a process where the compiler determines the type of object and resolves the method during the compile-time. Generally, the compiler binds the overloaded methods using static binding.

  There is a fact that the binding of static, private, and final methods are always done during compile-time using static-binding.

```
package com.techvidvan.binding;
class Person
{
  public void speak()
  {
    System.out.println("Person speaks");
  }
}
class Teacher extends Person
{
  //error
  // You can't override a non-static method with a static one, or vice versa.
  public static void speak()
  {
    System.out.println("Teacher speaks");
  }
}
public class StaticBinding
{
  public static void main( String args[ ])
  {
    // Reference is of Person type and object is Teacher type
    Person obj = new Teacher();
        obj.speak();  //Person speaks
    // Reference and object both are of Person type.
    Person obj2 = new Person();
        obj2.speak(); //Person speaks
  }
}
```

**From the above code, we got the same output from the parent class. This happened because:**

- The **reference for the parent class and the child class is the same(Person).** That is, a single object refers to both of them.
- **Since the method is static,** the **compiler is aware that this method can not be overridden in the child class and it knows which method to call**. Therefore there is no ambiguity and the output is the same for both cases.

# Encapsulation

Encapsulation **refers to the bundling of fields and methods inside a single class.**

Encapsulation is one of the key features of object-oriented programming.

- The details of implementation are encapsulated and hidden from the user

- Modules communicate only through their APIs and are oblivious to each other's inner workings
  - Eg using System.out.println without knowing how it is implemented
- Advantages
  - Decoupling the modules that comprise a system allows them to be developed, tested optimized, used, understood and modified in isolation
    - It helps to decouple components of a system. **For example**, we can encapsulate code into multiple bundles.

      **These decoupled components (bundle) can be developed, tested, and debugged independently and concurrently**. And, any changes in a particular component do not have any effect on other components.
  - **Information hiding** increase software reuse because modules that aren't tightly couple often prove useful in other contexts
  - It helps to **control the values** of our data fields.
  - encapsulation helps us to **keep related fields and methods together**, which makes our code cleaner and easy to read.
  - The **getter and setter** methods provide **read-only** or **write-only** access to our class fields.
  - We can also achieve data hiding using encapsulation. In the above example, if we change the length and breadth variable into private, then the access to these fields is restricted. And, they are kept hidden from outer classes. This is called **data hiding**.
    - It **prevents outer classes from accessing and changing fields and methods of a class**. This also helps to achieve **data hiding**.
- The access control mechanism in Java facilitates encapsulation
- There are four possible access levels for members, listed in order of increasing accessibility:

1. **private—**The member is accessible only from the top-level class where it is declared
2. **package-private—**The member is accessible from any class in the package where it is declared (default access)
3. **protected—**The member is accessible from subclasses of the class where it is declared and from any class in the package where it is declared
4. **public—**The member is accessible from anywhere
   • Rule of thumb: make each member as inaccessible as possible

```
class Area {

  // fields to calculate area
  int length;
  int breadth;

  // constructor to initialize values
  Area(int length, int breadth) {
    this.length = length;
    this.breadth = breadth;
  }

  // method to calculate area
  public void getArea() {
    int area = length * breadth;
    System.out.println("Area: " + area);
  }
}

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

    // create object of Area
    // pass value of length and breadth
    Area rectangle = new Area(5, 6);
    rectangle.getArea();    // Area: 30 printed
  }
}
```

In the above example, we have created a class named Area. The main purpose of this class is to calculate the area.

To calculate an area, we need two variables: length and breadth and a method: `getArea()`. Hence, we bundled these fields and methods inside a single class.

Here, the fields and methods can be accessed from other classes as well. Hence, this is not **data hiding**.

This is only **encapsulation**. We are just keeping similar codes together.

## Data Hiding

Data hiding is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding.

We can use access modifiers to achieve data hiding.

```
class Person {

  // private field
  private int age;

  // getter method
  public int getAge() {
    return age;
  }

  // setter method
  public void setAge(int age) {
    this.age = age;
  }
}

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

    // create an object of Person
    Person p1 = new Person();

    // change age using setter
    p1.setAge(24);

    // access age using getter
    System.out.println("My age is " + p1.getAge()); /// My age is 24
  }
}
```

In the above example, we have a `private` field age. Since it is `private`, it cannot be accessed from outside the class.

