CONCURRENCY

Ques 1) What is the meaning of concurrent program?

Ans: Concurrent Program

In a concurrent program, several streams of operations may execute concurrently. Each stream of operations executes as it would in a sequential program except for the fact that streams can communicate and interfere with one another. Each such sequence of instructions is called a **thread**. For this reason, sequential programs are often called single-threaded programs.

When a multi-threaded program executes, the operations in its various threads are interleaved in an unpredictable order subject to the constraints imposed by explicit synchronization operations that may be embedded in the code. The operations for each stream are strictly ordered, but the interleaving of operations from a collection of streams is undetermined and depends on the vagaries of a particular execution of the program. One stream may run very fast while another does not run at all. In the absence of fairness guarantees, a given thread can starve unless it is the only "runnable" thread.

A program is said to be concurrent if it may have more than one active execution context—more than one "thread of control." Concurrency has at least three important motivations:

1) To capture the logical structure of a problem.
2) To exploit extra processors, for speed.
3) To cope with separate physical devices.

Concurrent programming often results in superior program structure: write code for the different tasks and let some separate engine schedule the tasks.

Ques 2) What are threads?

Ans: Threads

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

Threads are sometimes called **lightweight processes**. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process. Threads exist within a process – every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Ques 3) Write a note on communication and synchronization.

Ans: Communication and Synchronization

In any concurrent programming model, two of the most crucial issues to be addressed are communication and synchronization. Communication refers to any mechanism that allows one thread to obtain information produced by another. Communication mechanisms for imperative programs are generally based on either shared memory or message passing. In a shared-memory programming model, some or all of a program's variables are accessible to multiple threads.

For a pair of threads to communicate, one of them writes a value to a variable and the other simply reads it. In a message-passing programming model, threads have no common state. For a pair of threads to communicate, one of them must perform an explicit send operation to transmit data to another.

Synchronization refers to any mechanism that allows the programmer to control the relative order in which operations occur in different threads. Synchronization is generally implicit in message-passing models: a message must be sent before it can be received. If a thread attempts to receive a message that has not yet been sent, it will wait for the sender to catch up. Synchronization is generally not implicit in shared-memory models: unless we do something special, a "receiving" thread could read the "old" value of a variable, before it has been written by the "sender."

The most common purpose of synchronization is to make some sequence of instructions, known as a **critical section,** appear to be atomic—to happen "all at once" from the point of view of every other thread. Process synchronization refers to the coordination of simultaneous threads or processes to complete a task in order to get correct runtime order and avoid unexpected **race conditions.** The most common way to make the sequence atomic is with a **mutual exclusion lock,** which we acquire before the first instruction of the sequence and release after the last.

Ques 4) Discuss the levels of abstraction with example.

Ans: Levels of Abstraction

With the spread of thread-level parallelism, different kinds of programmers will need to understand concurrency at different levels of detail, and use it in different ways. The simplest, most abstract case will arise when using "black box" parallel libraries. A sorting routine or a linear algebra package, **for example,** may execute in parallel without its caller needing to understand how.

At a slightly less abstract level, a programmer may know that certain tasks are mutually independent (because, for example, they access disjoint sets of variables). Such tasks can safely execute in parallel. In C# 3.0, **for example,** we can write the following using the Parallel FX Library.

```
Parallel.For(0, 10, i => { A[i] = foo(A[i]); } );
```

The first two arguments to Parallel.For are "loop" bounds; the third is a delegate, here written as a lambda expression. Assuming A is a 10-element array, and that the invocations of foo are truly independent, this code will have the same effect as the obvious traditional for loop, except that it will run faster, making use of as many processors (up to 10) as possible.

If tasks are not independent, it may still be possible to run them in parallel if we explicitly synchronize their interactions. Synchronization serves to eliminate races between threads by controlling the ways in which their actions can interleave in time. Suppose function foo in the previous example subtracts 1 from A[i] and also counts the number of times that the result is zero. Consider the following implement foo as

```
int zero_count;
public static int foo(int n) {
int rtn = n - 1;
if (rtn == 0) zero_count++;
return rtn;
}
```

Consider now what may happen when two or more instances of this code run concurrently.

```
Thread 1
                                    Thread 2
...
r1 := zero count               ...
r1 := r1 + 1                        r1 := zero count
zero count := r1             r1 := r1 + 1
...                                 zero count := r1
                                    ...
```

If the instructions interleave roughly as shown, both threads may load the same value of zero count, both may increment it by 1, and both may store the (only 1 greater) value back into zero count. The result may be 1 less than what we expect.

