



OBJECT ORIENTED PROGRAMMING WITH JAVA

1BCS302

Course Name

Course Code

Contact Hours

40

CIE Marks

50

SEE Marks

50

Course Coordinator

Mrs. MADHU NAGARAJ
Assistant Professor
Dept of CSEData Science
ATMECE, Mysuru



Course Outcomes

- CO1. Demonstrate proficiency in writing simple programs involving branching and looping structures.**
- CO2. Construct object-oriented solutions using classes, objects, and methods.**
- CO3. Develop programs using inheritance, packages, and interfaces.**
- CO4. Make use of exception handling and multithreading in solving complex problems**
- CO5. Apply concepts of autoboxing, enumerations, and generics in program development.**
- CO6. Choose a suitable IDE tool to design, develop, and debug Java programs.**



Course Learning Objectives

CLO 1: To learn primitive constructs JAVA programming language.

CLO2: To understand Object Oriented Programming Features of JAVA.

CLO3: To gain knowledge on: packages, multithreaded programing and exceptions.



Text Book

Herbert Schildt and Danny Coward, Java: The Complete Reference, 13th Edition, McGraw-Hill, 2024.

Reference Books

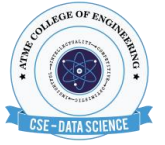
- 1. Cay S. Horstmann, Core Java Volume I – Fundamentals, 14th Edition, Oracle Press, 2026.**
- 2. E. Balagurusamy, Programming with Java, 7th Edition, McGraw-Hill Education, 2024.**

Prerequisite

To learn Java, you must have the basic knowledge of C/C++ programming language.



A T M E
College of Engineering



An Overview of Java: Module 1- Chapt 1

Mrs. Madhu Nagaraj
Assistant Professor
Dept of CSE-Data Science
ATMECE

What is Java

- Java is a most popular, object-oriented, widely used programming language and platform that is utilized for Android development, web development, artificial intelligence, cloud applications, and much more.
- So, mastering this gives you great opportunities in bigger organizations.

History

- Java was developed by Sun Microsystems in the year 1995.
- JAVA is now the owned and controlled by Oracle.
- James Gosling is known as the father of Java.
- Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the Oak name to Java.

Object-Oriented Programming

Java is a programming language, and it heavily relies on Object-Oriented Programming (OOP).

This means that when you write code in Java, you're essentially working with objects.

Objects are instances of classes, which are like blueprints defining the properties and behaviors of those objects.



Real world example for class and object

Let's consider a real-world example using a "Car" analogy:

Class: Car

A class is like a blueprint or template for creating cars. It defines the properties and behaviors that all cars of that type should have.

Properties (attributes): color, model, year, speed, fuel level.

Behaviors (methods): start, accelerate, brake, honk.

Object: myCar

An object is an instance created from the class. So, "myCar" is an object of the "Car" class.

Properties of "myCar": red color, model XYZ, year 2022, current speed 0, full fuel tank.

Behaviors: It can start, accelerate, brake, and honk.

In this example, the "Car" class serves as a blueprint for any car, defining what properties and behaviors a car should have. The "myCar" object is a specific instance of a car with its unique characteristics. This mirrors the idea of classes and objects in programming, where a class defines the structure, and objects are instances created based on that structure.

Object: myAnotherCar

An object is an instance created from the class. So, "myAnotherCar" is an object of the "Car" class.

Properties of "myAnotherCar": blue color, model ABC, year 2020, current speed 100KM/hr, 60% fuel tank.

Behaviors: It can start, accelerate, brake, and honk.

Basic Structure of Java Program

```
public class MyClass  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello World");  
    }  
}
```

Two Paradigms of Programming

1. Process-Oriented Programming (POP)
2. Object-Oriented Programming (OOP):

Process-Oriented Programming (POP)

Focus: POP focuses on the processes or functions that manipulate data. The emphasis is on procedures or routines that perform specific tasks.

Data Handling: Data and functions are separate entities. Data is often global and can be accessed by any function.

Top-Down Approach: Programs are typically designed using a top-down approach, breaking down the problem into smaller tasks and procedures.

Example: Traditional procedural languages like C are process-oriented. In C, you write functions that operate on data, and the flow of the program is directed by function calls.

Object-Oriented Programming (OOP):

Focus: OOP focuses on modeling real-world entities as objects that have both data (attributes) and behaviors (methods).

Data Encapsulation: Data and methods are encapsulated together into objects. Objects interact with each other through well-defined interfaces.

Abstraction and Inheritance: OOP supports concepts like abstraction, allowing you to hide complex implementation details, and inheritance, enabling you to create new classes based on existing ones.

Example: Java, Python, and C++ are examples of object-oriented languages. In Java, for instance, you define classes representing objects, and these classes encapsulate both data (attributes) and methods (behaviors).

Abstraction

Abstraction is a fundamental concept in computer science and Object-Oriented Programming (OOP) that involves simplifying complex systems by modeling classes based on the essential properties and behaviors they share.

Abstraction allows you to focus on the essential characteristics of an object while ignoring the unnecessary details.

Let's consider a real-world example to illustrate abstraction: a TV remote control.

In this example:
Abstraction of a TV Remote:

Essential Properties:

- Power button
- Volume control
- Channel control

Essential Behaviors:

- Turn on/off
- Increase/decrease volume
- Change channels

Abstraction in Programming Terms:

Class Definition:

```
class TVRemote {  
    // Properties  
    private boolean power; // On or Off  
    private int volume;    // Volume level  
    private int channel;   // Current channel  
  
    // Methods  
    public void turnOn() {  
        // Implementation to turn on the TV  
    }  
  
    public void turnOff() {  
        // Implementation to turn off the TV  
    }  
  
    public void increaseVolume() {  
        // Implementation to increase volume  
    }  
  
    public void decreaseVolume() {  
        // Implementation to decrease volume  
    }  
  
    public void changeChannel(int newChannel) {  
        // Implementation to change the channel  
    }  
}
```

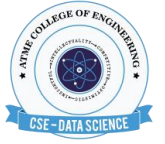
Abstraction in Action:

From a user's perspective, the TV remote is a simple device with a few buttons (abstraction).

- Users interact with the essential features: turning the TV on/off, adjusting volume, and changing channels.
- Users don't need to know the intricate details of how the remote works internally or how signals are transmitted to the TV. The complexity is abstracted away.



A T M E
College of Engineering



The Three OOP Principles

Encapsulation

Inheritance

Polymorphism



The Three OOP Principles

Encapsulation:

Definition: Encapsulation is the bundling of data (attributes) and the methods (functions or procedures) that operate on the data into a single unit known as an object.

Key Concepts:

Data Protection: Encapsulation helps protect the integrity of an object's data by preventing direct access from outside the object. Access is controlled through well-defined interfaces (public methods).

Abstraction: Encapsulation provides a level of abstraction by hiding the internal details of an object, allowing developers to focus on the essential characteristics while ignoring unnecessary details.

Modularity: Encapsulation promotes modularity by organizing the code into separate, self-contained objects, each responsible for its own state and behavior.



