Introduction to Java

History of Java

- ✓ Java is a general purpose object oriented programming language.
- ✓ Developed by Sun Microsystems. (James Gostling)
- ✓ Initially called “Oak” but was renamed as “Java” in 1995.
- ✓ Initial motivation is to develop a platform independent language to create software to be embedded in various consumer electronics devices.
- ✓ Become the language of internet. (portability and security).

Features of Java

1. Simple, Small and Familiar
2. Compiled and Interpreted
3. Object Oriented
4. Platform Independent and portable
5. Robust and Secure
6. Distributed / Network Oriented
7. Multithreaded and Interactive
8. High Performance
9. Dynamic

Simple, Small and Familiar

- Similar to C/C++ in syntax
- But eliminates several complexities of
  - No operator overloading
  - No direct pointer manipulation or pointer arithmetic
  - No multiple inheritance
  - No malloc() and free() – handles memory automatically

Compiled and Interpreted

- Java works in two stages
  - Java compiler translate the source code into byte code.
  - Java interpreter converts the byte code into machine level representation.

Byte Code:

-A highly optimized set of instructions to be executed by the java runtime system, known as java virtual machine (JVM).

-Not executable code.

Java Virtual Machine (JVM):

- Need to be implemented for each platform.
- Although the details vary from machine to machine,

Java Virtual Machine

- ✓ Java compiler produces an intermediate code known as byte code for a machine, known as JVM.
- ✓ It exists only inside the computer memory.

- ✓ Machine code is generated by the java interpreter by acting as an intermediary between the virtual machine and real machine.

Object Oriented

- Fundamentally based on OOP
  - Classes and Objects
  - Efficient re-use of packages such that the programmer only cares about the interface and not the implementation
  - The object model in java is simple and easy to extend.

Platform Independent and Portable

- “Write-Once Run-Anywhere”
- Changes in system resources will not force any change in the program.
- The Java Virtual Machine (JVM) hides the complexity of working on a particular platform
  - Convert byte code into machine level representation.

Robust and Secure

- Designed with the intention of being secure
  - No pointer arithmetic or memory management!
  - Strict compile time and run time checking of data type.
  - Exception handling
  - It verifies all memory access
  - Ensure that no viruses are communicated with an applet.

Distributed and Network Oriented

- Java grew up in the days of the Internet
  - Inherently network friendly
  - Original release of Java came with Networking libraries
  - Newer releases contain even more for handling distributed applications
    - RMI, Transactions

Multithreaded and Interactive

- Handles multiple tasks simultaneously.
- Java runtime system contains tools to support multiprocess synchronization and construct smoothly running interactive systems.

High Performance

- Java performance is slower than C
- Provisions are added to reduce overhead at runtime.
- Incorporation of multithreading enhance the overall execution speed.
- Just-in-Time (JIT) can compile the byte code into machine code.
- Can sometimes be even faster than compiled C code!

Dynamic

- Capable of dynamically linking new class libraries, methods and objects.
- Java can use efficient functions available in C/C++.
- Installing new version of library automatically updates all programs

Language of Internet Programming

- Java Applets
- Security
- Portability

1. **Applets:**

Special java program that can transmitted over the network and automatically executed by a java-compatible web browser.

2. Security:

Java compatible web browser can download java applets without fear of viral infection and malicious agent.