In general, a **race condition** occurs whenever two or more threads are "racing" toward points in the code at which they touch some common object, and the behavior of the system depends on which thread gets there first. In this particular example, the store of zero count in Thread 1 is racing with the load in Thread 2. If Thread 1 gets there first, we will get the "right" result; if Thread 2 gets there first, we won't.

Ques 5) What is race condition? Why it is occur? Give its solution.

Ans: Race Condition

A situation, where several processes access and manipulate the same data concurrently, and outcome of execution depends on particular order in which the access takes place, is called **race condition. For example,** concurrent access to share data. Let suppose that two processes A and B have access to a shared variable "Balance":

```
PROCESS A:                      PROCESS B:
Balance = Balance – 100   Balance = Balance – 200
```

Further, assume that Process A and Process B are executing concurrently in a time-shared, multi-programmed system.

The statement "Balance = Balance – 100" is implemented by several machine level instructions such as:

```
A1. LOAD R1, BALANCE   //load Balance from
                          memory into Register 1 (R1)
A2. SUB R1, 100           // Subtract 100 from R1
A3. STORE BALANCE, R1 // Store R1's contents
                          back to the memory
                          location of Balance.
```

Similarly, "Balance = Balance – 200" can be implemented by the following:

```
B1. LOAD R1, BALANCE
B2. SUB R1, 200
B3. STORE BALANCE, R1
```

Observe: In a time-shared or multi-processing system the exact instruction execution order cannot be predicted.

Scenario 1:	**Scenario 2:**
A1. LOAD R1, BALANCE	A1. LOAD R1, BALANCE
A2. SUB R1, 100	A2. SUB R1, 100
A3. STORE BALANCE, R1	Context Switch!
Context Switch!	B1. LOAD R1, BALANCE
B1. LOAD R1, BALANCE	B2. SUB R1, 200
B2. SUB R1, 200	B3. STORE BALANCE, R1
B3. STORE BALANCE, R1	Context Switch!
Balance is effectively decreased by 300!	A3. STORE BALANCE, R1
	Balance is effectively decreased by 100!

Situations like this, where multiple processes are writing or reading some shared data and the final result depends on who runs precisely when, are called race conditions.

Race condition occurs due to the following reasons:

1) Because of the timing and which process starts first.

2) There is a chance that different executions of the processes may end up with different results because of interleaving, one can not predict what will happen next while the processes are executing.

To guard against the race condition it is needed to ensure that only one process at a time can be manipulated the shared thing. To make such a guarantee, synchronization of processes is required.

Solutions to Race Conditions

1) **Disable Interrupts:** Disable CPU clock interrupt when a process is in critical region.

2) **Lock Variables:** Read and set lock before entering the critical region; reset after leaving.

3) **Strict Alternation:** between two processes. It needs strict synchronization and is quite hard to expand.

Ques 6) Explain critical section problem.

Or

Discuss critical region.

Ans: Critical-Section Problem/Critical-Region

The critical section problem arises from the need to coordinate the interactions of processes. Each process has critical section.

The **critical region** (critical section) is a control structure for data access synchronization. It is used to implement mutual exclusion over a shared variable. A critical region is very convenient for mutual exclusion. However it is less versatile than a semaphore. It cannot be used for control synchronization since it lacks the ability to block a process for any reason other than mutual exclusion. The **figure 6.1** shows the critical region.

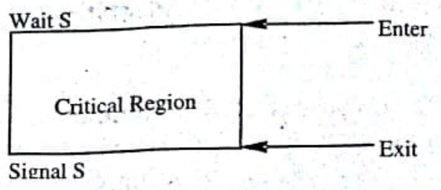

Figure 6.1: Critical Region

Critical Region is:

i) Codes that reference one or more variables in a "read-update-write" fashion while any of those variables is possibly being altered by another thread.

ii) Codes that alter one or more variables that are possibly being referenced in "read-update-write" fashion by another thread.

iii) Codes use a data structure while any part of it is possibly being altered by another thread.

iv) Codes alter any part of a data structure while it is possibly in use by another thread.