- Class consists of **code (methods) and data (variables)**
- Collectively, these elements (code and data) are called **members of the class.**
- Specifically, the data defined by the class are **referred to as member variables or instance variables.**
- The code that operates on that data is referred to as **member methods or just methods.**



Access Modifiers

In Java, access modifiers are keywords that determine the visibility or accessibility of classes, methods, and fields in a class. There are four access modifiers in Java:

1. Default (Package-private)
2. `public`
3. `private`
4. `protected`

Default-package private access modifier

If no access modifier is specified, it is considered default (also known as package-private).

Members (classes, methods, and fields) with default access are accessible within the same package but not outside of it.

```
class MyClass {  
    int myField; // default access  
    void myMethod() {  
        // default access  
    }  
}
```

public- access modifier

The public modifier makes a class, method, or field accessible from any other class.

There are no restrictions on accessing public members.

```
public class PublicClass {  
    public int publicField;  
    public void publicMethod() {  
        // code here  
    }  
}
```

private-access modifier

The private modifier makes a member (method or field) accessible only within the same class.

It provides the highest level of encapsulation.

```
public class MyClass {  
    private int privateField;  
    private void privateMethod() {  
        // code here  
    }  
}
```

protected-access modifier

The protected modifier allows a member to be accessible within its own package and by subclasses.

It is similar to default access but with the addition that it allows access to subclasses even if they are in a different package.

```
public class MyBaseClass {  
    protected int protectedField;  
    protected void protectedMethod() {  
        // code here  
    }  
}  
  
public class MySubClass extends MyBaseClass {  
    void anotherMethod() {  
        protectedField = 42; // Accessing protected member from the subclass  
    }  
}
```

Here's a summary:

public: Accessible from anywhere.

protected: Accessible within the same package and by subclasses, even if they are in a different package.

Default (Package-Private): Accessible within the same package.

private: Accessible only within the same class.

Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP), and Java supports it.

Inheritance allows a class (subclass or derived class) to inherit the characteristics (fields and methods) of another class (superclass or base class).

This promotes code reuse and helps in creating a hierarchical structure.

Inheritance- Example

```
public class BaseClass {  
    public String name = "Harish";  
    public int id = 123;  
}
```

```
public class Main extends BaseClass {  
    public static void main(String[] args) {  
        Main m = new Main();  
        System.out.println(m.name);  
        System.out.println(m.id);  
    }  
}
```

```
----jGRASP exec: java Main  
Harish  
123  
  
----jGRASP: operation complete.
```

Polymorphism

Polymorphism is one of the three fundamental principles of object-oriented programming (OOP), along with encapsulation, inheritance.

The word "polymorphism" is derived from Greek, where "poly" means many, and "morph" means form.

In the context of OOP, polymorphism allows objects of different types to be treated as objects of a common type.

Types of polymorphism

There are two main types of polymorphism in Java:

1. compile-time (static) polymorphism (overloading) and
2. runtime (dynamic) polymorphism (overriding)

compile-time (static) polymorphism (overloading) and

Compile-time polymorphism, also known as method overloading, occurs when there are multiple methods in the same class with the same name but different parameter lists.

The compiler determines which method to call based on the number and types of arguments provided.

Example of method overloading:



ATME

College of Engineering



ISO 9001:2015



```
public class MathOperations {  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public double add(double a, double b) {  
        return a + b;  
    }  
  
    public String add(String a, String b) {  
        return a + b;  
    }  
  
    public static void main(String[] args)  
    {  
        MathOperations mo=new MathOperations();  
  
        int res=mo.add(4,5);  
        double res1=mo.add(56.6,78.9);  
        String res2=mo.add("4","6");  
  
        System.out.println(res);  
        System.out.println(res1);  
        System.out.println(res2);  
  
    }  
}
```

```
----jGRASP exec: java MathOperations  
9  
135.5  
46  
  
----jGRASP: operation complete.
```



ATME

College of Engineering

A First Simple Program



ISO 9001:2015



```
/*  
This is a simple Java program. Call this file "Example.java".  
*/  
class Example {  
    // Your program begins with a call to main().  
    public static void main(String args[]) {  
        System.out.println("This is a simple Java program.");  
    }  
}
```

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

All Java applications begin execution by calling main().

main() must be declared as public, since it must be called by code outside of its class when the program is started.

The keyword static allows main() to be called without having to instantiate a particular instance of the class.

This is necessary since main() is called by the Java Virtual Machine before any objects are made.

The keyword void simply tells the compiler that main() does not return a value

A Second Short Program

```
/* Here is another short example. Call this file "Example2.java".*/  
class Example2 {  
    public static void main(String args []) {  
        int num; // this declares a variable called num  
        num = 100; // this assigns num the value 100  
        System.out.println("This is num: " + num);  
        num = num * 2;  
        System.out.print("The value of num * 2 is ");  
        System.out.println(num);  
    }  
}
```

The if Statement

`if(condition) statement;`

Here, condition is a Boolean expression. If condition is true, then the statement is executed. If condition is false, then the statement is bypassed. Here is an example:

```
if(num < 100) System.out.println("num is less than 100");
```

Operator	Meaning
<	Less than
>	Greater than
==	Equal to



if example

/*

Demonstrate the if.

Call this file "IfSample.java".

*/

```
class IfSample {  
    public static void main(String args[]) {  
        int x, y;  
        x = 10;  
        y = 20;
```

```
        if(x < y) System.out.println("x is less than y");  
        x = x * 2;  
        if(x == y) System.out.println("x now equal to y");  
        x = x * 2;  
        if(x > y) System.out.println("x now greater than y");  
        // this won't display anything  
        if(x == y) System.out.println("you won't see this");  
    }  
}
```

The for Loop

for (initialization; condition; iteration) statement;

In its most common form, the initialization portion of the loop sets a loop control variable to an initial value. The condition is a Boolean expression that tests the loop control variable.

If the outcome of that test is true, the for loop continues to iterate. If it is false, the loop terminates. The iteration expression determines how the loop control variable is changed each time the loop iterates.



for loop example

```
/*
```

Demonstrate the for loop.

Call this file "ForTest.java".

```
*/
```

```
class ForTest {  
    public static void main(String args[]) {  
        int x;  
        for(x = 0; x<10; x = x+1)  
            System.out.println("This is x: " + x);  
    }  
}
```



Using Blocks of Code

```
if(x < y) { // begin a block
```

```
    x = y;
```

```
    y = 0;
```

```
} // end of block
```

Block code another example

```
/*  
    Demonstrate a block of code.  
    Call this file "BlockTest.java"  
*/  
class BlockTest {  
    public static void main(String args[]) {  
        int x, y;  
        y = 20;
```

```
// the target of this loop is a block  
    for(x = 0; x<10; x++) {  
        System.out.println("This is x: " + x);  
        System.out.println("This is y: " + y);  
        y = y - 2;  
    }  
}
```



Lexical Issues

Java programs are a collection of atomic elements such as whitespace, identifiers, literals, comments, operators, separators, and keywords.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For instance, the Example program could have been written all on one line in Java, whitespace includes a space, tab, newline, or form feed.

Identifiers

Identifiers are used to name things, such as classes, variables, and methods.

An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.)