3. Portable:

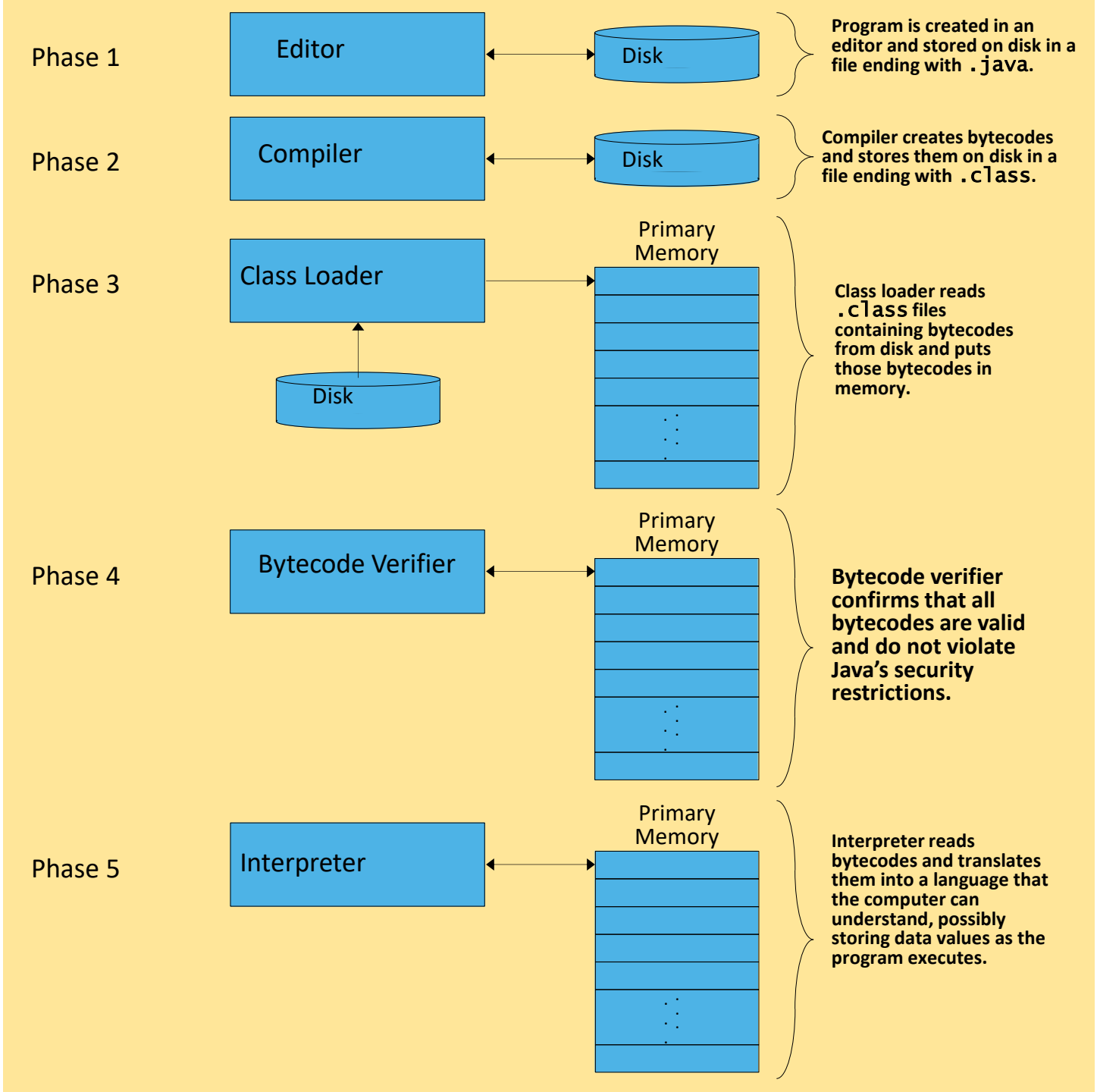
Java applets can be dynamically downloaded to all the various types of platforms connected to the internet

Why portable and Secure?

- The output of java compiler is not executable code.
- Once JVM exists for a platform, any java program can run on it.
- The execution of byte code by the JVM makes java programs portable.
- Java program is confined within the java execution environment and cannot access the other part of the computer.

Basics of Java Environments

- Java programs normally undergo five phases
  - Edit
    - Programmer writes program (and stores program on disk)
  - Compile
    - Compiler creates *bytecodes* from program
  - Load
    - Class loader stores bytecodes in memory
  - Verify
    - Verifier ensures bytecodes do not violate security requirements
  - Execute
    - Interpreter translates bytecodes into machine language


Phase 1
Editor
Disk
Program is created in an editor and stored on disk in a file ending with .java.
Phase 2
Compiler
Disk
Compiler creates bytecodes and stores them on disk in a file ending with .class.
Phase 3
Class Loader
Disk
Primary Memory
Class loader reads .class files containing bytecodes from disk and puts those bytecodes in memory.
Phase 4
Bytecode Verifier
Primary Memory
Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions.
Phase 5
Interpreter
Primary Memory
Interpreter reads bytecodes and translates them into a language that the computer can understand, possibly storing data values as the program executes.