In order to access age, we have used `public` methods: `getAge()` and `setAge()`. These methods are called getter and setter methods.

Making age private allowed us to restrict unauthorized access from outside the class. This is **data hiding**.

# Abstract Classes

- **Cannot be instantiated using the new operator**
- **Usually contain abstract methods** that are implemented in concrete subclasses
  - E.g. computeArea() in GeometricObject
- Abstract classes and abstract methods are **denoted using the abstract modifier in the header**
- **A class that contains abstract methods must be defined as abstract**
  - **Abstract class can contain no abstract method**
- **If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract**

The **abstract class in Java cannot be instantiated** (we cannot create objects of abstract classes). We use the `abstract` keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {

  // abstract method
  abstract void method1();
//In each of subclasses we implement this method

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

## Implementing Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

```
abstract class Animal {
  abstract void makeSound();   <- Abstract method

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}

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

    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }
}
```

**Output**

```
Bark bark
I can eat.
```

In the above example, we have created an abstract class Animal. The class contains an abstract method `makeSound()` and a non-abstract method `eat()`.

We have inherited a subclass Dog from the superclass Animal. Here, the subclass Dog provides the implementation for the abstract method `makeSound()`.

We then used the object d1 of the Dog class to call methods `makeSound()` and `eat()`.

## Accesses Constructor of Abstract Classes

**An abstract class can have constructors like the regular class. And, we can access the constructor of an abstract class from the subclass using the `super` keyword**. For example,

```
abstract class Animal {
   Animal() {
      ….
   }
}

class Dog extends Animal {
   Dog() {
      super();
      ...
   }
}
```

Here, we have used the `super()` inside the constructor of Dog to access the constructor of the Animal.

Note that the `super` should always be the first statement of the subclass constructor.

```
abstract class MotorBike {
  abstract void brake();
}

class SportsBike extends MotorBike {

  // implementation of abstract method
```

```
  public void brake() {
    System.out.println("SportsBike Brake");
  }
}

class MountainBike extends MotorBike {

  // implementation of abstract method
  public void brake() {
    System.out.println("MountainBike Brake");
  }
}

class Main {
  public static void main(String[] args) {
    MountainBike m1 = new MountainBike();
    m1.brake();
    SportsBike s1 = new SportsBike();
    s1.brake();
  }
}
```

## Key Points to Remember

- We use the `abstract` keyword to create abstract classes and methods.
- It **can have constructors and static methods also.**
- It **can have final methods** which will force the subclass not to change the body of the method.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We **cannot create objects** of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A **subclass must override all abstract methods** of an abstract class. However, **if the subclass is declared abstract, it's not mandatory to override abstract methods.**
- We **can access the static attributes and methods of an abstract class using the reference of the abstract class.** For example,

```
Animal.staticMethod();
```

# Interfaces

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

an interface is different from a class in several ways, including –

- You **cannot instantiate** an interface.
- An interface does **not contain any constructors.**
- All of the **methods** in an interface **are abstract.**
- An **interface cannot contain instance fields.** The **only fields that can appear in an interface must be declared both static and final.**
- An interface is **not extended by a class**; it is implemented by a class.
- An interface **can extend multiple interfaces**.

We use the `interface` keyword to create an interface in Java. For example,

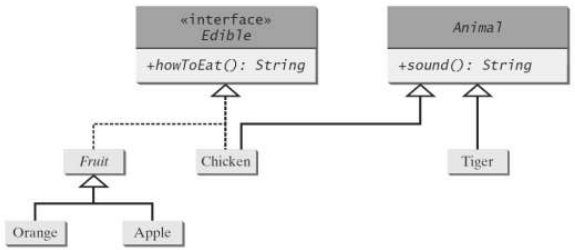
```
interface Language {
  public void getType();

  public void getVersion();
}
```

Here,

- Language is an interface.
- It includes abstract methods: `getType()` and `getVersion()`.

```
// create an interface
interface Language {
  void getName(String name);
}

// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: " + name);
  }
}

class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

- An interface can be used to define common behaviour for classes (including unrelated classes)
- **Contains only constants and abstract methods**
- Interfaces are denoted using the interface modifier in the header
- public interface Edible{
    public abstract String howToEat();
  }




UML (Unified Modelling Language) Model