They must not begin with a number, lest they be confused with a numeric literal. Java is case-sensitive, so `VALUE` is a different identifier than `Value`. Some examples of valid identifier are

AvgTemp	count	a4	\$test	this_is_ok
---------	-------	----	--------	------------

Invalid identifier names include these:

2count	high-temp	Not/ok
--------	-----------	--------

Literals

A constant value in Java is called literal.

For example, here are some literals:

100	98.6	'X'	"This is a test"
-----	------	-----	------------------

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

single-line comment

```
// comment line
```

multiline comment

```
/* here goes the
```

```
multi line
```

```
comment */
```

The third type is called a documentation comment.

This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a `/**` and ends with a `*/`.



Example of a document comment

```
/**  
 * This class draws a bar chart.  
 * @author Herbert Schildt  
 * @version 3.2  
 */
```

Separators

Symbol	Name	Purpose
()	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference.
...	Ellipsis	Indicates a variable-arity parameter.
@	Ampersand	Begins an annotation.



The Java Keywords

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	open	opens	package
private	protected	provides	public	requires	return
short	static	strictfp	super	switch	synchronized
this	throw	throws	to	transient	transitive
try	uses	void	volatile	while	with
—					

The Java Class Libraries

Java's built-in methods used in the sample programs so far:

`println( )` and
`print( )`.

These methods are available through **System.out**. **System** is a class predefined by Java that is automatically included in your programs.

Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as **I/O, string handling, networking, and graphics.**

Data Types, Variables, and Arrays

Java Is a Strongly Typed Language

- Java is a strongly typed language, contributing to its safety and robustness.
- In Java, every variable and expression has a strictly defined type.
- All assignments, whether through explicit assignment statements or parameter passing in method calls, undergo type compatibility checks.
- Unlike some languages, Java does not allow automatic conversions of conflicting types.
- The Java compiler ensures that expressions and parameters are compatible, and any type mismatches result in errors that must be addressed for successful compilation.

The Primitive Types

Java defines eight primitive types of data:

byte, short, int, long, char, float, double, and boolean.

The primitive types are also commonly referred to as **simple types**.

These are grouped into 4 groups:

- **Integers** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers** This group includes float and double, which represent numbers with fractional precision.
- **Characters** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean** This group includes boolean, which is a special type for representing true/false values.

Integers

Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values.

Java does not support unsigned values. Java's designers felt that unsigned integers were unnecessary.

Name	Width	Range
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	-2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32,767
byte	8	-128 to 127

byte

- The smallest integer type is byte.
- This is a signed 8-bit type that has a range from **-128 to 127**.
- Variables of type byte are especially useful when you're working with a stream of data from a network or file.
- Byte variables are declared by use of the **byte keyword**.
- For example, the following declares two byte variables called b and c:
 - **byte b, c;**

short

short is a signed **16-bit type**.

It has a range from **-32,768 to 32,767**.

It is probably the least-used Java type.

Here are some examples of short variable declarations:

```
short s;
```

```
short t;
```

int

The **most commonly used** integer type is `int`.

It is a signed 32-bit type that has a range from **−2,147,483,648 to 2,147,483,647**. Variables of type `int` are commonly employed to control **loops** and to **index arrays**.

Although you might think that using a byte or short would be more efficient than using an `int` in situations in which the larger range of an `int` is not needed, this may not be the case.

The reason is that when byte and short values are used in an expression, they are promoted to `int` when the expression is evaluated. (Type promotion is described later in this chapter.)

Therefore, **`int` is often the best choice** when an integer is needed.

long

long is a **signed 64-bit type** and is useful for those occasions where an int type is not large enough to hold the desired value.

The range of a long is quite large. This makes it useful **when big, whole numbers are needed.**

long

For example, here is a program that computes the number of miles that light will travel in a specified number of days:

```
// Compute distance light travels using long variables.
class Light {
    public static void main(String args[]) {
        int lightspeed;
        long days;
        long seconds;
        long distance;

        // approximate speed of light in miles per second
        lightspeed = 186000;

        days = 1000; // specify number of days here

        seconds = days * 24 * 60 * 60; // convert to seconds

        distance = lightspeed * seconds; // compute distance

        System.out.print("In " + days);
        System.out.print(" days light will travel about ");
        System.out.println(distance + " miles.");
    }
}
```

This program generates the following output:

In 1000 days light will travel about 160704000000000 miles.

Floating-Point Types

Also known as **real numbers**.

Used when evaluating expressions that require **fractional precision**.

For example, calculations such as **square root, sine and cosine**.

Java implements the standard (**IEEE– 754**) set of floating-point types and operators.

There are two kinds of floatingpoint types,

float (single-precision) and

double (double-precision),

Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

float

The type float specifies a **single-precision** value that uses 32 bits of storage.

Single precision is faster on some processors and **takes half as much space as double precision**, but will become **imprecise when the values are either very large or very small**.

Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.

For example, float can be useful when representing dollars and cents. Here are some example float variable declarations:

```
float hightemp, lowtemp;
```

double

Double precision, as denoted by the double keyword, **uses 64 bits to store a value.**

Double precision is **actually faster than single precision on some modern processors** that have been optimized for high-speed mathematical calculations.

All math functions, such as `sin( )`, `cos( )`, and `sqrt( )`, return **double values.**

When you need to **maintain accuracy** over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

double-example

Here is a short program that uses double variables to compute the area of a circle

```
// Compute the area of a circle.
class Area {
    public static void main(String args[]) {
        double pi, r, a;

        r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately
        a = pi * r * r; // compute area

        System.out.println("Area of circle is " + a);
    }
}
```

Characters

In Java, the data type used to store characters is `char`.

`char` in Java is not the same as `char` in C or C++.

In C/C++, `char` is 8 bits wide.

Java uses **Unicode to represent characters**.

Unicode defines a fully international character set that can represent all of the characters found in all human languages.

It is a unification of dozens of character sets, such as **Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul**, and many more.

At the time of Java's creation, Unicode required 16 bits. Thus, in Java **`char` is a 16-bit type**.



CharDemo.java

/ Demonstrate char data type.

```
class CharDemo {  
    public static void main(String args[]) {  
        char ch1, ch2;  
        ch1 = 88; // code for X  
        ch2 = 'Y';  
  
        System.out.print("ch1 and ch2: ");  
        System.out.println(ch1 + " " + ch2);  
    }  
}
```

----jGRASP exec: java CharDemo
ch1 and ch2: X Y

----jGRASP: operation complete.



CharDemo2.java

```
// char variables behave like integers.  
class CharDemo2 {  
    public static void main(String args[]) {  
        char ch1;  
        ch1 = 'X';  
        System.out.println("ch1 contains " + ch1);  
        ch1++; // increment ch1  
        System.out.println("ch1 is now " + ch1);  
    }  
}
```

```
----jGRASP exec: java CharDemo2  
ch1 contains X  
ch1 is now Y  
  
----jGRASP: operation complete.
```

Booleans

Java has a primitive type, called boolean, for logical values.
It can have only one of two possible values, true or false.