Java Environment

- Development tools-part of java development kit (JDK).
- Classes and methods-part of Java Standard Library (JSL), also known as Application Programming Interface (API).

1. **JDK:**

- Appletviewer (for viewing applets)
- Javac (Compiler)
- Java (Interpreter)
- Javap (Java disassembler)
- Javah (for C header files)
- Javadoc (for creating HTML description)
- Jdb (Java Debugger)

Java Environment

2. Application Package Interface (API)

Contains hundreds of classes and methods grouped into several functional packages:

- Language Support Package
- Utility Packages
- Input/Output Packages
- Networking Packages
- AWT Package
- Applet Package

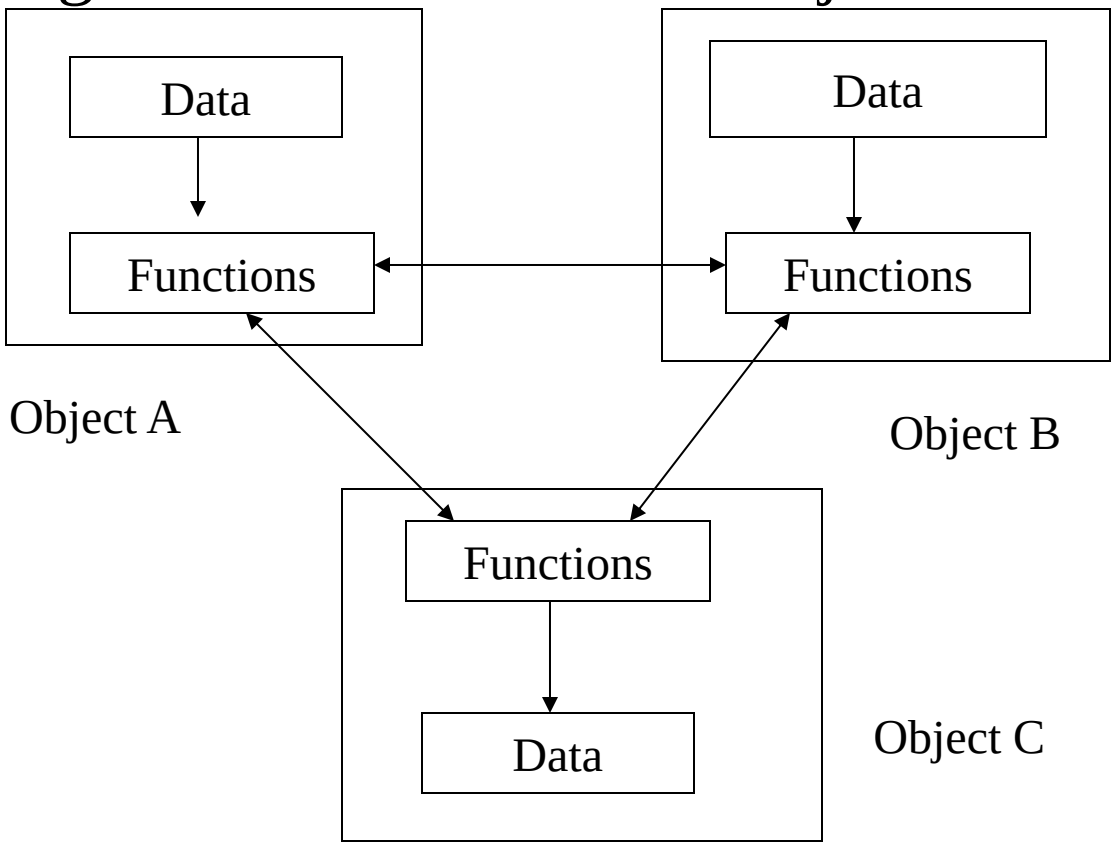
Concept of Object oriented Programming

&

Writing First Java Program

Object –Oriented Programming

- ✓ Emphasis is on data rather than procedure.
- ✓ Programs are divided into objects.



Object –Oriented Programming

- ✓ Objects can communicate with each other through functions.
- ✓ New data and functions can be easily added whenever necessary.
- ✓ Follow bottom up approach in program design.