The important feature of system is that when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus the execution of critical sections by processes is mutually exclusive in time.

```
do
{
    entry section
        critical section
    exit section
        remainder section
} while (TRUE)
```

For example, consider critical region problem as shown in **figure 6.2**. The process A enters its critical region at time T_1. A little later at time T_2 process B attempts to enter its critical region but fails because another process is already in critical region. B is temporarily suspended until time T_3 when A leaves its critical region, allowing B to enter immediately.

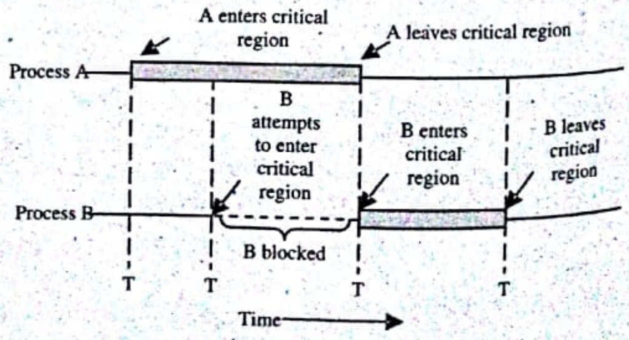

Figure 6.2: Critical Region Problem

Thus the critical section problem is to design a protocol that the process can use to co-operate.

Ques 7) Describe the solution to critical section problem.

Ans: Solution to Critical Section Problem

Each process must request permission to enter its critical section. The section of code implementing this is the **entry section**. The critical section must be followed by an **exit section**. The remaining code is the **remainder section**. These sections are shown in **figure 6.3**.

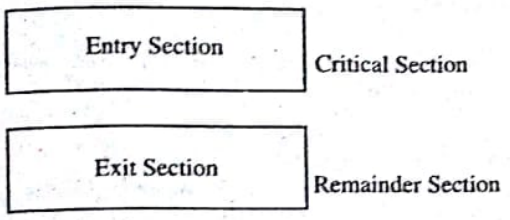

Figure 6.3: Structure of a Typical Process

Thus solution to critical section problem involves following three requirements:

1) **Mutual Exclusion:** When one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

   If process Pi is executing in its critical section, then no other processes can be executing in their critical sections. It means that no two processes may be simultaneously inside critical section.

2) **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

   By this solution no process running outside its critical section may block other process. Means for entry in the critical section only those processes will be considered who are not running in remainder section.

3) **Bounded Waiting:** In bounded waiting solution no process should have to wait forever to enter its critical section. A bound must exists on the number of times that other processes are allowed to enter their critical sections, after a process has made a request to enter its critical section and before that request is granted.
   i) Assume that each process executes at a non-zero speed,
   ii) No assumption of relative speed of the n processes.

RUN-TIME PROGRAM MANAGEMENT

Ques 8) What do you mean by run time?

Ans: Run Time

Run time is a phase of a computer program in which the program is run or executed on a computer system. Run time is part of the program life cycle, and it describes the time between when the program begins running within the memory until it is terminated or closed by the user or the operating system.

Programming languages can be classified based on the time binding takes place between program variables and their memory locations. If this happens early on, at compile time, and this binding stays in place throughout the execution of the program, then we say that these languages are **static languages**. Two **examples** of such languages are Fortran 77 and Cobol. Fortran 90 has evolved to a point where not all binding takes place at compile time. Hence it cannot be considered a static language.

At the other extreme, there are languages where all bindings take place during run time. In other words, no variable locations are decided when the program is compiled. Such languages are called **dynamic languages. Examples** in this category include LISP, ML, Perl.

Some languages (notably C) have very small run-time systems. Most of the user-level code required to execute a given source program is either generated directly by the compiler or contained in language-independent libraries. Other languages have extensive run-time systems. C#, **for example,** is heavily dependent on a run-time system defined by the Common Language Infrastructure (CLI) standard.