BoolTest.java

/ Demonstrate boolean values.

```
class BoolTest {  
    public static void main(String args[]) {  
        boolean b;  
        b = false;  
        System.out.println("b is " + b);  
        b = true;  
        System.out.println("b is " + b);  
    }  
}
```

// a boolean value can control the if statement

if(b)

System.out.println("This is executed.");

b = false;

if(b)

System.out.println("This is not executed.");

// outcome of a relational operator is a boolean value

System.out.println("10 > 9 is " + (10 > 9));

}

}

```
----jGRASP exec: java BoolTest
b is false
b is true
This is executed.
10 > 9 is true

----jGRASP: operation complete.
```

Integer Literals

Any whole number value is an integer literal.

Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number.

Two other bases that can be used in integer literals are **octal (base eight) and hexadecimal (base 16)**.

Octal values are denoted in Java by a leading zero.

Normal decimal numbers cannot have a leading zero.

octal's have range from 0 to 7.

A more common base for numbers used by programmers is hexadecimal.

Hexadecimal constant are signified with a leading zero-x, (0x or 0X).

The range of a hexadecimal digit is 0 to 15, so A through F (or a through f) are substituted for 10 through 15.

Starting with JDK 7, you can also specify integer literals using binary.

To do so, prefix the value with 0b or 0B.

For example, this specifies the decimal value 10 using a binary literal:

```
int x = 0b1010;
```

starting with JDK 7, you can embed one or more underscores in an integer literal.

Doing so makes it easier to read large integer literals.

When the literal is compiled, the underscores are discarded.

```
For example, given int x = 123_456_789;
```

the value given to x will be 123,456,789.

Underscores can only be used to separate digits.

They cannot come at the beginning or the end of a literal.

It is, permissible to have more than one underscore between two digits.

For example, this is valid: `int x = 123___456___789;`

For example, binary values are often visually grouped in four-digits units, as shown here:

`int x = 0b1101_0101_0001_1010;`

Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component.

They can be expressed in two ways:

1. Standard Notation
2. Scientific Notation

Standard Notation:

Standard notation consists of a **whole number component** followed by a **decimal point** followed by a **fractional component**.

For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers.



A T M E

College of Engineering

Boolean Literals



Boolean literals are simple.

There are only two logical values that a boolean value can have, **true** and **false**.

The values of true and false do not convert into any numerical representation.

The true literal in Java does not equal 1, nor does the false literal equal 0.

In Java, the **Boolean** literals can only be assigned to variables declared as **boolean** or used in expressions with **Boolean** operators.

Character Literals

Java uses Unicode character set. (2 bytes per character / 16 bits)

A literal character is represented inside a pair of single quotes.

such as 'a', 'z', and '@'.

Character literals can be prefixed with \ (escape sequence)

\n → new line

There is also a mechanism for directly entering the value of a character in octal or hexadecimal.

For octal notation, use the backslash followed by the three-digit number. For example, '\141' is the letter 'a'. [octal value of 97 is 141]

For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits.

For example, '\u0061' is the 'a'

String Literals

String literals in Java are specified —by enclosing a sequence of characters between a pair of double quotes.

Examples of string literals are

"Hello World"

"two\nlines"

"\"This is in quotes\""

this prints along with the double quotes as "This is in quotes"

Character Escape Sequences

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Table 3-1 Character Escape Sequences

Variables

variable is the basic unit of storage in a Java program.

A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable

type identifier [= value][, identifier [= value] ...];

Examples:

```
int a, b, c; // declares three ints, a, b, and c.
```

```
int d = 3, e, f = 5; // declares three more ints, initializing // d and f.
```

```
byte z = 22; // initializes z.
```

```
double pi = 3.14159; // declares an approximation of pi.
```

```
char x = 'x'; // the variable x has the value 'x'.
```

Dynamic Initialization

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;
        // c is dynamically initialized
        double c = Math.sqrt(a * a + b * b);
        System.out.println("Hypotenuse is " + c);
    }
}
```



The Scope and Lifetime of Variables

// Demonstrate block scope.

```
class Scope {  
    public static void main(String args[]) {  
        int x; // known to all code within main  
        x = 10;  
        if(x == 10) { // start new scope  
            int y = 20; // known only to this block  
            // x and y both known here.  
            System.out.println("x and y: " + x + " " + y);  
            x = y * 2;  
        }  
        // y = 100; // Error! y not known here  
        // x is still known here.  
        System.out.println("x is " + x); } }
```

Variable Lifetime Example

// Demonstrate lifetime of a variable.

```
class LifeTime {  
    public static void main(String args[]) {  
        int x;  
        for(x = 0; x < 3; x++) {  
            int y = -1; // y is initialized each time block is entered  
            System.out.println("y is: " + y); // this always prints -1  
            y = 100;  
            System.out.println("y is now: " + y);  
        }  
    }  
}
```

The output generated by this program is shown here:

y is: -1

y is now: 100

y is: -1

y is now: 100

y is: -1

y is now: 100



Example on scope error

```
// This program will not compile
class ScopeErr {
    public static void main(String args[]) {
        int bar = 1;
        { // creates a new scope
            int bar = 2; // Compile-time error – bar already defined!
        }
    }
}
```

Type Conversion and Casting

Introduction

Type conversion and casting are fundamental concepts in Java programming that involve altering the data type of a value.

Understanding these concepts is crucial for writing efficient and error-free code.

Types of Type Conversion

Widening Conversion:

Automatically converts values from smaller to larger data types.

Considered safe as no data loss occurs.

Example: Converting an int value to a long value.

Narrowing Conversion:

Manually converts values from larger to smaller data types.

Potentially risky due to possible data loss.

Example: Converting a double value to an int value.

Widening Conversion (Implicit Conversion)

- Occurs automatically when assigning a smaller data type value to a larger data type variable.
- The compiler implicitly promotes the value to match the larger data type.
- Example:

```
int x = 10;
```

```
long y = x; // Implicit widening conversion
```

Narrowing Conversion (Explicit Casting)

- Requires explicit casting using the (type) operator to convert a larger data type value to a smaller data type.
- The programmer must ensure that the conversion is safe and no data loss occurs.
- Example:

```
double x = 10.5;
```

```
int y = (int) x;    // Explicit narrowing casting
```

Casting Techniques

- **Widening Casting (Upcasting):**

- Consists of casting a value from a smaller to a larger data type using the `()` operator.
- Safe operation as no data loss is involved.
- Example:

```
int x = 10;
```

```
long y = (long) x; // Widening casting using `()`
```

Narrowing Casting (Downcasting):

Involves casting a value from a larger to a smaller data type using the (type) operator.

- Requires careful handling as data loss may occur.
- Example:

Java

```
double x = 10.5;
```

```
int y = (int) x; // Narrowing casting using `(type)`
```

Example on conversion

// Demonstrate casts.