Basic Concepts of OOP

- ✓ Objects and classes
- ✓ Data Encapsulation
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Dynamic Binding
- ✓ Message Communication

Objects and Classes

- ✓ Program objects should be chosen such that they match closely with the real-world objects.
- ✓ Any programming problem is analyzed in terms of objects and the nature of communication between them.
- ✓ Objects contain data and code to manipulate that data.
- ✓ A **class** is a data type and an object is a variable of that data type.
- ✓ Class define the data and code that should be included in each object of that class.
- ✓ It is a user defined type

Data Encapsulation

- ✓ The wrapping of data and methods into a single unit is known as encapsulation.
- ✓ This ensures data hiding.
- ✓ The methods of an object provides interface between the data of the object and the program.

Inheritance

✓Inheritance is the process by which objects of one class can acquire the properties of objects of another class.

✓It provides the idea of reusability.

Polymorphism

- ✓ Polymorphism means the ability to take more than one form.
- ✓ A general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ.

Dynamic Binding

✓The code associated with a procedure call is not known until the time of the call at runtime.

✓It is associated with polymorphism and inheritance.

Message Communication

- ✓ Objects communicate with each other by sending and receiving messages.

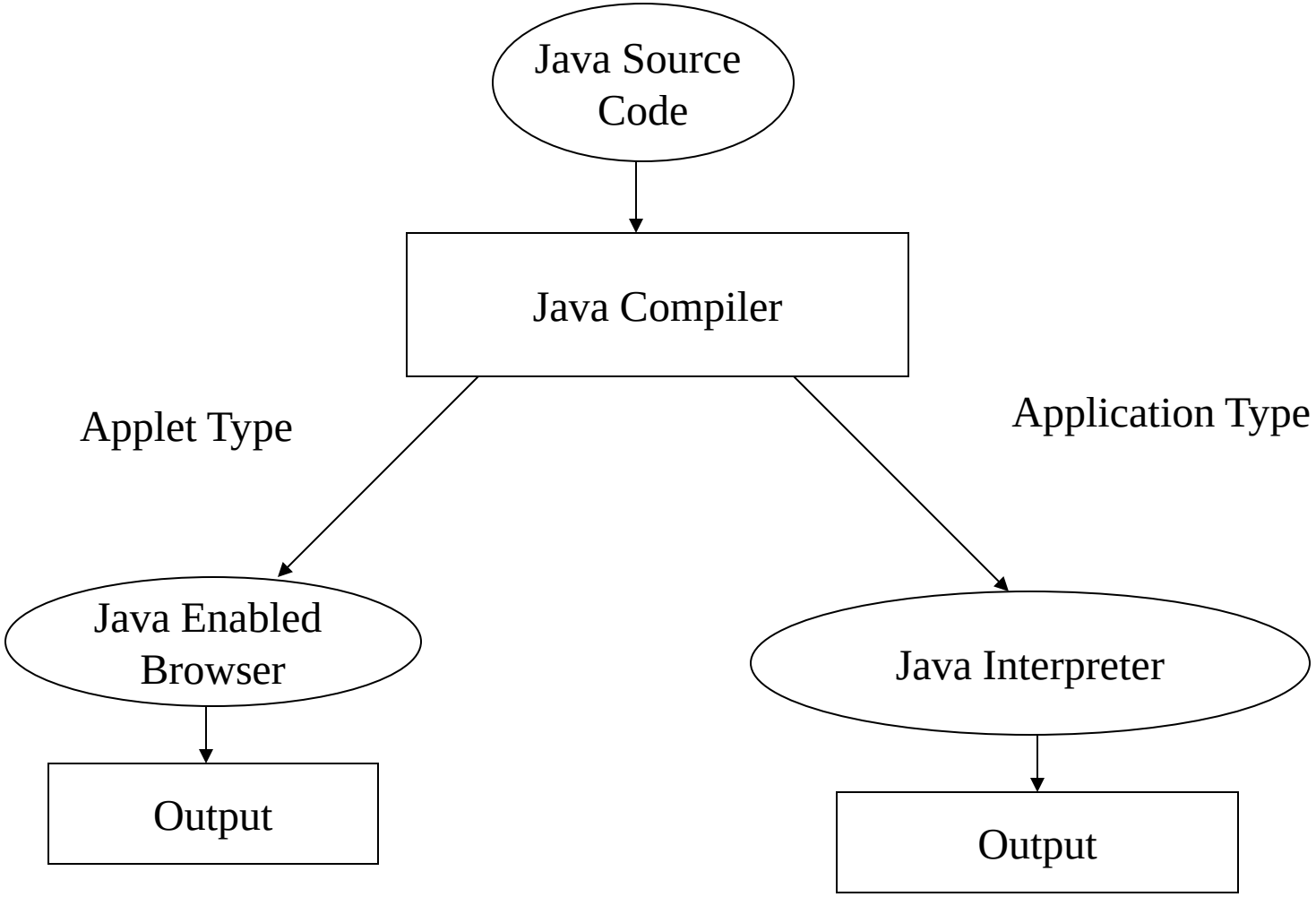
- ✓ Message passing involves specifying the object name, the name of the method and the information to be sent.