Like any run-time system, the **CLI** (Common Language Interface or Common Language Runtime) depends on data generated by the compiler (e.g., type descriptors, lists of exception handlers, and certain content from the symbol table). It also makes extensive assumptions about the structure of compiler-generated code (e.g., parameter-passing conventions, synchronization mechanisms, and the layout of run-time stacks). The coupling between compiler and runtime runs deeper than this, however: the CLI programming interface is so complete as to fully hide the underlying hardware.1 Such a runtime is known as a **virtual machine**. Virtual machines are part of a growing trend toward run-time management and manipulation of programs using compiler technology. To avoid the overhead of emulating a non-native instruction set, many virtual machines use a **just-in-time (JIT)** compiler to translate their instruction set to that of the underlying hardware.

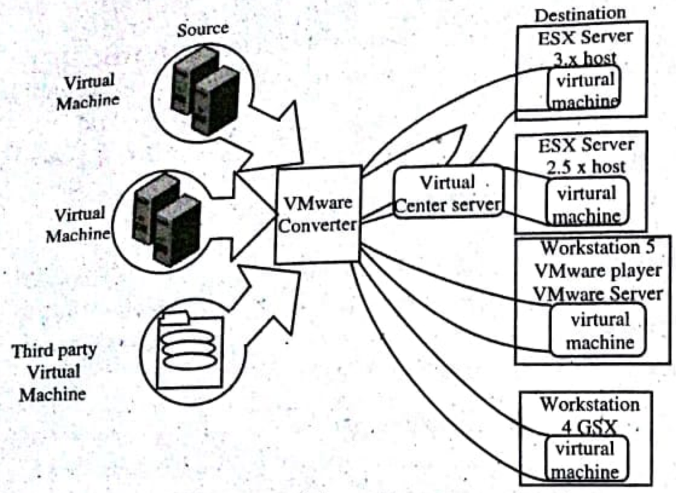

Figure 6.4

Ques 9) Write a note on virtual machines.

Ans: Virtual Machines

A virtual machine (VM) is a software implementation of a machine (i.e. a computer) that executes program like a physical machine. Virtual machines are separated into two major categories, based on their use and degree of correspondence to any real machine:

1) A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS). These usually emulate an existing architecture, and are built with either the purpose of providing a platform to run programs where the real hardware is not available for use (for example, executing software on otherwise obsolete platforms), or of having multiple instances of virtual machines lead to more efficient use of computing resources.

2) A Process virtual machine (also, language virtual machine) is designed to run a single program, which means that it supports a single process. Such virtual machines are usually closely suited to one or more programming languages and built with the purpose of providing program portability and flexibility (amongst other things). An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine–it cannot break out of its virtual environment. A virtual machine was originally defined by Popek and Goldberg as "an efficient, isolated duplicate of a real machine". Current use includes virtual machines which have no direct correspondence to any real hardware.

Ques 10) What is JVM? Explain the storage management in JVM.

Ans: Java Virtual Machine (JVM)

As we know when a **Java file** compiled the output is not an '.exe' but it's a '.class' file and that '.class' file consists of **Java byte codes** which are understandable by JVM.

Java Virtual Machine interprets the byte code into the machine code depending upon the underlying operating system and hardware combination. It is responsible for all the things like garbage collection, array bounds checking, etc. JVM is **platform dependent.**

The **JVM** is called "virtual" because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture.

This independence from hardware and operating system is a basis of the write-once run-anywhere (WORA) value of Java programs.

Storage Management in JVM

Figure 6.5 shows a block diagram of the Java virtual machine that includes the major subsystems and memory areas described in the specification.