```
class Conversion {  
    public static void main(String args[]) {  
        byte b;  
        int i = 257;  
        double d = 323.142;  
        System.out.println("\nConversion of int to  
byte.");  
        b = (byte) i;  
        System.out.println("i and b " + i + " " +  
b);  
        System.out.println("\nConversion of  
double to int.");
```

```
        i = (int) d;  
        System.out.println("d and i " + d + " " + i);  
        System.out.println("\nConversion of double to  
byte.");  
        b = (byte) d;  
        System.out.println("d and b " + d + " " + b);  
    }  
}
```

This program generates the following output:

Conversion of int to byte.

i and b 257 1

Conversion of double to int.

d and i 323.142 323

Conversion of double to byte.

d and b 323.142 67

Explanation

```
int i = 257;
```

```
byte b = (byte) i;
```

```
System.out.println("i and b " + i + " " + b);
```

In Java, the byte data type can only represent values in the range of -128 to 127 (8 bits, signed two's complement). When you try to cast the int value 257 to a byte, it overflows because 257 is beyond the valid range of a byte.

The overflow wraps around, so 257 becomes equivalent to $257 - 256$, which results in 1. Therefore, when you print the value of b, it will be 1. This is an example of overflow and wrap-around behavior in Java when casting values to a smaller data type.



ATME

College of Engineering

Explanation



```
double d = 323.142;  
byte b = (byte) d;  
System.out.println("d and b " + d + " " + b);
```

In this specific example, the double value 323.142 will be truncated to 323 when cast to a byte. However, the value 323 is outside the valid range of a byte (-128 to 127), so it will overflow.

The overflow behavior in Java is such that the value wraps around within the valid range of the target type. In this case, 323 wraps around and becomes equivalent to $323 - 256$, resulting in 67. Therefore, when you print the value of b, it will be 67.

Automatic Type Promotion in Expressions

```
byte a = 40;
```

```
byte b = 50;
```

```
byte c = 100;
```

```
int d = a * b / c;
```

The result of the intermediate term $a * b$ easily exceeds the range of either of its byte operands.

Java automatically promotes each byte operand to int when evaluating an expression.

This means that the subexpression $a*b$ is performed using integers—not bytes.

Thus, 2,000, the result of the intermediate expression, $50 * 40$, is legal even though a and b are both specified as type byte.

Automatic promotions can cause confusing compile-time errors.

For example, this seemingly correct code causes a problem

```
byte b = 50;
```

```
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store $50 * 2$, a perfectly valid byte value, back into a byte variable.

However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int.

Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.

In cases where you understand the consequences of overflow, you should use an explicit cast,

such as

```
byte b = 50;  
b = (byte)(b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions.

They are as follows:

1. All byte, short, and char values are promoted to int, as just described.
2. If one operand is a long, the whole expression is promoted to long.
3. If one operand is a float, the entire expression is promoted to float.
4. If any of the operands are double, the result is double.

Example on Type Promote

```
class Promote {  
    public static void main(String args[]) {  
        byte b = 42;  
        char c = 'a';  
        short s = 1024;  
        int i = 50000;  
        float f = 5.67f;  
        double d = .1234;  
        double result = (f * b) + (i / c) - (d * s);  
        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));  
        System.out.println("result = " + result);  
    }  
}
```

---jGRASP exec: java Promote
238.14 + 515 - 126.3616
result = 626.7784146484375

---jGRASP: operation complete.

Arrays

Collection of elements of similar type under one name.

One-Dimensional Arrays:

Declaration

```
type var-name[ ];
```

Example: `int month_days[];`

But this will not allocate memory. To allocate the memory you have to use **new** keyword.

The general form of new as it applies to one-dimensional arrays appears as follows:

```
array-var = new type [size];
```

Ex:- `month_days = new int[12];`

Example on 1D-arrays

// Demonstrate a one-dimensional array.

```
class Array {  
    public static void main(String args[]) {  
        int month_days[];  
        month_days = new int[12];  
        month_days[0] = 31;  
        month_days[1] = 28;  
        month_days[2] = 31;  
        month_days[3] = 30;  
        month_days[4] = 31;  
        month_days[5] = 30;
```

```
month_days[6] = 31;  
month_days[7] = 31;  
month_days[8] = 30;  
month_days[9] = 31;  
month_days[10] = 30;  
month_days[11] = 31;  
System.out.println("April has " + month_days[3] + " days.");  
}  
}
```

**----jGRASP exec: java Array
April has 30 days.**

----jGRASP: operation complete.



An improved version of the previous program. [No need of new keyword]

// An improved version of the previous program.

```
class AutoArray {  
    public static void main(String args[]) {  
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,  
                               30, 31 };  
        System.out.println("April has " + month_days[3] + " days.");  
    }  
}
```

**----jGRASP exec: java AutoArray
April has 30 days.**

----jGRASP: operation complete.

Average example using arrays

// Average an array of values.

```
class Average {  
    public static void main(String args[]) {  
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};  
        double result = 0;  
        int i;  
        for(i=0; i<5; i++)  
            result = result + nums[i];  
        System.out.println("Average is " + result / 5);  
    }  
}
```

----jGRASP exec: java Average
Average is 12.299999999999999

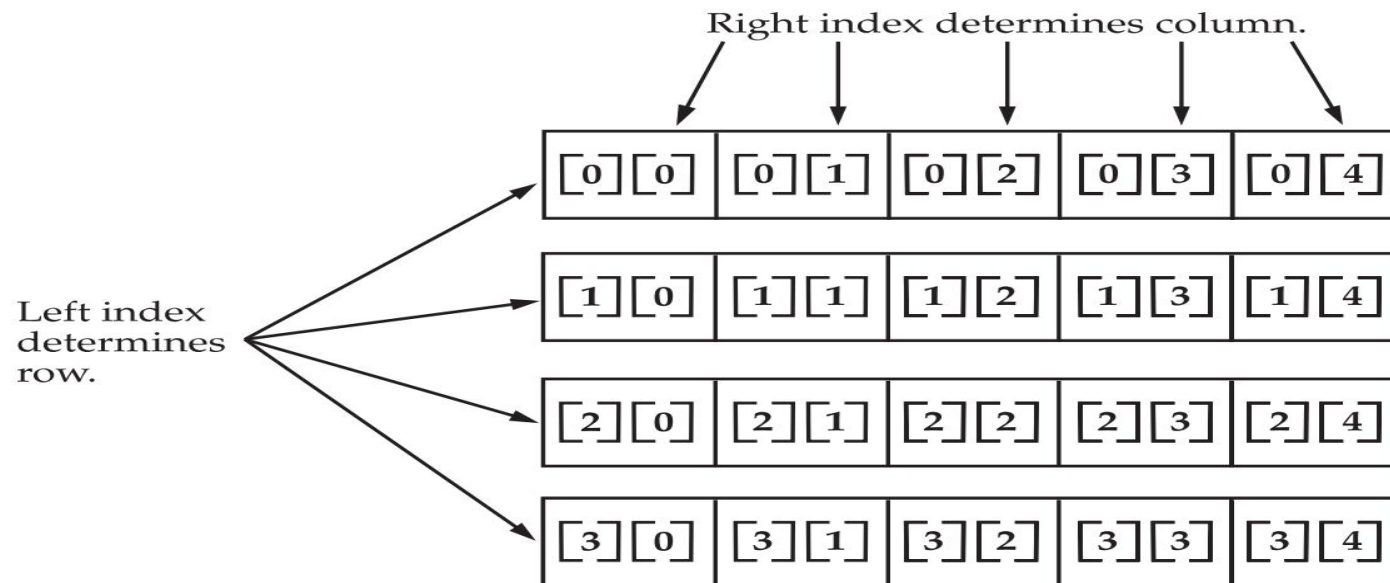
----jGRASP: operation complete.

Multidimensional Arrays

The following declares a two dimensional array variable called twoD: **int twoD[][]**
= new int[4][5];

This allocates a 4 by 5 array and assigns it to twoD.

Internally, this matrix is implemented as an array of arrays of int.
Conceptually, this array will look like the one shown



Given: `int twoD [ ] [ ] = new int [4] [5] ;`

Figure 3-1 A conceptual view of a 4 by 5, two-dimensional array

Example on TwoDArray

```
// Demonstrate a two-dimensional array.
class TwoDArray {
    public static void main(String args[]) {
        int twoD[][]= new int[4][5];
        int i, j, k = 0;
        for(i=0; i<4; i++)
            for(j=0; j<5; j++) {
                twoD[i][j] = k;
                k++;
            }
    }
}
```

```
for(i=0; i<4; i++) {
    for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
    System.out.println();
}
```

----jGRASP exec: java TwoDArray

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

----jGRASP: operation complete.