```
abstract class Animal{
  public abstract String sound();
}

class Chicken extends Animal implements Edible{
  @Override
  public String howToEat(){
    return "Chicken: Fry it";
  }
  @Override
  public String sound(){
    return "Chicken: cock-a-doodle-doo";
  }
}
```

```
abstract class Fruit implements Edible{
  //Datafields, constructor, and metgods omitted here
}

class Apple extends Fruit {
  @Override
  public String howToEat(){
    return "Apple: Make apple cidar";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat(){
    return "Orange: Make orange juice";
  }
}
```

```
class Tiger extends Animal{
  @Override
  public String sound(){
    return "Tiger:
";
  }
}
```

## Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

```
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

## Extending an Interface

Similar to classes, interfaces can extend other interfaces. The `extends` keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

## Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

```
interface A {
  ...
}
interface B {
  ...
}

interface C extends A, B {
  ...
}
```

## Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

- Similar to abstract classes, interfaces help us to achieve **abstraction in Java**.

  Here, we know `getArea()` calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of `getArea()` is independent of one another.

- Interfaces **provide specifications** that a class (which implements it) must follow.

  In our previous example, we have used `getArea()` as a specification inside the interface `Polygon`. This is like setting a rule that we should be able to get the area of every polygon.

  Now any class that implements the `Polygon` interface must provide an implementation for the `getArea()` method.

- **Interfaces are also used to achieve multiple inheritance in Java**. For example, Here, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

  ```
  interface Line {
  …
  }

  interface Polygon {
  …
  }
  ```

```
class Rectangle implements Line, Polygon {
…
}
```

Rectangle

**Note: All the methods inside an interface are implicitly `public` and all fields are implicitly `public static final`. For example,**

```
interface Language {

  // by default public static final
  String type = "programming language";
  final int = 2;

  // by default public
  void getName();   // Doesnt matter if u use keyword abstract here or not
}
```

# Generics

- Enable Type Parameterization
  - Generic Interfaces
  - Generic Classes
  - Generic Methods
- Example: **ArrayList** class
  - ArrayList<Integer> A = new ArrayList<Integer>();
  - ArrayList<String> B = new ArrayList<String>();
- **Generics types must be reference types**
- Generic types **could be bounded using the extends keyword**

The Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).

This helps us to reuse our code.

**Note**: **Generics** does not work with primitive types (`int`, `float`, `char`, etc).

## Java Generics Class

We can **create a class that can be used with any type of data**. Such a class is known as Generics Class.

Here's is how we can create a generics class in Java:

**Example: Create a Generics Class**

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

    // initialize generic class
    // with Integer data
    GenericsClass<Integer> intObj = new GenericsClass<>(5);
    System.out.println("Generic Class returns: " + intObj.getData());
    // initialize generic class
    // with String data
    GenericsClass<String> stringObj = new GenericsClass<>("Java Programming");
    System.out.println("Generic Class returns: " + stringObj.getData());
  }
}

// create a generics class
class GenericsClass<T> {

  // variable of T type
  private T data;

  public GenericsClass(T data) {
    this.data = data;
  }

  // method that return T type variable
  public T getData() {
    return this.data;
  }
}
```

**Output**

```
Generic Class returns: 5
Generic Class returns: Java Programming
```

In the above example, we have created a generic class named GenericsClass. This class can be used to work with any type of data.

```
class GenericsClass<T> {...}
```

Here, T used inside the angle bracket `<>` indicates the **type parameter**. Inside the `Main` class, we have created two objects of GenericsClass

- **intObj - Here, the type parameter T is replaced by `Integer`. Now, the GenericsClass works with integer data.**
- **stringObj - Here, the type parameter T is replaced by `String`. Now, the GenericsClass works with string data.**

## Java Generics Method

Similar to the generics class, we can also create a method that can be used with any type of data. Such a class is known as Generics Method.

Here's is how we can create a generics class in Java:

**Example: Create a Generics Method**

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

    // initialize the class with Integer data
    DemoClass demo = new DemoClass();

    // generics method working with String
    demo.<String>genericsMethod("Java Programming");

    // generics method working with integer
    demo.<Integer>genericsMethod(25);
```

```
  }
}

class DemoClass {

  // create a generics method
  public <T> void genericsMethod(T data) {
    System.out.println("Generics Method:");
    System.out.println("Data Passed: " + data);
  }
}
```

**Output**

```
Generics Method:
Data Passed: Java Programming
Generics Method:
Data Passed: 25
```

In the above example, we have created a generic method named genericsMethod.

```
public <T> void genericMethod(T data) {...}
```

Here, the type parameter `<T>` is inserted after the modifier `public` and before the return type `void`.