  Example : Employee . salary (name);

Advantages of OOP

- ✓ Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- ✓ Inheritance leads to saving of time and higher productivity.
- ✓ The principle of data hiding produces more secure program.
- ✓ It is possible to map objects in the problem domain to those objects in program.
- ✓ It is easy to partition the work in a project based on object.
- ✓ System can easily be upgraded from small to large system.
- ✓ Software complexity can easily handled.

Java Program



First Java Program-Example 1

```
/*This is a simple java program*/
class Example
{
    public static void main (String args[])
    {
        System.out.println (“This is a simple Java program”);
    }
}
```

Simple Java Program-Some important points

- ✓ **public:** Access specifier. main() must be made public, since it must be called by code defined outside it's class.
- ✓ **Static:** It is required because main() is called without creating an object of it's class
- ✓ **String args[]:** An array of objects of type String class. Each object of type string contains a character string. It is used to manipulate command line argument.
- ✓ Java is case sensitive.
- ✓ System ⟶ predefined class that refers to system.

  out ⟶ It is static data member of System class

  println() ⟶ It is a member of out object

Implementing a Java Program

1. Creating a java program
2. Compiling a java program
3. Running the program

Creating a Java Program:

1. All java source file has an extension **.java**.
2. If a program contains multiple classes, the file name must be the class name of the class containing the **main** method. This class must be the only public class.

[Rule: File name should match the name of the public class of a source code.]

Implementing a Java Program

Compiling a Java Program:

- ✓ To compile a java program, execute the java compiler **javac**, specifying the name of the source file on the command line. C:\> javac Example.java
- ✓ When java source code is compiled, each individual class is put into it's own output file named after the class and using the **.class** extension.
- ✓ Each file with **.class** extension contains the bytecode corresponding to that class

Implementing a Java Program

- ✓ To run the program, interpreter **java** is used the name of the class that contains the **main** function.

  c:\> java Example

- ✓ Actually it searches for **Example. class** file.

Constants, Data Types and Variables

Java is a Strongly typed Language

- ✓ Every variable and expression has a strongly defined type.
- ✓ All assignments are checked for type compatibility.
- ✓ Java compiler checks all expressions and parameters to ensure that the types are compatible.

The Primitive Types

- ✓ There are exactly eight primitive data types in Java
- ✓ Four of them represent whole valued signed numbers:
  - ✓ byte, short, int, long
- ✓ Two of them represent floating point numbers:
  - ✓ float, double
- ✓ One of them represents characters:
  - ✓ char
- ✓ And one of them represents boolean values:
  - ✓ boolean

Numeric Primitive Types

- ✓ The difference between the various numeric primitive types is their size, and therefore the values they can store:

Type	Storage	Min Value	Max Value
byte	**8 bits**	**-128**	**127**
short	**16 bits**	**-32,768**	**32,767**
int	**32 bits**	**-2,147,483,648**	**2,147,483,647**
long	**64 bits**	**< -9 x 10^{18}**	**> 9 x 10^{18}**
float	**32 bits**	**+/- 3.4 x 10^{38} with 7 significant digits**	
double	**64 bits**	**+/- 1.7 x 10^{308} with 15 significant digits**	

Character Primitive Type

- ✓ It uses unicode to represent character.
- ✓ The char type is unsigned 16 bit values ranging from 0 to 65536.
- ✓ ASCII still ranges from 0 to 127.

