Teacher's Sign / Remarks

Module - 02

Operators, Control statements ch 4, ch 5

Operators

Java provides a rich operator environment. divided into the following four groups:
- Arithmetic
- bitwise
- relational and
- logical

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

operator	Result
+	Addition
-	Substraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	division assignment
%=	Modulus assignment
--	Decrement

The operands of the arithemetic operator must be of 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 sub set of int.

05/06/2024 10:48

* The Basic Arithmetic Operators

→ The basic arithmetic operations - addition, subtraction, multiplication and division - all behaves as you would expect for all numeric types

→ The minus operator also has a unary form that negates its single operand.

→ When the division operator is applied to an integer type, there will be no fractional component attached to the result.

→ Example illustrates the difference between floating-point division and integer division

Ex: // 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);
 }
}
```

O/P: Integer Arithmetic

a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic

da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5

* The modulus operator

The modulus operator % returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

Ex:
```
// Demonstrate the % operator
class Modulus {
  public static void main(String args[])
  {
    int x = 42;
    double y = 42.25;
```

```
        System.out.println ("x mod 10 = " + x%10);
        System.out.println ("y mod 10 = " + y%10);
    }
}
```

O/p : x mod 10 = 2
y mod 10 = 2.25

*Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

```
a = a + 4;
```

rewrite as `a += 4;`

This is version of += compound assignment operator

both statements perform the same action

Another example

```
a = a % 2;
```

which can be expressed as

```
a %= 2;
```

There are compound assignment operators for all of the arithmetic, binary operators

Thus any statement of the form

```
var = var op expression;
```

can be written as

```
var op = expression;
```

Benefits of compound assignment :

1) they save you bit of typing, shorthand
2) they are implemented more efficiently by the Java-run-time system than are their equivalent long forms.

ex6// 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);
}
}
```

o/p : a = 6
b = 8
c = 3

*Increment and Decrement

The Increment operator increases its operand by one.

The decrement operator decreases its operand by one.

```
x = x + 1;
```

can be rewritten like this by use of the Increment operator

```
x++;
```

similarly

```
x = x - 1;
```

is equivalent to

```
x--;
```

These operators are unique in that they can appear both in postfix form, where they follow the operand as just shown and prefix form, where they precede the operand.

→ In the prefix form, the operand is incre-mented or decremented before the value is obtained for use in the expression.

→ In postfix form, the previous value is obtained for use in the expression and then the operand is modified.

```
x = 42;
y = ++x;          x = 43   y = 43.

y = ++x; is the equivalent of these two state-ments
  x = x+1;
  y = x;
```
Prefix

```
x = 42;
y = x++;          x = 43   y = 42

y = x++; is the equivalent of these two state-ments.
  y = x;
  x = x+1;
```
postfix

```
// Demonstrate ++
class IncDec {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c;
        int d;
        c = ++b;      // b = 3   c = 3
        d = a++;      // a = 2   d = 1
        c++;          // c = 4
        System.out.println("a = " + a);   // 2
        System.out.println("b = " + b);   // 3
        System.out.println("c = " + c);   // 4
        System.out.println("d = " + d);   // 1
    }
}
```

The Bitwise Operators

Java defines several bitwise operators that can be applied to the integer types long, int, short char, and byte.

These operators act upon the individual bits of their operands.

Operator → Result

~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	shift right
>>>	shift right zero fill
<<	left shift
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	shift right assignment
>>>=	shift right zero fill
<<=	shift left assignment

All of the integer types are represented by binary numbers of varying bit widths.

The byte value for 42 in binary is 00101010 where each position represents a power of two, starting with 2^0 at the rightmost bit. The next bit position to the left would be 2^1 or 2, 1, 3 and 5

so 42

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	0

All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones.

The Bitwise Logical operators

→ The bitwise logical operators are &, |, ^ and ~

→ bitwise operators are applied to each individual bit within each operand.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand.

Ex: 42

00101010 becomes 11010101

The Bitwise AND

The AND operator & produces a 1 bit if both operands are also 1. A zero is produced in all other cases.

```
  00101010   42
& 00001111   15
----------
  00001010   10
```

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1

```
00101010   42
00001111   15
--------
00101111   47
```

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. otherwise the result is zero.

```
00101010   42
^ 00001111   15
--------
00100101   37
```

Using the Bitwise Logical operators.

```
// Demonstrate the bitwise logical operators
class BitLogic {
    public static void main(String args[]) {
        String binary[] = { "0000", "0001", "0010",
"0011", "0100", "0101", "0110", "0111", "1000", "1001",
"1010", "1011", "1100", "1101", "1110", "1111" };
        int a = 3;
        int b = 6;
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;
        System.out.println("a = " + binary[a]);
        System.out.println("b = " + binary[b]);
        System.out.println("a|b = " + binary[c]);
        System.out.println("a&b = " + binary[d]);
        System.out.println("a^b = " + binary[e]);
        System.out.println("~a&b | a&~b = " + binary[f]);
        System.out.println("~a = " + binary[g]);
    }
}
```

05/06/2024

O/P : a = 0011

b = 0110

a | b = 0111

a & b = 0010

a ^ b = 0101

~a & b | a & ~b = 0101

~a = 1100

Relational Operators.

The relational operators determine the relationship that one operand has to the other. Specifically they determine equality and ordering

operator	Result
==	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

```
ex: Int a = 4;
    Int b = 1;
    boolean c = a<b;   // c = false
```

Boolean 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.

Operator → Result

& → Logical AND
| → Logical OR
^ → Logical XOR
|| → short-circuit OR
&& → short-circuit AND
! → Logical unary NOT
&= → AND assignment
|= → OR assignment
^= → XOR assignment
== → Equal to
!= → Not equal to
?: → Ternary if-then-else

The logical Boolean operators &, | and ^ operate on boolean values in the same way that they operate on the bits of an integer.

The logical ! operator inverts the Boolean state:
!true == false and !false == true

A	B	A | B	A & B	A ^ B	! A
False	False	false	false	false	True
True	False	True	false	True	false
false	True	True	false	True	True
True	True	True	True	false	false

Ex: // Demonstrate the boolean logical operators

```
class BoolLogic {
  public static void main(String args[]){
    boolean a = true;
    boolean b = false;
    boolean c = a|b;
    boolean d = a&b;
    boolean e = a^b;
    boolean f = (!a&b)|(a&!b);
    boolean g = !a;
    System.out.println(" a = " + a);
    System.out.println(" b = " + b);
    System.out.println("a|b = " + c);
    System.out.println("a&b = " + d);
    System.out.println("a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println(" !a = " + g);
```

o/p : a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false.

The Assignment Operator

The assignment operator is the single equal sign

=

```
var = expression;
```

The assignment operator allows to create a chain of assignments

```
int x, y, z;
x = y = z = 100;
```

Module - 04 (02)

Exception Handling

→ An exception is an abnormal condition that arises in a code sequence at run time.

→ 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.

→ 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 occured in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in a 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.

→ 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

→ 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.

→ 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
}
```

→ ExceptionType is the type of exception that has occured.

Exception Types

→ All exception types are subclasses of the built-in class Throwable.

→ 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.

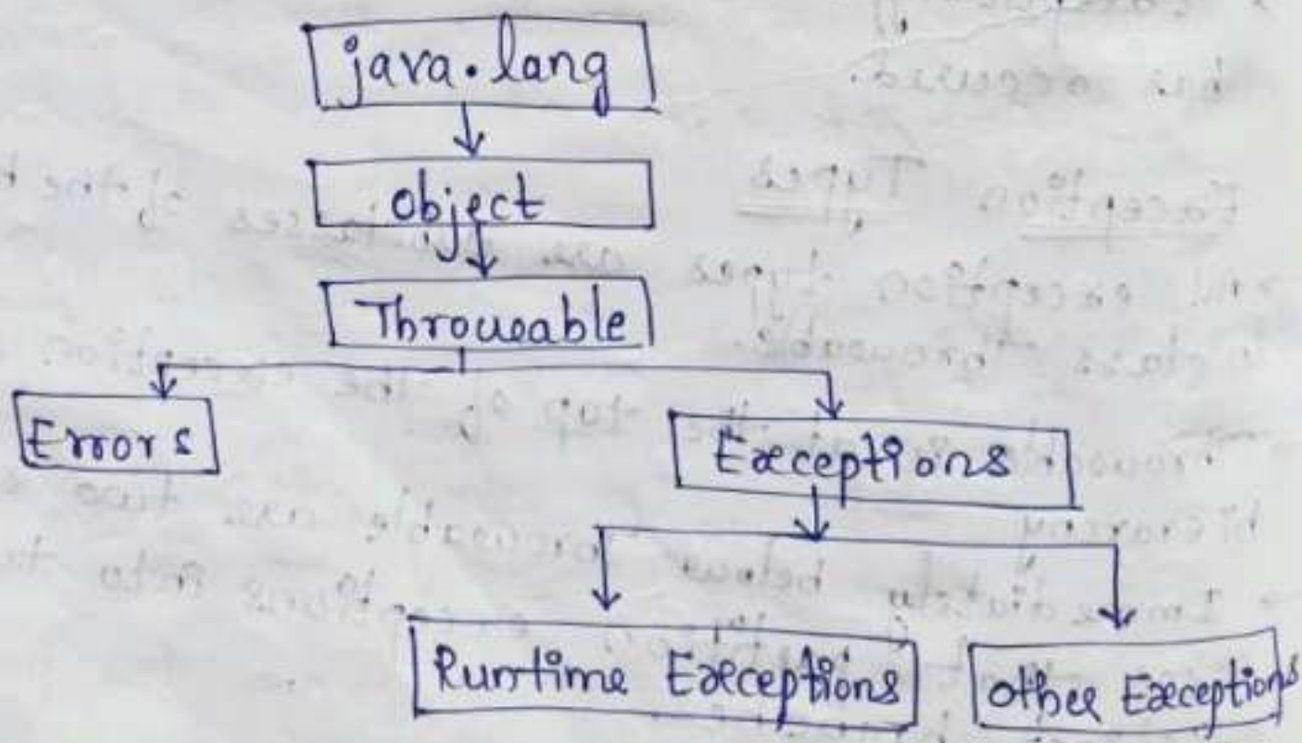
→ This is also the class that you will subclass to create your own custom exception types.

→ There is an important subclass of Exception called RuntimeException

→ 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.

→ StackOverflow is an example of such an error



Uncaught Exceptions

If we have program that expression intentionally have divide-by-zero error. If we not handle what happens will see.

```
class Exc0 {
    public static void main(String args[]){
        int d = 0;
        int a = 42/d;
    }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.

This causes the execution of Exc0 to stop because once an exception has been thrown it must be caught by an exception handler and dealt with immediately.

In this example we have not supplied any exception handlers of our own; so the exception is caught by the default handler provided by the Java run-time system.

Any exception that is not caught by your program will ultimately processed by the default handler.

The default handler provided which displays a string describing the exception, prints a stack trace from the point at which the exception occured and terminates the program.

exception generated by above example:

```
java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:4)
```

→ one more example program that introduces the same error but in a method separate from main():

```
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    public static void main(String args[]) {
        Exc1.subroutine();
    }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed.

