**INDUSTRIAL TRAINING REPORT**

**AT NICE COMPUTER INSTITUTE PITAMPURA, DELHI**



Submitted in partial fulfillment of the diploma of Computer Engineering

**SUBMITTED BY:**

PRIYADARSHINI SHARMA

BOARD ROLL NO. 130610304030

5TH SEM COMPUTER ENGINEERING

**SUBMMITED TO:**

Department of Computer Engineering

Govt. Polytechnic for Women, Kandaghat,Solan(H.P.)

**ACKNOWLEDGEMENT**

I am extremely grateful and remain indebted to my guide Mr. Deepak Sharma for my four week industrial training. I am very thankful to them for their constant constructive criticism and invaluable suggestions, which benefited us a lot while developing the project on “VOTING SYATEM” and my training on “CORE JAVA”.

The Success behind anything always involves many people. The same thing is with my project & my training. But the first personality that comes to my mind is “Mr. Deepak Sharma”. He is teacher of NICE institute. He has been a constant source of inspiration and motivation for hard work. He has helped me in all the way that a guide can. My sincerer thank for his help & inspiration guidance.

I leave the second for the people who mean the most to parents, my elder brother for their constant support and encouragement and me at last.

Priyadarshini

INDEX

* **Overview of Java**
* **Key Features of Java**
* **Operators**
* **Control Statement**
* **Classes**
* **Array and Strings**
* **Inheritance**
* **Packages And Interfaces**
* **Exception Handling**
* **Multithreading**
* **Applets**

HISTORY OF JAVA

**The Birth of Modern Programming:**

**C** The C language shook the computer world. Its impact should not be underestimated, because it fundamentally changed the way programming was approached and thought about. The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs. As you probably know, when a computer language is designed, trade-offs are often made, such as the following: • Ease-of-use versus power • Safety versus efficiency • Rigidity versus extensibility

Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other. For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for system code. And while BASIC was easy to learn, it wasn’t very powerful, and its lack of structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively. Further, debugging assembly code can be quite difficult.

**C++: The Next Step**

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is complexity. Throughout the history of programming, the increasing complexity of programs has driven the need for better way stoma age that complexity. C++is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++, consider the following. Approaches to programming have changed dramatically since the invention of the computer. For example, when computers were first invented, programming was done by manually toggling in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language “C with Classes.” However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, It includes all of C’s features, attributes, and benefits. This is a crucial reason for the successof C++ as a language. The invention of C++ was not an attempt to create a completely new programming language. Instead, it was an enhancement to an already highly successful one.

**The Creation of Java**

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design an devolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype. Somewhat surprisingly, the origin limpet us for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave oven sand remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although It is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt of in such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPU sunder differing environments. This effort ultimately led to the creation of Java. About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the “Internet version of C++. ” However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++was designed to solve a different set of problems. Both will coexist for many years to come.

**Security**

As you are likely aware, every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack. Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java. Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers.

**Simple**

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

**Object-Oriented**

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non objects.

**Robust**

The multi plat formed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

**Multithreaded**

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multi process synchronization that enables you to construct smoothly running interactive systems. Java’s easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

**Architecture-Neutral**

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, anytime, forever.” To a great extent, this goal was accomplished.

**Interpreted and High Performance**

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

**Distributed**

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

**Dynamic**

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

OVERVIEW OF JAVA

**The Three OOP Principles**

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism. Let’s take a look at these concepts now.

**Encapsulation**

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission nan automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever.

**Inheritance**

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

**Polymorphism**

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. The concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action.

**A First Simple Program**

Now that the basic object-oriented underpinning of Java has been discussed, let’s look at some actual Java programs. Let’s start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

/\*

This is a simple Java program.

Call this file "Example.java".

\*/

class Example {

// Your program begins with a call to main ().

public static void main (String args[]) {

System.out.println("This is a simple Java program.");

}

}

**The Java Class Libraries**

The sample programs shown in this chapter make use of two of Java’s built-in methods: println( ) and print( ). As mentioned, these methods are members of the System class, which is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for windowed output. Thus, Java as a totality is a combination of the Java language itself, plus its standard classes. As you will see, the class libraries provide much of the functionality that comes with Java. Indeed, part of becoming a Java programmer is learning to use the standard Java classes.

