P. A. EDUCATIONAL TRUST'S (PAET)

P.A. COLLEGE OF ENGINEERING

MANGALURU -574153 , KARNATAKA (INDIA)

www.pace.edu.in

Approved by A.I.C.T.E. New Delhi, Recognized by Government of Karnataka,

Affiliated to Visvesvarya Technological University, Belagavi INDIA

OBJECT ORIENTED CONCEPTS
IV SEMESTER 2018-19

DR. ZAHID ANSARI

Introduction to Java

Text: Java - The Complete Reference by Herbert Schildt, 7th Edition, Tata McGraw Hill, 2007. (Chapters 1-5)

OVERVIEW

- Java Introduction
- Java's magic: the Byte code
- Java Development Kit (JDK)
- Java Buzzwords
- Object-oriented programming
- Simple Java programs
- Data types
- variables and arrays
- Operators
- Control Statements

WHY JAVA IS IMPORTANT

- Two reasons
  - Trouble with C/C++ language is that they are not portable and are not platform independent languages.
  - Emergence of World Wide Web, which demanded portable programs
- Portability and security necessitated the invention of Java

P·A·C·E
HIGHER EDUCATION
engineering future

JAVA ORIGIN

- Many Java features are inherited from the earlier languages:

  B → C → C++ → Java
  - From C, Java derives its syntax
  - Many of Java's object-oriented features were influenced by C++

- Designed by James Gosling et. al of Sun Microsystems in 1991.
- Initially called Oak, in honor of the tree outside Gosling's window, its name was changed to Java because there was already a language called Oak.

MOTIVATION

- The primary motivation was the need for a platform-independent (that is, architecture-neutral) language to create software for consumer electronic devices, such as microwave ovens and remote controls.

- About the same time, World Wide Web was emerging.
  - This realization caused the focus of Java to switch from consumer electronics to Internet programming.

P·A·C·E
HIGHER EDUCATION
engineering future

JAVA LANGUAGE FEATURES JAVA **BUZZWORDS**

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

P·A·C·E
HIGHER EDUCATION
engineering future

SIMPLE

- Java is designed to be easy for the professional programmer to learn and use.
- Java omits many rarely used, poorly understood, confusing features of C++. Say : No Pointer! No dynamic delete.
- Automatic garbage collection
- Rich pre-defined class library
- Similar to C/C++ and it makes an effort not to have surprising features

P·A·C·E
HIGHER EDUCATION
engineering future

OBJECT ORIENTED

- The object model in Java is simple and easy to extend
  - simple types, such as integers are kept as high-performance non-objects
- Focus on the data (objects) and methods manipulating the data
- All functions are associated with objects
- Almost all data types are objects (files, strings, etc.)
- Potentially better code organization and reuse

P·A·C·E HIGHER EDUCATION engineering future

SECURE

- Since Java is used in networked environment, it requires more security.
- Code security is attained in Java through the implementation of its Java Runtime Environment (JRE)
- Programs are confined to the JRE and cannot access other parts of the computer
- Memory layout of the executable is determined during run time to add protection against unauthorized access to restricted areas of the code

ROBUST

- Restricts the programmer to find the mistakes early
  - performs compile-time (strong type checking)
  - run-time (exception-handling) checks
- Manages memory automatically
  - Java uses "a inner safe pointer- model",
  - It eliminates the possibility of overwriting memory and corrupting data, so programmers feel very safe in coding.

P·A·C·E HIGHER EDUCATION engineering future

WHY JAVA IS ROBUST?

- Consider two of the main reasons for program failure:
  - **memory management mistakes** and
  - **mishandled exceptional conditions** (that is, run-time errors)
- Memory management is a difficult and tedious. In C/C++, the programmer must manually allocate and free all dynamic memory
  - Programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using
  - Java virtually eliminates these problems by managing memory allocation and deallocation for you
- Java provides object-oriented exception handling

MULTITHREADED

- Java supports multithreaded programming, which allows you to write programs that do many things simultaneously
- Multiple concurrent threads of executions can run simultaneously
- Java run-time system provides mechaniosm for multithread synchronization that enables you to construct smoothly running interactive systems
- Uses a set of synchronization primitives (based on monitors and condition variables paradigm) to achieve this.

P·A·C·E
HIGHER EDUCATION
engineering future

PLATFORM INDEPENDENT / ARCHITECTURE-NEUTRAL / PORTABLE

- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction
- The goal of Java language creation was **"write once; run anywhere, any time, forever."**
- Java Virtual Machine provides a **platform independent environment** for the execution of Java bytecode
- Same application runs on all platforms
- The sizes of the primitive data types are always the same
- The libraries define portable interfaces

P·A·C·E
HIGHER EDUCATION
engineering future

INTERPRETED AND HIGH PERFORMANCE

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called **Java bytecode**
- Java bytecode can be **interpreted** on any system that provides a Java Virtual Machine (JVM)
- Bytecode can be also translated into the native machine code for better efficiency
- The Java bytecode is translated directly into native machine code using a **just-in-time (JIT) compiler** to achieve **high performance**

P·A·C·E
HIGHER EDUCATION
engineering future

DISTRIBUTED

- Java is extensively used for **Network applications** and **WEB projects**
- Java is designed for the distributed environment of Internet, because
  - it handles **TCP/IP protocols** to access a resource through its **URL**
  - Java can access "objects" across the Net via URLs e.g. **"http//:gamut.neiu.edu/~ylei/home.html",** with the same ease as when accessing a local file system

- **Remote Method Invocation (RMI)** feature of Java simplifies the client/server programming

P·A·C·E
HIGHER EDUCATION
engineering future

DYNAMIC

- Java programs carry with them a substantial amounts of run-time type information to verify and resolve accesses to objects at run time.
- This makes it possible to dynamically link code in a safe manner
- Class type of an object can be checked at runtime
- Small fragments of bytecode may be dynamically updated on a running system
- Libraries can freely add new methods and instance variables without any effect on their clients

P·A·C·E
HIGHER EDUCATION
engineering future

JAVA PROGRAM EXECUTION

- Java programs are both compiled and interpreted
- Steps
  - write the Java program
  - compile the program into bytecode
  - execute (interpret) the bytecode on the computer through the Java Virtual Machine (JVM)
- Compilation happens once.
- Interpretation occurs each time the program is executed.

JAVA IS COMPILED AND INTERPRETED

```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("I Rule!");
     System.out.println("The World");
   }

}
```