```
int twoD[][] = new int[4][];  
twoD[0] = new int[5];  
twoD[1] = new int[5];  
twoD[2] = new int[5];  
twoD[3] = new int[5];
```

the above code allocates memory for the first dimension of twoD when it is declared.
It allocates the second dimension manually.

Program creates a two-dimensional array in which the sizes of the second dimension are unequal:

```
// Manually allocate differing size second dimensions.
```

```
class TwoDAgain {  
    public static void main(String args[]) {  
        int twoD[][] = new int[4][];  
        twoD[0] = new int[1];  
        twoD[1] = new int[2];  
        twoD[2] = new int[3];  
        twoD[3] = new int[4];  
        int i, j, k = 0;  
        for(i=0; i<4; i++)  
            for(j=0; j<i+1; j++) {  
                twoD[i][j] = k;  
                k++;  
            }  
    }  
}
```

```
for(i=0; i<4; i++) {  
    for(j=0; j<i+1; j++)  
        System.out.print(twoD[i][j] +  
            " ");  
    System.out.println();  
}  
}
```

The array created by this program looks like this:

0	0		
1	0	1	1
2	0	2	1
3	0	3	2

This program generates the following output:

```
0  
1 2  
3 4 5  
6 7 8 9
```

// Initialize a two-dimensional array.

```
class Matrix {  
    public static void main(String args[]) {  
        double m[][] = {  
            { 0*0, 1*0, 2*0, 3*0 },  
            { 0*1, 1*1, 2*1, 3*1 },  
            { 0*2, 1*2, 2*2, 3*2 },  
            { 0*3, 1*3, 2*3, 3*3 }  
        };  
        int i, j;  
        for(i=0; i<4; i++) {  
            for(j=0; j<4; j++)  
                System.out.print(m[i][j] + " ");  
            System.out.println();  
        }  
    }  
}
```

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

```
0.0 0.0 0.0 0.0  
0.0 1.0 2.0 3.0  
0.0 2.0 4.0 6.0  
0.0 3.0 6.0 9.0
```

Direct Initialization of 2D-Array

3D-array Example on creating a 3 X 4 X 5, three-dimensional array.

// Demonstrate a three-dimensional array.

```
class ThreeDMatrix {  
    public static void main(String args[]) {  
        int threeD[][][] = new int[3][4][5];  
        int i, j, k;  
        for(i=0; i<3; i++)  
            for(j=0; j<4; j++)  
                for(k=0; k<5; k++)  
                    threeD[i][j][k] = i * j * k;  
        for(i=0; i<3; i++) {  
            for(j=0; j<4; j++) {  
                for(k=0; k<5; k++)  
                    System.out.print(threeD[i][j][k]  
                    + " ");  
                System.out.println();  
            }  
            System.out.println();  
        }  
    }  
}
```

This program generates
the following output:

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 1 2 3 4

0 2 4 6 8

0 3 6 9 12

0 0 0 0 0

0 2 4 6 8

0 4 8 12 16

0 6 12 18 24

Alternative Array Declaration Syntax

`type[ ] var-name;`

1D-Declaration

`int a1[] = new int[3]; // both are same`

`int[] a2 = new int[3];`

2D-declaration

`char twod1[][] = new char[3][4];`

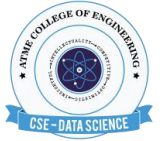
`char[][] twod2 = new char[3][4];`

`int[] nums, nums2, nums3; // create three arrays`

`int nums[], nums2[], nums3[]; // create three arrays`



A T M E
College of Engineering



Operators and Control statements

- ❖ Operators: Arithmetic Operators,
- ❖ The Bitwise Operators
- ❖ Relational Operators
- ❖ Boolean Logical Operators
- ❖ The Assignment Operator
- ❖ The ? Operator
- ❖ Operator Precedence
- ❖ Using Parentheses
- ❖ Control Statements: Java's Selection Statements
- ❖ Iteration Statements
- ❖ Jump Statements.

Operators

Arithmetic Operators

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



ATME

College of Engineering



The Basic Arithmetic Operators

```
// 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);
    }
}
```

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.

// 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);  
    }  
}
```

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

x mod 10 = 2
y mod 10 = 2.25

Arithmetic Compound Assignment Operators

`a = a + 4;`

In Java, you can rewrite this statement as shown here:

`a += 4;`

This version uses the += compound assignment operator.

`var = var op expression;`

can be rewritten as

`var op= expression;`

Increment and Decrement

- The ++ and the -- are Java's increment and decrement operators.
- The increment operator increases its operand by one. The decrement operator decreases its operand by one.
- For example, this statement:

$x = x + 1;$

can be rewritten like this by use of the increment operator:

$x++; \quad ++x;$

Similarly, this statement:

$x = x - 1;$

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 is modified.

- For example: $x = 42;$ $y = ++x;$
- In this case, y is set to 43 as you would expect, because the increment occurs *before* x is *assigned* to y .
- Thus, the line $y = ++x;$ is the equivalent of these two statements:
 - ❖ $x = x + 1;$
 - ❖ $y = x;$
- However, when written like this,
 - $x = 42;$
 - $y = x++;$the value of x is obtained before the increment operator is executed, so the value of y is 42.
- Of course, in both cases x is set to 43. Here, the line $y = x++;$ is the equivalent of these two statements:
 - $y = x;$
 - $x = x + 1;$

// Demonstrate ++.

```
class IncDec {  
    public static void main(String args[]) {  
        int a = 1;  
        int b = 2;  
        int c;  
        int d;  
        c = ++b;  
        d = a++;  
        c++;  
        int e=d--;  
        int f=--c;  
        int G=c++ + --f;  
    }  
}
```

```
System.out.println("a = " + a); //a=2  
System.out.println("b = " + b); //b=3  
System.out.println("c = " + c); //c=4  
System.out.println("d = " + d); //d=0  
System.out.println("e="+e); //e=1  
System.out.println("f="+f); //f=2  
System.out.println("G="+G); //G=5  
}  
}
```

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
<<	Shift left
&=	Bitwise AND assignment
=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

- The bitwise logical operators are **&**, **|**, **^**, and **~**.
- The 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