```
java.lang.ArithmeticException: / by zero
        at Exc1.subroutine(Exc1.java:4)
        at Exc1.main(Exc1.java:7)
```

Using try and catch

The default exception handler helps for debugging which is provided by Java run-time systems. It providing 2 benefits

1) It allows you to fix the error
2) It prevents the program from automatically terminating.

→ To guard against and handle a run-time error, simply enclose 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.

→ Below example for try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error.

```
class Exc2 {
    public static void main(String args[]){
        int d, a;
        try { // monitor a block of code
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }
        System.out.println("After catch statement.");
    }
}
```

o/p: Division by zero.
After catch statement.

→ println() inside the try block, is never executed once an exception is thrown program control transfers out of the try block into the catch block.

→ Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

→ A try and its catch statement form a unit.

→ The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.

→ The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

→ for example, in the next program each iteration of the for loop obtains two random integers. Those two integers are divided by each other and the result is used to divide the value 12345.

→ The final result is put into a. If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.

```
import java.util.Random;
class HandleError {
    public static void main(String args[]){
        int a=0, b=0, c=0;
        Random r= new Random();
```

```
for (int i=0; i<32000; i++){
    try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
    } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a=0; // set a to zero and continue
    }
    System.out.println("a: "+a);
}
}
```

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.

```
class Multicatch {
    public static void main(String args[]){
        try {
            int a = args.length;
            System.out.println("a = "+a);
```

17/07/2024 13:35

```
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.");
}
}
```

→ the above 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].

```
c:\> java MultiCatch
o/p: a = 0
     Divide by 0: java.lang.ArithmeticExcepti
          -on: / by zero
     After try/catch blocks.

c:\> java MultiCath TestArg
     a=1
Array index oob: java.lang.ArrayOutOfBou
     -ndsException: 42
After try/catch blocks.
```

Ex 6 /* This program contains an error
A subclass must come before its superclass
in a series of catch statements If not, unreach
-able code will be created and a compile-time
error will result.
*/
class SuperSubCatch {
public static void main (String args[]){
try {
int a=0;
int b= 42/a;
} catch (Exception e) {
System.out.println ("Generic Exception catch.");
}
/* this is catch is never reached because
ArithmeticException is a subclass of Exception.
*/
catch (ArithmeticException e) { // Error -unre
-achable
System.out.println ("This is never reached.");
}
}

→ If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable because the exception has already been caught since ArithmeticException is a subclass of Exception.

The first catch statement will handle all exception based errors including ArithmeticExcepti on. This means that the second catch statement will never execute.

To fix the problem, reverse the order of the catch statements.

* Nested try statements.

→ The try statement can be nested. This 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.

→ If an 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 ~~for a part~~ are inspected for a match.

→ If no catch statement matches, then the Java run-time system will handle the exception

Ex:-

```
// An example of nested try statements
class NestTry {
    public static void main(String args[]){
        try {
            int a = args.length;
/* If no command-line args are present
   the following statement will generate
   a divide-by-zero exception. */
            int b=42/a;
            System.out.println("a= "+a);
            try { // nested try block
/* If one command-line arg is used
   then a divide-by-zero exception
   will be generated by the following
   code. */
```

```
if (a==1) a= a1/(a-a); // division by zero
/* If two command-line args are used.
   then generate an out-of-bounds exception */
   if (a==2){
      int c[] = {1};
      c[42] = 99; // generate an out-of-bounds exception
   }
} catch (ArrayIndexOutOfBoundsException e){
   System.out.println("Array index out-of-bounds: "+e);
} catch (ArithmeticException e) {
   System.out.println("Divide by 0: "+e);
}
}
```

o/p: C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry one
a=1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry one Two
a=2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException: 42

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;

→ ThrowableInstance must be an object of type Throwable or 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 transfered 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.

→ Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler

ex:- class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch (NullPointerException e) {
System.out.println("Caught inside demo
-proc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch (NullPointerException e) {
System.out.println("Recaught: " + e);
}
}

→ This program gets two chances to deal with the same error. First main() sets up an exception context and then calls demoproc().
→ The demoproc() method then sets up another exception-handling context and immediately throws a new instance of NullPointerException.
→ Here is the resulting output:

Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

→ The program also illustrates how to create one of Java's standard exception objects.

throw new NullPointerException("demo");

Here, new is used to construct an instance of NullPointerException.

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 RuntimeException, or any of their subclasses.

→ 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.

→ General form of method declaration that inclu-des a throws clause

```
type method-name (parameter-list) throws
        exception-list
{
    // body of method
}
```

exception-list is a comma-separated list of the exceptions that a method can throw.

→ An example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

ex:- This program contains an error and will not compile.

```
class ThrowDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

→ To make this example compile, you need to make two changes. first, you need to declare that throwOne() throws IllegalAccessException.

→ Second, main() must define a try/catch statement that catches this exception.

→ The corrected program here:

```
class ThrowDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
```

```
        System.out.println("caught " + e);
    }
}
```

O/P: Inside throwOne
caught java.lang.IllegalAccessException: demo

finally

→ When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.

→ 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.

- The finally clause is optional. However each try statement requires at least one catch or a finally clause.

→ example program that shows three methods that exit in various ways, none without executing their finally clauses.

ex:
```
// Demonstrate finally
class FinallyDemo {
// Through an exception out of the method
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }
// Return from within a try block
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }
// Execute a try block normally
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
```

```
        System.out.println("procC's finally");
    }
}
public static void main(String args[]){
    try{
        procA();
    } catch(Exception e){
        System.out.println("Exception caught");
    }
    procB();
    procC();
}
```

O/P : Inside procA
procA's finally
Exception caught
Inside procB
procB's finally
Inside procC
procC's finally.

We see 6 Java's Built-in Exceptions.

Chained Exceptions

→ The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception.

→ To allows chained exceptions, two constructors and two methods were added to throwable

The constructors are shown here:

Throwable (Throwable causeExc)

Throwable (String msg, Throwable causeExc)

In the first form, causeExc is the exception that causes the current exception. That is causeExc is the underlying reason that an exception occurred.

→ The second form allows you to specify a description at the same time that you specify a cause exception.

→ These two constructors have also been added to the Error, Exception and RuntimeException classes.

Ex:

```
class ChainExcDemo {
    static void demoproc(){
    // create an exception
    NullPointerException e = new NullPointerExce
        -ption ("top layer");
    // add a cause
    e.initCause(new ArithmeticException("cause"));
    throw e;
    }
public static void main(String args[]){
    try{
        demoproc();
    } catch(NullPointerException e) {
        // display top level exception
        System.out.println("caught: " + e);
```

```
// display cause exception
System.out.println("original cause : " +
                    e.getCause());
}
}
```

o/p: Caught: java.lang.NullPointerException: top layer

Original cause: java.lang.ArithmeticException: cause.

* Creating Your Own Exception Subclasses

The exception class does not define any methods of its own. It does of course inherit those methods provided by Throwable.

Exception defines four constructors:

Exception()

Exception(String msg)

ex:
```
// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;
    MyException(int a) {
        detail = a;
    }
    public String toString() {
        return "MyException[" + detail + "]";
    }
}
```

```
class ExceptionDemo{
  static void compute (int a) throws MyException{
    System.out.println ("Called compute ("+a+") ");
    if (a>10)
      throw new MyException (a);
    System.out.println ("Normal exit");
  }
  public static void main (String args[]){
    try{
      compute (1);
      compute (20);
    } catch (MyException e){
      System.out.println ("Caught "+e);
    }
  }
}
```

o/p:
```
Called compute (1)
Normal exit
Called compute (20)
Caught MyException [20]
```

Module - 02

operators, Control statements ch4, ch5

Operators

Java provides a rich operator environment. Devided into the following four groups:
- Arithmetic
- bitwise
- relational and
- logical

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

operator	Result
+	Addition
-	Substraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	division assignment
%=	Modulus assignment
--	Decrement

The operands of the arithemetic operators must be of 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 sub set of int.

* The Basic Arithmetic Operators

→ The basic arithmetic operations - addition, subtraction, multiplication and division - all behaves as you would expect for all numeric types

→ The minus operator also has a unary form that negates its single operand.

→ When the division operator is applied to an integer type, there will be no fractional component attached to the result.

→ Example illustrates the difference between floating-point division and integer division

Ex: // 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);
 }
}
```

O/P: Integer Arithmetic

a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic

da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5

* The modulus operator

The modulus operator % returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

Ex:

```
// Demonstrate the % operator
class Modulus {
  public static void main(String args[])
  {
    int x = 42;
    double y = 42.25;
```

```
System.out.println("x mod 10 = " + x%10);
System.out.println("y mod 10 = " + y%10);
}
}
```

O/p : x mod 10 = 2
y mod 10 = 2.25

*Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

a = a + 4;

rewrite as a += 4;

This is version of += compound assignment operator

both statements perform the same action

Another example

a = a % 2;

which can be expressed as

a %= 2;

There are compound assignment operators for all of the arithmetic, binary operators

Thus any statement of the form

var = var op expression;

can be written as

var op = expression;

Benefits of compound assignment:

1) they save you bit of typing, shorthand
2) they are implemented more efficiently by the Java-run-time system than are their equivalent long forms.

Ex: // 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);
  }
}
```

o/p: a = 6
b = 8
c = 3

*Increment and Decrement

The Increment operator increases its operand by one.

The decrement operator decreases its operand by one.

x = x + 1; can be rewritten like this by use of the Increment operator

x++;

Similarly

x = x - 1;

is equivalent to

x--;

These operators are unique in that they can appear both in postfix form, where they follow the operand as just shown and prefix form, where they precede the operand.

→ In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.

→ In postfix form, the previous value is obtained for use in the expression and then the operand is modified.

```
x = 42;
y = ++x;          x = 43   y = 43.
```

y = ++x; is the equivalent of these two statements

```
x = x + 1;
y = x;
```

Prefix

```
x = 42;
y = x++;          x = 43   y = 42
```

y = x++; is the equivalent of these two statements.

```
y = x;
x = x + 1;
```

postfix

```
// Demonstrate ++
class IncDec {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c;
        int d;
        c = ++b;      // b = 3   c = 3
        d = a++;      // a = 2   d = 1
        c++;          // c = 4
        System.out.println("a = " + a);   // 2
        System.out.println("b = " + b);   // 3
        System.out.println("c = " + c);   // 4
        System.out.println("d = " + d);   // 1
    }
}
```

The Bitwise Operators

Java defines several bitwise operators that can be applied to the integer types long, int, short char, and byte.

These operators act upon the individual bits of their operands.

Operator → Result

~ → Bitwise unary NOT
& → Bitwise AND
| → Bitwise OR
^ → Bitwise exclusive OR
>> → shift right
>>> → shift right zero fill
<< → left shift
&= → Bitwise AND assignment
|= → Bitwise OR assignment
^= → Bitwise exclusive OR assignment
>>= → shift right assignment
>>>= → shift right zero fill
<<= → shift left assignment

All of the integer types are represented by binary numbers of varying bit widths.

The byte value for 42 in binary is 00101010 where each position represents a power of two, starting with 2^0 at the rightmost bit. The next bit position to the left would be 2^1 or 2, 1, 3 and so on.

42

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	1	0	1	0	1	0

All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones.

19/06/2024

Java uses an encoding known as two's complement

−42

00101010

↓ Invert

11010101

$$\begin{array}{r} +\quad\quad {}^{1}1 \\ \hline 11010110 \end{array} = -42$$

To decode a negative number, first invert all of the bits and then add 1.