OPERATORS

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations

**Arithmetic Operators**

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

+ Addition

* Subtraction (also unary minus)

\* Multiplication

/ Division

% Modulus

++ Increment

+= Addition assignment

–= Subtraction assignment

\*= Multiplication assignment

/= Division assignment

%= Modulus assignment

– – Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on Boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

The basic arithmetic operations—addition, subtraction, multiplication, and division— all behave as you would expect for all numeric types. Them in us operator also has a unary form that negates its single operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result. The following simple example program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

// Demonstrate the basic arithmetic operators.

class BasicMath {

public static void main(String args[]) {

// arithmetic using integers

System.out.println("Integer Arithmetic");

int a = 1 + 1;

int b = a \* 3;

int c = b / 4;

int d = c - a;

int e = -d;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

System.out.println("e = " + e);

// arithmetic using doubles

System.out.println("\nFloating Point Arithmetic");

double da = 1 + 1;

double db = da \* 3;

double dc = db / 4;

double dd = dc - a;

double de = -dd;

System.out.println("da = " + da);

System.out.println("db = " + db);

System.out.println("dc = " + dc);

System.out.println("dd = " + dd);

System.out.println("de = " + de);

}

}

When you run this program, you will see the following output:

Integer Arithmetic

a = 2

b = 6

c = 1

d = -1

e = 1

Floating Point Arithmetic

da = 2.0

db = 6.0

**Assignment Operators**

Java provides special operators that can be used to combine an arithmetic operation with an assignment. The compound assignment operators provide two benefits. First, they save you a bit of typing, because they are “shorthand” for their equivalent long forms. Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms.

// Demonstrate several assignment operators.

class OpEquals {

public static void main(String args[]) {

int a = 1;

int b = 2;

int c = 3;

a += 5;

b \*= 4;

c += a \* b;

c %= 6;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

}

}

The output of this program is shown here:

a = 6

b = 8

c = 3

**Relational Operators**

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

== Equal to

! = Not equal to

> Greater than

< Less than

>= Greater than or equal to

<= Less than or equal to

The outcome of these operations is a Boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.

**Logical Operators**

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.

& Logical AND

| Logical OR

^ Logical XOR (exclusive OR)

|| Short-circuit OR

&& Short-circuit AND

! Logical unary NOT

&= AND assignment

|= OR assignment

^= XOR assignment

== Equal to

!= Not equal to

?: Ternary if-then-else

CONTROL STATEMENT

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java’s program control statements can be put into the following categories: selection, iteration, and jump. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion.

**Java’s Selection Statements**

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program’s execution based upon conditions known only during runtime. You will be pleasantly surprised by the power and flexibility contained in the set statements.

**if**

The if statement is Java’s conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

if (condition) statement1; else statement2;

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional. The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

int a, b;

// ... if

(a < b) a = 0;

else b = 0;

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case are they both set to zero.

**Nested ifs**

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

if(i == 10) {

if(j < 20) a = b;

if(k > 100) c = d; // this if is

else a = c; // associated with this else

}

else a = d; // this else refers to if(i == 10)

As the comments indicate, the final else is not associated with if(j<20) because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else refers to if(k>100) because it is the closest if within the same block.

**The if-else-if Ladder**

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

if(condition) statement; else if(condition) statement; else if(condition) statement; . . . else statement;

The If statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.

**switch**

The switch statement is Java’s multi way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements.

Here is the general form of a switch statement:

switch (expression) {

case value1:

// statement sequence

break;

case value2:

// statement sequence

break;

. . . case valueN:

// statement sequence break;

default:

// default statement sequence

}

The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression.