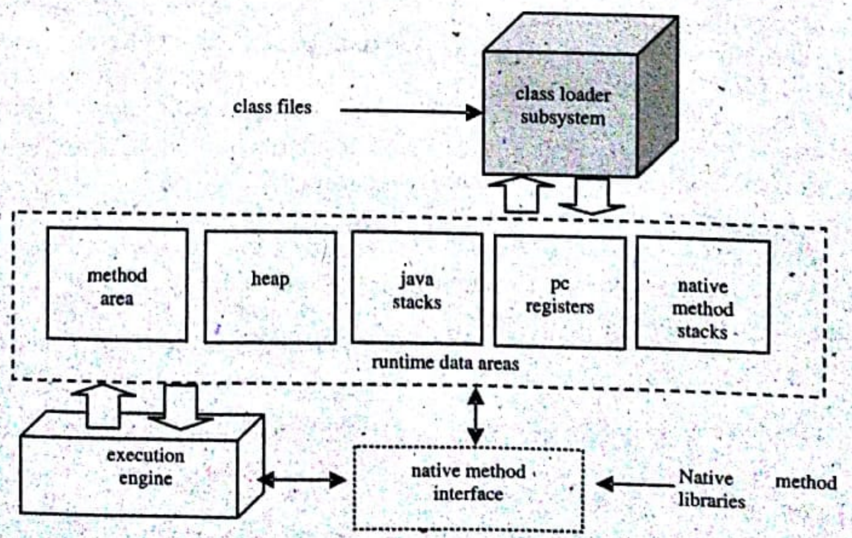

Figure 6.5: Internal Architecture of Java Virtual Machine

Ques 11) Give the elements used in java virtual machine.

Ans: Elements of Java Virtual Machine

1) **Class Loader Subsystem:** it's a mechanism for loading types (classes and interfaces) given fully qualified names.

2) **Method Area:** This area is also shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area

3) **Heap:** Heap is second part of memory and shared by all threads running inside the virtual machine. It contains objects Heap is used for creating objects.

4) **Java Stack:** The Java stack is composed of stack frames (or frames). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

5) **PC Register:** Stores the value to indicate the next instruction to execute.

6) **Native Method Stack:** It stores the state of native method invocations in an implementation-dependent way as well as possibly in registers or other implementation-dependent memory areas

7) **Execution Engine:** A mechanism responsible for executing the instructions contained in the methods of loaded classes.

Ques 12) What is a just-in-time (JIT) compiler? Explain in brief.

Ans: JIT

The key of java power "Write once, run everywhere" is the bytecode. The way bytecodes get converted to the appropriate native instructions for an application has a huge impact on the speed of an application. These bytecode can be interpreted, compiled to native code or directly executed on a processor whose Instruction Set Architecture is the bytecode specification. Interpreting the bytecode which is the standard implementation of the Java Virtual Machine (JVM) makes execution of programs slow.

To improve performance, JIT compilers interact with the JVM at run time and compile appropriate bytecode sequences into native machine code. When using a JIT compiler, the hardware can execute the native code, as opposed to having the JVM interpret the same sequence of bytecode repeatedly and incurring the penalty of a relatively lengthy translation process. This can lead to performance gains in the execution speed, unless methods are executed less frequently.

The time that a JIT compiler takes to compile the bytecode is added to the overall execution time, and could lead to a higher execution time than an interpreter for executing the bytecode if the methods that are compiled by the JIT are not invoked frequently. The JIT compiler performs certain optimizations when compiling the bytecode to native code.

Since the JIT compiler translates a series of bytecode into native instructions, it can perform some simple optimizations. Some of the common optimizations performed by JIT compilers are data-analysis, translation from stack operations to register operations, reduction of memory accesses by register allocation, elimination of common sub-expressions etc. The higher the degree of optimization done by a JIT compiler, the more time it spends in the execution stage. Therefore a JIT compiler cannot afford to do all the optimizations that is done by a static compiler, both because of the overhead added to the execution time and because it has only a restricted view of the program.

The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time. Java programs consist of classes, which contain platform neutral bytecode that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation.

The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecode into native machine code at run time.

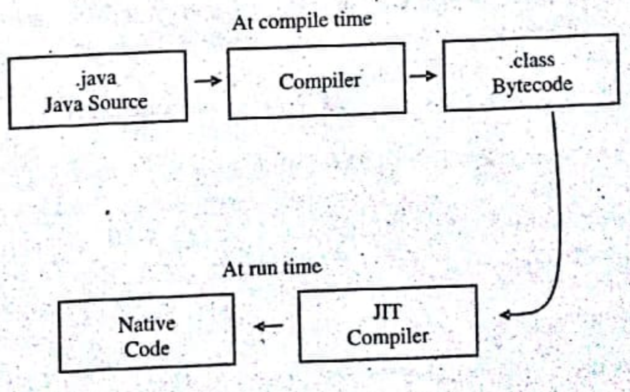

Figure 6.6

The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called.

Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

Ques 13)Why Common Language Runtime (CLR) is better than Java Virtual Machine (JVM)?

Ans: Comparison between CLR and JVM
Similarities between the CLR and JVM include:

1) Both Virtual Machines (VMs).
2) Both include garbage collection.
3) Both employ stack-based operations.
4) Both include runtime-level security.
5) Both have methods for exception handling.

Differences between the CLR and JVM include:

1) CLR was designed to be language-neutral, JVM was designed to be Java-specific.
2) CLR, was originally only Windows-compatible, JVM works with all major OSs.
3) CLR uses a JIT compiler, JVM uses a specialized JIT compiler called Java HotSpot.
4) CLR includes instructions for closures, coroutines and declaration/manipulation of pointers, the JVM does not.
5) JVM is compatible with more robust error resolution and production monitoring tools.

Ques 14)Explain late binding of machine code.

Ans: Late Binding of Machine Code
Late Binding (Dynamic building) is a computer programming mechanism in which the method being called upon an object is looked up by name at runtime. This is informally known as duck typing or name binding.

With Early binding (Static binding) the compilation phase fixes all types of variables and expressions. This is usually stored in the compiled program as an offset in a virtual method table ("v-table") and is very efficient. With late binding the compiler does not have enough information to verify the method even exists, let alone bind to its particular slot on the v-table. Instead the method is looked up by name at runtime.

Ques 15) What is inspection? How reflection is used in java for inspection?

Ans: Inspection/Introspection
Introspection is the ability of a program to examine the type or properties of an object at runtime. Some programming languages possess this capability.

Introspection should not be confused with reflection, which goes a step further and is the ability for a program to manipulate the values, meta-data, properties and/or functions of an object at runtime. Some programming languages also possess that capability.

Some languages even allow you to traverse the inheritance hierarchy to see if your object is derived from an inherited base class. Several languages have the type introspection capability, such as Ruby, Java, PHP, Python, C++, and more.

Reflection
If type introspection allows you to inspect an object's attributes at runtime, then reflection is what allows you to manipulate those attributes at runtime.

As a concrete definition, reflection is the ability of a computer program to examine and modify the structure and behavior (specifically the values, meta-data, properties and functions) of a program at runtime.

Using reflection, Java can support tools like the ones to which users of Visual Basic have grown accustomed. In particular, when new classes are added at design or runtime, rapid application development tools can dynamically inquire about the capabilities of the classes that were added.

A program that can analyse the capabilities of classes is called reflective. The reflection mechanism is extremely powerful.

You can use it to:

1) Analyse the capabilities of classes at runtime;
2) Inspect objects at runtime, e.g., to write a single toString method that works for all classes;
3) Implement generic array manipulation code; and
4) Take advantage of Method objects that work just like function pointers in languages such as C++.

While your program is running, the Java runtime system always maintains what is called runtime type identification on all objects.

This information keeps track of the class to which each object belongs. Runtime type information is used by the virtual machine to select the correct methods to execute.

However, you can also access this information by working with a special Java class. The class that holds this information is called, somewhat confusingly, Class. The getClass() method in the Object class returns an instance of Class type.

```
Employee e;
...
Class cl = e.getClass();
```

Just like an Employee object describes the properties of a particular employee, a Class object describes the properties of a particular class. Probably the most commonly used method of Class is getName. This returns the name of the class.

For example, the statement:
System.out.println(e.getClass().getName() + " " + e.getName());

prints
Employee Harry Singh
if e is an employee, or
Manager Harry Singh
if e is a manager.