We can call the generics method by placing the actual type `<String>` and `<Integer>` inside the bracket before the method name.

**Note**: We can **call the generics method without including the type parameter**. For example,

```
demo.genericsMethod("Java Programming");
```

In this case, the compiler can match the type parameter based on the value passed to the method.

## Bounded Types

In general, the **type parameter** can accept any data types (except primitive types).

However, if we want to use **generics for some specific types (such as accept data of number types) only,** then we can use **bounded types.**

In the case of bound types, we use the `extends` keyword. For example,

```
<T extends A>
```

This means T can only accept data that are subtypes of A.

**Example: Bounded Types**

```
class GenericsClass <T extends Number> {

  public void display() {
    System.out.println("This is a bounded type generics class.");
  }
}
```

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

    // create an object of GenericsClass
    GenericsClass<String> obj = new GenericsClass<>(); //Error
  }
}
```

In the above example, we have created a class named GenericsClass. Notice the expression, notice the expression

```
<T extends Number>
```

Here, GenericsClass is created with bounded type. This means GenericsClass can only work with data types that are children of `Number` (`Integer`, `Double`, and so on).

However, we have created an object of the generics class with `String`. In this case, we will get the following error.

```
GenericsClass<String> obj = new GenericsClass<>();
                                                ^
    reason: inference variable T has incompatible bounds
      equality constraints: String
      lower bounds: Number
  where T is a type-variable:
    T extends Number declared in class GenericsClass
```

## Advantages of Java Generics

### 1. Code Reusability

With the help of generics in Java, **we can write code that will work with different types of data**. For example,

```
public <T> void genericsMethod(T data) {...}
```

Here, we have created a generics method. This same method can be used to perform operations on integer data, string data, and so on.

### 2. Compile-time Type Checking

The **type parameter** of generics provides information about the type of data used in the generics code. For example,

```
// using Generics
GenericsClass<Integer> list = new GenericsClass<>();
```

Here, we know that GenericsClass is working with `Integer` data only.

Now, if we try to pass data other than Integer to this class, the program will generate an error at compile time.

### 3. Used with Collections

The collections framework uses the concept of generics in Java. For example,

```
// creating a string type ArrayList
ArrayList<String> list1 = new ArrayList<>();

// creating a integer type ArrayList
ArrayList<Integer> list2 = new ArrayList<>();
```

In the above example, we have used the same ArrayList class to work with different types of data.

Similar to `ArrayList`, other collections (`LinkedList`, `Queue`, `Maps`, and so on) are also generic in Java.

## Generics (The Comparable Interface)

- **Comparable** is a generic interface
  - Defines the **compareTo** method for comparing objects
- Defined as follows:

```
public interface Comparable<T>{
    public int compareTo(T t);
}
```

- The **compareTo** method determines the order of the calling object with **t** and returns a negative integer zero, or a positive integer if the calling object is less than, equal to. or greater than **t**
- Many classes implement Comparable (eg String Integer)
- Implementing comparable :

```
public class Point implements Comparable<Point> {
    // class body
    // blah blah