**Nested switch Statements**

You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch. For example, the following fragment is perfectly valid:

switch(count) {

case 1:

switch(target) { // nested switch

case 0:

System.out.println("target is zero");

break;

case 1: // no conflicts with outer switch

System.out.println("target is one");

break;

} break;

case 2: // ...

Here, the case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch. The count variable is only compared with the list of cases at the outer level. If count is 1, then target is compared with the inner list cases.

In summary, there are three important features of the switch statement to note:

• The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.

• No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.

• A switch statement is usually more efficient than a set of nested ifs.

**Iteration Statements**

Java’s iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

**while**

The while loop is Java’s most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

while(condition) {

// body of loop

}

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.

**do-while**

Sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

do {

// body of loop

} while (condition);

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java’s loops, condition must be a Boolean expression. The do-while loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

**For**

It is a powerful and versatile construct. Beginning with JDK5, there are two forms of the for loop. The first is the traditional form that has been in use since the original version of Java. The second is the new “for-each” form. Both types off or loops are discussed here, beginning with the traditional form.

Here is the general form of the traditional for statement:

for(initialization; condition; iteration) {

// body

}

If only one statement is being repeated, there is no need for the curly braces. The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

**Using the Comma**

There will be times when you will want to include more than one statement in the initialization and iteration portions of the for loop. For example, consider the loop in the following program:

class Sample {

public static void main(String args[]) {

int a, b;

b = 4;

for(a=1; a<b; a++)

{

System.out.println("a = " + a);

System.out.println("b = " + b); b--;

}

}

}

As you can see, the loop is controlled by the interaction of two variables. Since the loop is governed by two variables, it would be useful if both could be included in the for statement, it self, instead of **b** being handled manually. Fortunately, Java provides away to accomplish this. To allow two or more variables to control a for loop, Java permits you to include multiple statement sin both the initialization and iteration portions of the for. Each statement is separated from the next by a comma.

**Nested Loops** Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests for loops:

// Loops may be nested.

class Nested {

public static void main(String args[]) {

int i, j;

for(i=0; i<10; i++)

{

for(j=i; j<10; j++)

System.out.print(".");

System.out.println();

}

}

}

**Jump Statements**

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of your program. Each is examined here.

**NOTE**

In addition to the jump statements discussed here, Java supports one other way that you can change your program’s flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords try, catch, throw, throws, and finally. In essence, the exception handling mechanism allows your program to perform a nonlocal branch.

**Using break**

In Java, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a “civilized” form of goto.

**Using continue**

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’ send. The continue statement performs such an action. In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

**Using return**

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, return causes execution to return to the Java run-time system, since it is the run-time system that calls main( ).

classes

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class. Because the class is so fundamental to Java. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the this keyword.

A class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

The General Form of a Class When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. A simplified general form of a class definition is shown here:

class classname {

type instance-variable1;

type instance-variable2;

// ...

type instance-variableN;

type methodname1(parameter-list) {

// body of method

}

type methodname2(parameter-list) {

// body of method

}

// ...

type methodnameN(parameter-list) {

// body of method } }

classes

**Declaring Objects**

As just explained, when you create a class, you are creating a new datatype. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let’s look at the details of this procedure. In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

Box mybox = new Box();

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

Box mybox; // declare reference to object

mybox = new Box(); // allocate a Box object

The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object. Any attempt to use mybox at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to mybox. After the second line executes, you can use my box as if it were a Box object. But in reality, my box simply holds the memory address of the actual Box object.

**Constructors**

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class’ constructor is the class type itself. It is the constructor’s job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

The this Keyword Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. You can use this anywhere a reference to an object of the current class’ type is permitted. To better understand what this refers to, consider the following version of Box( ) :

// A redundant use of this.

Box(double w, double h, double d) {

this.width = w;

this.height = h;

this.depth = d;

}

**Garbage Collection**

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

**The finalize( ) Method**

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize() method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize() method on the object. The finalize() method has this general form:

protected void finalize( )

{

// finalization code here

}

**A Stack Class**

The class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new datatype that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class’ data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed with in the class. In a sense, a class is like a “data engine.” No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.