The Bitwise Logical Operators

→ The bitwise logical operators are &, |, ^ and ~

→ bitwise operators are applied to each individual bit within each operand.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand.

Ex: 42

00101010 becomes 11010101

The Bitwise AND

The AND operator & produces a 1 bit if both operands are also 1. A zero is produced in all other cases

$$\begin{array}{lll} 00101010 & 42 \\ \& \ 00001111 & 15 \\ \hline 00001010 & 10 \end{array}$$

The Bitwise OR

The OR operates, |, combines bits such that if either of the bits in the operands is a 1

```
00101010   42
10000111   15
--------
00101111   47
```

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. otherwise the result is zero.

```
00101010   42
^ 00001111   15
--------
00100101   37
```

Using the Bitwise logical operators.

```
// Demonstrate the bitwise logical operators
class BitLogic {
    public static void main(String args[]) {
        String binary[] = { "0000", "0001", "0010",
"0011", "0100", "0101", "0110", "0111", "1000", "1001",
"1010", "1011", "1100", "1101", "1110", "1111" };
        int a = 3;
        int b = 6;
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;
        System.out.println("a = " + binary[a]);
        System.out.println("b = " + binary[b]);
        System.out.println("a|b = " + binary[c]);
        System.out.println("a&b = " + binary[d]);
        System.out.println("a^b = " + binary[e]);
        System.out.println("~a&b | a&~b = " + binary[f]);
        System.out.println("~a = " + binary[g]);
    }
}
```

o/p : a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b | a&~b = 0101
~a = 1100

The left shift

The left shift operator << shifts all of the bits in a value to the left a specified number of times.

value << num

→ Here num specifies the number of positions to left-shift the value in value.
→ the << moves all of the bits in the specified value to the left by the number of bit positions specified by num.
→ for each shift left, the higher-order bit is shifted out (and lost) and a zero is brought in on the right.

Ex: // left shifting a byte value

```
class ByteShift {
    public static void main (String args[])
        byte a=64, b;
        int i;
        i = a << 2;
        b = (byte) (a<<2);
    System.out.println ("Original value of a: "+a);
    System.out.println ("i and b: " + i + " "+b);
}
}
```

64

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

128 64 32 16 8 4 2 1

0 1 0 0 0 0 0 0

i = a << 2;

0 1 0 0 0 0 0 0 << 2;

1 0 0 0 0 0 0 0 → 1st time

1 0 0 0 0 0 0 0 0 → 2nd time

o/p: Original value of a: 64

i and b: 256 0

The Right shift

The right shift operator >>, shifts all of the bits in a value to the right a specified number of times

value >> num

Here num specifies the number of positions to the right-shift the value in value.

>> moves all of the bits in the specified value to the right the number of bits positions specified by num.

int a = 32;

a = a >> 2; // 8

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

128 64 32 16 8 4 2 1

0 0 1 0 0 0 0 0

0 0 0 1 0 0 0 0 → 1st shift

0 0 0 0 1 0 0 0 → 2nd shift

```
int a = 35;
a = a >> 2 ;   // 8
```

00100011
>> 1
00010001
>> 2
00001000 = 8

When we are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called sign extension

Ex: 8
2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
0 0 0 0 0 1 0 0 0 = 8

Negative 8 = −8

1 1 1 1 0 1 1 1
+ 1
1 1 1 1 1 0 0 0 = −8
>> 1
1 1 1 1 1 1 0 0 = −4

The Unsigned Right Shift (>>>)

→ We have seen the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value.

→ When you are working with pixel-based values and graphics you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift.

→ Java's unsigned shift-right operator >>>, which always shifts zeros into the higher-order bit

```
int a = -1;
a = a >>> 24;
```

Here a is set to -1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. sets a to 255.

1 = 00000000 00000000 00000000 00000001

↓ 1's compliment

11111111 11111111 11111111 11111110

↓ add +1 = 2's compliment -ve value

11111111 11111111 11111111 11111111 = -1 in binary as int

>>> 24

00000000 00000000 00000000 11111111 = 255 in binary as int

>>> → operates meaningful for 32 and 64-bit values.

→ Smaller values are automatically promoted to int in expressions. This means that sign-extension occurs and that the shift will take place on a 32-bit rather than on an 8 or 16-bit value.

→ One might expect an unsigned right shift on a byte value to zero-fill beginning at bit 7. But this is not the case, since it is a 32-bit value that is actually being shifted.

```
// Unsigned shifting a byte value
class ByteUShift {
  public static void main (String args[]){
    char hex[] = {'0','1','2','3','4','5','6','7',
    '8','9','a','b','c','d','e','f'};
    byte b = (byte) 0xf1;
    byte c = (byte) (b>>4);
    byte d = (byte) (b>>>4);
    byte e = (byte) ((b&0xff)>>4);
    System.out.println (" b = 0x" + hex[(b>>4)&0x0
      + hex [b & 0x0f]);
    System.out.println (" b >> 4 = 0x" + hex[(c>>4)
    0x0f] + hex [c & 0x0f]);
    System.out.println (" b >>> 4 = 0x" + hex[(d>>4
    & 0x0f] + hex [d & 0x0f]);
    System.out.println ("(b & 0xff) >> 4 = 0x" +
    hex[(e>>4) & 0x0f] + hex [e & 0x0f]);
  }
}
```

Bitwise Operator Compound Assignments

All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.

```
a = a >> 4;   } Right shift
a >>= 4;

a = a | b;   } OR
a |= b;
```

ex:
```
class OpBitEquals {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a |= 4;
    b >>= 1;
    c <<= 1;
    a ^= c;
    System.out.println("a = " + a); // 3
    System.out.println("b = " + b); // 1
    System.out.println("c = " + c); // 6
  }
}
```

o/p:

a |= 4 — a = 0001, 0100 → 0101 = 5

b >>= 1 — b = 0010 >> 1, 0001 = 1

c <<= 1 — c = 0011 << 1, 0110 = 6

a ^= c → 0101 = 5, 0110 = 6, 0011 = 3

Relational Operators.

The relational operators determine the relation that one operand has to the other. Specifically determine equality and ordering

operator	Result
==	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 va

ex: `int a=4;`
`int b=1;`
`boolean c = a<b;` // c = false

Boolean Logical Operators

The boolean logical operators shown here oper only on boolean operands

All of the binary logical operators combine two boolean values to form a resultant boolean value

Operator → Result

& → Logical AND
| → Logical OR
^ → Logical XOR
|| → short-circuit OR
&& → short-circuit AND
! → Logical unary NOT
&= → AND assignment
|= → OR assignment
^= → XOR assignment
== → Equal to
!= → Not equal to
?: → Ternary if-then-else

The logical Boolean operators &, | and ^ operate on boolean values in the same way that they operate on the bits of an integer.

The logical ! operator inverts the Boolean state:
!true == false and !false == true

A	B	A | B	A & B	A ^ B	! A
False	False	false	false	false	True
True	False	True	false	True	false
false	True	True	false	True	True
True	True	True	True	false	false

Ex: // Demonstrate the boolean logical operators
```
class BoolLogic {
    public static void main(String args[]){
        boolean a = true;
        boolean b = false;
        boolean c = a|b;
        boolean d = a&b;
        boolean e = a^b;
        boolean f = (!a&b)|(a&!b);
        boolean g = !a;
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
        System.out.println("a|b = " + c);
        System.out.println("a&b = " + d);
        System.out.println("a^b = " + e);
        System.out.println("!a&b|a&!b = " + f);
        System.out.println(" !a = " + g);
```

o/p: a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false.

Short-Circuit logical Operators

→ Java provides two interesting Boolean operators not found in many other computer languages.

→ These are secondary versions of the Boolean AND and OR operators and are known as short-circuit logical operators.

Extra just for Reference

→ Short-circuit logical operators are used in programming languages to perform logical operations with the advantage of stopping evaluation as soon as the result is determined.

→ This can improve performance and prevent potential errors from evaluating unnecessary expressions.

→ Logical AND (&&)
If the first operand is 'false' the overall expression is 'false' without evaluating the second operand.

→ Logical OR (||)
If the first operand is 'true' the overall expression is 'true' without evaluating the second operand.

Ex: if (denom != 0 && num / denom > 10)

Benefits

1) Performance optimization.
Avoid unnecessary computations by not evaluating the second operand when the result can already be determined by the first operand.

2. Avoiding errors

prevent runtime errors by skipping expressions that may cause errors if evaluated

```
boolean a = false
boolean result = a && (10/0 == 0);
```

Normal non-short circuit

```
boolean a = false
boolean result = a & (10/0 == 0);
```

The Assignment Operator

The assignment operator is the single equal sign

=

```
var = expression;
```

The assignment operator allows to create a chain of assignments

```
int x, y, z;
x = y = z = 100;
```

The ? Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?

```
expression 1 ? expression 2 : expression 3
```

Here expression 1 can be any expression that evaluates to a boolean value.

If expression 1 is true, then expression 2 is evaluated; otherwise expression 3 is evaluated.

ratio = denom == 0 ? 0 : num / denom;

Ex 6 // Demonstrate ?

```
class Ternary {
    public static void main(String arg
    {
    int p, k;

    p = 10;
    k = p < 0 ? -p : p;
    System.out.print("Absolute value of
    System.out.println(p + " is " + k);

    p = -10;
    k = p < 0 ? -p : p;
    System.out.print("Absolute value of ");
    System.out.println(p + " is " + k);
    }
}
```

o/p: Absolute value of 10 is 10
Absolute value of -10 is 10

Operator Precedence & Using Parentheses

parentheses, square brackets and the dot operator are separators. but they act like operators in an expression.

Parentheses are used to alter the precedence of an operation.

a >> b + 3 ; This expression first adds 3 to b and then shifts a right by that result.

a >> (b + 3) →

(a >> b) + 3 → first shift a right by b positions and then add 3 to that result.

Control Statements

Selection : allows your program to choose different paths of execution based upon the outcome of an expression or state of a variable

Iteration : enables program execution to repeat one or more statements

Jump : allows your program to execute in a non linear 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 run time.

if

The if statement is Java's conditional branch statement.

```
if (condition) statement 1;
else statement 2;
```

→ each statement may be a single statement or compound statement enclosed in curly braces.

→ If 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 statement 1 is executed otherwise statement 2 (if it exists) is executed.

→ It is possible to control the if using a single boolean variable as shown in this code fragment

```
boolean dataAvailable ;
//.....
if (dataAvailable)
    ProcessData ;
  else
    waitforMoreData ( );
```

→
```
int bytesAvailable ;
//.....
if (bytesAvailable > 0) {
      ProcessData ( );
      byteAvailable -= n;
  } else
      waitforMoreData();
```

→
```
int bytesAvailable ;
//.....
if (bytesAvailable >0) {
      ProcessData ( ) ;
      bytesAvailable -= n;
  } else {
      waitforMore Data ( );
      bytesAvailable = n ;
  }
```

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.

```
if (i == 10) {
    if (j < 20) a = b;
    if (k > 100) c = d;
    else a = c;
}
else a = d;
```

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.

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

→ if statements are executed from the top down
→ 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.

Eg:- // Demonstrate if-else-if statements

```
class IfElse {
    public static void main(String args[]){
        int month = 4;
        String season;
if (month == 12 || month == 1 || month == 2)
    season = "Winter";
else if (month == 3 || month == 4 || month == 5)
    season = "Spring";
else if (month == 6 || month == 7 || month == 8)
    season = "Summer";
else if (month == 9 || month == 10 || month == 11)
    season = "Autumn";
else
    season = "Bogus Month";
System.out.println("April is in the" + season +
                   ".");
    }
}
```

o/p: April is in the Spring.