Example:

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

Output: ch1 and ch2: X Y

Character Primitive Type

Example:

```
class test
{
    public static void main (String args[])
    {
        char ch1;
        ch1= ‘X’;
        Sytem.out.println (“ch contains “+ch1);
        ch1++;
        System.out.println (“ch1 is now “ + ch1);
    }
}
```

Output:

ch1 contains X

Ch1 is now Y

Booleans

- ✓ Size is 1 bit – two value: true and false.
- ✓ This type is returned by all relational operators.
- ✓ Example:

```
boolean b;
b= true;
1. System.out.println("b is "+b);
2. System.out.println("10>9 is " +(10>9));
```

Output:

```
b is true
10>9 is true
```

Literals

- ✓ Integer Literals

  1. base 10 – 1,2,43 etc.
  2. base 8 – octal values are denoted in java by a leading 0.
  3. base 16 – hexadecimal values are denoted by leading 0x or 0X.

- Any whole number is by default integer (32 bits).
- To specify a long literal, the number should appended with an upper- or lowercase L.

Literals

- ✓ Floating point Literals

  1. Standard Notation – 3.14159, 0.6667, 2.0 etc.
  2. Scientific Notation – 6.022E23, 2e+100.

- Floating point literals are by default of type double.
- To specify a float literal, we must append an F or f to the constant.
- ✓ Boolean Literals
- Two values – true and false.
- True is not equal 1 and false is not equal to 0.
- They can be assigned to variable declared as boolean.

Literals

- ✓ Character Literals:
- A literal character is represented inside a pair of single quotes.

Escape sequence	Unicode Representation	Description
1. \’	\u0027	Single quote
2. \”	\u0022	Double quote
3. \\	\u005c	Backslash
4. \r	\u000d	Carriage Return
5. \n	\u000a	New line
6. \f	\u000b	Form feed
7. \t	\u0009	Tab
8. \b	\u0008	Backspace
9. \ddd		Octal Character
10. \uxxxx		Hexadecimal Unicode character

Literals

- ✓ String Literals
- A sequence of characters between a pair of double quotes.
- In java string must begin and end on the same line.

Variables

- ✓ Variable is a name for a location in memory.
- ✓ Declaring a variable:

  type identifier [=value][,identifier [=value]….];
- ✓ The initialization expression must result in a value of the same or compatible type as that specified for the variable.
- ✓ When a variable is not initialized, the value of that variable is undefined.

Scope and Lifetime of a variable

- ✓ A block begins with an opening curly brace and ends by a closing curly brace.
- ✓ A block determines scope, that defines which objects are visible to other parts of your program.
- ✓ Variables declared within a block localize themselves.
- ✓ In case of nested block, the outer block encloses the inner block. The variables declared in the outer block is visible to the inner block but the reverse is not true.
- ✓ A variable will not hold it’s value once it has gone out of it’s scope.
- ✓ In an inner block, it is not possible to declare a variable of the same name as in outer block.

Scope and Lifetime of a variable

Example:

```
public static void main( String args[])
{
    int x =10;
    if ( x == 10)
    {
        int y =20;
        System.out.println(“x and y: “ +x +” “+y);
        x= y * 2;
    }
    y= 100;  //Error
    System.out.println (“x is “+x);
}
```

Java Operators

Operators in Java

Classified into four groups:

1. Arithmetic Operator
2. Bitwise Operator
3. Relational Operator
4. Logical Operator

Arithmetic Operators

Addition	**+**
Subtraction	**-**
Multiplication	*****
Division	**/**
Remainder	**%**
Increment	**++**
Addition Assignment	**+=**
Subtraction Assignment	**-=**
Multiplication Assignment	***=**
Division Assignment	**/=**
Modulus Assignment	**%=**
Decrement	**--**

Arithmetic Operators

- ✓ If either or both operands associated with an arithmetic operator are floating point, the result is a floating point.
- ✓ % operator applies both to floating-point type and integer types.
- ✓ Example:

```
class modulus
{
    public static void main (String args [])
    {
        int x = 42;
        double y = 42.3;
        System.out.println("x mod 10 =" + x%10);
        System.out.println("y mod 10 = " + y%10);
    }
}
```

