DEPARTMENT OF INFORMATION SCIENCE

Session: Aug 2015 – Dec 2015

LAB MANUAL

Semester:	V
Sub Code:	12CS307
Sub:	**PRINCIPLES OF PROGRAMMING LANGUAGES LABORATORY**

LIST OF EXPERIMENTS/PROGRAMS

Week No	***Program No***	***Title of the program***
1	Instruction / Introduction class	Introduction to Principles of Programming Languages (PPL) Lab – debugging in gcc using gdb debugger
2	Program 1	Examine name and scope: - static variables in 'C' - call resolution in Java
3	Program 2	Examine garbage and memory leak in C - Develop a mechanism to avoid/detect memory leak
4	Program 3	Examine assignment operation - assignment of arrays in Java - assignment of lists in Python - assignment of structures in C
5	Program 4	Examine goto statement - scope of goto in 'c' - jump into/out of the block - non-local goto.
6	Program 5	Examine callbacks - callbacks in C - interface and inner classes in Java
7	Program 6	Examine closure - in python - in C
8	Program 7	Examine functions - variable # of args in C - variable # of args in Java - variable # of args in Python
9	Program 8	Examine functions - tail recursion - keyword parameter in python - stack smashing in 'C'
10	Program 9	Examine Generics - implicit and explicit instantiation in Java - type erasure in Java - classes and methods
11	Program 10	- Examine Generics (cont…) - Lists – linked lists Array lists - Sets – hash set Pre set Link hash set

		- Map
12	Program 11	Examine inheritance - @override in Java - Final in Java - Multiple inheritance in python - Downcasting in Java
13	Program 12	Examine Java thread model / pthread / Python - racing - synchronization - interthread communication - Thread local storage

These experiments will be conducted on LINUX platform using C, Java and Python programming languages.

Week #1 : Instruction class - Introduction to PPL Lab

Learning objectives	• Overview of PPL lab • Debugging in Linux

Debugging in LINUX

Write a sample C program with errors for debugging purpose:

```
$ vi factorial.c
# include <stdio.h>

int main()
{
        int i, num, j;
        printf ("Enter the number: ");
        scanf ("%d", &num );

        for (i=1; i<num; i++)
                j=j*i;

        printf("The factorial of %d is %d\n",num,j);
}
$ cc factorial.c

$ ./a.out
Enter the number: 3
The factorial of 3 is 12548672
```

Follow the steps mentioned below for debugging the above program in gdb.

Step 1. Compile the C program with debugging option -g

Compile your C program with -g option. This allows the compiler to collect the debugging information.

```
$ cc -g factorial.c
```

Note: The above command creates a.out file which will be used for debugging.

Step 2. Launch gdb

Launch the C debugger (gdb) as shown below.

```
$ gdb a.out
```

Step 3. Set up a break point inside C program

```
Syntax: break line_number
```

This places break point in the C program on the line number specified.

```
break 10
Breakpoint 1 at 0x804846f: file factorial.c, line 10.
```

Step 4. Execute the C program in gdb debugger

```
run [args]
```

You can start running the program using the run command in the gdb debugger. You can also give command line arguments to the program via run args. The example program we used here does not requires any command line arguments so let us give run, and start the program execution.

```
run
Starting program: /home/student/Debugging/c/a.out
```

Once you executed the C program, it would execute until the first break point, and give you the prompt for debugging.

```
Breakpoint 1, main () at factorial.c:10
10                     j=j*i;
```

You can use various gdb commands to debug the C program as explained in the sections below.

Step 5. Printing the variable values inside gdb debugger

```
Syntax: print {variable}

Examples:
print i
print j
print num
(gdb) p i
$1 = 1
(gdb) p j
$2 = 3042592
(gdb) p num
$3 = 3
(gdb)
```

As you see above, in the factorial.c, we have not initialized the variable j. So, it gets garbage value resulting in a big numbers as factorial values.

Fix this issue by initializing variable j with 1, compile the C program and execute it again.

Even after this fix there seems to be some problem in the factorial.c program, as it still gives wrong factorial value.

So, place the break point in 10th line, and continue as explained in the next section.

Step 6. Continue, stepping over and in – gdb commands

There are three kind of gdb operations you can choose when the program stops at a break point. They are continuing until the next break point, stepping in, or stepping over the next program lines.

- c or continue: Debugger will continue executing until the next break point.
- n or next: Debugger will execute the next line as single instruction.
- s or step: Same as next, but does not treats function as a single instruction, instead goes into the function and executes it line by line.

gdb command shortcuts

Use following shortcuts for most of the frequent gdb operations.

- l – list
- p – print
- c – continue
- s – step
- ENTER: pressing enter key would execute the previously executed command again.

Miscellaneous gdb commands

- **l command:** Use gdb command l or list to print the source code in the debug mode. Use l line-number to view a specific line number (or) l function to view a specific function.
- **bt: backtrack** – Print backtrace of all stack frames, or innermost COUNT frames.
- **help** – View help for a particular gdb topic — help TOPICNAME.
- **quit** – Exit from the gdb debugger.

Week #2 – Examine name and scope

Learning objectives:	**Static Variables** • Understanding the internal and external scope of a static variable. **Call resolution in Java and C++** • Understanding call resolution with different function signatures. • Difference between call resolution in C++ and Java. **Mangling of names in C/C++** • Understanding the purpose of name mangling • Viewing the mangled names in the program stack • Ways to prevent mangling

Static Variables:

The following cases are examined:

i) Accessing a static variable which has function scope
ii) Accessing a static variable which has file scope
iii) To examine the default value of the static variable.
iv) Can the static variable be externed?
v) Declaring the static variable in the header file and examining the behavior in multiple implementation files
vi) Examining the scope of a static variable within a compilation unit
vii) External and internal linkages

- Declare a static var in a function and access it from another function

```
/***case 1: A static variable defined within a function and accessing
it outside a function ***/

#include<stdio.h>

int callStatic(void);

int main()
{
  callStatic();
  printf("count in Main= %d\n", count);
}

int callStatic()
{
  static int count =0;
  count++;
  printf("count = %d", count);
  return 1;
}

cc staticinfunc.c
staticinfunc.c: In function â€˜mainâ€™:
staticinfunc.c:7: error: â€˜countâ€™ undeclared (first use in this
function)
staticinfunc.c:7: error: (Each undeclared identifier is reported only
once
staticinfunc.c:7: error: for each function it appears in.)
staticinfunc.c:5: warning: return type of â€˜mainâ€™ is not â€˜intâ€™
```

Conclusion: A static variable defined within a function scope cannot be referred from another function in the same file.

- View the default value of static variable.