Switch :- The switch statement is Java's multiway branch statement.
It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.

```
switch (expression) {
    case value 1 :
        // statement sequence
        break;
    case value 2 :
        // statement sequence
        break;
    .
    .
    .
    case value N :
        // statement sequence
        break;
    default :
        // default statement sequence
}
```

→ The switch statement works like this : The value of the expression is compared with each of the literal values in the case statements

→ If a match is found, the code sequence following that case statement is executed.

→ If none of the constants matches the value of expression, then the default statement is executed

→ The default statement is optional.

→ The break statement is used inside the switch to terminate a statement sequence.

Ex1// A simple example of the switch

```
class SampleSwitch {
    public static void main (String args[]) {
        for (int i=0 ; i<6 ; i++)
            switch (i) {
```

```
case 0:
 System.out.println (" i is zero.");
 break;
case 1:
 System.out.println (" i is one.");
 break;
case 2:
  System.out.println ("i is two.");
  break;
case 3:
  System.out.println (" i is three .");
  break;
default:
  System.out.println ("i is greater than 3.");
} } }
```

Ex2 // In a switch, break statements are optional

```
class MissingBreak {
  public static void main (String args[]) {
    for (int i = 0; i < 12; i++)
        switch (i) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
            System.out.println ("i is less than 5");
            break;
          case 5:
          case 6:
          case 7:
          case 8:
          case 9:
        System.out.println("i is less than 10");
         break;
        default:
} } } System.out.println ("i is 10 or more");
```

Ex 1 : output

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Ex 2 : output

```
i is less than 5.
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

Ex :

```
// An improved version of the season pro
class switch {
    public static void main (String args[]
        int month = 4;
        String season;
        switch (month) {
            case 12:
            case 1:
            case 2:
                season = "Winter";
                break;
```

```
        case 3:
        case 4:
        case 5:
            season = "Spring";
            break;
        case 6:
        case 7:
        case 8:
            season = "Summer";
            break;
        case 9:
        case 10:
        case 11:
            season = "Autum";
            break;
        default:
            season = "Bogus Month";
    }
    System.out.println("April is in the "+ season
                       + ".");
  }
}
```

* Iteration statements

→ Java's iteration statements are for, while and do-while.

→ These statements create what we commonly call loops

→ a loop repeatedly executes the same set of instructions until a termination condition is met

while

The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.

```
while (condition) {
    // body of loop
}
```

→ The condition can be any Boolean expression. 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.

Ex1:
```
// Demonstrate the while loop
class While {
    public static void main(String args[]) {
        int n = 10;
        while (n > 0) {
            System.out.println("tick " + n);
            n--;
        }
    }
}
```

o/p:
```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

Ex2:
```
int a = 10, b = 20;
while (a > b)
    System.out.println("This will not be displayed");
```

Ex 3 // The target of a loop can be empty

```
class NoBody {
public static void main(String args[]){
    int i, j;
    i = 100;
    j = 200;
    // find midpoint between i and j
    while (++i < --j) ;
    System.out.println("Midpoint is " + i);
  }
}
```

o/p: Midpoint is 150

do-while

→ We seen, if the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all.

→ Sometimes it is desirable to execute the body of a loop at least once, even if the condition -al expression is false to begin with.

→ The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

```
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.

Ex 6 // Demonstrate the do-while loop.

```
class Dowhile {
  public static void main(String args[])
    int n = 10;
    do {
      System.out.println("tick " + n);
      n--;
    } while (n>0);
  }
}
```

→
```
do {
  System.out.println("tick" + n);
  while (--n > 0);
}
```

for loop

It is a powerful and versatile construct.

```
for (initialization ; condition ; iteration)
{
  // body
}
```

Ex 7 // Demonstrate the for loop.

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

```
    int n;
for (n = 10 ; n > 0 ; n--)
    System.out.println ("tick" + n);
}
}
```

Ex 2 // Declare a loop control variable inside the for

```
class ForTick {
    public static void main (String args[]){
// here, n is declared inside of the for loop
    for (int n = 10 ; n>0 ; n--)
        System.out.println ("tick" + n);
}
}
```

Ex 3: Simple program that tests for prime numbers, that the loop control variable i, is declared inside the for since it is not needed elsewhere.

```
// Test for primes.
class FindPrime {
public static void main(String args[]){
        int num;
        boolean isPrime = true;

        num = 14;
    for (int i = 2; i <= num/i ; i++){
            if((num%i) == 0) {
                isPrime = false;
                break;
            }
    }
    if (isPrime) System.out.println ("Prime");
} else System.out.println ("Not Prime");
}
```

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.

```
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--;
        }
    }
}
```

Ex 6 // using the comma

```
class Comma {
    public static void main (String args[]) {
        int a, b;
        for (a = 1, b = 4; a < b; a++, b--) {
            System.out.println ("a = " + a);
            System.out.println ("b = " + b);
        }
    }
}
```

o/p:
```
a = 1
b = 4
a = 2
b = 3
```

* Some for loop Variations

The for loop supports a number of variations that increase its power and applicability

Ex 5 boolean done = false;
for (int i = 1; !done; i++) {
// ....
if (Interrupted()) done = true;
}

Ex 6 // parts of the for loop can be empty
class ForVar {
public static void main(String args[]){
int i;
boolean done = false;
i = 0;
for(; !done;) {
System.out.println("i is " + i);
if (i == 10) done = true;
i++;
}
}
}

Ex 7 If you leave all three parts of the for empty, an infinite loop (a loop that never terminates)
for (; ;) {
// ....
}
This loop will run forever because there is no condition under which it will terminate.

* The for-Each version of the for loop
a for-each loop by using the keyword foreach. Java adds the for-each capability by enhancing the for statement. The advantage of this approach is that no new keyword is required, and no ~~the for statement~~. preexisting code is broken.

The for-each style of for is also referred as the enhanced for loop.

for (type itr-var : collection) statement-block

→ Here type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time from beginning to end.

→ The collection being cycled through is specified by collection

→ There are various types of collections that can be used with the for, but the only type used in this chapter is the array.

Ex// traditional for loop to compute the sum of the values in an array.:

```
int nums[] = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
for (int i=0; i<10; i++)
    sum += nums[i];
```

Ex: for-each version of the for:

```
int nums[] = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
for (int x : nums) sum += x;
```

Ex: Entire program that demonstrates the for each version of the for.

```
// Use a for-each style for loop
class ForEach {
    public static void main(String args[]){
        int nums[] = {1,2,3,4,5,6,7,8,9,10};
        int sum = 0;

        for (int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
        }
        System.out.println("Summation: " + sum);
    }
}
```

o/p: Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55

Ex: terminate loop early using break in for-each style for

```
// Use break with a for-each style for
class ForEach2 {
    public static void main(String args[]){
        int sum = 0;
        int nums[] = {1,2,3,4,5,6,7,8,9,10};
        for (int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
            if (x == 5) break;
        }
        System.out.println("Summation of first 5 elements: " + sum);
    }
}
```

o/p: value is : 1
value is : 2
value is : 3
value is : 4
value is : 5
Summation of first 5 elements : 15

* Iterating over Multidimensional Arrays

The enhanced version of the for also works on multidimensional arrays.

ex: // Use for-each style for on a two-dimensional array.

```
class forEach3 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[][] = new int[3][5];

        // give nums some values.
        for (int i=0; i<3; i++)
            for (int j=0; j<5; j++)
                nums[i][j] = (i+1) * (j+1);
        // use for-each for to display and sum the values
        for (int x[] : nums) {
            for (int y : x) {
                System.out.println("Value is: " + y);
                sum += y;
            }
        }
        System.out.println("Summation: " + sum);
    }
}
```

Nested loops

Ex: // 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.

Using break

Uses of break statement

1. It terminates a statement sequence in a switch statement
2. It can be used to exit a loop
3. It can be used as a civilized form of goto

Ex 1:

```
class BreakLoop {
  public static void main(String args[]){
    for (int i=0 ; i< 100; i++) {
      if (i==10)
        break ; // terminate loop if i is 10
        System.out.println ("i: " + i);
    }
    System.out.println ("Loop complete.");
  }
}
```

Ex 6
```
public static void main (String args[]){
    int i=0;
    while (i<100){
        if (i==10) break;
        System.out.println ("i: "+i);
        i++;
    }
    System.out.println ("Loop Complete.");
}
}
```

Using break as a form of Goto

The general form of the labeled break statement is
```
break label;
```
→ label is the name of a label that identifies a block of code. When this form of break executes, control is transferred out of the named block.
→ The labeled block must enclose the break statement, but it does not need to be the immediately enclosing block.
→ To name a block, put a label at the start of it. a label is any valid Java identifier followed by colon.

Ex 6
```
class Break {
    public static void main(String args[]){
        boolean t = true;
        first : {
            second : {
                third : {
                    System.out.println ("Before the break.");
```

```
if (t) break second;  // break out of second block
System.out.println ("This won't execute");
}
System.out.println ("This won't execute");
}
System.out.println ("This is after second block.");
}
}
```

o/p: Before the break.
This is after second block.

Using Continue

Sometimes it is useful to force an early iteration of a loop

Ex:
```
class Continue1{
public static void main (String args[]){
for (int i= 0; i<10 ; i++){
System.out.print (i + " ");
if (i%2 == 0) continue;
System.out.println (" ");
}
}
}
```

o/p:
0 1
2 3
4 5
6 7
8 9

- The continue statement in Java never terminates the execution of any loops.
- The nature of the continue statement is to skip the current iteration and force for the next one.

Ex:
```
for (int i=0; i<=10; i++)
{
if (i<3){
continue;
}
System.out.println (i);
```

o/p:
3
4
5
6
7
8
9
10

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

Ex:

```
class Return {
    public static void main (String args[]);
        boolean t = true ;
    System.out.println ("Before the return.");
        if (t) return ;
    System.out.println ("This wont execute.");
    }
}
```

o/p : Before the return.

Module-03

Introducing classes.

class Fundamentals

→ class is a template for an object

→ an object is an instance of a class.

→ class is that it defines a new datatype once defined, this new type can be used to create objects of that type.

The General Form of a Class

→ A class is declared by use of the class keyword.

→ Very simple classes may contain only code or only data, most real-world classes contain both.

→ a class code defines the interface to its data.

→ A simplified general form of a class definition

```
class classname {
    type instance-variable 1;
    type instance-variable 2;
    // ....
    type instance-variable N;

    type methodname 1 (parameter-list) {
        // body of method
    }
    type methodname 2 (parameter-list) {
        // body of method
    }
    :
```

```
type methodname N (parameter-list){
    // body of method
}
}
```

→ The data or variables, defined within a class are called instance variables. The code is contained within methods.

→ the methods and variables defined within a class are called members of the class.

→ variables defined within a class are called instance variables because each instance of the class contains its own copy of these variables thus, the data for one object is seperate and unique from the data for another

A simple class

```
class Box {
    double width;
    double height;
    double depth;
}
```

Ex1e The complete program for Box class

```
class Box {
    double width;
    double height;
    double depth;
}
```

```
// This class declares an object of type Box.
class BoxDemo{
  public static void main(String args[]){
    Box mybox = new Box();
    double vol;
  // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;
  // compute volume of box
  vol = mybox.width * mybox.height * mybox.
        depth;
  System.out.println("Volume is " + vol);
  }
}
```

Ex2 ← // This program declares two box objects.