1 Save

`MyFirstApp.java`

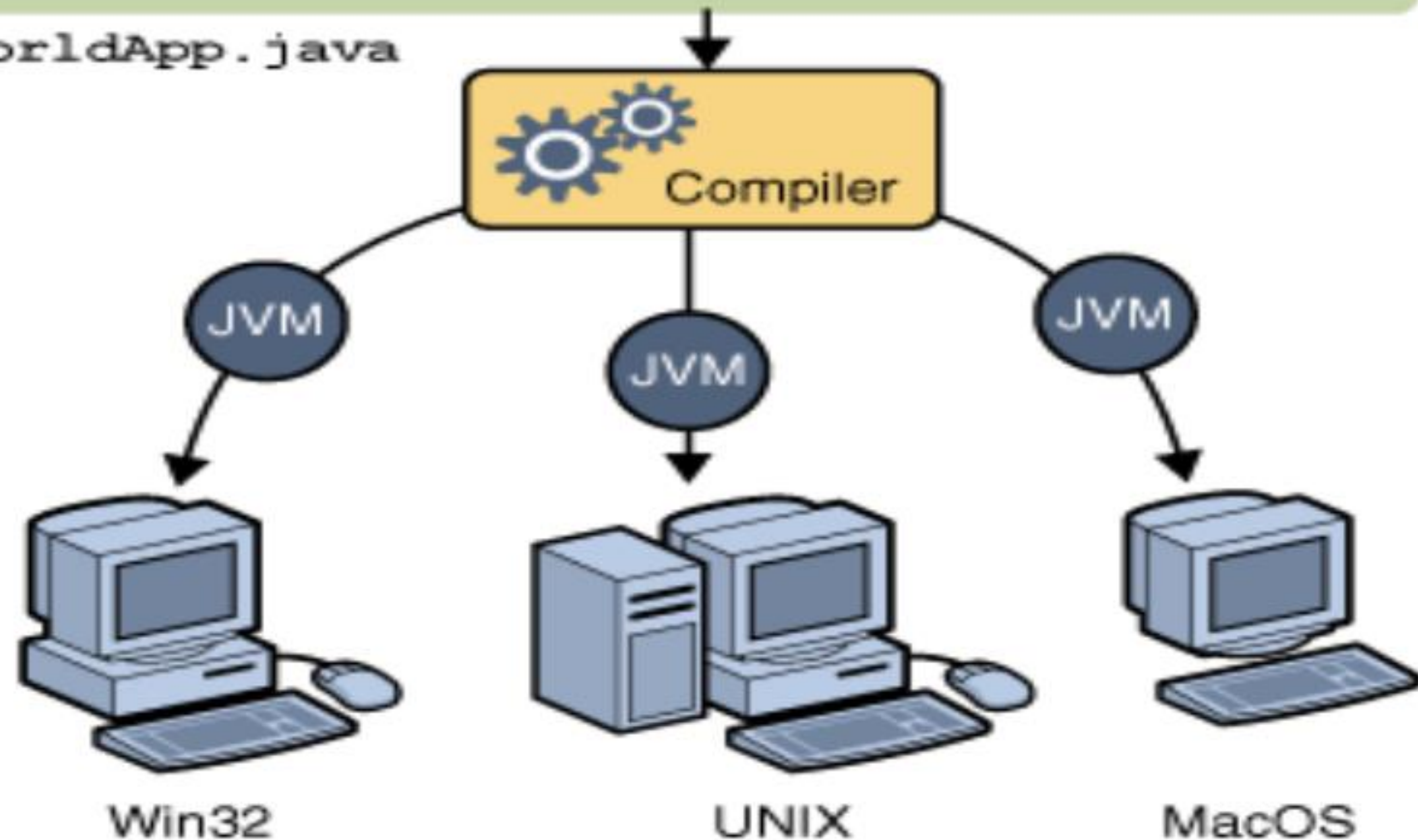
2 Compile

`javac MyFirstApp.java`

3 Run

File Edit Window Help Scream

```
%java MyFirstApp
I Rule!
The World
```



P.A. COLLEGE OF ENGINEERING
P·A·C·E
HIGHER EDUCATION
engineering future

JAVA'S MAGIC: THE BYTECODE

- The output of a Java compiler is not executable code, rather, it is **bytecode**
- Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system called **Java Virtual Machine** (JVM).
- Bytecode generated after compiling in Mac, Windows, Linux or Unix will be same. Bytecode is Independent of any particular computer hardware, so any computer with a JVM can execute the compiled Java program, no matter what type of computer, the program was compiled on
- Since Bytecode compiled in one platform can be executed into another platform, Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments. (only the JVM needs to be implemented for each platform)

JAVA VIRTUAL MACHINE (JVM)

- Java Virtual Machine (JVM) is An imaginary machine that is implemented by emulating software on a real machine
- It provides the hardware platform specifications to which you compile all Java technology code
- Java Virtual Machine **is an interpreter for bytecode**
- To run a program in a wide variety of environments, only the JVM needs to be implemented for each platform
- When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code
- The Java bytecode is translated directly into native machine code at run time on demand using a **just-in-time (JIT) compiler** to achieve **high performance**

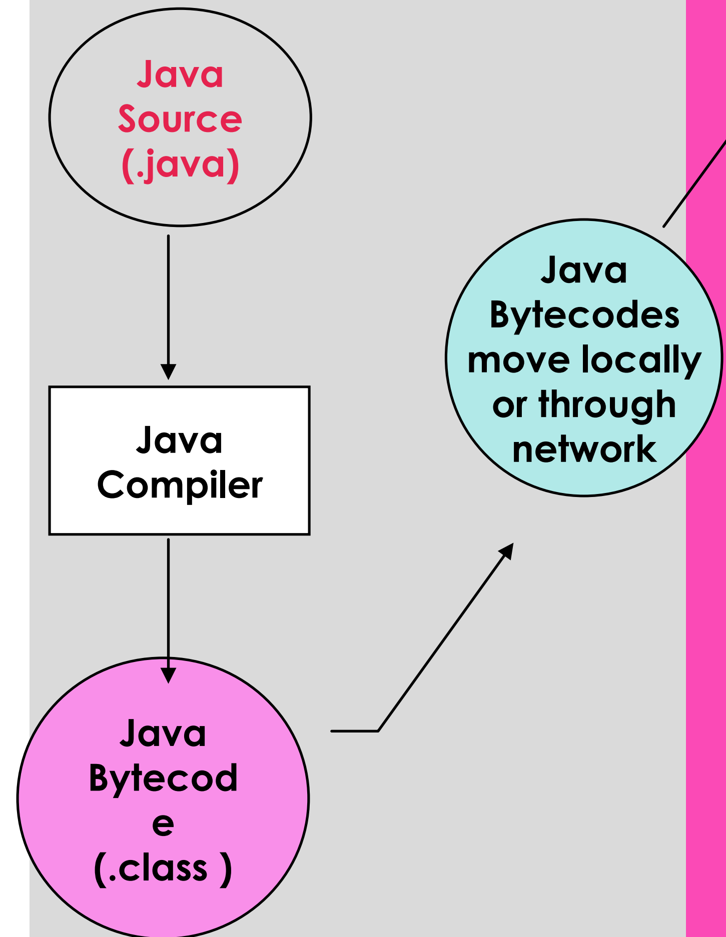
JUST IN TIME (JIT) COMPILER FOR BYTECODE

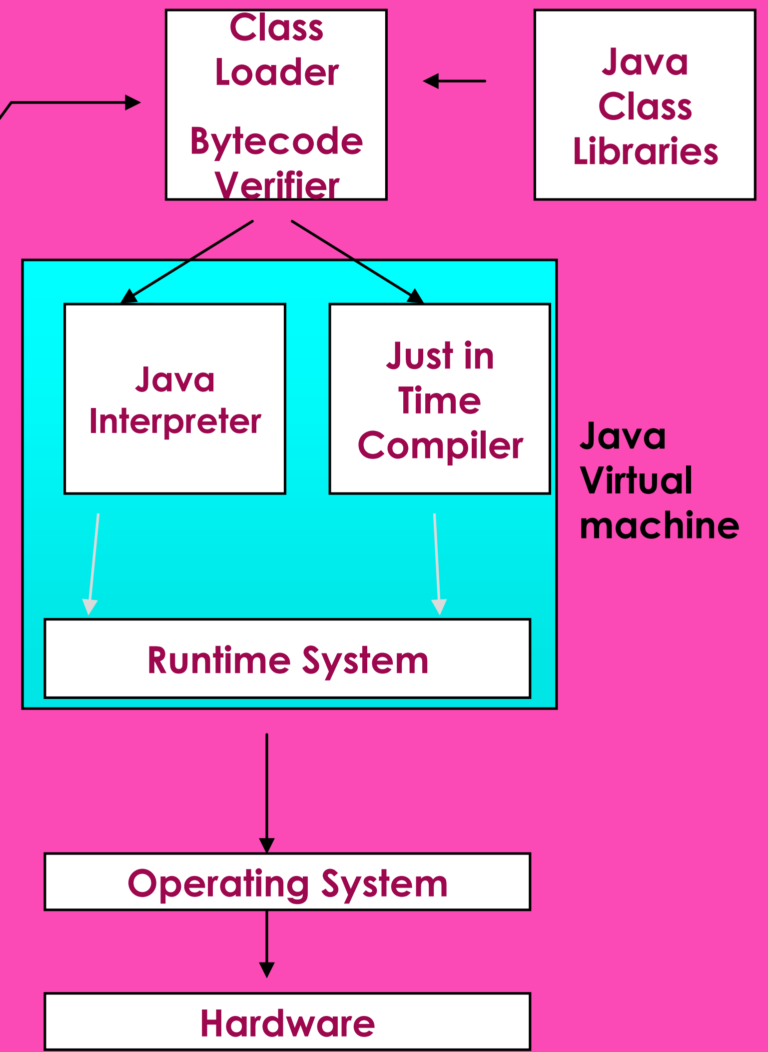
- Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release
- When JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis

P·A·C·E
HIGHER EDUCATION
engineering future

HOW IT WORKS!

- Java is independent only for one reason:
- Only depends on the Java Virtual Machine (JVM),
- Code is compiled to *bytecode*, which is interpreted by the resident JVM
- JIT (just in time) compilers attempt to increase speed.