Command:

Static variable when defined within a file but outside of any function

```
#include<stdio.h>

int callStatic(void);
static int count =0;
int main()
{
  int i;
  for(i=0; i<2; i++)
    callStatic();
  printf("count in Main= %d\n", count);
}

int callStatic()
{
  count++;
  printf("count = %d ", count);
  return 1;
}

/**output***/
count = 1 count = 2 count in Main= 2
```

Conclusion: A static variable defined within the file scope can be accessed across functions in the same file.

To examine if static variables be externed?

```
/***statextern.c****/

#include<stdio.h>
int main()
{
static int z=10;
printf("value of z is %d",z);
}
/*****output*****/
value of z is 10

/***statextern1.c****/
#include<stdio.h>
int main()
{
extern z;
printf("value of z is%d",z);
}

/tmp/ccYixHDk.o: In function `main':
statextern1.c:(.text+0x12): undefined reference to `z'
collect2: ld returned 1 exit status
```

Conclusion: A static variable cannot be externed.

Behavior of static variables when declared in a header file and used in more than one .c files

```
/***a1.h****/
static int a=10;
/***b1.c******/
#include<stdio.h>
#include"a1.h"
int main()
{
  a++;
  printf("value of a in b1.c is %d",a);
}
/*****output****/
value of a in b1.c is 11

/*****c1.c*******/
#include<stdio.h>
#include"a1.h"

int main()
{
  a=a+100;
  printf("value of a in c1.c is%d",a);
}
/*****output****/
value of a in c1.c is 110
```

Conclusion: When a .h file containing a static variable is included in the several .c files, a new memory is allocated for that variable for that particular file, which means that different files will use their respective copies of the static variable during the program execution.

Examine the behavior of static variable within a compilation unit

```
/****comunit.c******/
#include<stdio.h>
int main()
{
   int a = 10;
   if(a>0)
   {
      static int b=10;
      printf("value of b within compilation unit is %d",b);
   }
   printf("value of b outside compilation unit is %d",b);
}

comunit.c: In function â€˜mainâ€™:
comunit.c:10: error: â€˜bâ€™ undeclared (first use in this function)
comunit.c:10: error: (Each undeclared identifier is reported only
once
comunit.c:10: error: for each function it appears in.)

/*case2*/
/******comunit1.c*********/
#include<stdio.h>
int main()
{
   int s;
   for(s=0;s<2;s++)
   {
      static int z=10;
      printf("\nvalue of z is %d\n",z);
      z=z+20;
      printf("\nvalue of z+20 is %d",z);
   }
   //printf("value of outside for loop is %d",z);
}

/*****output**********/
value of z is 10

value of z+20 is 30
value of z is 30

value of z+20 is 50
```

/*******conclusion*******/
The scope of static variable declared within a compilation unit is within that unit only. Outside the loop the static variable is inaccessible.

Returning a pointer to static variable.(internal and external linkages)

```
#include<stdio.h>

int * callStatic(void);
//static int count =0;
int main()
{
  int *i;
 //for(i=0; i<2; i++)
  i = callStatic();
  printf("count in Main= %d\n", *i);
}

int *callStatic()
{
  static int count =0;
  count = count + 10;
  printf("count = %d", count);
  return &count;
}

/***output***/
count = 10 count in Main= 10
```

Call resolution in C++ & Java

Compare the order of execution between C++ and Java

C++

- find candidate functions
- find best candidate
- check for access

JAVA

- find candidate functions
- check for access
- find best candidate

```
/*call resolution - the normal way */

#include <iostream>
using namespace std;
int add(int x, int y)
{
  int sum;
  sum = x+y;
 return sum;
}

int add(int x)
{
  int sum;
  sum = x + 100;
  return sum;
}
int main()
{
        int total;
        total = add(10,20);
        cout << "Total1 = "<< total << endl;
        total = add(10);
         cout << "Total2 = "<< total << endl;
        return 0;
}

/****output*****/
Total1 = 30
Total2 = 110

/****function signature differing only in return types******/
/*namemangle1.cpp*/
#include <iostream>
using namespace std;

int add(int x, int y)
{
  int sum;
  sum = x+y;
  return sum;
}
```

```
float add(int a,int b)
{
  int sum;
  sum = a+b;
  return sum;
}
int main()
{
        int total;
        total = add(10,20);
        cout << "Total1 = "<< total << endl;
        total = add(10,40);
        cout << "Total2 = "<< total << endl;
        return 0;
}

namemangle1.cpp: In function â€˜float add(int, int)â€™:
namemangle1.cpp:14: error: new declaration â€˜float add(int,
int)â€™
namemangle1.cpp:5: error: ambiguates old declaration â€˜int
add(int, int)â€™
namemangle1.cpp: In function â€˜int main()â€™:
namemangle1.cpp:27: warning: converting to â€˜intâ€™ from
â€˜floatâ€™
namemangle1.cpp:30: warning: converting to â€˜intâ€™ from
â€˜floatâ€™

/*******call resolution with automatic type conversion******/
#include <iostream>
using namespace std;

int add(int x,float y)
{
  int sum;
  sum = x+y;
  return sum;
}
int add(float a,int b)
{
  int sum;
  sum = a+b+100;
  return sum;
}

int main()
{
        int total;
        total = add(10.0,'a');
        cout << "Total1 = "<< total << endl;
        total = add(10,40.0);
        cout << "Total2 = "<< total << endl;
        return 0;
}
/*********output************/
```

```
Total1 = 207
Total2 = 50

/***function signature with default arguments********/

#include<iostream>
using namespace std;
int func1(int x,int y=24)
{
  int s;
  s=x+y;
  return(s);
}

int func2(int x,int y)
{
  int s1;
  s1=x+y;
  return(s1);
}
int  main()
{
  int sum;
  sum=func1(20);
  cout<<"Sum is\n"<<sum;
  sum=func1(20,30);
  cout<<"\nsum is\n "<<sum;
}

/*********output*********/
Sum is 44
sum is 50

/*constant int and const as argument******/
#include<iostream>
#include<stdio.h>

using namespace std;

int func ( int x) {
    return x+10;
}

int func( const int x) {
    return x+10;
}

int main() {
  func(20);
  getchar();
  return 0;
}
```

```
callresol.cpp: In function â€˜int func(int)â€™:
callresol.cpp:10: error: redefinition of â€˜int func(int)â€™
callresol.cpp:6: error: â€˜int func(int)â€™ previously defined
here
```

```
/*********function signature with a int pointer and int array
ambiguity*********/

#include<iostream>
using namespace std;
int fun(int *ptr)
{
  cout << "IN FUNC1" << endl;

}

int fun(int ptr[])
{
  cout << "IN FUNC2" << endl;

}
int main()
{
  int a[4]={10,20,30};

  fun(a);

}
```

```
callresol1.cpp: In function â€˜int fun(int*)â€™:
callresol1.cpp:9: error: redefinition of â€˜int fun(int*)â€™
callresol1.cpp:3: error: â€˜int fun(int*)â€™ previously
defined here
```

Mangling of names in C++

- **Variable name mangling**
- **Function name mangling**

```
void one(int);
void two(char,char);
int main()
{
          one(1);
          two('a','b');
          return 0;
}
$ g++ -c linkage.cpp
/* Examine the object file using nm command */
```

The characters that identify symbol type describe :

- **A** : Global absolute symbol.
- **a** : Local absolute symbol.