One last point about the Stack class. As it is currently implemented, it is possible for the array that holds the stack, stck, to be altered by code outside of the Stack class.

ARRAY

AND

STRINGS

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers [0], numbers [1], and ..., numbers [99] to represent individual variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

dataType[] arrayRefVar; // preferred way.

or

dataType arrayRefVar[]; // works but not preferred way

**Creating Arrays**

You can create an array by using the new operator with the following syntax −

Syntax

arrayRefVar = new dataType[arraySize];

The above statement does two things −

* It creates an array using new dataType[arraySize].
* It assigns the reference of the newly created array to the variable arrayRefVar.

The Arrays Class

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

|  |  |
| --- | --- |
| 1 | **public static int binarySearch(Object[] a, Object key)**  Searches the specified array of Object ( Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, it returns ( – (insertion point + 1)). |
| 2 | **public static boolean equals(long[] a, long[] a2)**  Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.) |
| 3 | **public static void fill(int[] a, int val)**  Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.) |
| 4 | **public static void sort(Object[] a)**  Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types ( Byte, short, Int, etc.) |

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write −

String greeting = "Hello world!";

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

**Note** − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use [String Buffer & String Builder](https://www.tutorialspoint.com/java/java_string_buffer.htm) Classes.

String Length

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

Concatenating Strings

The String class includes a method for concatenating two strings −

string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in −

"My name is ".concat("Zara");

Strings are more commonly concatenated with the + operator, as in −

"Hello," + " world" + "!"

inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a super class. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the super class and adds its own, unique elements.

Inheritance Basics To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword. To see how, let’s begin with a short example. The following program creates a superclass called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.

The general form of a class declaration that inherits a superclass is shown here:

class subclass-name extends superclass-name {

// body of class

}

**super**

In the preceding examples, classes derived from Box were not implemented as efficiently or as robustly as they could have been. For example, the constructor for BoxWeight explicitly initializes the width, height, and depth fields of Box( ). Not only does this duplicate code found in its super class, which is in efficient, but it implies that a sub class must be granted access to these members. However, there will be times when you will want to create a super class that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super. super has two general forms. The first calls the superclass’ constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

A subclass can call a constructor defined by its superclass by use of the following form of super:

super(arg-list);

The second form of super acts somewhat like this, except that it always refers to the super class of the sub class in which it is used. This usage has the following general form:

super.member

**Method Overriding**

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the sub class. The version of the method defined by the superclass will be hidden. Consider the following:

// Method overriding.

class A {

int i, j;

A(int a, int b) {

i = a;

j = b;

}

// display i and j

void show() {

System.out.println("i and j: " + i + " " + j);

}

}

class B extends A {

int k;

B(int a, int b, int c) {

super(a, b); k = c; }

// display k – this overrides show() in A

void show() {

System.out.println("k: " + k);

}

}

class Override {

public static void main(String args[]) {

B subOb = new B(1, 2, 3);

subOb.show(); // this calls show() in B

}

}

The output produced by this program is shown here:

k: 3

**Abstract Classes**

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a super class is unable to create a meaningful implementation for a method. This is the case with the class Figure used in the preceding example. The definition of area() is simply a placeholder. It will not compute and display the area of any type of object. As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its super class. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the sub class to have any meaning. Consider the class Triangle. It has no meaning if area() is not defined. In this case, you want some way to ensure that a sub class does, indeed, override all necessary methods. Java’s solution to this problem is the abstract method. You can require that certain methods be overridden by subclasses by specifying the abstract type modifier. These methods are sometimes referred to as subclasses responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

To declare an abstract method, use this general form:

abstract type name(parameter-list);

**Using final with Inheritance**

The keyword final has three uses. First, it can be used to create the equivalent of a named constant. The other two uses of final apply to inheritance. Both are examined here. Using final to Prevent Overriding While method overriding is one of Java’s most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

class A { final void meth() {

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

}

}

class B extends A {

void meth() { // ERROR! Can't override.