PLATFORM INDEPENDENCE

JAVA COMPILER
(translator)

JAVA BYTE CODE

(same for all platforms)

JAVA INTERPRETER
(one for each different system)

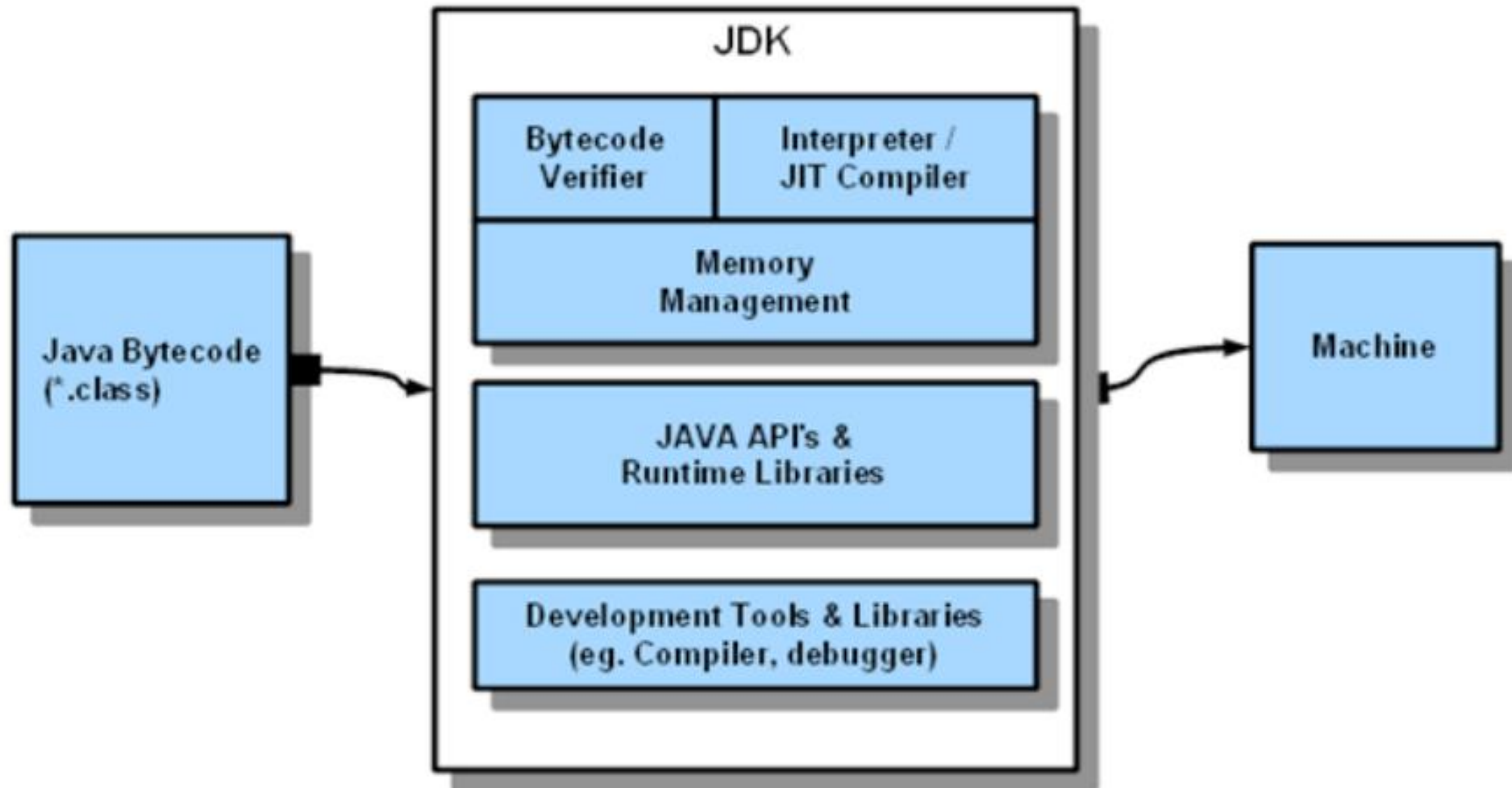
Windows 95 | Macintosh | Solaris | Windows NT

P·A·C·E
HIGHER EDUCATION
engineering future

JAVA DEVELOPMENT KIT (JDK)

- **JDK** consists of the Java Compiler and related development tools which enable users to create applications in Java.
- **JRE** is the Java Runtime Environment. i.e., the JRE is an implementation of the Java Virtual Machine which actually executes Java programs.
- Typically, each JDK contains one (or more) JRE's along with the various development tools like the Java source compilers, bundling and deployment tools, debuggers, development libraries, etc.
- JDK = JRE + Java Development Tools + Libraries

JAVA DEVELOPMENT KIT (JDK)



P.A. COLLEGE OF ENGINEERING
P·A·C·E HIGHER EDUCATION engineering future

JDK TOOLS

- Some of the tools provided by JDK are:

  javac the Java compiler

  java VM for running stand-alone Java applications

  appletviewer a utility to view applets

  javadoc HTML generator from Java source code

  jdb a rudimentary Java debugger

  javah Header file generator for interlanguage linking

  javap A disassembler

JAVA TECHNOLOGIES

- Different technologies depending on the target applications
  1) Desktop applications - Java 2 Standard Edition (J2SE)
  2) Enterprise applications – Java 2 Enterprise Edition (J2EE)
  3) Mobile applications – Java 2 Mobile Edition (J2ME)
  4) Smart Card applications – JavaCard

- Each edition puts together a large collections of packages offering functionality needed and relevant to a given application.
- The Java Virtual Machine remains essentially the same.

JAVA PROGRAM TYPES

- **Applications**: standalone (desktop) Java programs, executed from the command line, only need the Java Virtual Machine to run

- **Applets**: Java program that runs within a Java-enabled browser, invoked through a “applet” reference on a web page, dynamically downloaded to the client computer. It can be transmitted over the internet.

- **Servlets**: Java program running on the web server, capable of responding to HTTP requests made through the network etc.

An Overview of Java

JAVA – OBJECT ORIENTED

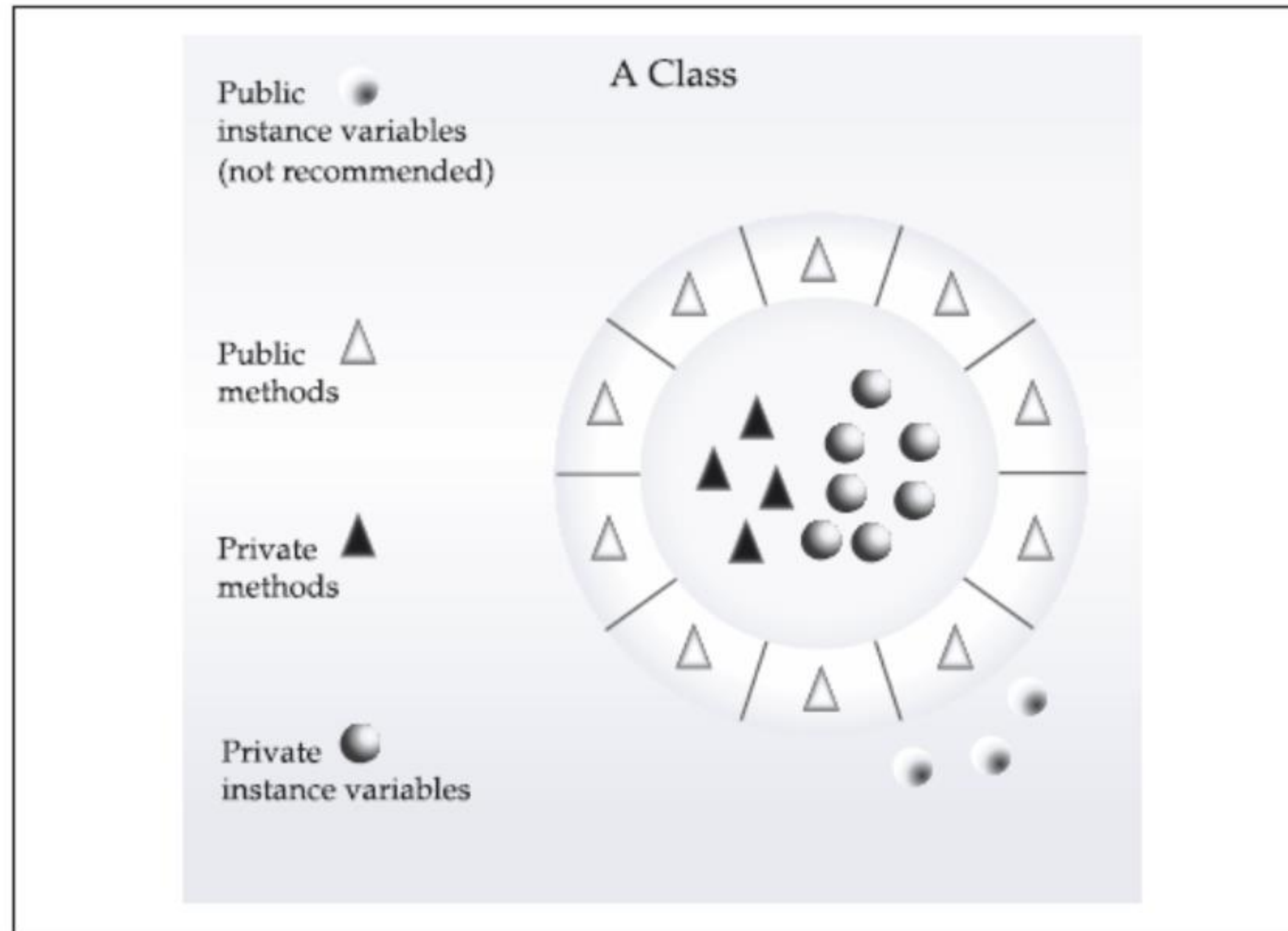
- Computer programs consist of two elements: ***code*** and ***data***.