- **B** : Global bss symbol.
- **b** : Local bss symbol.
- **D** : Global data symbol.
- **d** : Local data symbol.
- **f** : Source file name symbol.
- **L** : Global thread-local symbol (TLS).
- **l** : Static thread-local symbol (TLS).
- **T** : Global text symbol.
- **t** : Local text symbol.
- **U** : Undefined symbol

```
$ nm linkage.o
         U _Z3onei
        U _Z3twocc
        U __gxx_personality_v0
        00000000 T main
```

Ways to prevent name mangling(Can C++ code be accessed within extern C linkage)

```
extern "C"
{
       void one(int);
       void two(char,char);
}
 int main()
{
      one(1);
      two('a','b');
      return 0;
}
$ g++ -c linkage2.cpp
$ nm linkage2.o
               U __gxx_personality_v0
               00000000 T main
               U one
      U two
```

- **Name mangling on different compilers**

Command to see the mangled names in the stack:

nm command

Viva Questions:

1. What is the default value of static variable and Why?
   Uninitialized static variables are allocated form BSS(Block Started Symbol) section of the memory and initializes to 0 by default.

2. How does a static variable remember its previous value?

A static variable can retain it's previous value because it is not allocated from the stack. It's allocated from the data section of the memory.

3. Is the static variable allocated from stack or heap?
   Stack

4. What is name mangling? How to prevent name mangling?
   Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language.

   To prevent the C++ compiler from mangling the name of a function, you can apply the extern "C" linkage specifier to the declaration or declarations, as shown in the following example:

```
extern "C" {
        int f1(int);
        int f2(int);
        int f3(int);
};
```

   This declaration tells the compiler that references to the functions f1, f2, and f3 should not be mangled.

5. When do you prevent name mangling?
   Name mangling is not desirable when linking C modules with libraries or object files compiled with a C++ compiler.
   The `extern "C"` linkage specifier can also be used to prevent mangling of functions that are defined in C++ so that they can be called from C.

6. Is name mangling same across all compilers?
   No

Week #3 – Examine Garbage and Memory leak

Problem Statement	Mechanism to detect memory leak. Implementation of reference counted garbage collector.
Learning objectives	• Understanding memory leak • Ways to prevent memory leak

Reference counting is a common optimization (also called "lazy copy" and "copy on write").

There are many ways to detect a memory leak and thereby avoid it. Many tools have been developed to detect mem leaks. Eg: Valgrind is one such open source tool which detects the mem leaks.

```
/**memleak.c**/
#include <stdlib.h>
#include <mcheck.h>

int main(void) {
        mtrace();

        int * a;

        a = malloc(sizeof(int)); /* allocate memory and assign it to
the pointer */

        return 0; /* we exited the program without freeing memory */

        /* we should have released the allocated memory with the
statement "free(a)" */
muntrace();


}
```

```
/** setting the environment */
MALLOC_TRACE=/home/YourUserName/path/to/program/MemLeakReport.txt
export MALLOC_TRACE;

gcc memLeak.c -g
./a.out

mtrace ./a.out MemLeakReport.txt

/**this prints the line number which has caused the memory leak else
it will print NO MEMORY LEAKS **/
```

```
#include <stdio.h>


void display(int *a[5])

{

for (int b=0;b<5;b++)

{

printf("%d th element is ::%d\n",b,*a[b]);

        }
}


int main()

{
        int* a[5];

        int i=1,j=2,k=3,l=4,m=5;

a[0]=&i;

a[1]=&j;

a[2]=&k;

a[3]=&l;

a[4]=&m;

        display(a);

        free(*a[0]);

        *(a[0])=8;

         free(a);
        display(a);
        return 0;
}
```

The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct. The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour.
The rest of this guide gives the minimum information you need to start detecting memory errors in your program with Memcheck. For full documentation of Memcheck and the other tools, please read the User Manual.

2. Preparing your program
Compile your program with -g to include debugging information so that Memcheck's error messages include exact line numbers. Using -O0 is also a good idea, if you can tolerate the slowdown. With -O1 line numbers in error messages can be inaccurate, although generally speaking running Memcheck on code compiled at -O1 works fairly well, and the speed improvement compared to running -O0 is quite significant. Use of -O2 and above is not recommended as Memcheck occasionally reports uninitialised-value errors which don't really exist.
3. Running your program under Memcheck
If you normally run your program like this:

```
myprog arg1 arg2
```

Use this command line:

```
valgrind --leak-check=yes myprog arg1 arg2
```

Memcheck is the default tool. The --leak-check option turns on the detailed memory leak detector.
Your program will run much slower (eg. 20 to 30 times) than normal, and use a lot more memory. Memcheck will issue messages about memory errors and leaks that it detects.http://valgrind.org/docs/manual

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

Things to notice:

There is a lot of information in each error message; read it carefully.
The first no as in 19182 is the process ID; it's usually unimportant.

The first line ("Invalid write...") tells you what kind of error it is. Here, the program wrote to some memory it should not have due to a heap block overrun.

Below the first line is a stack trace telling you where the problem occurred. Stack traces can get quite large, and be confusing, especially if you are using the C++ STL. Reading them from the bottom up can help. If the stack trace is not big enough, use the --num-callers option to make it bigger.

The code addresses (eg. 0x804838F) are usually unimportant, but occasionally crucial for tracking down weirder bugs.

Memory leak messages look like this:

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

The stack trace tells you where the leaked memory was allocated. Memcheck cannot tell you why the memory leaked, unfortunately. (Ignore the "vg_replace_malloc.c", that's an implementation detail.)

There are several kinds of leaks; the two most important categories are:

"definitely lost": your program is leaking memory -- fix it!

"probably lost": your program is leaking memory, unless you're doing funny things with pointers (such as moving them to point to the middle of a heap block).

Memcheck also reports uses of uninitialised values, most commonly with the message "Conditional jump or move depends on uninitialised value(s)". It can be difficult to determine the root cause of these errors. Try using the --track-origins=yes to get extra information. This makes Memcheck run slower, but the extra information you get often saves a lot of time figuring out where the uninitialised values are coming from.