```
class Box {
  double width;
  double height;
  double depth;
}

class BoxDemo2{
  public static void main(String args[]){
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
  // assign value to mybox1's instance
     variables.
```

```
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
    /* assign different values to mybox2's instance
       variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
    // compute volume of first box
        vol = mybox1.width * mybox1.height
              * mybox1.depth;
        System.out.println("Volume is " + vol);

    // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

o/p: Volume is 300.0
Volume is 162.0

Declaring Objects

→ When you create a class, you are creating a new data type. you can use this type to declare objects of that type

→ 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. 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. for this use new operator

→ The new operator dynamically allocates (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

→ In Java, all class objects must be dynamically allocated.

→ declare an object of type Box

```
Box myBox = new Box();
```

→
```
Box myBox; // declare reference to object
myBox = new Box(); // allocate a Box object
```

figure 6.1 6 Declaring an object of type Box

Statement

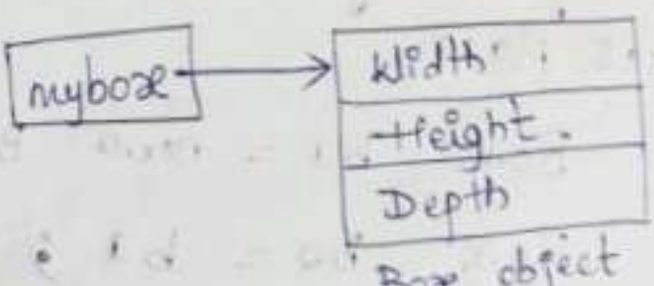
```
Box myBox;
```

Object

null

myBox

```
myBox = new Box();
```



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

→ a closer look at new

The new operates dynamically allocates memory for an object

```
class-var = new classname ();
```

→ class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The classname follow -ed by parentheses specifies the constructor for the class.

→ A constructor defines what occurs when an object of a class is created.

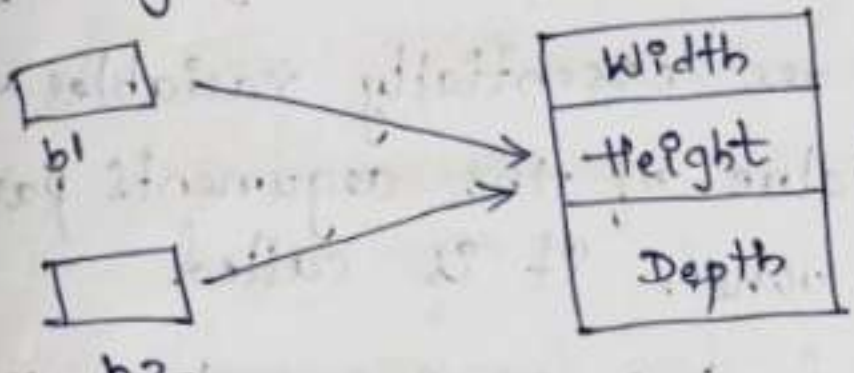
Assigning object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place.

```
Box b1 = new Box ();
Box b2 = b1 ;
```

Here b1 & b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object.

It simply makes b2 refer to the same object as does b1. any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.




```
Box b1 = new Box ();
Box b2 = b1;
 ||
b1 = null;
```

Here b1 has set to null, but b2 still points to the original object.

Introducing Methods.

Class consists of two things: instance variables and methods.

The general form of a method:

```
type name (parameter-list) {
    // body of method
}
```

→ Here type specifies the type of data returned by the method. The type should be valid.

→ If the method does not return a value, its return type must be void.

→ The name of the method is specified by name.

→ The parameter-list is a sequence of type and identifier pairs seperated by commas.

→ Parameters are essentially variables that receive the value of the arguments passed to the method. when it is called.

→ If the method has no parameters, then the parameter list will be empty.

→ Methods that have a return type other than void return a value to the calling routine using the following form of the return statement.

return value ;

Adding a Method to the Box Class

```
// This program includes a method inside the
box class.

class Box {
    double width;
    double height;
    double depth;
    // display volume of a box
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}
```

```
class BoxDemo3{
public static void main(String args[]){
 Box mybox1 = new Box();
 Box mybox2 = new Box();
 // assign values to mybox1's instance variables
 mybox1.width = 10;
 mybox1.height = 20;
 mybox1.depth = 15;
 /* assign different values to mybox2's
    instance variables */
 mybox2.width = 3;
 mybox2.height = 6;
 mybox2.depth = 9;
 // display volume of first box
  mybox1.volume();
 // display volume of second box
 mybox2.volume();
}
}
```

O/P: Volume is 3000.0
Volume is 162.0

Returning a Value

```
class Box {
 double width;
 double height;
 double depth;
```

```
    double volume(){
        return width * height * depth;
    }
}

class BoxDemo4{
    public static void main(String args[]){
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

1.      { vol = mybox1.volume();
        System.out.println("Volume is " + vol);

2.      { vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}

// other way
// System.out.println("Volume is " + mybox1.volume());
// System.out.println("Volume is " + mybox2.volume());
```

Adding a Method that Takes Parameters

a parameterized method can operate on a variety of data and/or be used on a number of slightly different solutions

```
int square ()
{
  return 10 * 10;
}

int square (int i)
{
  return i * i;
}
```

Example:
```
int x, y;
x = square (5);
x = square (9);
y = 2;
x = square (y);
```

A parameter is a variable defined by a method that receives a value when the method is called. ⇒ parameter

An argument is a value that is passed to a method when it is invoked. ⇒ square (100) ↑ arg

```
class Box {
  double width;
  double height;
  double depth;
```

```
    double volume () {
        return width * height * depth;
    }
void setDim (double w, double h, double d)
    {
     width = w;
     height = h;
     depth = d;
}}

class BoxDemo5 {
    public static void main (String args[]){
        Box mybox1 = new Box();
        Box mybox2 = new Box();
         double vol;
      mybox1.setDim (10,20,15);
      mybox2.setDim (3,6,9);
      vol = mybox1.volume ();
    System.out.println ("Volume is" +vol);
      vol = mybox2.volume ();
    System.out.println ("Volume is" +vol);
}}
```

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.

→ Constructor's job is to initialize the internal state of an object, so that the code creating an instance will have a fully initialized, usable object immediately.

```
/* Here, Box uses a constructor to initialize
   the dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;
// This is the constructor for Box.
Box() {
System.out.println("constructing Box");
    width = 10;
    height = 10;
    depth = 10;
}
double volume() {
    return width * height * depth;
}
}
```

```
class BoxDemo6 {
    public static void main (String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

O/P: Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0

→ class-var = new classname();

after classname we needed parentheses because the constructor for the class is being called.

Box mybox1 = new Box();

new Box() is calling the Box() constructor. When you do not explicitly define a constructor for a class then Java creates a default constructor for the class.

Parameterized Constructors

→ Construct Box objects of various dimensions.
→ preceding example, providing Box object is not useful, because all boxes have the same dimensions
→ The easy solution is to add parameters to the constructor.
* Here Box uses a parameterized constructor to initialize the dimensions of a box.
*/

```
class Box {
    double width;
    double height;
    double depth;
Box (double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}
double volume () {
    return width * height * depth;
}
}
class BoxDemo7 {
public static void main (String args[]) {
Box mybox1 = new Box (10, 20, 15);
Box mybox2 = new Box (3, 6, 9);
    double vol;
```

```
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

o/p: Volume is 3000.0
Volume is 162.0

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. this always a reference to the object on which the method was invoked.

```
Box(double w, double h, double d){
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Instance Variable Hiding

When a local variable has the same name as an instance variable, the local variable hides the instance variable.

```
Box (double width, double height, double depth){
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

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

→ C++ dynamically allocated objects must be manually released by use of a delete operator

→ In Java, it handles deallocation for you automatically, this technique called as garbage collection.

→ garbage collection 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.

→ Garbage collection only occurs if at all during the execution of your program.

The finalize() Method

→ Sometimes an object will need to perform some action when it is destroyed.

→ 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 finalize() method has the general form

```
protected void finalize()
{
  // finalization code here
}
```

→ the keyword protected is a specifier that prevents access to finalize() by code defined outside its class.

→ finalize() is only called just prior to garbage collection.

A stack class

→ By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it.

→ methods defines a consistent and controlled interface to the class data.

→ one of the archetypal examples of encapsulation is the stack.

→ A stack stores data using first-in, last-out ordering.

→ stacks are controlled through two operations called push and pop.

→ To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop.

// This class defines an integer stack that can hold 10 values.

```
class Stack {
    int stck[] = new int [10];
    int tos;

// Initialize top-of-stack
    Stack () {
        tos = -1;
    }
// push an item onto the stack
```

```
void push (int item){
    if (tos == 9)
    System.out.println ("stack is full.");
    else
     stck [++tos] = item;
  }
//pop an item from the stack
  int pop() {
   if (tos<0) {
    System.out.println ("stack underflow");
     return 0;
   }
   else
     return stck [tos--];
  }
}
```

→ The class TestStack shown here demonstrates the Stack class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
class TestStack {
public static void main(String args[]){
 Stack mystack1 = new Stack();
 Stack mystack2 = new Stack();
// push some numbers onto the stack
for(int i=0; i<10 ; i++)
  mystack1.push(i);
```

```
for (int i=10; i<20; i++)
    mystack2.push(i);
// pop those numbers off the stack
System.out.println("stack of mystack1:");
for (int i=0; i<10; i++)
    System.out.println(mystack1.pop());
System.out.println("stack of mystack2:");
for (int i=0; i<10; i++)
    System.out.println(mystack2.pop());
}
}
```

O/P: stack of mystack1:

9
8
7
6
5
4
3
2
1
0

stack of mystack2:

19
18
17
16
15
14
13
12
11
10

* A Closer look at Methods and Classes

→ In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this case, the methods are said to be overloaded. and the process is refered to as method overloading

→ Method overloading is one of the ways that Java supports polymorphism.

→ The overloaded methods must differ in the type and/or number of their parameters.

→ While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

→ When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Eg:- Ex illustrates method overloading

```
// Demonstrate method overloading
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
```

```
// overload test for one integer parameter
    void test (int a) {
      System.out.println ("a: "+a);
    }
// overload test for two integer parameters
    void test(int a, int b) {
      System.out.println ("a and b: "+a+
                          " "+b);
    }

// overload test for a double parameter
    double test (double a) {
      System.out.println ("double a: "+a);
      return a*a;
    }
}

class Overload {
    public static void main(String args[]){
        OverloadDemo ob = new OverloadDemo();
        double result;
    // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10,20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25)
                           " + result);
    }
}
```

O/P: No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test (123.25): 15190.5625

- test() is overloaded four times.
- The first version takes no parameters, the second takes one integer, the third takes two integer parameters and the fourth takes one double parameter
- The fourth version of test() also returns a value as of no consequence relative to overloading.

→ When an overloaded method is called, Java looks for a match between the arguments used to call the method and method's parameters.

→ *Java's automatic type conversions can play a role in overload resolution.

Ex: // Automatic type conversions apply to overloading.