- Some programs are organized around its code i.e. "**what is happening**". This approach is called the **Process-oriented model**: The process-oriented model can be thought of as **code acting on data**

- Other programs are organized around its data i.e. "**who is being affected**." This approach is called **Object-oriented programming**
- **Object-oriented programming** organizes a program around its data (objects) and a set of well-defined interfaces to that data: characterized as **data controlling access to the code**
- **Object-oriented programming is at the core of Java**

OOP PRINCIPLES: ENCAPSULATION

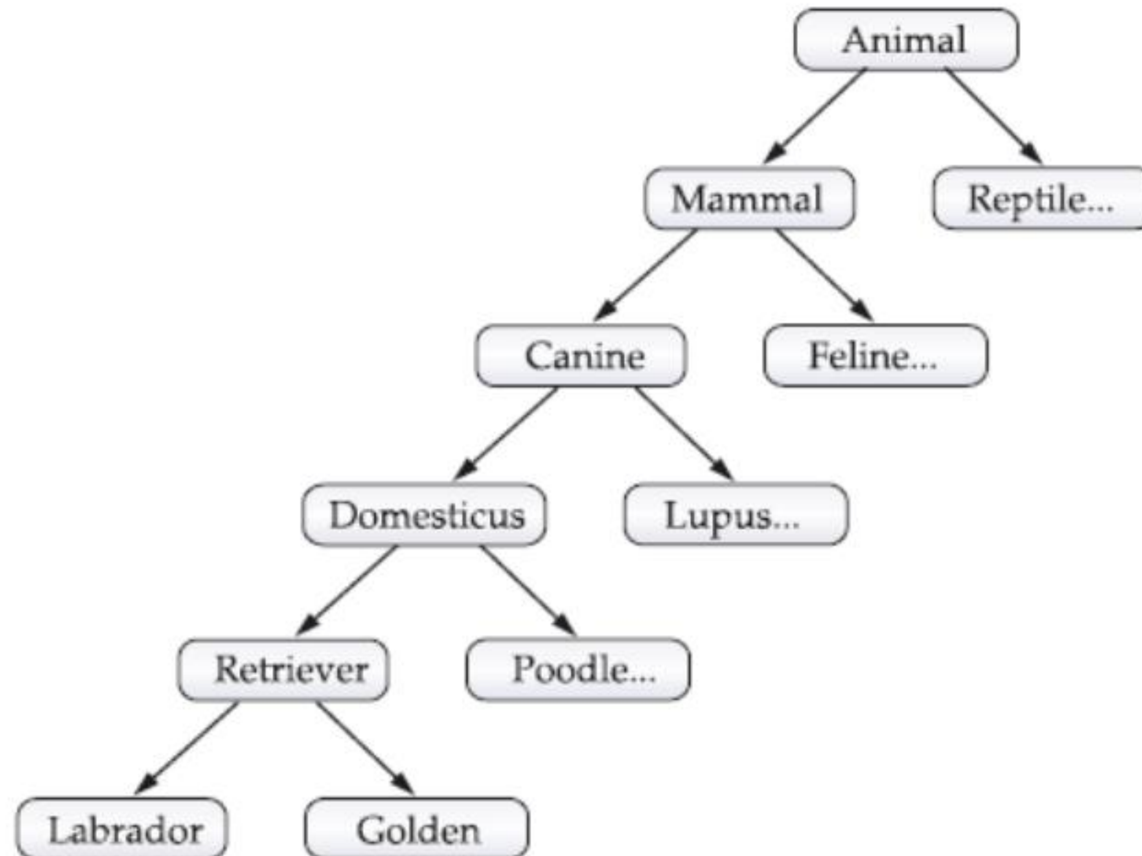
- **Encapsulation - is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse – class, member variables and methods**



P·A·C·E
HIGHER EDUCATION
engineering future

OOP PRINCIPLES: INHERITANCE

- **Inheritance** - **the process by which one object acquires the properties of another object**



OOP PRINCIPLES: POLYMORPHISM

- **Polymorphism** - is a feature that allows one interface to be used for a general class of actions – "one interface , multiple methods"
- Polymorphism, encapsulation and inheritance works together - every java program involves these.

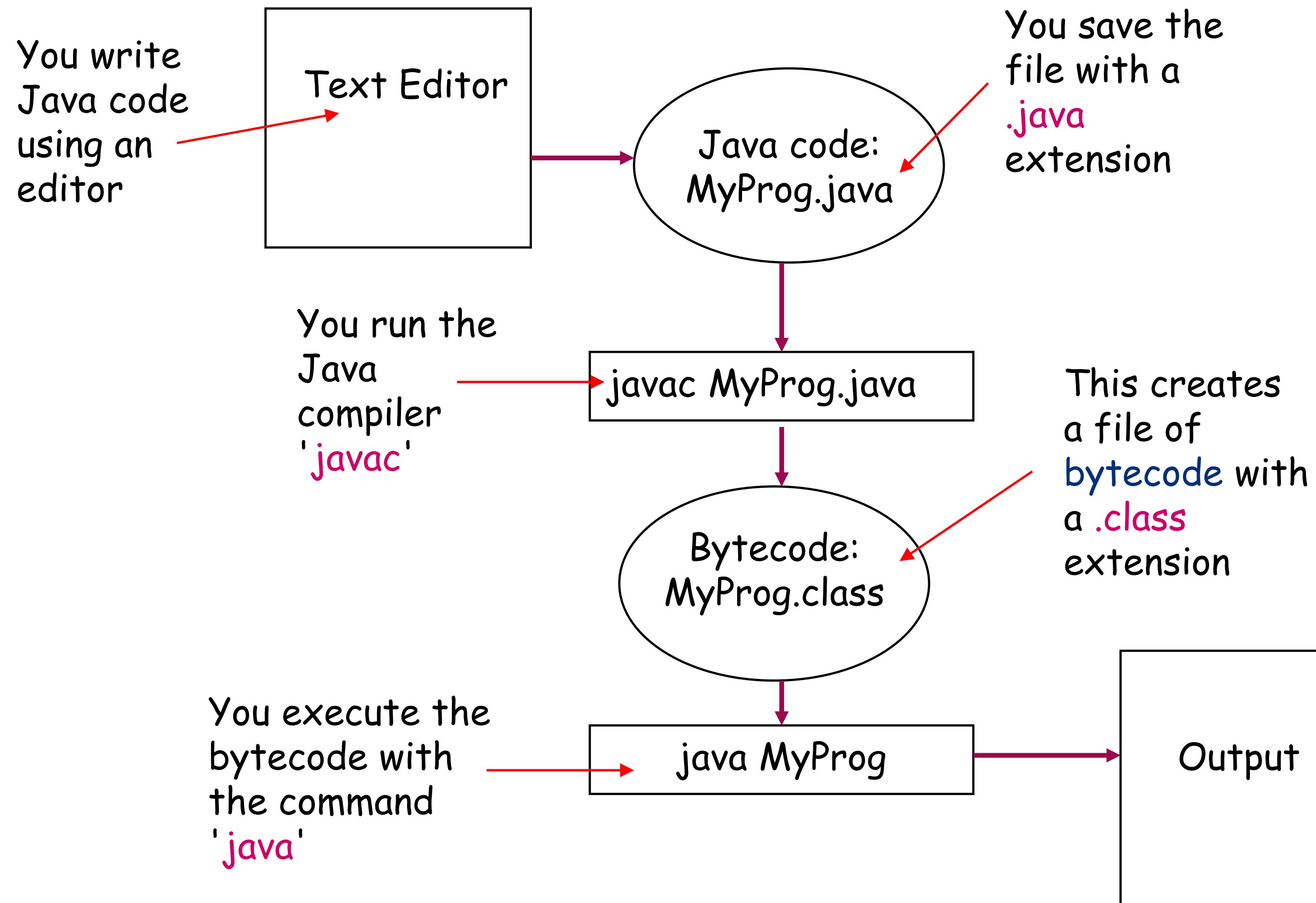
FIRST SIMPLE PROGRAM

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

A FIRST SIMPLE PROGRAM

- When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared
- Keyword **static** allows **main()** to be called without having to instantiate a particular instance of the class
- Keyword **void** tells the compiler that **main()** does not return a value
- **String[] args** declares a parameter named **args**, which is an array of instances of the class **String**. **args** receives any command-line arguments.
- The command **System.out.println()**, prints the text enclosed by quotation on the screen.
  - **System** is a predefined class which provides access to the system
  - **out** is output stream connected to console.
  - **println** displays the string which is passed to it.