Viva Questions:

1. Can delete be called on a void pointer? What would be the behavior ?
2. What is the difference between dangling pointer and NULL pointer?
3. What happens if we call free() twice on a pointer?
4. List down the problems in the below "C" code:
   ```
   Int * ptr;
   for(int i=0; i<n; i++) {
           ptr = (int*) malloc(sizeof(int));
           scanf("%d",ptr);
           printf("%d", *ptr);
   ```
   line1: ptr is a dangling pointer
   ptr is not freed
5. "static int *ptr;"   Is this a dangling pointer?
6. Is garbage collection in Java under the control of the user?
7. What are the commands to request JVM to do garbage collection?
8. What is the disadvantage with the reference counting algorithm?

Week #4 : Assignment operation in C, Java and Python

Problem Statement	Assignment of arrays in Java Assignment of lists in Python Assignment of structures in C
Learning objectives	• Understand Shallow copy and Deep copy.

Assignment of lists in Python

(a) In Python, the assignment operator (=) assign the reference to the list instead of making copy of it.

```
>>> first=[10,20,30]
>>> firstCopy=first
>>> firstCopy
[10, 20, 30]
>>> firstCopy.append(40)
>>> firstCopy
[10, 20, 30, 40]
>>> first
[10, 20, 30, 40]
```

(b) To make a new copy of a list, there are two different approaches: using the list() function and using the sub list mechanism.

```
>>> org=[1,2,3,4,5]
>>> cpy=list(org)
>>> cpy.append(6,7)
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    cpy.append(6,7)
TypeError: append() takes exactly one argument (2 given)
>>> cpy.append(6)
>>> cpy
[1, 2, 3, 4, 5, 6]
>>> org
[1, 2, 3, 4, 5]
```

Sublist copy:

```
>>> org
[1, 2, 3, 4, 5]
>>> sublistcpy=org[:]
>>> sublistcpy.append(6)
>>> sublistcpy
[1, 2, 3, 4, 5, 6]
>>> org
[1, 2, 3, 4, 5]
```

Assignment in python is always by reference.

```
>>> x=[1,2]
>>> y=x
>>> id(x)
31585224
>>> id(y)
31585224
```

Accessing the list:
If you pass in a negative index, Python adds the length of the list to the index. L[-1] can be used to access the last item in a list.

```
>>> x
[1, 2]
>>> x[0]
1
>>> x[1]
2
>>> x[2]
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    x[2]
IndexError: list index out of range
>>> x[-1]
2
>>> x[-2]
1
```

Assignment of Arrays in Java:

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

class pgm3
{
 public static void main(String args[])throws Exception
 {
  int a[]=new int[5];
  int []b=new int[a.length];
  int c[];


  for(int i=0; i<4; i++)
          a[i]=i;

  System.out.println("Content 0f both the arrays after direct assigment" + " are as follows");

   b=a;
   for(int i=0; i<4; i++)
   System.out.println(a[i]+ ">>" + b[i]);

        System.out.println("Chaging fourth element of B to 100");
```

```
            b[4]=100;
            System.out.println(a[4] + ">>" + b[4]);

            System.out.println("Using arraycopy() function..");
            System.arraycopy(a,0,b,0,5);

             for(int i=0; i<5; i++)
        System.out.println(a[i]+ ">>" + b[i]);

            System.out.println("Using arrayList feature..");
            ArrayList a1=new ArrayList(),b1=new ArrayList();

            a1.add(1);
        a1.add(2);
            a1.add(3);
            a1.add(4);

            b1=a1;

            System.out.println(a1);
            System.out.println(b1);


        System.out.println("Adding two elements to list A");
            a1.add(5);
            a1.add(6);

            System.out.println(a1);
            System.out.println(b1);


            System.out.println("Using clone() feature..");

        c=a.clone();
            System.out.print(c.toString());
        System.out.println("Changing the fifth element of A");
        a[5]=444;
        System.out.print(c.toString());

}
}
```

Assignment of Structures in C:

```
#include<stdio.h>

struct student
{
   char name[20];
   int roll_no;
};

int main()
```

```
{
   struct student s1,s2;
   printf("Enter the name of student:");
   scanf("%s",s1.name);
   printf("Enter the roll no. of student:");
   scanf("%d",&s1.roll_no);
   s2=s1;
   printf("%s = %d\n",s1.name,s1.roll_no);
  printf("%s = %d\n",s2.name,s2.roll_no);
   return 0;
}
```

```
#include<iostream>
#include<string.h>
using namespace std;

struct student
{
    char name[20];
    int roll;

     student(){};
     student(char s[],int n)
     {
         strcpy(name,s);
         roll=n;
     }
     void display()
     {
        cout<<name<<" "<<roll<<endl;
     }
};


int main()
{
    student s1("vinay",411),s2;
    s1.display();

   cout<<"Enter the student's name and number\n";
    cin>>s2.name>>s2.roll;
   s1=s2;

    cout<<"Structure data..\n";
    s1.display();
    s2.display();
    return 0;
}
```

Week #5 – Examine goto

Learning objectives:	**Scope of goto in C** • Understanding scope of goto in functions • Understanding scope of goto in files **Jump into/out of the block** • Is it possible to jump into and out of blocks? **Non-local goto** • Understanding the use of setjmp and longjmp functions

Scope of goto in C

-Use goto statement in one function and try to transfer control to another function by defining label in that function.
/case1**/**

```
/***scope of goto in functions******/
/****gotofn1.c***/
#include<stdio.h>
int main()
{
int c=0;
func();
v:
c=c+1;
printf("%d",c);
}
int func()
{

goto v;
}

cc gotofn1.c

gotofn1.c: In function â€˜funcâ€™:
gotofn1.c:13: error: label â€˜vâ€™ used but not defined
```

Conclusion: The scope of goto is only within a function.

Use goto statement in one file and try to transfer control to another file by defining label in that file.

```
/*********scope of goto in files*******/
/*gotoinfile.c*/
#include<stdio.h>
int  main()
{
int z=10;
printf("%d",z);
goto label;

}
```

```
cc gotoinfile.c


gotoinfile.c: In function â€˜mainâ€™:
gotoinfile.c:6: error: label â€˜labelâ€™ used but not defined

/*gotoinfile1.c*/
#include<stdio.h>
int main()
{
label:
}


cc gotoinfile1.c

gotoinfile1.c: In function â€˜mainâ€™:
gotoinfile1.c:5: error: label at end of compound statement
```

Conclusion: The label defined in a function in one file cannot be referred from a function in the another file.

Jump into/out of the block

- Use goto statement outside a block and try to refer to label defined inside the block.