    @Override
    public int compareTo(Point p){
        // implementation
    }

}
```

## Generics (The ArrayList class)

- Arrays cant be usred to store lists oif objects. Howevere, once an array is created, its size is fixed
- Java provides the generic class **ArrayList** whose **size is variable**
- imported using: import java.util.ArrayList
- Commonly used methods (ArrayList<E>)
  - boolean add(E e)
  - E get (int index)
  - int size()
  - boolean contains(Object o)
  - int indexOf(Object o)
- An ArrayList could be traversed using a for-each loop

For more info go to Lecture→ CSCB07 → collection framework

## Generics (The HashSet class)

- Generic class that can be used to store elements **without duplicates**
  - No two elements e1 and e2 can be in the set such that e1.equals(e2) is true
- Imported using: import *java.util.HashSet;*
- **Objects added to the hash set should override *equals* and *hashCode* properly**
- Commonly used methods (HashSet)
  - boolean add(E e)
  - int size()
  - boolean contains(Object o)
- A HashSet could be traversed using a for-each loop

For more info go to Lecture→ CSCB07 → collection framework

## Generics (The LinkedHashSet class)

- Elements of a HashSet are not necessarily stored in the same order they were added
- LinkedHashSet is a subclass of HashSet with a linked-list implementation that **supports an ordering of the elements** in the set
- Imported using: import java.util.LinkedHashSet;

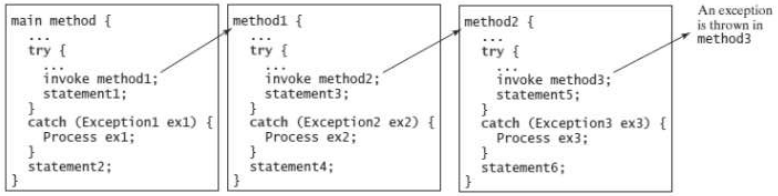
# Exceptions

- Exception handling enables a **program to deal with exceptional situations and continue its normal execution**
- An **exception** is an object that represents an error or a condition that prevents execution from proceeding normally
- Exceptions are represented in the **Exception class, which describes errors caused by the program and by external circumstances**
- **Developers can create their own exception classes by extending Exception or a subclass of Exception**
- In Java, **runtime exceptions are represented in the RuntimeException** class. Subclasses include:
  - ArrayIndexOutOfBoundsException
  - NullPointerException
- **RuntimeException and its subclasses are known as unchecked exceptions**
- **All other exceptions are known as checked exceptions**
  - The compiler forces the programmer to check and deal with them in a trycatch block or declare them in the method header

- Declaring exceptions
  - **Every method must state the types of checked exceptions it might throw using the throws keyword in the header**
  - E.g. **public void myMethod() throws Exception1, Exception2, ..., ExceptionN**
- Throwing exceptions
  - A program that detects an error can create an instance of an appropriate exception type and throw it using the throw keyword
  - E.g. **throw new IllegalArgumentException("Wrong Argument");**
- When an exception is thrown, it can be caught and handled in a trycatch block. For example:

```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

- If no exceptions arise during the execution of the try block, the catch blocks are skipped
- If **one of the statements inside the try block throws an exception**
  - The **remaining statements in the try block are skipped**
  - **Each catch block is examined in turn, from first to last,** to see whether the type of the exception object is an instance of the exception class in the catch block
  - **If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler**
  - **If no handler is found in the chain of methods** being invoked, the **program terminates** and **prints an error message** on the console




- **Java has a finally clause that can be used to execute some code regardless of whether an exception occurs or is caught.** For example:

```
try {
  //statements;
}
catch Exception ex) {
  //handling ex;
}
finally {
  //final statements;
}
```

Custom exception

```
class Lecture4Exception extends Exception{
  String message;
  public Lecture4Exception(String message){
    this.message = message
  }
}
```

```
public static void foo(int n) throws lecture4Exception{
  if (n< 0){
    throw new Lecture4Exception("Negative values are not al
  }
  else{
    sysout("...");
  }
}
```

```
public class driver{

  //public static void bar(int n) throws Lecture4Exception{
  //    Person.foo(0);
  //}

//OR

  public static void bar(int n) {
    try{
      Person.foo(0);
      // someother exception then it goes to last exception
      sysout("!!!!!");   // Will be skipped if above throws exception
    }catch(Lecture4Exception ex){
        sysout("...")
    }catch(Exception ex){
        sysout(".../")
    }

  public static void main(String[] args){
    bar(0);
  }
}
```


# Exception Handling



## Exception:-

The abnormal condition/situation of the program at runtime which disturbs the normal flow and then rest of the code of program is known as Exception.