If the class is in a package, the package name is part of the class name:

```
Date d = new Date();
Class cl = d.getClass();
String name = cl.getName(); //name is set to "java.util.Date"
```

You can obtain a Class object corresponding to a class name by using the static forName method:

```
String className = "java.util.Date";
Class cl = Class.forName(className);
```

You would use this method if the class name is stored in a string that varies at runtime. This works if className is the name of a class or interface. Otherwise, the forName method throws a checked exception.

Because of the runtime-specific nature of reflection, it's more difficult to implement reflection in a statically-typed language compared to a dynamically-typed language because type checking occurs at compile time in a statically-typed language instead of at runtime. However it is by no means impossible, as Java, C#, and other modern statically-typed languages allow for both type introspection and reflection (but not C++, which allows only type introspection and not reflection).

Ques 16)What are annotations? Why they are used?

Ans: Annotations

Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate.

Both Java and C# allow the programmer to extend the metadata saved by the compiler. In Java, these extensions take the form of annotations attached to declarations. Several annotations are built into the programming language. These play the role of pragmas.

Uses of Annotation

1) **Information for the Compiler:** Annotations can be used by the compiler to detect errors or suppress warnings.
2) **Compiler-time and Deployment-time Processing:** Software tools can process annotation information to generate code, XML files, and so forth.
3) **Runtime Processing:** Some annotations are available to be examined at runtime.

Annotations can be applied to a program's declarations of classes, fields, methods, and other program elements.

Ques 17)Which annotations are used in java?

Ans: Java has two special shortcuts can simplify annotations.

1) **Marker Annotation:** If no elements are specified, either because the annotation doesn't have any or because all of them use the default value, then you don't need to use parentheses.

   For example,
   @BugReport

   is the same as
   @BugReport(assignedTo="[none]", severity=0)

   Such an annotation is called a marker annotation.

2) **Single Value Annotation**: If an element has the special name value and no other element is specified, then you can omit the element name and the = symbol.

   For example,
   ```
   public @interface ActionListenerFor
   {
   String value(); }
   ```
   then we could have written the annotations as
   @ActionListenerFor("yellowButton")

   instead of
   @ActionListenerFor(value="yellowButton")

For example: A valid element can be declartion as:

```
public @interface BugReport
{
```

```
enum Status { UNCONFIRMED, CONFIRMED,
FIXED, NOTABUG };
boolean showStopper() default false;
String assignedTo() default "[none]";
Class<? extends Testable> testCase() default
Testable.class;
Status status() default Status.UNCONFIRMED;
TestCase testCase();
String[] reportedBy();
}
```

If an element value is an array, you enclose its values in braces, like below:

```
@BugReport(. . ., reportedBy={"kumar",
"abhay"})
```

You can omit the braces if the element has a single value:

```
@BugReport(. . ., reportedBy="peter") // OK, same
as {"rob"}
```

Since an annotation element can be another annotation, you can build arbitrarily complex annotations.

Ques 18) What is symbolic debugging? How does symbolic debugging work?

Ans: Symbolic Debugging

Symbolic debugging is a method by which the programmer can receive information about bugs as well as look for bugs at the same level as the program was originally written. The successful symbolic debugger maintains the illusion of the computational model created by the high level language. By keeping the programmer at the source level during debugging, the symbolic debugger makes the same impact on the debugging process as the compiler made on the coding process. Symbolic debuggers are built into most programming language interpreters, virtual machines, and integrated program development environments. They are also available as stand-alone tools, of which the best known is GNU's gdb.

Working of Symbolic Debugging

The symbolic debugger is actually only one part of a series of cooperating programs. The symbolic debugger (depending on implementation techniques) requires information from both the compiler and the linker. Remember, during normal compiling, most of the source information is lost when the object modules are produced. In order to have symbolic debugging, the information usually discarded by the compiler must be passed through into the object modules. In addition, because object modules may be relocated in memory as a result of the linking process, the linker (or loader or segmenter) must output information to a special file called a debug information file or a debug map file. The result of the compile-link operation then is a completed application program along with a debug information file.

Ques 19) Write the features of Symbolic Debugger.

Ans: Features of Symbolic Debugger

A good symbolic debugger should have as many of the following features as is possible:

1) **Source Based:** This means that debugger commands and displays should be oriented around the original source program. Commands should reference source statement numbers and source user identifiers rather than machine addresses.