RUNNING A JAVA APPLICATION



COMPILING AND RUNNING

- To compile the program call java compiler **javac**

  C:\>javac Example.java

- The javac compiler creates a file called **Example.class** that contains the bytecode version of the program
- The output of **javac** is not code that can be directly executed

- To actually run the program, you must use the Java interpreter called **java**

  C:\>java Example

P·A·C·E HIGHER EDUCATION engineering future

A SECOND SHORT PROGRAM

```
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);
     }
}
```

COMMENTS

Three kinds of comments:

1. Ignore the text between /* and */

   ```
   /* text */
   ```

2. Documentation comment (javadoc tool uses this kind of comment to automatically generate software documentation)

   ```
   /** documentation */
   ```

3. Ignore all text from // to the end of the line

   ```
   // text
   ```

CODING GUIDELINES

1. Your Java programs should always end with the .java extension.

2. Filenames should match the name of your public class. So for example, if the name of your public class is Hello, you should save it in a file called Hello.java.

3. You should write comments in your code explaining what a certain class does, or what a certain method does.

JAVA STATEMENTS AND BLOCKS CODING GUIDELINES

1. In creating blocks, you can place the opening curly brace in line with the statement. For example:

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

   or you can place the curly brace on the next line, like,

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

2. You should **indent** the next statements after the start of a block.

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

P·A·C·E HIGHER EDUCATION engineering future

JAVA IDENTIFIERS

- Identifiers are tokens that represent names of variables, methods, classes, etc.

  Examples: Hello, main, System, out.
- Java identifiers are case-sensitive.

  i.e. Identifier **Hello** is not the same as **hello**.
- Identifiers must begin with either a letter, an underscore "_", or a dollar sign "$". Letters may be lower or upper case. Subsequent characters may use numbers 0 to 9.
- Identifiers cannot use Java keywords like class, public, void, etc.

P·A·C·E HIGHER EDUCATION engineering future

JAVA IDENTIFIERS CODING GUIDELINES

- For names of classes, capitalize the first letter of the class name. Example: ThisIsAnExampleOfClassName
- For names of methods and variables, the first letter of the word should start with a small letter. Example: thisIsAnExampleOfMethodName.
- In case of multi-word identifiers, use capital letters to indicate the start of the word except the first word. Example, charArray, fileNumber, className.
- Avoid using underscores at the start of identifier such as _read or _write.

JAVA KEYWORDS

- Keywords are reserved words recognized by Java that cannot be used as identifiers.

abstract	continue	goto	package	synchronize
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	

P·A·C·E HIGHER EDUCATION engineering future

PRIMITIVE DATA TYPES

Java defines eight simple types:

1.	byte	- 8-bit integer type
2.	short	- 16-bit integer type
3.	int	- 32-bit integer type
4.	long	- 64-bit integer type
5.	float	- 32-bit floating-point type
6.	double	- 64-bit floating-point type
7.	char	- symbols in a character set
8.	boolean	- logical values true and false

P·A·C·E HIGHER EDUCATION engineering future

PRIMITIVE TYPE: BYTE

8-bit integer type.

Range:
  -128 to 127.

Example:
  byte b = -15;

Usage: particularly when working with data streams.

PRIMITIVE TYPE: SHORT

16-bit integer type.

Range:

-32768 to 32767

Example:

```
short c = 1000;
```

Usage: probably the least used simple type.

PRIMITIVE TYPE: INT

32-bit integer type.

Range:

-2147483648 to 2147483647.

Example:

```
int b = -50000;
```

Usage:

1) Most common integer type. Typically used to control loops and to index arrays.

3) Expressions involving the byte and short values are promoted to int before calculation.

P·A·C·E HIGHER EDUCATION engineering future

PRIMITIVE TYPE: LONG

64-bit integer type.

Range:

-9223372036854775808 to 9223372036854775807.

Example:

```
long l = 10000000000000000;
```

Usage:

1) useful when int type is not large enough to hold the desired value

EXAMPLE: LONG

```
// compute the light travel distance
class Light {
 public static void main(String args[]) {
      int lightspeed = 186000;
      long days = = 1000;
      long seconds = days * 24 * 60 * 60;
      long distance = lightspeed * seconds;
      System.out.print("In " + days);
      System.out.print(" light will travel about ");
      System.out.println(distance + " miles.");
  }
}
```

PRIMITIVE TYPE: FLOAT

32-bit floating-point number.

Range:

1.4e-045 to 3.4e+038.

Example:

```
float f = 1.5;
```

Usage:

1) fractional part is needed
2) large degree of precision is not required

PRIMITIVE TYPE: DOUBLE

64-bit floating-point number.

Range:

4.9e-324 to 1.8e+308.

Example:

```
double pi = 3.1416;
```

Usage:

1) accuracy over many iterative calculations
2) manipulation of large-valued numbers

EXAMPLE: DOUBLE

```
// Compute the area of a circle.
class Area {
  public static void main(String args[]) {
       double pi = 3.1416; // approximate pi value
       double r = 10.8; // radius of circle
       double a = pi * r * r; // compute area
       System.out.println("Area of circle is " + a);
  }
}
```

PRIMITIVE TYPE: CHAR

16-bit data type used to store characters.

Range:

0 to 65536.

Example:

```
char c = 'a';
```

Usage:

1) Represents both ASCII and Unicode character sets; Unicode defines a character set with characters found in (almost) all human languages.

2) Not the same as in C/C++ where char is 8-bit and represents ASCII only.

EXAMPLE CHAR

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

P·A·C·E HIGHER EDUCATION engineering future

PRIMITIVE TYPE: BOOLEAN

Two-valued type of logical values.

Range: values true and false.

Example:

```
boolean b = (1<2);
```

Usage:

1) returned by relational operators, such as 1<2
2) required by branching expressions such as if or for

EXAMPLE BOOLEAN

```
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);
        if (b) System.out.println("executed");
        b = false;
        if (b) System.out.println(“not executed");
        System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

P·A·C·E HIGHER EDUCATION engineering future

JAVA LITERALS

- Literals are tokens that do not change - they are constant.
- The different types of literals in Java are:
  - Integer Literals
  - Floating-Point Literals
  - Boolean Literals
  - Character Literals
  - String Literals

P·A·C·E HIGHER EDUCATION engineering future

INTEGER LITERALS

- Integer literals come in different formats:
  - decimal, example: 12
  - hexadecimal, example: 0xC
  - octal, example: 014
- Integer literals are of type int by default.
- Integer literal written with "L" are of type long e.g. 123L

FLOATING POINT LITERALS

- Represents decimals with fractional parts

  Example: 3.1416
- Can be expressed in standard or scientific notation e.g.
  583.45 (standard),
  5.8345e2 (scientific)
- Floating-point literals are of type double by default.
- Floating-point literal written with "F" are of type float e.g. 2.0005e3F

BOOLEAN LITERALS

- Boolean literals have only two values, true or false.
- Those values do not convert to any numerical representation.
- In particular:
    1) true is not equal to 1
    2) false is not equal to 0

PACE HIGHER EDUCATION engineering future

CHARACTER LITERALS

- Character Literals represent single Unicode characters.
- Unicode character
  - A 16-bit character set that replaces the 8-bit ASCII character set.
  - Unicode allows the inclusion of symbols and special characters from other languages.
- To use a character literal, enclose the character in single quote delimiter. For example
  - the letter a, is represented as ‘a’.
  - special characters such as a newline character, a backslash is used followed by the character code. For example, ‘\n’ for the newline character, ‘\r’ for the carriage return, ‘\b’ for backspace.

STRING LITERALS

- String is not a simple type.
- String literals are character-sequences enclosed in double quotes.
- An example of a string literal is, “Hello World!”.
- Notes:

1) Escape sequences can be used inside string literals
2) String literals must begin and end on the same line
3) Unlike in C/C++, in Java String is not an array of characters

VARIABLES

- A variable is an item of data used to store the state of objects.
- A variable has a:
  - data type, which indicates the type of value that the variable can hold
  - name, it must follow rules for identifiers.
- Declare a variable as follows:

  <data type> <name> [=initial value];

- Note: Values enclosed in <> are required values, while those values in [] are optional.

P·A·C·E HIGHER EDUCATION engineering future

VARIABLE DECLARATION

- We can declare several variables at the same time

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

- Examples:

```
int a, b, c;
int d = 3, e, f = 5;
byte hog = 22;
double pi = 3.14159;
char kat = 'x';
```

CONSTANT DECLARATION

- A variable can be declared as final:

```
final double PI = 3.14;
```

- The value of the final variable cannot change after it has been initialized:

```
PI = 3.13; // Not allowed
```

VARIABLE INITIALIZATION

- During declaration, variables may be optionally initialized.

- Initialization can be static or dynamic:
  1. static initialize with a literal:
     ```
     int n = 1;
     ```
  2. dynamic initialize with an expression composed of any literals, variables or method calls available at the time of initialization:
     ```
     int m = n + 1;
     ```

- The types of the expression and variable must be the same.

P·A·C·E HIGHER EDUCATION engineering future

CODING GUIDELINES

- It always good to initialize your variables as you declare them.
- Use descriptive names for your variables. Like for example, if you want to have a variable that contains a grade for a student, name it as, grade and not just some random letters you choose.
- Declare one variable per line of code. For example, the variable declarations,

```
double exam=0;
double quiz=10;
double grade=0;
```

is preferred over the declaration

```
double exam=0, quiz=10, grade=0;
```

P·A·C·E HIGHER EDUCATION engineering future

OUTPUTTING VARIABLE DATA

```
public class OutputVariable {
  public static void main( String[] args ){
        int value = 10;
        char x;
        x = 'A';

        System.out.println( value );
        System.out.println( "The value of x=" + x );
  }
}
```

The program will output the following text on screen:

```
10
The value of x=A
```

P·A·C·E HIGHER EDUCATION engineering future

PRIMITIVE AND REFERENCE VARIABLES

- Two types of variables in Java i) **Primitive** Variables ii) **Reference** Variables

- **Primitive Variables**
  - variables with primitive data types such as int or long.
  - stores data in the actual memory location of the variable

```
Example: int num = 10; // primitive type
```

- **Reference Variables**
  - variables that stores the address in the memory location
  - points to another memory location where the actual data is
  - When you declare a variable of a certain class, you are actually declaring a reference variable to the object with that certain class.

```
Example: String name = "Hello"; // reference type
```

P·A·C·E HIGHER EDUCATION engineering future

ARRAYS

- An array is a group of liked-typed variables referred to by a common name, with individual variables accessed by their index.
- Arrays are:
  1. declared
  2. created
  3. initialized
  4. used
- Also, arrays can have one or several dimensions.

ARRAY DECLARATION

- Array declaration involves:
  1. declaring an array identifier
  2. declaring the number of dimensions
  3. declaring the data type of the array elements
- Two styles of array declaration:

```
type array-variable[];
```

or

```
type [] array-variable;
```

P·A·C·E HIGHER EDUCATION engineering future

ARRAY CREATION

- After declaration, no array actually exists.
- In order to create an array, we use the new operator:

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

This creates a new array to hold size elements of type, whose reference will be kept in the variable array-variable.

ARRAY INDEXING

- Later we can refer to the elements of this array through their indexes:

```
array-variable[index]
```

- The array index always starts with zero!
- The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

ARRAY USE

```
class Array {
  public static void main(String args[]) {
        int monthDays[];
        monthDays = new int[12];
        monthDays[0] = 31;
        monthDays[1] = 28;
        monthDays[2] = 31;
        monthDays[3] = 30;
        monthDays[4] = 31;
        monthDays[5] = 30;
        …
        monthDays[11] = 31;
        System.out.print("April has ");
        System.out.println(monthDays[3] +" days.");
  }
}
```

ARRAY INITIALIZATION

- Arrays can be initialized when they are declared:

```
int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

- Comments:

1) there is no need to use the new operator
2) the array is created large enough to hold all specified elements

MULTIDIMENSIONAL ARRAYS

- Multidimensional arrays are arrays of arrays:
  1. declaration

     ```
     int array[][];
     ```
  2. Creation

     ```
     int array = new int[2][3];
     ```
  3. initialization

     ```
     int array[][] = { {1, 2, 3}, {4, 5, 6} };
     ```

P·A·C·E HIGHER EDUCATION engineering future

EXAMPLE: MULTIDIMENSIONAL ARRAYS

```
class Array {
  public static void main(String args[]) {
        int array[][] = { {1, 2, 3}, {4, 5, 6} };
        int i, j;
        for(i=0; i<2; i++) {
              for(j=0; j<3; j++)
                 System.out.print(array[i][j] + " ");
              System.out.println();
        }
  }
}
```

P·A·C·E HIGHER EDUCATION engineering future

OPERATORS

- Java operators are used to build value expressions.
- Java provides a rich set of operators:
  1. assignment
  2. arithmetic
  3. relational
  4. logical
  5. bitwise
  6. other

ASSIGNMENT OPERATOR

- A binary operator:

```
variable = expression;
```

- It assigns the value of the expression to the variable.
- The types of the variable and expression must be compatible.

- The value of the whole assignment expression is the value of the expression on the right, so it is possible to chain assignment expressions as follows:

```
int x, y, z;
x = y = z = 2;
```

P·A·C·E HIGHER EDUCATION engineering future

ARITHMETIC OPERATORS

Operator	Use	Description
+	op1 + op2	Adds op1 and op2
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2
-	op1 - op2	Subtracts op2 from op1

ARITHMETIC OPERATORS

- Java supports arithmetic operators for:

  - integer numbers

  - floating-point numbers

- Note: When an integer and a floating-point number are used as operands to a single arithmetic operation, the result is a floating point. The integer is implicitly converted to a floating-point number before the operation takes place.

INCREMENT AND DECREMENT OPERATORS

- unary increment operator (**++**)
- unary decrement operator (**--**)
- Increment and decrement operators increase and decrease a value stored in a number variable by 1.
- For example, the expression,

`count=count + 1;` // increment the value of count by 1

is equivalent to,

`count++;`

PACE HIGHER EDUCATION engineering future

INCREMENT AND DECREMENT OPERATORS

Operator	*Use*	*Description*
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

P·A·C·E HIGHER EDUCATION engineering future

INCREMENT AND DECREMENT OPERATORS

- The increment and decrement operators can be placed before or after an operand.
- When used before an operand, it causes the variable to be incremented or decremented by 1, and then the new value is used in the expression in which it appears. For example,

```
int i = 10;
int j = 3;
int k = 0;
k = ++j + i; //will result to k = 4+10 = 14
```

- When the increment and decrement operators are placed after the operand, the old value of the variable will be used in the expression where it appears. For example,

```
int i = 10;
int j = 3;
int k = 0;
k = j++ + i; //will result to k = 3+10 = 13
```

EXAMPLE: INCREMENT/DECREMENT

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

P·A·C·E
HIGHER EDUCATION
engineering future

RELATIONAL OPERATORS

- Relational operators determine the relationship that one operand has to the other operand, specifically equality and ordering.
- The outcome is always a value of type boolean i.e. true or false.
- They are most often used in branching and loop control statements.