// 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; // 0 + 2 + 1 or 0011 in binary  
        int b = 6; // 4 + 2 + 0 or 0110 in binary  
        int c = a | b;  
        int d = a & b;  
        int e = a ^ b;  
        int f = (~a & b) | (a & ~b);  
        int i = 0X0f;  
    }  
}
```

```
int g = ~a & 0x0f;  
System.out.println(" a = " + binary[a]); //binary[3]=0011  
System.out.println(" b = " + binary[b]); //binary[6]=0110  
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]);  
System.out.println(" i = " + binary[i]);  
  
}  
}
```

The Left Shift

- The left shift operator, **<<**, shifts all of the bits in a value to the left a specified number of times.
- It has this general form:
syntax: **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 high-order bit is shifted out (and lost), and a zero is brought in on the right.
- This means that when a left shift is applied to an **int operand**, **bits are lost**
- once they are shifted past bit position 31. If the operand is a **long**, then **bits are lost after bit position 63**.



// 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);  
    }  
}
```

The Right Shift

- The right shift operator, `>>`, shifts all of the bits in a value to the right a specified number of times.
- Its general form is shown here:

Syntax: value >> num

- Here, *num* specifies the number of positions to right-shift the value in *value*.
- *The >>* moves all of the bits in the specified value to the right the number of bit positions specified by *num*.
- The following code fragment shifts the value 32 to the right by two positions, resulting in **a** being set to 8:

```
int a = 32;  
a = a >> 2;
```

Right Shift Example

// Left shifting a byte value.

```
class ByteShiftRight {  
    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);  
    }  
}
```

program demonstrates the bitwise operators

```
public class Test {  
    public static void main(String args[]) {  
        int a = 60; /* 60 = 0011 1100 */  
        int b = 13; /* 13 = 0000 1101 */  
        int c = 0; c = a & b; /* 12 = 0000 1100 */  
        c = a << 2; /* 240 = 1111 0000 */  
        System.out.println("a << 2 = " + c );  
        c = a >> 2; /* 15 = 1111 */  
        System.out.println("a >> 2 = " + c );  
        c = a >>> 2; /* 15 = 0000 1111 */  
        System.out.println("a >>> 2 = " + c );  
    }  
}
```

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.
- For example, the following two statements, which shift the value in **a** right by four bits, are equivalent:
 - $a = a \gg 4;$
 - $a \gg= 4;$
- Likewise, the following two statements, which result in a being assigned the bitwise expression a OR b, are equivalent:
 - $a = a | b;$
 - $a |= b;$

```
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);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
    }  
}
```

- program creates a few integer variables and then uses compound bitwise operator assignments to manipulate the variables:

THE RELATIONAL OPERATORS

- The *relational operators determine the relationship that one operand has to the other.*
- Specifically, they determine equality and ordering. The relational operators are shown here:
The outcome of these operations is a **boolean value**. **The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.**

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

```
public class Test {  
    public static void main(String args[])  
    {  
        int a = 10;  
        int b = 20;  
        System.out.println("a == b = " + (a == b) );  
        System.out.println("a != b = " + (a != b) );  
        System.out.println("a > b = " + (a > b) );  
        System.out.println("a < b = " + (a < b) );  
        System.out.println("b >= a = " + (b >= a) );  
        System.out.println("b <= a = " + (b <= a) );  
    }  
}
```

Boolean Logical Operators

Operator	Result
&	Logical AND
	Logical OR
^	Logical XOR (exclusive OR)
	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

Example

// 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);  
}  
}
```

```
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, =.

The assignment operator works in Java much as it does in any other computer language. It has this general form:

var = expression;

Here, the type of var must be compatible with the type of expression.

It allows you to create a chain of assignments.

For example, consider this fragment:

```
int x, y, z;
```

```
x = y = z = 100; // set x, y, and z to 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 ?.

The ? has this general form:

`expression1 ? expression2 : expression3;`

`ratio = (denom == 0) ? 0 : num / denom;`

Example

```
class Ternary {  
    public static void main(String args[]) {  
        int i, k;  
        i = 10;  
        k = i < 0 ? -i : i; // get absolute value of i  
        System.out.print("Absolute value of ");  
        System.out.println(i + " is " + k);  
        i = -10;  
        k = i < 0 ? -i : i; // get absolute value of i  
        System.out.print("Absolute value of ");  
        System.out.println(i + " is " + k);  
    }  
}
```

The output generated by the program is shown here:

Absolute value of 10 is 10

Absolute value of -10 is 10

Table shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence.

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	- (unary)	(type-cast)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
->						
=	op=					
Lowest						

Examples

For example, consider the following expression:

$$a \gg b + 3$$

This expression first adds 3 to b and then shifts a right by that result. That is, this expression can be rewritten using redundant parentheses like this:

$$a \gg (b + 3)$$

However, if you want to first shift a right by b positions and then add 3 to that result, you will need to parenthesize the expression like this:

$$(a \gg b) + 3$$

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression.

For anyone reading your code, a complicated expression can be difficult to understand.

For example, which of the following expressions is easier to read?

`a | 4 + c >> b & 7`

`(a | (((4 + c) >> b) & 7))` // This is easier to read and gives clarity



Control Statements

Outline:

Java's Selection Statements	
if	
switch	
Iteration Statements.....	
while	
do-while	
for	
The For-Each Version of the for Loop .	
Nested Loops	
Jump Statements	
Using break	
Using continue.....	

if Statement

It can be used to route program execution through two different paths.
Here is the general form of the if statement:

if (condition) statement1;
else statement2;

```
int a, b;  
//...  
if(a < b) a = 0;  
else b = 0;
```

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; // this if is  
    else a = c;        // associated with this else  
}  
else a = d;           // this else refers to if(i == 10)
```



The if-else-if Ladder

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

```
// Demonstrate if-else-if statements.  
class IfElse {  
    public static void main(String args[]) {  
        int month = 4; // April  
        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 + ".");  
    }  
}
```

Here is the output produced by the program:

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.

A better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {  
    case value1:  
        // statement sequence  
        break;  
    case value2:  
        // statement sequence  
        break;  
    .  
    .  
    .  
    case valueN:  
        // statement sequence  
        break;  
    default:  
        // default statement sequence  
}
```

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

The output produced by this program is shown here:

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

```
// 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");
            }
    }
}
```

This program generates the following 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
```



```
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 = "Autumn";  
            break;  
        default:  
            season = "Bogus Month";  
        }  
        System.out.println("April  
is in the " + season + ".");  
    }  
}
```



Using Strings to control Switch

beginning with JDK 7, you can use a string to control a switch statement.

```
class StringSwitch {  
    public static void main(String args[]) {  
  
        String str = "two";  
  
        switch(str) {  
            case "one":  
                System.out.println("one");  
                break;  
            case "two":  
                System.out.println("two");  
                break;  
            case "three":  
                System.out.println("three");  
                break;  
            default:  
                System.out.println("no match");  
                break;  
        }  
    }  
}
```

Nested switch Statements

Switch inside another switch is nested switch statement.

```
switch(count) {  
    case 1:  
        switch(target) { // nested switch  
            case 0:  
                System.out.println("target is zero");  
                break;  
            case 1: // no conflicts with outer switch  
                System.out.println("target is one");  
                break;  
        }  
        break;  
    case 2: // ...  
}
```

Three important features of the switch statement to note:

- The switch differs from the if stmt, switch can only test for equality, whereas if stmt can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
- No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.
- A switch statement is usually more efficient than a set of nested if stmts

Iteration Statements

while loop: (Entry Controlled)

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

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

When you run this program, it will “tick” ten times:

```
tick 10  
tick 9  
tick 8  
tick 7  
tick 6  
tick 5  
tick 4  
tick 3  
tick 2  
tick 1
```

Body of the while loop can be empty

```
// 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); // no body in this loop

        System.out.println("Midpoint is " + i);
    }
}
```

This program finds the midpoint between **i** and **j**. It generates the following output:

```
Midpoint is 150
```



do-while loop

//Exit Controlled

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

Equivalent and Efficient Code

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

```
// 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);  
    }  
}
```

Output:

10
9
8
7
6
5
4
3
2
1

for loop

Before JDK5 there was only one for loop (traditional for loop).