```
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
```

```
// Overload test for two integer parameters:
  void test (int a, int b) {
    System.out.println("a and b:" + a+" "+b);
  }
// overload test for a double parameter
  void test (double a) {
    System.out.println("Inside test (double)a:"
                        + a);
  }
}

class Overload {
    public static void main(String args[]){
  OverloadDemo ob = new OverloadDemo();
    int i = 88;
      ob.test();
      ob.test(10,20);
    ob.test(i); // this will invoke test(double)
    ob.test(123.2); // this will invoke test(double)
  }
}
```

o/p: No parameters
a and b: 10 20
Inside test (double) a: 88
Inside test (double) a: 123.2

→ When test() is called with an integer argument inside Overload, no matching method is found. test(int) is not found, Java elevates
→ after test(int) is not found, Java elevates

; to double and then calls test (double)

→ If test (int) had been defined, it would have been called instead.

→ Java will employ its automatic type conversions only if no exact match is found.

⇒ "Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm

Overloading Constructors

In fact, for most real-world classes that you create, overloaded constructors will be the norm. latest version of Box program

```
class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box (double w, double h, double d){
        width = w;
        height = h;
        depth = d;
    }
    // compute and return volume.
    double volume () {
        return width * height * depth;
    }
}
```

Box() constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box() constructor.

example below statement is currently invalid

```
Box ob = new Box ();
```

Since Box() requires three arguments, it's an error to call it without them.

→ Box class is currently written; these other options are not available to you.

→ solution for this problem is easy: simply overload the Box constructor so that it handles the ~~sit~~ situations just described.

```
/* Here, Box defines three constructors
to initialize the dimensions of a box various
ways.
*/
class Box {
    double width;
    double height;
    double depth;
  // constructor used when all dimensions specified
  Box (double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
  }
```

```
// constructor used when no dimensions
   specified.
Box() {
   width = -1; // use -1 to indicate
   height = -1; // an uninitialized
   depth = -1; // box
}
// constructor used when cube is created
Box (double len) {
   width = height = depth = len;
}
// compute and return volume
double volume () {
   return width * height * depth;
}
}

class OverloadCons {
public static void main (String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box (10, 20, 15);
Box mybox2 = new Box ();
Box myclube = new Box (7);
double vol;
// get volume of first box
vol = mybox1.volume ();
System.out.println ("Volume of mybox1 is "+vol);
// get volume of second box
vol = mybox2.volume ();
```

```
System.out.println ("Volume of myboxe is "+vol);
// get volume of cube
vol = mycube.volume();
System.out.println ("Volume of mycube is " +vol);
}
}
```

```
O/P: Volume of mybox1 is 3000.0
     Volume of mybox2 is -1.0
     Volume of mybox2 is 343.0
```

Using Objects as Parametees

we have only been using simple types as parametees to methods. It is both correct and common to pass objects to methods.

```
// objects may be passed to methods
class Test {
    int a, b;
    Test (int i, int j) {
        a = i;
        b = j;
    }
// return true if o is equal to the invoking object
    boolean equals (Test o) {
        if (o.a == a && o.b == b)
            return true;
        else
            return false;
    }
}
```

```
class Passob {
public static void main (String args[]){
    Test ob1 = new Test (100, 22);
    Test ob2 = new Test (100, 22);
    Test ob3 = new Test (-1, -1);
System.out.println ("ob1 == ob2: " + ob1.equals(
                                    ob2));
System.out.println ("ob1 == ab3: " + ob1.equals(
                                    ob3));
}}
```

o/p: ob1 == ob2 : true

ob1 == ob3 : false

- the equals() method inside Test compares two objects for equality and returns the result.
- It compares the invoking object with the one that it is passed.
- If they contain the same values, then the method returns true otherwise it returns false
- the parameter o to equals() specifies Test as its type.
- one of the most common uses of object parameters involves constructors.
- you must define a constructor that takes an object of its class as parameter.

ex:// Here, Box allows one object to initialize another.

```
class Box {
  double width;
  double height;
  double depth;
// Notice this constructor. It takes an object of type Box.
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
// constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
// constructor used when no dimensions specified
  Box() {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
  }
// constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }
// compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class OverloadCons2 {
  public static void main(String args[]) {
// create boxes using the various
   constructors
```

```
Box mybox1 = new Box(10,20,15);
Box mybox2 = new Box();
Box mycube = new Box(7);

Box myclone = new Box(mybox1); // create copy
                                  of mybox1
double vol;
// get volume of first box
   vol = mybox1.volume();
   System.out.println("Volume of mybox1 is "+vol);
// get volume of second box
   vol = mybox2.volume();
   System.out.println("Volume of mybox2 is "+vol);

   // get volume of cube
      vol = mycube.volume();
      System.out.println("volume of cube is "+vol);
   // get volume of clone
      vol = myclone.volume();
      System.out.println("Volume of clone is "+vol);
   }
}
```

A closer Look at Argument Passing

→ There are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine. Changes made to the parameter of the subroutine have no effect on the argument

→ The second way an argument can be passed is call-by-reference. In this approach a reference to an argument is passed to the parameter (not a value of the argument). Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

Ex // primitive types are passed by value

```
class Test{
    void meth (int i, int j){
        i *= 2;
        j /= 2;
    }
}
class CallByValue{
    public static void main (String args[]){
        Test ob = new Test ();
        int a = 15, b = 20;
        System.out.println ("a and b before call:"
            + a + " " + b);
```

```
        ob.meth(a,b);
        System.out.println("a and b after call: "+
                           a+" "+b);
    }
}
```

o/p: a and b before call: 15 20
a and b after call: 15 20

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.

Ex 6

```
// objects are passed by reference
class Test{
    int a,b;
    Test(int i, int j){
        a=i;
        b=j;
    }
    // pass an object
    void meth(Test o){
        o.a *= 2;
        o.b /= 2;
    }
}
class CallByRef{
    public static void main(String args[]){
        Test ob=new Test(15,20);
```

```
System.out.println("ob.a and ob.b before call:"
        +ob.a +" " +ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call:"
        +ob.a+" "+ob.b);
}
}
```

o/p: ob.a and ob.b before call : 15 20
ob.a and ob.b after call : 30 10

Remember: When a primitive type is passed to a method, it is done by use of call-by-value. objects are implicitly by use of call-by-reference

Returning Objects

A method can return any type of data, including class types that you create.
the IncrByTen() method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
    int a;
    Test(int i) {
        a = i;
    }
    Test IncrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}
```

```
class Retob {
public static void main(String args[]){
    Test ob1 = new Test(2);
    Test ob2;
    ob2 = ob1.IncrByTen();
System.out.println("ob1.a: "+ob1.a);
System.out.println("ob2.a: "+ob2.a);
    ob2 = ob2.IncrByTen();
System.out.println("ob2.a after second
        increase: "+ ob2.a);
}
}
```

o/p: ob1.a : 2
ob2.a : 12
ob2.a after second increase : 22

Recursion.

Java supports recursion. Recursion is the proces of defining something in terms of itself.

Recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

Ex: // A simple example of recursion

```
class Factorial {
  // this is a recursive method
  int fact(int n) {
    int result;
    if (n==1) return 1;
    result = fact(n-1) * n;
    return result;
```

```
}}
class Recursion {
    public static void main(String args[]){
        Factorial f = new Factorial();
        System.out.println("Factorial of 3 is " + f.fact(3));
        "          "          4 is " + f.fact(4));
        "          "          5 is " + f.fact(5));
    }
}
```

op: Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

Introducing Access Control

Java's access specifiers are public, ~~static~~ private and protected.

Java also defines a default access level. protected applies only when inheritance is involved.

When a member of a class is modified by the public specifier, then that member can be accessed by any other code.

When a member of a class is specified as private, then that member can only accessed by other members of its class.

ex 6 /* This program demonstrates the difference between public and private */

```
class Test {
    int a; // default access
    public int b; // public access
    private int c; // private access
```

```
// methods to access c
void setc(int i) { // set c's value
    c = i;
}
int getc() { // get c's value
    return c;
}
}

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

// These are OK, a and b may be accessed directly
        ob.a = 10;
        ob.b = 20;

// this is not OK and will cause an error
// ob.c = 100; // Error!

// You must access c through its methods
        ob.setc(100);
        System.out.println("a, b, and c: " + ob.a + " " +
                           ob.b + " " + ob.getc());
    }
}
```

Understanding static

→ It is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member that can be done by precede its declaration with the keyword static.

→ When a member is declared static, it can be access-ed before any objects of its class are created and without reference to any object.

→ You can declare both methods and variables to be static.

→ The most common example of a static member is main(). main() is declared as static because it must be called before any objects exist.

→ Instance variables declared as static are essentially global variables. When objects of its class are declared, no copy of a static variable is made.

→ Instead, all instances of the class share the same static variable

Methods declared as static have several restrictions:

- They can only call other static methods
- They must only access static data
- They cannot refer to this or super in any way.

Ex:- // Demonstrate static variables, methods and blocks

```
class UseStatic {
    static int a=3;
    static int b;
    static void meth(int x) {
        System.out.println("x= "+x);
        System.out.println("a= "+a);
```

```
    System.out.println ("b = " + b);
  }
  static {
    System.out.println ("static block initialized");
    b = a * 4;
  }
  public static void main (String args[]) {
    meth (42);
  }
}
```

o/p: static block initialized.
x = 42
a = 3
b = 12

→ outside of the class in which they are defined, static methods and variables can be used independently of any object

→ if you wish to call a static method from outside its class, you use below general form

classname.method ()

→ A static variable can be accessed in the same way - by use of the dot operator on the name of the class.

→ Here is an example. Inside main() the static method callme() and the static variable b are accessed through their class name

```
class StaticDemo
```

```
class StaticDemo{
    static int a=42;
    static int b=99;
    static void callme(){
        System.out.println("a="+a);
    }
}
class StaticByName{
    public static void main(String args[]){
        StaticDemo.callme();
        System.out.println("b="+StaticDemo.b);
    }
}
```

o/p: a=42
b=99

Arrays Revisited

→ You know about classes, an important point can be made about arrays: they are implemented as objects.

→ Size of an array — that is the number of elements that an array can hold is found by its length instance variable.

→ All arrays have this variable and it will always hold the size of the array.

Ex: // This program demonstrates the length array member

```
class Length {
    public static void main (String args[]){
        int a1[] = new int[10];
        int a2[] = {3,5,7,1,8,99,44,-10};
        int a3[] = {4,3,2,1};
    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
    }
}
```

o/p : length of a1 is 10
length of a2 is 8
length of a3 is 4

Module - 05 (Part - 01)

Enumerations

ch 12.1, 12.2

* Enumerations

→ In simplest form an Enumeration is a list of named constants, enum in java is a data type that contains fixed set of constants

def° IMP

→ In java, an enumeration defines a class type

→ It can be used for days of the week (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday and Saturday), directions (North, South, East, and West) etc

→ It is available from JDK 1.5

→ An Enumeration can have constructors, methods and instance variables. It is created using enum keyword.

→ Each enumeration constant is public, static and final by default.