## Exception Handling:-

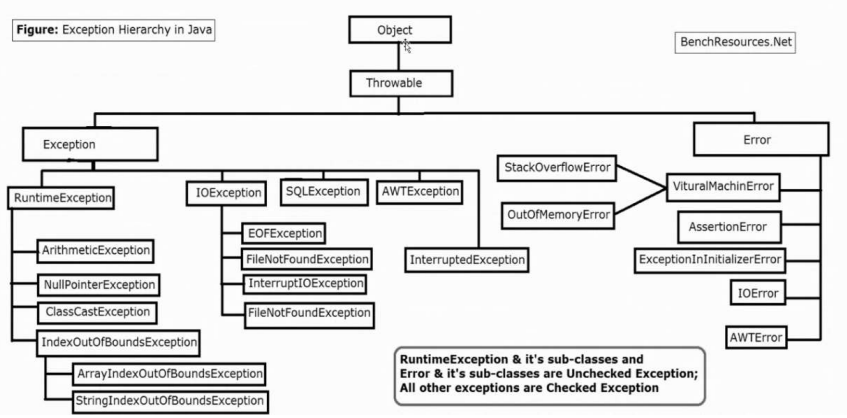
Exception handling simply means handling the runtime exception so that our program do not get terminate abnormally.

- Exception can be handled but errors cant be handled

## Errors:-

Errors are problem that disturbs the programs and terminate our program abnormally. It occurs due to lack of system resources. It is not recoverable.

From coding point of view **exception and errors** is an object in java and is represented by a class 'java.lang.Exception' and 'java.lang.Error'.




```
package com.yashank;
public class Calculate {
  public static void main(String[] args) {
    int a = 45;
    int b = 0;
    int c= a/b;

    System.out.println("Processing");
    System.out.println(c);
  }

}
Result:
  Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.yashank.Calculate.main(Calculate.java:7)
```

**Solution:**

```
package com.yashank;

public class Calculate {
  public static void main(String[] args) {
    int a = 45;
    int b = 0;
    try {
      int c= a/b;
      System.out.println("Processing");
      System.out.println(c);
    }catch(Exception e){
      System.out.println(e);
    }

    //OR
    //catch(ArithemeticException e){
      //something
    //}

    System.out.println("Completed");

  }

}


Result:
  java.lang.ArithmeticException: / by zero
  Completed
```

## More than one catch block:

```
package com.yashank;
import java.util.InputMismatchException;
import java.util.Scanner;

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


    try {
      Scanner s = new Scanner(System.in);
      System.out.println("Enter value for a ");
      int a = s.nextInt(); // Input -- s
      System.out.println("Enter value for b ");
      int b = s.nextInt();

      int c= a/b;
      System.out.println("Processing");
      System.out.println(c);
    }catch (ArithmeticException e){
      System.out.println(e);
    }catch(InputMismatchException e) {
      System.out.println(e);
      System.out.println("Input error");
    }
    catch(Exception e){
      System.out.println(e);
      System.out.println("Another Error");
    }

    System.out.println("Completed");

  }

}

Result:
  Enter value for a
  xyz
  java.util.InputMismatchException
  Input error
  Completed
```

## Nested try-catch

```
public class p1 {
  public static void main(String[] args) {
    try {
      int a = 832/0;
        // code below this wont work

          try {
            Class.forName("Calculato");
          }catch(Exception e) {
            System.out.println("Nested try catch");
          }


    }catch(Exception e){
      System.out.println("Main try catch");
    }

    System.out.println("Outside try catch");
  }
}

Result --
  Main try catch
  Outside try catch
```

```
public class p1 {
  public static void main(String[] args) {
    try {
          try {
            Class.forName("Calculato");
          }catch(Exception e) {
            System.out.println("Nested try catch");
          }
    }catch(Exception e){
      System.out.println("Main try catch");
    }
```

```
    System.out.println("Outside try catch");
  }
}

Result --
  Nested try catch
  Outside try catch
```

```
public class p1 {
  public static void main(String[] args) {
    try {
          try {
            Class.forName("Calculato");
          }catch(ArithematicException e) {
            System.out.println("Nested try catch");
          }
    }catch(Exception e){
      System.out.println("Main try catch");
    }

    System.out.println("Outside try catch");
  }
}

Result --
  Main try catch
  Outside try catch
```

# Throws Keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

## Syntax of Java throws

```
int method_name() throws exception_class_name{
  //method code
}
```

In Java, exceptions can be categorized into two types:

- **Unchecked Exceptions:** They are **not checked at compile-time but at run-time**.
  - Forexample: `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`, exceptions under `Error` class, etc.
- **Checked Exceptions:** They are checked at **compile-time**.
  - For example, `IOException`, `InterruptedException`, etc.