Operator	***Use***	***Description***
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

LOGICAL OPERATOR

<table>
<tr><td>&</td><td>op1 & op2</td><td>Logical AND</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>Logical OR</td></tr>
<tr><td>&&</td><td>op1 && op2</td><td>Short-circuit AND</td></tr>
<tr><td>||</td><td>op1 || op2</td><td>Short-circuit OR</td></tr>
<tr><td>!</td><td>! op</td><td>Logical NOT</td></tr>
<tr><td>^</td><td>op1 ^ op2</td><td>Logical XOR</td></tr>
</table>

LOGICAL OPERATOR

- Logical operators act upon boolean operands only.
- The outcome is always a value of type boolean.
- In particular, AND and OR logical operators occur in two forms:

1. **Full** - `op1 & op2` and `op1 | op2` where both op1 and op2 are evaluated
2. **Short-circuit** - `op1 && op2` and `op1 || op2` where op2 is only evaluated if the value of op1 is insufficient to determine the final outcome

EXAMPLE: LOGICAL OPERATORS

```
class LogicalDemo {
  public static void main(String[] args) {
        int n = 2;
        if (n != 0 && n / 0 > 10)
              System.out.println("This is true");
        else
              System.out.println("This is false");
  }
}
```

P·A·C·E HIGHER EDUCATION engineering future

BITWISE OPERATORS

- Bitwise operators apply to integer types only.
- They act on individual bits of their operands.
- There are three kinds of bitwise operators:
    1) basic bitwise AND, OR, NOT and XOR
    2) shifts left, right and right-zero-fill
    3) bitwise assignment for all basic and shift operators

P·A·C·E HIGHER EDUCATION engineering future

BITWISE OPERATORS

~	~ op	inverts all bits of its operand
&	op1 & op2	produces 1 bit if both operands are 1
|	op1 | op2	produces 1 bit if either operand is 1
^	op1 ^ op2	produces 1 bit if exactly one operand is 1
>>	op1 >> op2	shifts all bits in op1 right by the value of op2
<<	op1 << op2	shifts all bits in op1 left by the value of op2
>>>	op1 >>> op2	shifts op1 right by op2 value, write zero on the left

EXAMPLE: BITWISE OPERATORS

```
class BitLogic {
  public static void main(String args[]) {
        String binary[] = { "0000","0001", … , "1111"};
        int a = 3; // 0011 in binary
        int b = 6; // 0110 in binary
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        System.out.print("a =" + binary[a]);
        System.out.println("and b =" + binary[b]);
        System.out.println("a|b = " + binary[c]);
        System.out.println("a&b = " + binary[d]);
        System.out.println("a^b = " + binary[e]);
  }
}
```

P·A·C·E HIGHER EDUCATION engineering future

OTHER OPERATORS

Operator	Description
?:	shortcut if-else statement / conditional operator
[ ]	used to declare arrays, create arrays, access array elements
.	used to form qualified names
(params)	delimits a comma-separated list of parameters
(type)	casts a value to the specified type
new	creates a new object or a new array
instanceof	determines if its first operand is an instance of the second

CONDITIONAL OPERATOR

- General form:

  `expr1 ? expr2 : expr3`

  Where:

  1) **`expr1`** is of type boolean
  2) **`expr2`** and **`expr3`** are of the same type

- If **`expr1`** is true, **`expr2`** is evaluated, otherwise **`expr3`** is evaluated.

EXAMPLE: CONDITIONAL OPERATOR

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

P·A·C·E HIGHER EDUCATION engineering future

OPERATOR PRECEDENCE

- Java operators are assigned precedence order.
- Precedence determines that the expression

```
1 + 2 * 6 / 3 > 4 && 1 < 0
```

is equivalent to

```
(((1 + ((2 * 6) / 3)) > 4) && (1 < 0))
```

- When operators have the same precedence, the earlier one binds stronger.

OPERATOR PRECEDENCE

highest			
( )	[ ]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
lowest			

TYPE DIFFERENCES

- Suppose a value of one type is assigned to a variable of another type.

```
T1 t1;
T2 t2 = t1;
```

- What happens? Different situations:

  1) types T1 and T2 are incompatible

  2) types T1 and T2 are compatible:

     a) T1 and T2 are the same

     b) T1 is larger than T2

     c) T2 is larger than T1

P·A·C·E HIGHER EDUCATION engineering future

TYPE COMPATIBILITY

- When types are compatible?
  1. integer types and floating-point types are compatible with each other
  2. numeric types are not compatible with char or boolean
  3. char and boolean are not compatible with each other
- Examples:

```
byte b;
int i = b;
char c = b; // Not allowed
```

WIDENING TYPE CONVERSION

- Java performs automatic type conversion when:

  1) two types are compatible

  2) destination type is larger than the source type

- Example:

```
int i;
double d = i;
```

NARROWING TYPE CONVERSION

- When:
  1. two types are compatible
  2. destination type is smaller then the source type then Java will not carry out type-conversion:

```
int i;
byte b = i;
```

- Instead, we have to rely on manual type-casting:

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

P·A·C·E HIGHER EDUCATION engineering future

TYPE CASTING

- General form: `(targetType) value`
- Examples:

1) integer value will be reduced module byte's range:

```
int I = 500;
byte b = (byte) i;
```

2) floating-point value will be truncated to integer value:

```
float f = 3.4;
int i = (int) f;
```

P·A·C·E HIGHER EDUCATION engineering future

EXAMPLE: TYPE CASTING

```
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("\ndouble to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);
  }
}
```

P·A·C·E HIGHER EDUCATION engineering future

TYPE PROMOTION

- In an expression, precision required to hold an intermediate value may sometimes exceed the range of either operand:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

- Java promotes each byte operand to int when evaluating the expression.

TYPE PROMOTION RULES

1. byte and short are always promoted to int
2. if one operand is long, the whole expression is promoted to long
3. if one operand is float, the entire expression is promoted to float
4. if any operand is double, the result is double

P·A·C·E
HIGHER EDUCATION
engineering future

EXAMPLE: TYPE PROMOTION

```
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("result = " + result);
 }
}
```

P·A·C·E HIGHER EDUCATION engineering future

EXERCISE: OPERATORS

1) What operators do the code snippet below contain?

```
arrayOfInts[j] > arrayOfInts[j+1];
```

2) Consider the following code snippet:

```
int i = 10;
int n = i++%5;
```

a) What are the values of *i* and *n* after the code is executed?

b) What are the final values of *i* and *n* if instead of using the postfix increment operator (i++), you use the prefix version (++i))?

3) What is the value of *i* after the following code snippet executes?

```
int i = 8;
i >>=2;
```

4) What's the result of

```
System.out.println(010| 4); ?
```

P·A·C·E HIGHER EDUCATION engineering future

CONTROL FLOW

P·A·C·E HIGHER EDUCATION engineering future

CONTROL FLOW

- Writing a program means typing statements into a file.
- Without control flow, the interpreter would execute these statements in the order they appear in the file, left-to-right, top-down.
- Control flow statements, when inserted into the text of the program, determine in which order the program should be executed.

CONTROL FLOW STATEMENTS

- Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.
- Control statements are divided into three groups:

1. selection statements allow the program to choose different parts of the execution based on the outcome of an expression
2. iteration statements enable program execution to repeat one or more statements
3. jump statements enable your program to execute in a non-linear fashion

SELECTION STATEMENTS

- Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.
- Java provides four selection statements:

1) `if`
2) `if-else`
3) `if-else-if`
4) `switch`

P·A·C·E HIGHER EDUCATION engineering future

IF STATEMENTS

- General form:

```
if (expression) statement
```

- If `expression` evaluates to `true`, execute `statement`, otherwise do nothing.
- The expression must be of type `boolean`.

SIMPLE/COMPOUND STATEMENTS

- Simple

```
if (expression) statement;
```

- Compound

```
if (expression) {
   statement;
}
```

IF-ELSE STATEMENTS

- Suppose you want to perform two different statements depending on the outcome of a boolean expression. if-else statement can be used.
- General form:

```
if (expression)
    statement1
else
    statement2
```

- Again, statement1 and statement2 may be simple or compound.

IF-ELSE-IF STATEMENTS

- General form:

```
if (expression1) statement1
else if (expression2) statement2
else if (expression3) statement3
…
else statement
```

- Semantics:

1) statements are executed top-down
2) as soon as one expressions is true, its statement is executed
3) if none of the expressions is true, the last statement is executed

P·A·C·E HIGHER EDUCATION engineering future

EXAMPLE: IF-ELSE-IF

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

PACE HIGHER EDUCATION engineering future

SWITCH STATEMENTS

- switch provides a better alternative than if-else-if when the execution follows several branches depending on the value of an expression.

- General form:

```
switch (expression) {
  case value1: statement1; break;
  case value2: statement2; break;
  case value3: statement3; break;
  …
  default: statement;
}
```

P·A·C·E HIGHER EDUCATION engineering future

SWITCH ASSUMPTIONS/SEMANTICS

- Assumptions:
  1) expression must be of type byte, short, int or char
  2) each of the case values must be a literal of the compatible type
  3) case values must be unique
- Semantics:
  1) expression is evaluated
  2) its value is compared with each of the case values
  3) if a match is found, the statement following the case is executed
  4) if no match is found, the statement following default is executed
- break makes sure that only the matching statement is executed.
- Both default and break are optional.