Starting from JDK5 there is one more for loop added (for-each version).

Here is the general form of the traditional for statement:

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

Example on traditional for

// 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); } }
```



Declaring Loop Control Variables Inside the for Loop

```
// 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);  
    }  
}
```



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

        num = 14;

        if(num < 2) isPrime = false;
        else isPrime = true;

        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");
    }
}
```

FindPrime

example using
traditional for

Using the Comma in for loop

```
// 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);  
        }  
    }  
}
```



Some for Loop Variations

```
// 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++;
        }
    }
}
```

The For-Each Version of the for Loop

Languages, such as C#, implements 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 preexisting code is broken.

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

The general form of the for-each version of the for is shown here:

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

Using tradition for loop

```
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];
```

Using for-each version

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



ATME

College of Engineering

Example



```
// 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;

        // use for-each style for to display and sum the values
        for(int x : nums) {
            System.out.println("Value is: " + x);
            sum += x;
        }

        System.out.println("Summation: " + sum);
    }
}
```

The output from the program is shown here:

```
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
```

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 };  
        // use for to display and sum the values  
        for(int x : nums) {  
            System.out.println("Value is: " + x);  
            sum += x;  
            if(x == 5) break; // stop the loop when 5 is obtained  
        }  
        System.out.println("Summation of first 5 elements: " + sum);  
    }  
}
```

This is the output produced:

```
Value is: 1  
Value is: 2  
Value is: 3  
Value is: 4  
Value is: 5  
Summation of first 5 elements: 15
```

```
// The for-each loop is essentially read-only.
class NoChange {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        for(int x: nums) {
            System.out.print(x + " ");
            x = x * 10; // no effect on nums
        }

        System.out.println();

        for(int x : nums)
            System.out.print(x + " ");

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

The for-each loop is essentially read-only

The output, shown here, proves this point:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

```
// 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);
    }
}
```

Use for-each style for on a two-dimensional array.

The output from this program is shown here:

Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90

```
// Search an array using for-each style for.
class Search {
    public static void main(String args[]) {
        int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
        int val = 5;
        boolean found = false;

        // use for-each style for to search nums for val
        for(int x : nums) {
            if(x == val) {
                found = true;
                break;
            }
        }

        if(found)
            System.out.println("Value found!");
    }
}
```

**Search an array
using for-each
style for.**



ATME

College of Engineering

Jump Statements



ISO 9001:2015



Java supports three jump statements:

break,
continue, and
return.

These statements transfer control to another part of your program.

Uses of break in java

The break statement has three uses.

- First, as you have seen, it terminates a statement sequence in a switch statement.
- Second, it can be used to exit a loop.
- Third, it can be used as a “civilized” form of goto.

Using break to Exit a Loop

```
// Using break to exit a loop.
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.");
    }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```



Using break to exit a while loop.

```
// Using break to exit a while loop.
class BreakLoop2 {
    public static void main(String args[]) {
        int i = 0;

        while(i < 100) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
            i++;
        }
        System.out.println("Loop complete.");
    }
}
```

```
// Using break with nested loops.
```

```
class BreakLoop3 {  
    public static void main(String args[]) {  
        for(int i=0; i<3; i++) {  
            System.out.print("Pass " + i + ": ");  
            for(int j=0; j<100; j++) {  
                if(j == 10) break; // terminate loop if j is 10  
                System.out.print(j + " ");  
            }  
            System.out.println();  
        }  
        System.out.println("Loops complete.");  
    }  
}
```

Using break with nested loops.

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
```

```
Pass 1: 0 1 2 3 4 5 6 7 8 9
```

```
Pass 2: 0 1 2 3 4 5 6 7 8 9
```

```
Loops complete.
```

Using break as a Form of Goto

label general form:

```
label: {  
  
}
```

using break to jump out of the label general form

```
break label;
```



```
// Using break as a civilized form of goto.
```

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

Running this program generates the following output:

Before the break.

This is after second block.

Example

The break statement causes execution to jump forward, past the end of the block labeled second, skipping the two println() statements.



breaking out of the nest loops using label and break

// Using break to exit from nested loops

```
class BreakLoop4 {  
    public static void main(String args[]) {  
        outer: for(int i=0; i<3; i++) {  
            System.out.print("Pass " + i + ": ");  
            for(int j=0; j<100; j++) {  
                if(j == 10) break outer; // exit both loops  
                System.out.print(j + " ");  
            }  
            System.out.println("This will not print");  
        }  
        System.out.println("Loops complete.");  
    }  
}
```

This program generates the following output:

Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

Example on error

Cannot break to any label which is not defined for an enclosing block.

```
// This program contains an error.
class BreakErr {
    public static void main(String args[]) {

        one: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
        }

        for(int j=0; j<100; j++) {
            if(j == 10) break one; // WRONG
            System.out.print(j + " ");
        }
    }
}
```

Program is invalid and will not compile, Since the loop labeled one does not enclose the break statement, it is not possible to transfer control out of that block.

Using continue

Continue is used to skip the current loop iteration and get into the next loop iteration.

Example:

// Demonstrate continue.

```
class Continue {  
    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("");  
        }  
    }  
}
```

0	1
2	3
4	5
6	7
8	9

continue may specify a label to describe which enclosing loop to continue.

Example program that uses continue to print a triangular multiplication table for 0 through 9:

```
// Using continue with a label.
class ContinueLabel {
    public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
        for(int j=0; j<10; j++) {
            if(j > i) {
                System.out.println();
                continue outer;
            }
            System.out.print(" " + (i * j));
        }
        System.out.println();
    }
}
```

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```



return stmt

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.

```
// Demonstrate return.
class Return {
    public static void main(String args[]) {
        boolean t = true;

        System.out.println("Before the return.");

        if(t) return; // return to caller

        System.out.println("This won't execute.");
    }
}
```

The output from this program is shown here:

Before the return.



ATME

College of Engineering

Nested Loops



```
// Loops may be nested.
class Nested {
    public static void main(String args[]) {
        int i, j;

        for(i=0; i<10; i++) {
            for(j=i; j<10; j++)
                System.out.print(".");
            System.out.println();
        }
    }
}
```

The output produced by this program is shown here:

```
.....
.....
.....
.....
.....
.....
....
...
..
.
```