```
/*jump into if*/
#include<stdio.h>
int main()
{
  int a;
  printf("enter a num greater than 10\n");
  scanf("%d",&a);
  goto x;
  if(a<10)
  {
     x:
     a=a+10;
     printf("\n%d",a);

  }
}

/*output*/
enter a num greater than 10
23
33
```

Conclusion: It is possible to jump into and out of a block using goto avoiding conditional execution.

Non-local goto

-Understanding the use of setjmp and longjmp functions

```
#include <setjmp.h>
#include <stdio.h>

  jmp_buf ebuf;
  void f2(void);

  int main(void)
  {
    int i;

    printf("1 ");
    i = setjmp(ebuf);
    if(i == 0) {
      f2();
      printf("This will not be printed.");
    }
    printf("%d", i);

    return 0;
  }

  void f2(void)
  {
    printf("2 ");
    longjmp(ebuf, 3);
  }

/********output*********/

1 2 3
```

Conclusion: Using setjmp and longjmp functions we can transfer the control from one function to another.

VIVA

1. What happens if the longjmp is called with a value(2nd parameter) of 0?
2. What is the difference between goto and setjmp & longjmp?
3. How to implement non-local jump in C?
4. What construct does Java provide to jump out of nested loops?

Week #6– Examine Callbacks

Learning objectives:	• Callbacks in C • Interface and inner classes in Java -Familiarization of Static nested class -Familiarization of Non-Static nested class

Callbacks in C

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
// comparison-function for the sort-algorithm
// two items are taken by void-pointer, converted and compared
int CmpFunc(const void* _a, const void* _b)
{
  // you've got to explicitly cast to the correct type
  const float* a = (const float*) _a;
  const float* b = (const float*) _b;
  if(*a > *b)
    return 1; // first item is bigger than the second one -> return 1
 else
    if(*a == *b)
        return 0; // equality -> return 0
   else
        return -1; // second item is bigger than the first one -> return -1
}

// example for the use of qsort()
void QSortExample()
{
    float field[100];
     ::randomize(); // initialize random-number-generator
    for(int c=0;c<100;c++) // randomize all elements of the field
       field[c]=random(99);

    qsort((void*) field, /*number of items*/ 100, /*size of an item*/ sizeof(field[0]),
             /*comparison-function*/ CmpFunc);

   printf("The first ten elements of the sorted field are ...\n");
  for(int c=0;c<10;c++)
       printf("element #%d contains %.0f\n", c+1, field[c]);
  printf("\n");
}
```

Functors in C++

```
#include<iostream>
using namespace std;

/***abstract base class ***/
class TFunctor
{
        public:
```

```
 /*** two possible functions to call member function. virtual cause
derived classes will use a pointer to an object and a pointer to a
member function to make the function call **/

        virtual void operator()(const char* string)=0; // call using
operator
        virtual void Call(const char* string)=0;       // call using
function
};

// derived template class
template <class TClass> class TSpecificFunctor : public TFunctor
{
   private:
    void (TClass::*fpt)(const char*);   // pointer to member function
    TClass* pt2Object;                  // pointer to object

   public:
      // constructor - takes pointer to an object and pointer to a
member and stores them in two private variables
      TSpecificFunctor(TClass* _pt2Object, void(TClass::*_fpt)(const
char*))
      {
        pt2Object = _pt2Object; fpt=_fpt;
      };

        // override operator "()"
        virtual void operator()(const char* string)
        {
                (*pt2Object.*fpt)(string);
        };              // execute member function

        // override function "Call"
        virtual void Call(const char* string)
        {
                (*pt2Object.*fpt)(string);
        };             // execute member function
};

// dummy class A
class TClassA
{
public:
   TClassA(){};
   void Display(const char* text)
   {
      cout << text <<"   INDEED in ClassA"<< endl;
   };
   /* more of TClassA */
};

// dummy class B
class TClassB
{
  public:
    TClassB(){};

    void Display(const char* text) { cout << text << "   INDEED in
ClassB"<<endl; };
```

```
    /* more of TClassB */
};

// main program
int main(int argc, char* argv[])
{
      // 1. instantiate objects of TClassA and TClassB
        TClassA objA;
        TClassB objB;
      // 2. instantiate TSpecificFunctor objects ...
         //    a ) functor which encapsulates pointer to object and
to member of TClassA
        TSpecificFunctor<TClassA> specFuncA(&objA, TClassA::Display);
         //    b) functor which encapsulates pointer to object and to
member of TClassB
        TSpecificFunctor<TClassB> specFuncB(&objB,&TClassB::Display);

        TSpecificFunctor<TClassB> specFuncB(&objB,&TClassB::Display);
        // 3. make array with pointers to TFunctor, the base class,
and initialize it
        TFunctor* vTable[] = { &specFuncA, &specFuncB };
        // 4. use array to call member functions without the need of
an object
        vTable[0]->Call("TClassA::Display called!");        // via
function "Call"
        (*vTable[1])   ("TClassB::Display called!");        // via
operator "()"
        cout << endl << "Hit Enter to terminate!" << endl;
        cin.get();
        return 0;
}
/************************OUTPUT****
 * TClassA::Display called!   INDEED in ClassA
 * TClassB::Display called!   INDEED in ClassB

 *
 * Hit Enter to terminate!
 * *****************************/
```

Interface and inner classes in Java

- Familiarization of Static nested class

```
/**********static nested classes********/
/*********staticnes1.java*********/

 class InnerClassDemo
    {
    public static void main(String ar[])
    {
    //Outerclass o=new Outerclass();
    Outerclass.Inner in=new Outerclass.Inner();//Inner in=new
Outer().new Inner();
    //in.m1();
    //Outerclass.Inner in= new Inner();
    in.m1();
    }
    }
    class Outerclass
    {
    public static class Inner
    {
    public void m1()
    {
    System.out.println("I am in Inner Class");
    }
    }
    }
/**********output************/

I am in Inner Class
```

Conclusion: Function defined within a static nested class can be accessed only with the help of outer class object.

-Familiarization of non-static nested class(inner class)

```
/***********innerclasses in java***********/
/*****innerclass.java*********/

class Outerclass
    {
    private int x=10;
    private static int y=20;
    class Inner
    {
    public void m1()
    {
    System.out.println("instance variable..."+x+",static 
variable..."+y);
    }
    }
    public static void main(String args[])
    {
    Outerclass o=new Outerclass();
    Outerclass.Inner in=o.new Inner();//Inner in=new Outer().new 
Inner();
    in.m1();
    }
    }

/********output***********/
instance variable...10,static variable...20
```

Conclusion: Function defined within a non-static nested class (inner) can be accessed by specifying the outer class name.

VIVA QUESTIONS

1. What is a callback function?

   A callback function is one that is not invoked explicitly by the programmer; rather the responsibility for its invocation is delegated to another function that receives the callback function’s reference and instructions as to when to run the callback function.

Think of it as an “In case of fire, break glass” subroutine. Many computer programs tend to be written such that they expect a certain set of possibilities at any given moment. If “Thing That Was Expected”, then “Do something”, otherwise, “Do something else.” is a common theme. However, there are many situations in which events (such as fire) could happen at any time. Rather than checking for them at each possible step (“Thing that was expected OR Things are on fire”), it is easier to have a system which detects a number of events, and will call the appropriate function upon said event (this also keeps us from having to write programs like “Thing that was expected OR Things are on fire OR Nuclear meltdown OR alien invasion OR the dead rising from the grave OR…etc., etc.) Instead, a callback routine is a sort of insurance policy. If zombies attack, call this function. If the user moves their mouse over an icon, call HighlightIcon, and so forth.