EXAMPLE: SWITCH

```
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 " + season + ".");
  }
}
```

NESTED SWITCH STATEMENTS

- A switch statement can be nested within another switch statement:

```
switch( count ) {
        case 1:
                switch( target ) {
                        case 0:System.out.println("target is zero");
                        break;
                        case 1:System.out.println("target is one");
                        break;
                }
                break;
        case 2: …
}
```

- Since, every switch statement defines its own block, no conflict arises between the case constants in the inner and outer switch statements.

P·A·C·E HIGHER EDUCATION engineering future

COMPARING SWITCH AND IF

- Two main differences:

1) switch can only test for equality, while if can evaluate any kind of boolean expression

2) Java creates a “jump table” for switch expressions, so a switch statement is usually more efficient than a set of nested if statements

ITERATION STATEMENTS

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.
- Java provides three iteration statements:

1. while
2. do-while
3. for

WHILE STATEMENTS

- General form:

  ```
  while (expression) statement
  ```

- where `expression` must be of type boolean.

- Semantics:

1. repeat execution of `statement` until `expression` becomes false
2. `expression` is always evaluated before `statement`
3. if `expression` is false initially, `statement` will never get executed

EXAMPLE: WHILE

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

PACE HIGHER EDUCATION engineering future

DO-WHILE STATEMENTS

- If a comound statement has to be executed at least once, the do-while statement is more appropriate than the while statement.
- General form:

```
do statement
while (expression);
```

- where expression must be of type boolean.
- Semantics:

1. repeat execution of statement until expression becomes false
2. expression is always evaluated after statement
3. even if expression is false initially, statement will be executed

P·A·C·E HIGHER EDUCATION engineering future

EXAMPLE: DO-WHILE

```
class DoWhile {
 public static void main(String args[]) {
      int i;
      i = 0;
      do
           i++;
      while ( 1/i < 0.001);
      System.out.println("i is " + i);
 }
}
```

P·A·C·E
HIGHER EDUCATION
engineering future

FOR STATEMENTS

- When iterating over a range of values, for statement is more suitable to use than while or do-while.
- General form:

```
for (initialization; termination; increment)
   statement
```

- where:

1. initialization statement is executed once before the first iteration
2. termination expression is evaluated before each iteration to determine when the loop should terminate
3. increment statement is executed after each iteration

FOR STATEMENTS

- This is how the for statement is executed:

1. initialization is executed once
2. termination expression is evaluated:

   a) if false, the statement terminates

   b) otherwise, continue to (3)
3. increment statement is executed
4. component statement is executed
5. control flow continues from (2)

P·A·C·E
HIGHER EDUCATION
engineering future

LOOP CONTROL VARIABLE

- The for statement may include declaration of a loop control variable:

```
for (int i = 0; i < 1000; i++) {
  …
}
```

- The variable does not exist outside the for statement.

EXAMPLE: FOR

```
class FindPrime {
 public static void main(String args[]) {
      int num = 14;
      boolean isPrime = true;
      for (int i=2; i < num/2; i++) {
           if ((num % i) == 0) {
                isPrime = false;
                break;
           }
      }
      if (isPrime)
           System.out.println("Prime");
      else System.out.println("Not Prime");
 }
}
```

PACE HIGHER EDUCATION engineering future

FOR-EACH VERSION OF THE FOR LOOP

- 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
```

- *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.
- With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*.
- The loop repeats until all elements in the collection have been obtained.
- Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection.
- The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

P·A·C·E
HIGHER EDUCATION
engineering future

EXAMPLE: FOR-EACH

```
// Java program to sum the elements of the array  {1, 2, 3, 4, 5, 6, 7, 8.9, 10}
// using "for-each" version of the 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);
        }
}
```

PACE HIGHER EDUCATION
engineering future

EXAMPLE: FOR-EACH

- 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
```

P·A·C·E
HIGHER EDUCATION
engineering future

MANY INITIALIZATION/ITERATION PARTS

- The for statement may include several initialization and iteration parts.
- Parts are separated by a comma:

```
int a, b;
for (a = 1, b = 4; a < b; a++, b--) {
    …
}
```

PACE HIGHER EDUCATION engineering future

FOR STATEMENT VARIATIONS

- The for statement need not have all components:

```
class ForVar {
    public static void main(String args[]) {
        int i = 0;
        boolean done = false;
        for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

PACE HIGHER EDUCATION engineering future

EMPTY FOR

- In fact, all three components may be omitted:

```
public class EmptyFor {
 public static void main(String[] args) {
      int i = 0;
      for (; ;) {
           System.out.println("Infinite Loop " + i);
      }
  }
}
```

JUMP STATEMENTS

- Java jump statements enable transfer of control to other parts of program.
- Java provides three jump statements:
  1) `break`
  2) `continue`
  3) `return`
- In addition, Java supports exception handling that can also alter the control flow of a program.

P·A·C·E HIGHER EDUCATION engineering future

BREAK STATEMENTS

- The break statement has three uses:
  1. to terminate a case inside the switch statement
  2. to exit an iterative statement
  3. to transfer control to another statement

- (1) has been described.
- We continue with (2) and (3).

LOOP EXIT WITH BREAK

- When break is used inside a loop, the loop terminates and control is transferred to the following instruction.

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

P·A·C·E HIGHER EDUCATION engineering future

BREAK IN NESTED LOOP

- Used inside nested loops, break will only terminate the innermost loop:

```
class NestedLoopBreak {
    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;
                System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Loops complete.");
    }
}
```

P·A·C·E HIGHER EDUCATION engineering future

CONTROL TRANSFER WITH BREAK

- Java does not have an unrestricted “goto” statement, which tends to produce code that is hard to understand and maintain.
- However, in some places, the use of gotos is well justified. In particular, when breaking out from the deeply nested blocks of code.
- break occurs in two versions:
  1) unlabelled
  2) labeled
- The labeled break statement is a “civilized” replacement for goto.

P·A·C·E HIGHER EDUCATION engineering future

LABELED BREAK

- General form:

```
break label;
```

- where label is the name of a label that identifies a block of code:

```
label: { … }
```

- The effect of executing break label; is to transfer control immediately after the block of code identified by label.

PACE HIGHER EDUCATION engineering future

EXAMPLE: LABELED BREAK

```
class Break {
    public static void main(String args[]) {
        boolean t = true;
        first: {
            second: {
                third: {
                    System.out.println("Before the break.");
                    if (t) break second;
                    System.out.println("This won't execute");
                }
                System.out.println("This won't execute");
            }
            System.out.println("After second block.");
        }
    }
}
```

EXAMPLE: NESTED LOOP BREAK

```
class NestedLoopBreak {
    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++) {
                  // exit both loops
                  if (j == 10) break outer;
                  System.out.print(j + " ");
            }
            System.out.println("This will not print");
      }
      System.out.println("Loops complete.");
    }
}
```

CONTINUE STATEMENT

- The break statement terminates the block of code, in particular it terminates the execution of an iterative statement.
- The continue statement forces the early termination of the current iteration to begin immediately the next iteration.
- Like break, continue has two versions:

1. unlabelled – continue with the next iteration of the current loop
2. labeled – specifies which enclosing loop to continue

P·A·C·E HIGHER EDUCATION engineering future

EXAMPLE: UNLABELED 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("");
      }
    }
}
```

P·A·C·E HIGHER EDUCATION engineering future

EXAMPLE: LABELED CONTINUE

```
class LabeledContinue {
    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();
    }
}
```

P·A·C·E HIGHER EDUCATION engineering future

RETURN STATEMENT

- The return statement is used to return from the current method: it causes program control to transfer back to the caller of the method.
- Two forms:

1) return without value

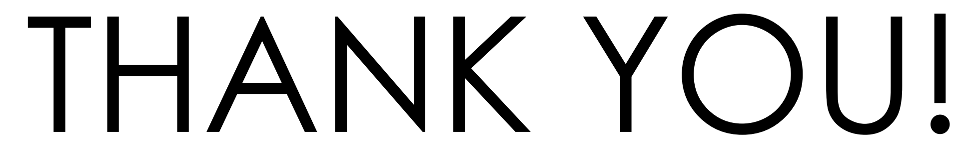
```
return;
```

2) return with value

```
return expression;
```

EXAMPLE: 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.");
    }
}
```


THANK YOU!