System.out.println("Illegal!");

}

}

Because meth() is declared as final, it cannot be overridden in B. If you attempt to do so, a compile-time error will result.

Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode forthesubroutinedirectlyinlinewiththecompiledcodeofthecallingmethod, thus eliminating the costly overhead associated with a method call. In lining is only an option with final methods. Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final method scan not be overridden, a call to one can be resolved at compile time. This is called early binding.

**The Object Class**

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is as upper class of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.

The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: equals( ) and toString( ). The equals( ) method compares the contents of two objects. It returns true if the objects are equivalent, and false otherwise.

PACAGES AND INTERFACES

**T**his chapter examines two of Java’s most innovative features: packages and interfaces. Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions. Methods define the interface to the data in a class. Through the use of the interface keyword, Java allows you to fully abstract the interface from its implementation. Using interface, you can specify a set of methods that can be implemented by one or more classes. The interface, itself, does not actually define any implementation. Although they are similar to abstract classes, interfaces have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

**Packages**

This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name “Foobar” as a class name. Or, imagine the entire Internet community arguing over who first named a class “Espresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

**Defining a Package**

To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared with in that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. (This is why you haven’t had to worry about packages before now). While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. This is the general form of the package statement:

package pkg;

Here, pkg is the name of the package. For example, the following statement creates a package called MyPackage.

package MyPackage;

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

package pkg1[.pkg2[.pkg3]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

**Interfaces**

Using the keyword interface, you can fully abstract a class’ interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces that don’t make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism. Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.

**NOTE** Interfaces add most of the functionality that is required for many applications that would normally resort to using multiple inheritance in a language such as C++.

**Defining an Interface**

An interface is defined much like a class. This is the general form of an interface:

access interface name {

return-type method-name1(parameter-list);

return-type method-name2(parameter-list);

type final-varname1 = value;

type final-varname2 = value;

// ...

return-type method-nameN(parameter-list);

type final-varnameN = value;

}

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. Name is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods. Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly public.

**Implementing Interfaces**

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

class classname [extends superclass] [implements interface [,interface...]] {

// class-body

}

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition. Here is a small example class that implements the Callback interface shown earlier.

class Client implements Callback {

// Implement Callback's interface

public void callback(int p) {

System.out.println("callback called with " + p);

}

}

Notice that callback( ) is declared using the public access specifier.

**REMEMBER:** When you implement an interface method, it must be declared as public.

**Interfaces Can Be Extended**

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

// One interface can extend another.

interface A {

void meth1();

void meth2(); }

// B now includes meth1() and meth2() -- it adds meth3().

interface B extends A {

void meth3();

}

// This class must implement all of A and B

class MyClass implements B {

public void meth1() {

System.out.println("Implement meth1().");

}

public void meth2() {

System.out.println("Implement meth2().");

}

public void meth3() {

System.out.println("Implement meth3().");

}

}

class IFExtend {

public static void main(String arg[]) {

MyClass ob = new MyClass();

ob.meth1();

ob.meth2();

ob.meth3();

}

}

As an experiment, you might want to try removing the implementation for meth1( ) in MyClass. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

EXCEPTION HANDLING

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java’s exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.

**Exception-Handling Fundamentals**

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method. Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block. This is the general form of an exception-handling block:

try {

// block of code to monitor for errors

}

catch (ExceptionType1 exOb) {

// exception handler for ExceptionType1

}

catch (ExceptionType2 exOb) {

// exception handler for ExceptionType2

}

// ...

finally {

// block of code to be executed after try block ends

}

**Exception Types**

All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called Runtime Exception. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing. The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type Error, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

**Using try and catch**

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this. To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

**Displaying a Description of an Exception**

Throwable overrides the toString( ) method (defined by Object) so that it returns a string containing a description of the exception. You can display this description in a println( ) statement by simply passing the exception as an argument. For example, the catch block in the preceding program can be rewritten like this:

catch (ArithmeticException e) {

System.out.println("Exception: " + e); a = 0;

// set a to zero and continue

}

When this version is substituted in the program, and the program is run, each divide-by zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

**Multiple catch Clauses**

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

// Demonstrate multiple catch statements.

class MultiCatch {

public static void main(String args[]) {

try {

int a = args.length;

System.out.println("a = " + a);

int b = 42 / a;

int c[] = { 1 };

c[42] = 99;

} catch(ArithmeticException e) {

System.out.println("Divide by 0: " + e);

} catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Array index oob: " + e);

}

System.out.println("After try/catch blocks.");

}

}

This program will cause a division-by-zero exception if it is started with no command-line arguments, since a will equal zero. It will survive the division if you provide a command-line argument, setting a to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].