Refer to Java Exceptions to learn in detail about checked and unchecked exceptions.

Usually, **we don't need to handle unchecked exceptions**. It's because unchecked exceptions occur due to programming errors. And, it is a good practice to correct them instead of handling them.

## Example:

```
import java.io.*;
class Main {
  public static void findFile() throws IOException {
    // code that may produce IOException
    File newFile=new File("test.txt");
    FileInputStream stream=new FileInputStream(newFile);
  }

  public static void main(String[] args) {
    try{
      findFile();
    } catch(IOException e){
      System.out.println(e);
    }
  }
}

Result --
  java.io.FileNotFoundException: test.txt (No such file or directory)
```

## Throwing multiple exceptions

```
import java.io.*;
class Main {
  public static void findFile() throws NullPointerException, IOException, InvalidClassException {

    // code that may produce NullPointerException
    … … …

    // code that may produce IOException
    … … …

    // code that may produce InvalidClassException
    … … …
  }

  public static void main(String[] args) {
    try{
      findFile();
    } catch(IOException e1){
      System.out.println(e1.getMessage());
```

```
    } catch(InvalidClassException e2){
      System.out.println(e2.getMessage());
    }
  }
}
```

Here, the `findFile()` method specifies that it can throw `NullPointerException`, `IOException`, and `InvalidClassException` in its `throws` clause.

Note that we have not handled the `NullPointerException`. **This is because it is an unchecked exception.** It is not necessary to specify it in the `throws` clause and handle it.

# Throw keyword

Custom exception is thrown through *throw keyword.

- Not using try catch.

The `throw` keyword is used to explicitly throw a **single exception**.

When an exception is thrown, the flow of program execution transfers from the `try` block to the `catch` block. We use the `throw` keyword within a method.

## Syntax:

```
throw throwableObject;
```

A throwable object is an instance of class Throwable or subclass of the Throwable class.

## Example

```
class Main {
  public static void divideByZero() {
    throw new ArithmeticException("Trying to divide by 0");
  }

  public static void main(String[] args) {
    divideByZero();
  }
}

Result --
  Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
      at Main.divideByZero(Main.java:3)
      at Main.main(Main.java:7)
  exit status 1
```

In this example, we are explicitly throwing an `ArithmeticException.`

**Note:** `ArithmeticException` is an unchecked exception. It's usually not necessary to handle unchecked exceptions.

```
import java.io.*;
class Main {
  public static void findFile() throws IOException {
    throw new IOException("File not found");
  }

  public static void main(String[] args) {
    try {
      findFile();
      System.out.println("Rest of code in try block");
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}

Result --
  File not found
```

The `findFile()` method throws an `IOException` with the message we passed to its constructor.

Note that **since it is a checked exception, we must specify it in the `throws` clause.**

The methods that call this `findFile()` method need to either handle this exception or specify it using `throws` keyword themselves.

We have handled this exception in the `main ()` method. The flow of program execution transfers from the `try` block to `catch` block when an exception is thrown. So, the rest of the code in the `try` block is skipped and statements in the `catch` block are executed.

# Final vs finalize vs finally

## Final:

```
Final int xyz = 32;  // Must be initialized here
```

## Finalize:

The finalize() method of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection, so as to perform clean-up activity.

Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation.

Remember it is not a reserved keyword. Once the finalize method completes immediately Garbage Collector destroy that object.

### Why finalize method is used()?

finalize() method releases system resources before the garbage collector runs for a specific object. JVM allows finalize() to be invoked only once per object.

Now, the finalize method which is present in the **Object class**, has an **empty implementation**, in our class clean-up activities are there, then we have to **override this method** to define our own clean-up activities.

In order to **Override this method**, we have to explicitly define and call finalize within our code.