→ Even though enumeration defines a class type and have constructors, you do not instantiate an enum using new

→ Enumeration variables are used and declared in much a same way as you do a primitive variable

Enumeration Fundamentals

→ An enumeration is created using the enum keyword

→ Simple example for enumeration that lists various apple varieties

// An enumeration of apple varieties
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap,
Cortland
};

The identifiers Jonathan, GoldenDel are enumeration constants

→ These constants are self-typed in which self refers to the enclosing enumeration

→ You can declare and use an enumeration variable in which same way as you do one of the primitive types

ex: ap as a enumeration type varia be of enum Apple

Apple ap;

Because ap is of type Apple, the only values that it can be assigned are those defined by the enumeration. Example this assigns ap the value RedDel:

ap = Apple.RedDel;

→ Two enumeration constants can be compared for equality by using the == relational operator

```
if (ap == Apple.GoldenDel) // ....
```

→ An enumeration value can also be used to control a switch statement, all of the case statements must use constants from the same enum as that used by the switch expression

```
// Use an enum to control a switch statement
switch (ap) {
  case Jonathan:
    // .....
  case Winesap:
    // .....
```

→ When an enumeration constant is displayed, such as in a println() statement, its name is output

```
System.out.println(Apple.Winesap);
```

the name Winesap is displayed.

→ The following program puts together all of the pieces and demonstrates the Apple enumeration (IMP)

```
// An enumeration of apple varieties
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland;
}
```

```
class EnumDemo {
    public static void main(String args[])
    {
        Apple ap;

        ap = Apple.RedDel;
// output an enum value
        System.out.println("Value of ap: " + ap);
        System.out.println();

        ap = Apple.GoldenDel;

        if (ap == Apple.GoldenDel) // compare two enum values
            System.out.println("ap contains GoldenDel.\n");

// Use an enum to control a switch statement
        switch(ap)
        {
            case Jonathan:
                System.out.println("Jonathan is red.");
                break;
            case GoldenDel:
                System.out.println("Golden Delicious is
                                    yellow.");
                break;
            case RedDel:
                System.out.println("Red Delicious is red.");
                break;
            case Winesap:
                System.out.println("Winesap is red.");
                break;
            case Cortland:
                System.out.println("Cortland is red.");
                break;
        }
    }
}
```

The output from the program
value of ap: Red Del

ap contains GoldenDel.

Golden Delicious is yellow

(IMP)* The values() and .valueOf() Methods

→ All enumerations automatically contain two predefined methods: values() and valueOf()

→ The java compiler internally adds the values() method when it creates an enum.

→ The values() method returns an array that contains a list of the enumeration constants. (all the values of the enum)

public static enum-type[] values()

→ The valueOf() method returns the enumeration constant whose value corresponding to the string passed in str. (return the enumeration constant whose value is equal to the string passed in an argument while calling this method.

public static enum-type valueOf(String str)

Example for values() and valueOf() methods

(IMP)

```
// Use the built-in enumeration methods
// An enumeration of apple varieties
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String args[]) {
        Apple ap;
        System.out.println("Here are all Apple
                            constants:");
        // use values()
        Apple allapples[] = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // use of valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}
```

The output from the program

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Java Enumerations Are Class Types

- In Java enumeration is a class type
- You don't instantiate an enum using new
- It is important to understand that each enumeration constant is an object of the enumeration type.
- When you define a constructor for an enum, the constructor is called when each enumeration constant is created
- Also each enumeration constant has its own copy of any instance variables defined by the enumeration.

// use an enum constructor, instance variable and method.

```
enum Apple {
    Jonathan (10), GoldenDel (9), RedDel (10),
    Winesap (15), Cortland (8);

    private int price; // price of each apple

    // constructor
    Apple (int p) { price = p; }

    int getPrice() { return price; }
}
```

Return

```
class EnumDemo3{
    public static void main(String args[])
    {
        Apple ap;

        // Display price of Winesap
        System.out.println("Winesap costs " +
                Apple.Winesap.getPrice() + " cents.\n");

        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(Apple a : Apple.values())
            System.out.println(a + " costs " +
                    a.getPrice() + " cents.");
    }
}
```

The output is shown here

Winesap costs 15 cents

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents
RedDel costs 12 cents
Winesap costs 15 cents
Cortland costs 8 cents.

→ Apple adds three things.
→ The first is the instance variable price, which is used to hold the price of each variety of apple.

21/07/2024 19:31

Rajguru

→ The second is the Apple constructor, which is passed the price of an apple

→ The third is the method getPrice(), which returns the value of price.

Enumerations Inherit Enum

All enumerations automatically inherit one: java.lang.Enum

You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the ordinal() method.

shown here:

final int ordinal()

→ It returns the ordinal value of the invoking constant

ordinal values begin at zero

→ In the Apple enumeration, Jonathan has an ordinal value of zero, GoldenDel has an ordinal value of 1. and so on

→ You can compare the ordinal value of two constants of the same enumeration by using the compareTo() method

final int compareTo(enum-type e)

Here, enum-type is the type of the enumeration and e is the constant being compared to the invoking constant.

→ Remember both the invoking constant and e must be of the same enumeration.

→ You can compare for equality an enumeration constant with any other object by using equals(), which overrides the equals() method defined by Object.

→ Remember you can compare two enumeration references for equality by using ==.

→ The following program demonstrates the ordinal(), compareTo() and equals() methods.

```
// Demonstrate ordinal(), compareTo() and
   equals().

// An enumeration of apple varieties
   enum Apple{
      Jonathan, GoldenDel, RedDel, Winesap,
         Cortland
   }

   class EnumDemo4{
      public static void main(String args[])
      {
      Apple ap, ap2, ap3;

// Obtain all ordinal values using ordinal().
System.out.println("Here are all apple constant
   +"and their ordinal values:");
```

```
for (Apple a : Apple.values())
    System.out.println(a + " " + a.ordinal());

ap = Apple.RedDel;
ap2 = Apple.GoldenDel;
ap3 = Apple.RedDel;

System.out.println();

// Demonstrate compareTo() and equals()
if (ap.compareTo(ap2) < 0)
    System.out.println(ap + " comes before " +
                       ap2);
if (ap.compareTo(ap2) > 0)
    System.out.println(ap2 + " comes before " +
                       ap);
if (ap.compareTo(ap3) == 0)
    System.out.println(ap + " equals " + ap3);

System.out.println();

if(ap.equals(ap2))
    System.out.println(ap + " equal!");

if (ap.equals(ap3))
    System.out.println(ap + " equals " + ap3);

if (ap == ap3)
    System.out.println(ap + " == " + ap3);
  }
}
```

The output from the program is shown here:

Here are all apple constants and their ordinal values:

Jonathan	0
GoldenDel	1
RedDel	2
Winesap	3
Cortland	4

GoldenDel comes before RedDel
RedDel equals RedDel

RedDel equals RedDel
RedDel == RedDel.

Refer: Another example for Enumeration. textbook

IMP Type Wrappers:

→ Java uses primitive types (also called simple types) such as int or double, to hold the basic data types supported by the language.

→ Primitive types, rather than objects are used for these quantities for the sake of performance.

→ Despite the performance benefit offered by the primitive types, there are times when you will need an object representation.

→ For example, you can't pass a primitive type by reference to a method. Also many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types.

→ To handle these situations, Java provides type wrappers, which are classes that encapsulate a primitive type within an object.

→ The type wrappers are Double, Float, Long, Integer, Short, Byte, Character and Boolean.

These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character

Character is a wrapper around a char. The constructor for Character is

Character (char ch)

Here ch specifies the character that will be wrapped by the character object being created.

To obtain the char value contained in a character object, call charValue(),

char charValue()

Boolean

Boolean is a wrapper around boolean values. It defines these constructors:

Boolean (boolean boolValue)
Boolean (String boolString)

first version, boolValue must be either true or false

second version, if boolString contains the string "true"; then the new Boolean object will be true. otherwise it will be false

To obtain a boolean value from a Boolean object, use booleanValue() shown here

boolean booleanValue()

The Numeric Type Wrappers

The most commonly used type wrappers are those that represent numeric values.

These are Byte, Short, Integer, Long, Float and Double.

All of the numeric type wrappers beside the abstract class Number.

21/07/2024 19:32

→ Number declares methods that return the value of an object in each of the different number formats.

byte	byteValue()
double	doubleValue()
float	floatValue()
int	intValue()
long	longValue()
short	shortValue()

For example, doubleValue() returns the value of an object as a double

→ All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value

```
Integer (int num)
Integer (String str)
```

→ All of the type wrappers override toString(). It returns the human-readable form of the value contained within the wrapper.

→ The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
class Wrap {
    public static void main(String args[])
    {
    Integer iob = new Integer(100);

    int i = iob.intValue();
    System.out.println(i + " " + iob);
    }
}
```

// displays 100 100

The process of encapsulating a value within an object is called boxing

```
Integer iob = new Integer(100);
```

The process of extracting a value from a type wrapper is called unboxing

```
int i = iob.intValue();
```

21/07/2024 19:33

Module - 05 (Part-02)

String Handling

The String Constructors

→ The String class supports several constructors. To create an empty string, you call the default constructor

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

→ To create a string initialized by an array of characters use the constructor shown here

```
String(char chars[])
```

Ex: `char chars[] = {'a', 'b', 'c'};`
`String s = new String(chars);`

→ You can specify a subrange of a character array as an initializer using the following constructor

```
String(char chars[], int startIndex, int numchars);
```

→ startIndex specifies the index at which the subrange begins and numchars specifies the number of characters to use.

```
char chars[] = {'a', 'b', 'c', 'd', 'e', 'f'};
String s = new String(chars, 2, 3);
```

This initializes s with the characters cde

→ You can construct a String object that contains the same character sequence as another String object using this constructor :

```
String(String strObj)
```

Ex :-
```
class MakeString {
    public static void main(String args[])
    {
    char c[] = {'J', 'a', 'v', 'a'};
    String s1 = new String(c);
    String s2 = new String(s1);

    System.out.println(s1);
    System.out.println(s2);
    }
}
```

output :-
```
Java
Java
```

→ Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.

The byte formats are these forms are shown here

```
String(byte asciiChars[])
String(byte asciiChars[], int startIndex,
       int numChars)
```

Here, asciiChars specifies the array of bytes

ex:-
```
class SubStringCons {
public static void main (String args[]){
    byte ascii[] = {65, 66, 67, 68, 69, 70};

    String s1 = new String (ascii);
    System.out.println (s1);

    String s2 = new String (ascii, 2, 3);
    System.out.println (s2);
  }
}
```

o/p :- ABCDEF
CDE

String Constructors Added by J2SE 5

J2SE 5 added two constructors to String. The first supports the extended Unicode character set and is shown here

```
String (int codePoints [], int startIndex,
        int numChars);
```

The second new constructor supports the new StringBuilder class

```
String (StringBuilder strBuildObj)
```

This constructs a String from the StringBuilder passed in strBuildObj.

String length

The length of a string is the number of characters that it contains. To obtain this value, call the length() method.

Syntax length()

Ex- char chars[] = {'a', 'b', 'c'};
String s = new String(chars);
System.out.println(s.length()); // 3

- there are 3 characters in the string s

* Special String Operations

Java has important and common part in programming. Java added special support for several string operations within the syntax of the language

- the java has special operations include the automatic creation of new string instances from the string literals, concatenation of multiple String objects by use of the + operator and the conversion of other data types to a string representation.

a) String Literals

To explicitly create a string instance from an array of characters by using the new operator. an easier way to do this using a string literal.

Java automatically constructs a String object

```
char chars[] = {'a', 'b', 'c'};
String s1 = new String(chars);

String s2 = "abc"; // Use string literal
```

b) String Concatenation

→ Java does not allow operators to be applied to string objects.

→ The one exception to this rule is the + operator which concatenates two strings, producing a String object

```
String age = "9";
String s = "He is" + age + " years old.";
System.out.println(s);
```

Ex 6

```
// Using concatenation to prevent long lines
class Concat {
    public static void main(String args[]) {
        String longStr = "This could have" +
        "a very long line that could have " +
        "wrapped around. But string concatenation" +
        + "prevents this.";

        System.out.println(longStr);
    }
}
```

c) String Concatenation with Other Data Types

You can concatenate strings with other types of data.

Ex
```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

Be careful when you mix other types of operations with string concatenation expressions

Ex
```
String s = "four: " + 2 + 2;
System.out.println(s);
```
This fragment displays
four: 22
rather than
four: 4

```
String s = "four: " + (2+2);
```
s contains the string "four: 4".

d) String Conversion and toString()

→ When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method valueOf() defined by String.