**Nested try Statements**

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. I fan inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement’s catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

**throw**

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, Throwable Instance must bean object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

**throws**

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method’s declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or Runtime Exception, or any of their sub classes. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:

type method-name(parameter-list) throws exception-list

{

// body of method

}

**finally**

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.

finally creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

**Using Exceptions**

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program’s logic. Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes. One last point: Java’s exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

MULTI-

THREADING

Unlike many other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process based and thread-based. It is important to understand the difference between the two. For most readers, process-based multitasking is the more familiar form. A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compile rat the same time that you are using a text editor. In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler. In a thread-based multitasking environment, the thread is the smallest unit of dispatch able code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details. Multitasking threads require less overhead than multitasking processes. Processes are heavy weight tasks that require their own separate address spaces. Inter process communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavy weight process. Inter thread communication is in expensive, and context switching from one thread to the next is low cost. While Java programs make use of process based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time. Multithreading lets you gain access to this idle time and put it to good use.

**The Java Thread Model**

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles. The value of a multithreaded environment is best under stood in contrast to its counterpart. Single-threaded systems use an approach called an event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a singled-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run. Threads exist in several states. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

• A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

• A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

**Synchronization** Because multithreading introduces an asynchronous behavior to your programs, there must be away for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don’t conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. Forth is purpose, Java implements an elegant twist on an age-old model of inter process synchronization: the monitor.

**Thread**

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons: • It is the thread from which other “child” threads will be spawned. • Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:

static Thread currentThread( )

Thismethodreturnsareferencetothethreadinwhichitiscalled.Onceyouhaveareference to the main thread, you can control it just like any other thread.

**Creating a Thread**

In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

• You can implement the Runnable interface.

• You can extend the Thread class, itself.

**Implementing Runnable**

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

public void run()

Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.

**Extending Thread**

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

**Deadlock**

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

•In general, it occurs only rarely, when the two threads time-slice in just the right way.

• It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, A and B, with methods foo( ) and bar( ), respectively, which pause briefly before trying to call a method in the other class. The main class, named Deadlock, creates an A and a B instance, and then starts a second thread to set up the deadlock condition. The foo( ) and bar( ) methods use sleep( ) as a way to force the deadlock condition to occur.

**Using Multithreading**

The key to utilizing Java’s multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems with in a program that can execute concurrently, make them individual threads. With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program.

**Bibliography:-**

**I have taken references for the Report Development were taken from the**

**Following Books and Web Sites**.

**JAVA Technologies**

* JAVA Complete Reference, JAVA2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ By Herbert Schildt
* Advanced Programming for the By Calvin Austin and

JavaTM 2 Platform \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Monica Pawlan

* JAVA Book by Pragya Publications \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ By Dr. Anurag Seetha Surbhi Rastogi

**REFERENCE SITES:**

* [**www.wikipedia.com**](http://www.wikipedia.com)
* [**www.javapoint.com**](http://www.javapoint.com)
* [**www.tutorialspoint.com**](http://www.tutorialspoint.com)

Weekly

schedule

|  |  |
| --- | --- |
| First week (1-2) | Overview of java |
| Second week(4,9) | Key features of java , Operators, Control statements, |
| Third week(10-16) | Classes, Array and Strings , Inheritance |
| Fourth week(18-23) | **Packages and Interfaces,** **Exception Handling** |
| Fifth week (25-30) | **Multithreading,** **Applets** |