2. When should callbacks be used?
3. What is a functor? How do you realize them in C++?
4. How is callback implemented in Python?
5. What is the difference between functor and function pointer?

Closures

1. What is meant by Referencing environment?
2. What is a closure? When should it be used?
3. What is the difference between anonymous inner class of java and closures?
4. Write atleast 2 applications of closures.
5. Mention how languages like C, C++ and C## support closures?

Week #7 : Examine Closures

Problem Statement	Closures in Python Closures in C
Learning objectives	• Understand the concept of Referencing environment • Understand the language support for closure

Closures in Python:

```
def makeInc(x):
    def inc(y):
# x is "closed" in the definition of inc
        return y + x

    return inc

inc5 = makeInc(5)
inc10 = makeInc(10)

inc5 (5) # returns 10
inc10(5) # returns 15
```

Closures in python are created by function calls. Here, the call to makeInc creates a binding for x that is referenced inside the function inc. Each call to makeInc creates a new instance of this function, but each instance has a link to a different binding of x. The example shows the closure of x being used to eliminate either a global or a constant, depending on the nature of x.

Week #8 : Examine Functions

Problem Statement	Variable # of arguments in C Variable # of arguments in Java Variable # of arguments in Python
Learning objectives	•

Variable # of arguments in C:

```
/*******variable number of args -c***********/
/*varagrs.c*/
#include <stdarg.h>
#include <stdio.h>
#include <stdarg.h>
int sum( int num, ... ) {
   va_list args;
   va_start(args, num);

   int i, total = 0;
   for( i = 0; i < num; ++i ) {

        total += va_arg(args, int);
    }
    va_end(args);
    return total;
}

int main() {
   int total = sum(5, 10,15,20,30,40);
    printf("SUM IS: %d\n", total);
    return 0;
}

/********output*********/
SUM IS: 115
```

The stdarg library provides us with the **va_list** data type, as well as the macros **va_start**, **va_arg**, and **va_end** for manipulating the list of arguments.
To declare the function, we use a triple dot (...) to signify that the function accepts a variable number of arguments. The first argument, is used to indicate how many arguments were passed into the function.

va_list is a data type used by stdarg to hold the list of arguments passed into the function.

va_start is a macro used to initialize the argument list so that we can begin reading arguments from it.

va_end is another macro that cleans up our object for us when we're done using it.

Variable # of arguments in Java:

```
class sum
{
     public static void main(String args[]){
     System.out.println("The sum is " + calculateSum(10,20,30));
```

```
   }

   private static int calculateSum(int... values)
   {
    int sum = 0;
    for (int count = 0; count < values.length; count ++)
    {
      sum = sum + values[count];
    }
    return sum;
   }
}
/********output*********/
The sum is 60
```

Writing your own printf function using strarg.h

```
#include <stdio.h>
#include <stdarg.h>

/* print all non-negative args one at a time;
   all args are assumed to be of int type */
void printargs(int arg1, ...)
{
  va_list ap;
  int i;

  va_start(ap, arg1);
  for (i = arg1; i >= 0; i = va_arg(ap, int))
    printf("%d ", i);
  va_end(ap);
  putchar('\n');
}

int main(void)
{
   printargs(5, 2, 14, 84, 97, 15, 24, 48, -1);
   printargs(84, 51, -1);
   printargs(-1);
   printargs(1, -1);
   return 0;
}

/**output**/

5 2 14 84 97 15 24 48
84 51

1
```

/****

To call other var args functions from within your function (such as sprintf) you need to use the var arg version of the function (vsprintf in this example):

***/

void MyPrintf(const char* format, ...)

```
{
  va_list args;
  char buffer[BUFSIZ];

  va_start(args,format);
  vsprintf (buffer, format, args );
  FlushFunnyStream(buffer);
  va_end(args);
}
```

Variable # of arguments in Python:

```
def findAvg(*args):
      myList = list(args)
      total = 0;
      size = len(args)

      while myList:
            total += myList.pop()

      avg = total/float(size)
      return args, myList, total, avg


def manyArgs(*arg):
      print "I was called with ", len(arg), "arguments : ", arg


#can send multiple key-value args too
def myfunc(**kwargs):
      #kwargs is a dictionary
      for k,v in kwargs.iteritems():
            print "%s = %s" %(k,v)

def myfunc2(*args, **kwargs):
   for a in args:
       print a
   for k,v in kwargs.iteritems():
       print "%s = %s" % (k, v)



print(findAvg(1, 2, 3))
manyArgs(1, 2, 3)
manyArgs(1, 2, 3, 4, 5)
myfunc(abc=123, cde=456)
myfunc2(1, 2, 3, banan=123)
```

Variable # of arguments in Java:

```
import java.util.*;

class varArgs
{
      public static void main(String[] args)
      {
            int result;

            result = findSum(10, 20, 30);
            System.out.println("Result = "+result);


            result = findSum(1, 2, 3, 4, 5);
            System.out.println("Result = "+result);
      }

      static int findSum(int ... numbers)
      {
            int total = 0;

            for(int i = 0; i<numbers.length; ++i)
            {
                  total = total + numbers[i];
            }

            return total;
      }
}
```

Week #9: Examine Functions

Problem Statement	- tail recursion - keyword parameter in python - stack smashing in 'C'
Learning objectives	• Understand the difference between tail recursion and traditional recursion • Understand the difference between default parameters and Keyword parameters • Understand the behavior of the stack

/**Stack Smashing****/**

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int i;
   char a1[2]="he";
        char a2[2]="ll";
        char b1[3]="he";
        char b2[3]="ll";

       int s[5]={1,2,3,4,5};
   int v[5]={7,8,9,10,11};

   printf("\nBeforestack smash");
   printf("\ns=");
   for(i=0;i<5;i++)
      printf("%d ",s[i]);

   printf("\nv=");
   for(i=0;i<5;i++)
      printf("%d ",v[i]);
   printf("\n");

   v[5]=100;
   v[6]=200;
   v[7]=300;
   printf("\nAfter stack smash");
   printf("\ns=");
        for(i=0;i<5;i++)
                printf("%d ",s[i]);

        printf("\nv=");
        for(i=0;i<8;i++)
                printf("%d ",v[i]);

        printf("\n%s",a1);
```

```
    printf("\n%s",a2);
    strcat(a2,"o world");

    printf("\n%s",a1);
         printf("\n%s",a2);

    printf("\nAvoid stack smash");
    printf("\n%s",b1);
         printf("\n%s",b2);

         strcat(b2,"o world");
         printf("\n%s",b1);
         printf("\n%s",b2);


    return 0;
}
```

VIVA

1. Draw the activation record of the call stack.
2. Write a tail recursive procedure for string reversal.
3. What is the output of the following program:

```
Def func(a, b=5, c=10):
        Print 'a is',a, 'and b is', b, 'and c is', c
Func(5,10)
Func(25, c=20)
Func(c=50,a=100)
```

4. What is the output of the following code:

```
def keywrd(name="roopa"):
  print("hi %s" % name)

keywrd()
```

5. Difference between default parameters in Java and keyword parameters of Python.

Week #10: Examine Inheritance

Problem Statement	Examine inheritance **@override in Java** **Final in Java** **Multiple inheritance in python and Java** **Downcasting in Java**
Learning objectives	•

@override in Java:

```
/****override1.java*********/

     class Base
     {
     public void foo(int i)
     {
     System.out.println("Inside foo(int i) i= " + i);
     }
     }

     class Derived extends Base
     {
     @Override
     public void fuoo(int f)
     {
     f = f+10;
     System.out.println("Inside foo(float f) f= " + f);
     }
     }

     class Overload
     {
     public static void main(String[] args)
     {
     float  f = 1.2f;
     int    i = 100;
     Derived obj = new Derived();

     ((Base)obj).foo(i);
     }
     }
```

```
override1.java:11: method does not override or implement a method
from a supertype
     @Override
     ^
1 error
```

Conclusion:

`@Override` annotation informs the compiler that the function is meant to override a function declared in a superclass. It helps to prevent errors. If a method with `@Override` fails to correctly override a method in one of its superclasses, the compiler generates an error.

Final in Java:

Final Variables:

```
/************final variables************/
/****finalvar.java**********/

     class FinalVariableExample {

     public static void main(String[] args) {

        final int hoursInADay=24;
         hoursInADay=12;
       System.out.println("Hours in 5 days = " + hoursInADay * 5);
      }
      }

finalvar.java:6: cannot assign a value to final variable hoursInADay
         hoursInADay=12;
         ^
1 error
```

Conclusion:

The final variable can be assigned only once. The value of a final variable is not necessarily known at compile time.

Final Class:

```
***********final class********/
/*****finalclass.java********/

  final class MyFinalClass
        {
         int a=10;
        void show()
        {

         System.out.println(a);
         }
         }
       class A extends MyFinalClass
         {
          void show()
          {
            System.out.println("HELLO");
          }
          }

       class mainclass
     {
     public static void main(String[] args)
     {
      MyFinalClass f=new MyFinalClass();
       f.show();
        }
}
```

```
finalclass.java:10: cannot inherit from final MyFinalClass
       class A extends MyFinalClass
                        ^
1 error
```

Conclusion:
Final class cannot be extended. This can be used to restrict inheritance.

Final Method:

```
/*******final method*********/
/*******overridefinal.java**/

class Base1
{
  final void base()
  {
    System.out.println("BASE CLASS");
  }
 }

 class Derived1 extends Base1
 {
    void base()
    {
       System.out.println("DERIVED CLASS");
    }
 }

 class main1
 {
   public static void main(String[] args)
   {
     Derived1 d=new Derived1();
     d.base();
   }
 }
```

```
overridefinal.java:11: base() in Derived1 cannot override base() in
Base1; overridden method is final
    void base()
         ^
1 error
```

Conclusion:
Final Method cannot be overridden. Private methods are equivalent to final methods.

Downcasting in Java / C++:

```
#include <iostream>
using namespace std;

class Shape
{
public:
     virtual void f()
     {
     }
};

class Circle : public Shape
{
};

class Square : public Shape
{
};

class Triangle : public Shape
{
};

int main()
{
    Shape* pShape = new Circle; /// Upcast

    Square* pSquare = dynamic_cast<Square*>(pShape); /// Downcast

    Circle* pCircle = dynamic_cast<Circle*>(pShape); /// Downcast

    cout << "Square address = " << (long)pSquare << endl;
    cout << "Circle address = " << (long)pCircle << endl;
}
```

The dynamic_cast is a type safe downcast operation. The dynamic_cast uses the information stored in the VTABLE to determine the actual type. There must be a virtual function when we are using a dynamic_cast. If the return value in dynamic_cast is non-zero, then the cast is proper and successful otherwise it will return zero.

- Can be done using TypeId in C++.

/***JAVA****/

```
import java.io.*;

class A
{
  void display()
 {
 System.out.println("In the Parent class");
 }
}


class B extends A
{
 void display()
 {
  System.out.println("In the First child class");
 }
}

class C extends A
{
 void display()
{
 System.out.println("In the Second child class");
}
}

class javacast
{
public static void main(String args[]) throws Exception
{
 A a1,a2=new A();
 B b1=new B(),b2;
 C c1=new C(),c2;

 a1=b1;
 a1.display();

 a1=c1;
 a1.display();

 b2=(B)a2;
 b2.display();

 c2=(C)a2;
 c2.display();

 }
}
```

Week #11 and #12: Generics in Java

Problem Statement	Examine Generics in Java
Learning objectives	Generic classes , methods, lists sets and map

Generic Methods:

We can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example:

Following example illustrates how we can print array of different type using a single Generic method:

```
public class GenericMethodTest
{
  // generic method printArray
  public static < E > void printArray( E[] inputArray )
  {
    // Display array elements
      for ( E element : inputArray ){
        System.out.printf( "%s ", element );
      }
      System.out.println();
   }

   public static void main( String args[] )
   {
     // Create arrays of Integer, Double and Character
```

```
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // pass a Character array
    }
}
```

This would produce the following result:

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4
Array characterArray contains:
H E L L O
```

Bounded Type Parameters:

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example:

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:

```
public class MaximumTest
{
  // determines the largest of three Comparable objects
  public static <T extends Comparable<T>> T maximum(T x, T y, T z)
  {
    T max = x; // assume x is initially the largest
    if ( y.compareTo( max ) > 0 ){
      max = y; // y is the largest so far
    }
    if ( z.compareTo( max ) > 0 ){
      max = z; // z is the largest now
    }
    return max; // returns the largest object
  }
  public static void main( String args[] )
  {
    System.out.printf( "Max of %d, %d and %d is %d\n\n",
              3, 4, 5, maximum( 3, 4, 5 ) );

    System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
              6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

    System.out.printf( "Max of %s, %s and %s is %s\n","pear",
      "apple", "orange", maximum( "pear", "apple", "orange" ) );
  }
}
```

This would produce the following result:

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

Generic Classes:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example:

Following example illustrates how we can define a generic class:

```
public class Box<T> {

  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }
  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    Box<String> stringBox = new Box<String>();

    integerBox.add(new Integer(10));
    stringBox.add(new String("Hello World"));

    System.out.printf("Integer Value :%d\n\n", integerBox.get());
    System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

This would produce the following result:

```
Integer Value :10

String Value :Hello World
```

VIVA QUESTIONS:

1 . What is Generics in Java

Generic in Java is added to provide compile time type-safety of code and removing risk of ClassCastException at runtime which was quite frequent error in Java code. Type-safety at compile time, is just a check by compiler that correct Type is used in correct place and there should not be any ClassCastException.

2. How Generics works in Java

- When Java compiler, when it sees code written using Generics it completely erases that code and convert it into raw type i.e. code without Generics.
- All type related information is removed during erasing.
- Also when the translated code is not type correct compiler inserts a type casting operator. Generics in Java doesn't store any type related information at runtime.
- All type related information is erased by Type Erasure, this was the main requirement while developing Generics feature in order to reuse all Java code written without Generics.

3. What is the benefit of Generics in Collections Framework?

Generics allow us to provide the type of Object that a collection can contain, so if you try to add any element of other type it throws compile time error. This avoids ClassCastException at Runtime because you will get the error at compilation.

Also Generics make code clean since we don't need to use casting and instance of operator. It also adds up to runtime benefit because the bytecode instructions that do type checking are not generated.

4. What is type erasure?

Type erasure is a JVM phenomenon that means that the runtime has no knowledge of the types of generic objects, like `List<Integer>` (the runtime sees all `List` objects as having the same type, `List<Object>`).

5. What is difference between List<? extends T> and List <? super T> ?

This is related to previous generics interview questions, some time instead of asking what is bounded and unbounded wildcards interviewer present this question to gauge your understanding of generics. Both of List declaration is example of bounded wildcards, List<? extends T> will accept any List with Type extending T while List<? super T> will accept any List with type super class of T. for Example List<? extends Number> can accept List<Integer> or List<Float> .

6. Can we use Generics with Array?

Array doesn't support Generics .So, List over Array is preferred because *List can provide compile time type-safety* over Array.

Generic class:

```
public class Entry<K, V> {

  private final K key;
  private final V value;

  public Entry(K k,V v) {
    key = k;
    value = v;
  }

  public K getKey() {
    return key;
  }

  public V getValue() {
    return value;
  }

  public String toString() {
    return "(" + key + ", " + value + ")";
  }
}
```

Generic method definitions:

Here is an example of a generic method using the generic class above:

```
public static <T> Entry<T,T> twice(T value) {
   return new Entry<T,T>(value, value);
}
```

To print all elements in a collection:

```
private static void print(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        Object o = i.next();
        System.out.println(o);
    }
}
```

The use of the `java.util.Collections.sort()` static method, which does an in-place sort of the elements in a List:

```
package util;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

public class Sort {
    public static void main(String[] args) throws IOException {
        if (args.length > 0) {
            Arrays.sort(args);
            for (int i = 0; i < args.length; i++) {
                System.out.println(args[i]);
            }
        } else {
            List lines = new ArrayList();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(
                    System.in));
            String line = null;
            while ((line = reader.readLine()) != null) {
                lines.add(line);
            }
            Collections.sort(lines);
            for (Iterator i = lines.iterator(); i.hasNext();) {
                System.out.println(i.next());
            }
        }
    }
}
```

Lists(Generics in Java):

```
public class TestList {

    interface Record {}
    interface SubRecord extends Record {}

    public static void main(String[] args) {
        List<? extends Record> l = new ArrayList<Record>();
        List<SubRecord> l2 = new ArrayList<SubRecord>();
        Record i = new Record(){};
        SubRecord j = new SubRecord(){};

        l = l2;
        Record a = l.get( 0 );
        ((List<Record>)l).add( i );       //<--will fail at run
time,see below
        ((List<SubRecord>)l).add( j );    //<--will be ok at run time

    }

}
```

Creating Hash set:

HashSet<String> hashSet = new HashSet<String>(); // using generics --
String in this case

Declaration and instantiation for map:

```
   import java.util.*;

    public class Test {

        public static void main(String[] args) {

                // insert code here

                map.put(new ArrayList<Integer>(), 1);

                map.put(new ArrayList<Integer>(), 12);

                map.put(new LinkedList<Integer>(), new Integer(1));

                map.put(new LinkedList<Integer>(), new Long(1));

        }}
```

Week #13: Examine Java Thread Model

Problem Statement	Examine Java Thread Model
Learning objectives	- Racing - Synchronization - Interthread communication - Thread local storage.

```
// check annotations
import java.io.*;
import java.lang.annotation.*;

public class annotate
{
        @SuppressWarnings("deprecation")
        public static void main(String[] args)
        {
                Testing.display(10);
                Testing test = new Testing();
                //test.print1(99);
                test.print(99);
        }
}

class Testing
{
      @Deprecated
      static void display(int a)
      {
            System.out.println(a);
      }

        public void print(int a)
        {
                System.out.println("in testing class");
                System.out.println(a);
        }
}

class OverrideTesting extends Testing
{
        @Override
        public void print(int a)
        {
        System.out.println(" in extended class");
        System.out.println(a);
        }
}
```

```
// check deprecated

import java.util.*;

class deprecate
{
        public static void main(String[] args)
        {
                Foo f = new Foo();
                f.foo();
        }

}

class Foo
{
        @Deprecated
        public static void foo()
        {
                int a = 10;
                System.out.println(a);
        }
}


// suppress warnings

import java.util.*;
import java.lang.annotation.*;

class suppress
{
        @SuppressWarnings("deprecated")
        public static void main(String[] args)
        {
                Testing.display(100);
                Testing t = new Testing();
                t.Test(200);
        }
}

class Testing
{
        @Deprecated
        static void display(int a)
        {
                System.out.println(a);
        }
        public void Test(int a)
        {
        System.out.println(a);
        }
}

class OverRide extends Testing
{
        @Override
        public void Test(int a)
        {
                System.out.println(a);
        }
}



// synchronization

// File Name : Callme.java
// This program uses a synchronized block.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

// File Name : Caller.java
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
```

```
        target.call(msg);

    }
  }
}

public class synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
   try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}


// synch example2




public class synch2
{
public static void main(String arg[])
{
Thread2 b=new Thread2();
b.start();
System.out.println(" Total is : "+b.total);
}
}

class Thread2 extends Thread
{
int total;


public void run()
{
for (int i=0;i<100;i++)
```

```
{
total += i;
}
}
}


//synch example 3

public class synch3
{
public static void main(String arg[])
{
Thread2 b= new Thread2();
b.start();

synchronized(b)
{
try {
System.out.println(" waiting for b ....");
b.wait();
}
catch(InterruptedException e) {
e.printStackTrace();
}

System.out.println(" total is "+b.total);
//}
}
}

class Thread2 extends Thread
{
int total;
public void run()
{
synchronized(this)
{
for(int i=0;i<100;i++)
{
total += i;
}
notify();
}
}
}
```