2) **Breakpoints:** The basic operation of the symbolic debugger is to insert breakpoints into the program under test and then execute the program until it hits a breakpoint. When a breakpoint is hit, the program suspends execution and you can look around. For example, if you suspect that a problem is at line 100 (the program aborts from that location), you can set a breakpoint at that line and execute the program. When the program attempts to execute line 100, the breakpoint is hit and you can check the values of various data locations to make sure everything is as it should be.

3) **Single Step:** In addition to breakpoints another vital feature is the ability to single step source statements. This is usually accompanied with a representation of the current statement on the screen. This might be a marker to one side of the statement (CCS TRAX) or, more spectacularly, by changing the color of the current statement (Microsoft QuickC). Whatever the display technique, the single step operation should have these important features:
   i) Simple command to repeat single stepping.
   ii) Single step should come in two different "flavours". You should be able to step into subroutines or to step over subroutines.
   iii) Although not provided by all symbolic debuggers (multiple breakpoints accomplish the same thing) single step execution is a real time saver when trying to follow complex logic flow.

4) **Variable Display and Modification:** There are several different ways to implement this important feature. Some debuggers (HP DEBUG/1OOO, CCS CView, HP TOOLSEn have commands to display the current contents of program variables whenever you want.

   A different approach is the one used by Microsoft's Codeview and to a lesser extent HP's TOOLSET. These debuggers allow you to define watch expressions which automatically display the contents of user identifiers whenever the identifier changes during the execution of the program.

5) **Flow Control:** Most debuggers offer several commands which effect the execution flow of the program under test There are usually two different types of flow control. The first is the proceed command. Using a proceed will cause the program you are debugging to begin execution under the control of the symbolic debugger. The program will continue execution until it either terminates, is terminated by the host operating system or hits a breakpoint. There are several variations on this theme which are available.

Another important (but dangerous) flow control command is the go to command. This allow the programmer to redirect control to another part of the source program. This is often used to re-execute statements after a variable has been changed to check the new outcome. The problem with this command is that its power often lets you abuse the source language by going where the compiler never intended you to

Ques 20) Give a note on performance analysis.

Ans: Performance Analysis

Before placing a debugged program into production use, one often wants to understand—and if possible improve—its performance.

Tools to profile and analyse programs are both numerous and varied. Development in the area of performance analysis tools got its start in the early 1980s. In 1982, the first paper was written on the subject of profilers entitled "Gprof: a Call Graph Execution Profiler".

Following are the some performance analysis tools:

1) **gprof Profile Tool:** One of first profilers developed was a tool called gprof, which is a call graph execution profiler. gprof is a performance analysis tool for Unix applications. It used a hybrid of instrumentation and sampling[1] and was created as an extended version of the older "prof" tool. Unlike prof, gprof is capable of limited call graph collecting and printing.

2) **Timers:** Timers are universally available on modern systems. The accuracy and resolution of timers is system and language specific. Most languages provide simple, high level calls which provide a coarse measurement. For most PCs, the resolution is in the microseconds. There are two main categories of time measurements: wallclock or CPU time.

   i) Wallclock time is defined as the total time it takes to execute a program or process from start to finish.

   ii) CPU time represents the time a given application is being actively processed on the system. This can be broken down into system and user times. System times represent time spent during the execution of operating system calls. User time is the total time spent in executing your program.

3) **Profiling Tools:** Statistical profilers are able to report back data that summarizes the code run within an application. It may report back function call count, call traces, time spent within procedures, memory consumption, etc. These tools essentially summarize complex interactions within a program that might take a developer weeks to analyze. Profilers can help isolate down to data types or algorithms that need further development or refinement.

4) **SGI Pixie:** Another common statistical profiler is SGI's Pixie utility. It is a tool that is used to measure execution frequency for procedures within an application. The Pixie program takes in a program and generates an equivalent program which contains additional code to perform the counting. When the application is executed, it generates an output file with the relevant data. This data can be combined with other tools, such as prof, to analyze the results and produce a more readable version of the data.

5) **CLR Profiler:** The Common Language Runtime (CLR) Profiler is used to generate profile information for application written for the .NET framework.