→ Every class implements toString() because it is defined by Object.

→ The toString() method has this general form

String toString()

Ex: // override toString() for Box class

```
class Box {
    double width;
    double height;
    double depth;

    Box (double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    public String toString() {
        return "Dimensions are " + width + " by " +
            depth + " by " + height + ".";
    }
}

class toStringDemo {
    public static void main (String args[]) {
        Box b = new Box (10, 12, 14);
        String s = "Box b: " + b; // concatenate
                                  Box object
        System.out.println (b); // convert Box to string
        System.out.println (s);
    }
}
```

o/p : Dimensions are 10.0 by 14.0 by 12.0

Box b: Dimensions are 10.0 by 14.0 by 12.0

Character Extraction

The string class provides a number of ways in which characters can be extracted from a string object

Like arrays, the string indexes begin at zero

a) charAt()

To extract a single character from a string you can refer directly to an individual character via the charAt() method.

```
char charAt (int where)
```

Here where is the index of the character that you want to obtain

Ex:-
```
char ch;
ch = "abc".charAt(1);
```
o/p : ch = b

b) getChars()

If you need to extract more than one characters at a time, you can use the getChars() method.

```
void getChars (int sourceStart, int sourceEnd,
    char target[], int targetStart)
```

sourceStart specifies the index of the beginning of the substring and sourceEnd specifies an index that is one past the end of the desired

```
class getCharsDemo{
    public static void main(String args[])
    {
    String s = "This is a demo of the
                get chars method.";
    int start = 10;
    int end = 14;
    char buf[] = new char[end - start];

    s.getChars(start, end, buf, 0);
    System.out.println(buf);
    }
}
```

O/P: demo

c) getBytes()

There is an alternative to getChars() that stores the characters in an array of bytes. This method is called getBytes()

```
byte[] getBytes()
```

d) toCharArray()

If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray(). It returns an array of characters for the entire string.

```
char[] toCharArray()
```

* String Comparison

The String class includes several methods that compare strings or substrings within strings.

a) equals() and equalsIgnoreCase()

→ To compare two strings for equality, use equals(). It has this general form

```
boolean equals(Object str)
```

→ To perform a comparison that ignores case differences, call equalsIgnoreCase().

```
boolean equalsIgnoreCase(String str)
```

Ex:-

```
class equalsDemo{
    public static void main(String args[]){
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Good-bye";
        String s4 = "HELLO";
        System.out.println(s1 + "equals " + s2 + "->" +
                           s1.equals(s2));
        System.out.println(s1 + "equals" + s3 + "->" +
                           s1.equals(s3));
        System.out.println(s1 + "equals " + s4 + "->" +
                           s1.equals(s4));
        System.out.println(s1 + "equalsIgnoreCase " + s4
                           + "->" + s1.equalsIgnoreCase(s4));
    }
}
```

O/P :- Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true

b) regionMatches ()

The regionMatches method compares a specific region inside a string with another specific region in another string

General form for these two methods:

boolean regionMatches (int startIndex, String str2, int str2StartIndex, int numChars)

boolean regionMatches (boolean ignoreCase, int startIndex, String str2, int str2startIndex, int numch -ars)

For both version, startIndex specifies the index at which the region begins within the invoking String object

c) startsWith () and endsWith()

→ String defines two routines that are, more or less specialized forms of regionMatches

→ The startsWith() method determines whether a given String begins with a specified string

→ The endsWith () determines whether the string

to question ends with a specified string. They have the following general forms

```
boolean startsWith(String str)
boolean endsWith(String str)
```

str is the string being tested.
If the string matches, true is returned otherwise false.

```
"Foobar".endsWith("bar")
```

and

```
"Foobar".startWith("Foo")
```

are both true.

A second form of startsWith()

```
boolean startsWith(String str, int startIndex)
```

d) equals() versus ==

→ It is important to understand that the equals() method and the == operator perform two different operations

→ The equals() method compares the characters inside a string object

→ The == operator compares two object references to see whether they refer to the same instance

PAGE NO.:
DATE: / /

Ex: // equals() vs ==

```
class EqualsNotEqualTo {
    public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = new String(s1);
        System.out.println(s1 + " equals " + s2 + " -> " +
                           s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " +
                           (s1 == s2));
    }
}
```

O/P: Hello equals Hello -> true
Hello == Hello -> false

e) compareTo()

→ To know which is less than, equal to, or greater than String method compareTo() serves this purpose

int compareTo(String str)

Value	Meaning
Less than zero	→ The invoking string is less than str
Greater than zero	→ The invoking string is greater than str
zero	→ The two strings are equal.

Ex: A bubble sort for strings

```
class SortString {
    static String arr[] = {
"Now", "is", "the", "time", "for", "all", "good", "men",
"to", "come", "to", "the", "aid", "of", "their"
    };
```

```
public static void main(String args[]){
    for(int i=0;i<arr.length;i++){
        if(arr[i].compareTo(arr[j])<0){
            String t = arr[j];
            arr[j] = arr[i];
            arr[i] = t;
        }
    }
    System.out.println(arr[j]);
}
}
```

O/P:
```
Now
and
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

* Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring

indexOf()	Searches for the first occurence of a character or substring
lastIndexOf()	Searches for the last occurence of a character or substring

→ To search for the first occurence of a character, use
int indexOf(int ch)

To search for the last occurrence of a character, use
int lastIndexOf(int ch).

Refer ex 6 in textbook page 369

Modifying a String

Because String objects are immutable, whenever you want to modify a string, you must either copy it into a StringBuffer or StringBuilder as following methods

a) substring()

You can extract a substring using substring(). It has two forms. The first is

String substring(int startIndex)

→ startIndex specifies the index at which the substring will begin.

The second form of substring() allows you to specify both the beginning and ending index of the substring:

String substring(int startIndex, int endIndex)

ex 6 — // Substring replacement

```
class StringReplace {
    public static void main(String args[]) {
        String org = "This is a test. This is too.";
        String search = "is";
        String sub = "was";
        String result = "";
        int i;
```

```
do {                    // replace all matching substrings
    system.out.println (org);
    i = org.indexof (search);
    if (i != -1) {
        result = org.substring (0, i);
        result = result + sub;
        result = result + org.substring (i + search.
                                length());
        org = result;
    }
} while (i != -1);
}
}
```

o/p : This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.

b) concat()

You can concatenate two strings using concat().

String concat (String str)

This method creates a new object that contains the invoking string with the contents of str appended to the end. concat() performs the same function as +.

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string "onetwo" into s2.

```
String s1 = "one";
String s2 = s1 + "two";
```

c) .replace ()

The replace () method has two forms: The first replaces all occurrences of one character in the invoking string with another character.

String replace (char original, char replacement)

String s = "Hello".replace ('l', 'w');
put the string "Hewwo" into s.

The second form of replace () replaces one character sequence with another

String replace (CharSequence original, CharSequence replacement)

d) trim ()

The trim () method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

String trim ()

String s = " Hello World ".trim ();

This puts the string "Hello World" into s.

Refer textbook example. page No: 372

PAGE NO.:
DATE: / /

* StringBuffer

StringBuffer is a peer class of string that provides much of the functionality of strings

String represents fixed-length, immutable character sequences.
StringBuffer represents growable and writeable character sequences

StringBuffer Constructors

StringBuffer defines these four constructors

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

The default constructor reserves room for 16 characters without reallocation

a) length() and capacity()

The current length of a StringBuffer can be found via the length() method, while the total allocated capacity can be found through the capacity() method.

```
int length()
int capacity()
```

Ex: // StringBuffer length vs. capacity

```
class StringBufferDemo{
    public static void main(String args[]){
StringBuffer sb = new StringBuffer("Hello");
```

```
System.out.println("buffer = " + sb);
System.out.println("length = " + sb.length());
System.out.println("capacity = " + sb.capacity());
}
}
```

o/p: buffer = Hello
length = 5
capacity = 21

b) ensureCapacity()

If you want to pre allocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity() to set the size of the buffer.

void ensureCapacity(int capacity)
Here capacity specifies the size of the buffer

c) setLength()

To set the length of the buffer within a StringBuffer object, use setLength().

void setLength(int len)
Here len specifies the length of the buffer

d) charAt() and setCharAt()

The value of a single character can be obtained from a StringBuffer via the charAt() method. You can set the value of a character within a StringBuffer using setCharAt().

char charAt(int where)
void setCharAt(int where, char ch)

e) getchars()

To copy a substring of a StringBuffer into an array, use the getchars() method.

```
void getchars(int sourcestart, int sourceEnd,
              char target[], int targetstart)
```

Here sourcestart specifies the index of the beginning of the substring and sourceEnd specifies an index that is one past the end of the desired substring.

f) append()

The append() method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

Ex:-

```
class appendDemo{
    public static void main(String args[]){
        String s;
        int a=42;
        StringBuffer sb=new StringBuffer(40);
        s= sb.append("a=").append(a).append
        ("!").toString();
        System.out.println(s);
    }
}
```

o/p: a=42!

a) insert()

The insert() method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings, Objects and charSequences

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Ex:-
```
class insertDemo{
    public static void main(String args[]){
        StringBuffer sb = new StringBuffer("I Java!");
        sb.insert(2,"like");
        System.out.println(sb);
}}
```
o/p: I like Java!

b) reverse()

You can reverse the characters within a StringBuffer object using reverse()

```
StringBuffer reverse()
```

Ex:-
```
class ReverseDemo{
    public static void main(String args[]){
        StringBuffer s=new StringBuffer("abcdef");
        System.out.println(s);
        s.reverse();
        System.out.println(s);
}}
```
o/p:- abcdef
fedcba

8) delete() and deleteCharAt()

You can delete characters within a StringBuffer by using the methods delete() and deleteCharAt().

StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)

→ The delete() method deletes a sequence of characters from the invoking object.

Ex: class deleteDemo{
public static void main(String args[]){
StringBuffer sb = new StringBuffer("This is a test.");
sb.delete(4,7);
System.out.println("After delete: " + sb);
sb.deleteCharAt(0);
System.out.println("After deleteCharAt: " + sb);
}}

O/P: After delete: This a test.
After deleteCharAt: his a test

9) replace()

You can replace one set of characters with another set inside a StringBuffer object by calling replace().

StringBuffer replace(int startIndex, int endIndex, String str)

Ex: class replaceDemo{
public static void main(String args[]){
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5,7, "was");
System.out.println("After replace: " + sb);
}}

O/P: After replace: This was a test.