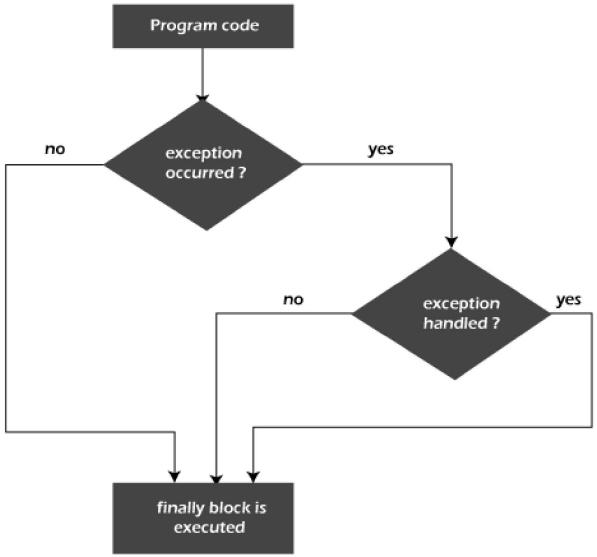
```
// Java code to show the
// overriding of finalize() method

import java.lang.*;

// Defining a class demo since every java class
// is a subclass of predefined Object class
// Therefore demo is a subclass of Object class
public class demo {  // Object class is Parent of all class

    protected void finalize() throws Throwable
    {
        try {

            System.out.println("inside demo's finalize()");
        }
        catch (Throwable e) {

            throw e;
        }
        finally {

            System.out.println("Calling finalize method"
                               + " of the Object class");

            // Calling finalize() of Object class
            super.finalize();
        }
    }

    // Driver code
    public static void main(String[] args) throws Throwable
    {

        // Creating demo's object
        demo d = new demo();

        // Calling finalize of demo
        d.finalize();
    }
}

Result --
  inside demo's finalize()
  Calling finalize method of the Object class
```

## Finally

**Java finally block** is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.




### Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

### Usage of Java finally

Let's see the different cases where Java finally block can be used.

## Case 1: When exception doesnt occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

**TestFinallyBlock.java**

```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

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


Result--
  5
  finally block is always executed
  rest of the code...
```

## Case 2: when an exception occurs but not handled by the catch block

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

**TestFinallyBlock1.java**

```
public class TestFinallyBlock1{
      public static void main(String args[]){

      try {

        System.out.println("Inside the try block");

        //below code throws divide by zero exception
       int data=25/0;
       System.out.println(data);
      }
      //cannot handle Arithmetic type exception
      //can only accept Null Pointer type exception
      catch(NullPointerException e){
        System.out.println(e);
      }
      //executes regardless of exception occured or not
      finally {
        System.out.println("finally block is always executed");
      }

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


Result--
  Inside the try block
  finally block is always executed
  Exception in thread "main" java.lang.ArithmeticException: / by zero at
            TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

## Case 3: When an exception occurs and is handled by the catch block

**Example:**

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

**TestFinallyBlock2.java**

```
public class TestFinallyBlock2{
      public static void main(String args[]){

      try {

        System.out.println("Inside try block");

        //below code throws divide by zero exception
       int data=25/0;
       System.out.println(data);
      }

      //handles the Arithmetic Exception / Divide by zero exception
      catch(ArithmeticException e){
        System.out.println("Exception handled");
        System.out.println(e);
      }

      //executes regardless of exception occured or not
      finally {
        System.out.println("finally block is always executed");
      }

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

Result --
  Inside try block
  Exception handled
  java.lang.ArithmeticException: / zero
  finally block is always executed
  rest of the code...
```

# Java Custom Exception

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

```
// class representing custom exception
class InvalidAgeException  extends Exception
{
    public InvalidAgeException (String str)
    {
        // calling the constructor of parent Exception
        super(str);
    }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

    // method to check the age
    static void validate (int age) throws InvalidAgeException{
       if(age < 18){

        // throw an object of user defined exception
        throw new InvalidAgeException("age is not valid to vote");
    }
       else {
        System.out.println("welcome to vote");
        }
     }

    // main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");

            // printing the message from InvalidAgeException object
            System.out.println("Exception occured: " + ex);
        }

        System.out.println("rest of the code...");
    }
// Or instead of try catch using throws
    public static void main(String args[]) throws InvalidAgeException
    {
        validate(13);
    }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.