Output:

```
x mod 10 =2
y mod 10 = 2.3
```

Increment and Decrement

- ✓ The increment and decrement operators are arithmetic and operate on one operand
- ✓ The *increment operator* (++) adds one to its operand
- ✓ The *decrement operator* (--) subtracts one from its operand
- ✓ The statement count++;

is functionally equivalent to count = count + 1;

Increment and Decrement

- ✓ The increment and decrement operators can be applied in *prefix form* (before the operand) or *postfix form* (after the operand)
- ✓ When used alone in a statement, the prefix and postfix forms are functionally equivalent. That is,

```
count++;
```

is equivalent to ++count;

Increment and Decrement

- ✓ When used in a larger expression, the prefix and postfix forms have different effects
- ✓ In both cases the variable is incremented (decremented)
- ✓ But the value used in the larger expression depends on the form used.
- ✓ If count currently contains 45, then the statement

```
total = count++;
```

assigns 45 to total and 46 to count

- ✓ If count currently contains 45, then the statement

```
total = ++count;
```

assigns the value 46 to both total and count

Assignment Operators

- ✓ The right hand side of an assignment operator can be a complex expression
- ✓ The entire right-hand expression is evaluated first, then the result is combined with the original variable

Assignment Operators

- ✓There are many assignment operators, including the following:

Operator	Example	Equivalent To
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Bitwise Operator

~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Shift Right
>>>	Shift Right zero fill
<<	Shift left
& =	Bitwise AND Assignment
|=	Bitwise OR Assignment
^=	Bitwise XOR Assignment
>>=	Shift Right Assignment
>>>=	Shift Right zero fill Assignment
<<=	Shift Left Assignment

Bitwise Operator

- ✓ Applied to integer type – long, int, short, byte and char.

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

The Left Shift

- ✓ byte a=8, b=24;
  int c;
  c=a<<2; → 00001000 << 2 = 00100000=32
- ✓ Java's automatic type conversion produces unexpected result when shifting **byte** and **short** values.
  Example:
  byte a = 64, b;
  int i;
  i = a<<2;
  b= (byte) (a<<2);
  i → 00000000 00000000 00000001 00000000 = 256
  b → 00000000 = 0
- ✓ Each left shift double the value which is equivalent to multiplying by 2.

The Right Shift

- ✓ byte a=8, b=24;

  int c ;

  c=a>>2; ⟶ 00001000 >> 2= 00000010=2

- ✓ Use sign extension.
- ✓ Each time we shift a value to the right, it divides that value by two and discards any remainder.

The Unsigned Right Shift

- ✓ byte a=8, b=24;

  int c;

  c=a>>>1 ⟶ 00001000 >>> 1= 00000100=4

Relational operators

- ✓ > greater than
- ✓ >= greater than or equal to
- ✓ < less than
- ✓ <= less than or equal to
- ✓ = = equal to
- ✓ != not equal to

- ✓ The outcome of these operations is a boolean value.
- ✓ = = , != can be applied to any type in java.
- ✓ Only numeric types are compared using ordering operator.

Relational Operator

- ✓ int done;

  …….

  if(!done) …. // Valid in C /C++ but not in java

  if(done)….

- ✓ if (done == 0)….

  if(done!=0) …….. //Valid in Java

Boolean Logical Operator

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

Boolean Logical Operator

- ✓ The logical boolean operators &, | and ^ operates in the same way that they operate on the bits of integer.

a	b	a & b	a | b	a ^ b	!a
true	true	true	True	false	false
true	false	false	true	true	false
false	true	false	true	ture	true
false	false	false	false	false	true

Short Circuit Logical Operators

- ✓ || Short circuit logical OR
- ✓ && Short circuit logical AND
- ✓ If the left operand is sufficient to determine the result, the right operand is not evaluated

```
if (demon !=0 && num / demon >10)
```

This type of processing must be used carefully

The Conditional Operator

- ✓ The conditional operator is similar to an if-else statement, except that it forms an expression that returns a value
- ✓ For example:
- ✓ `larger = ((num1 > num2) ? num1 : num2);`
- ✓ If num1 is greater that num2, then num1 is assigned to larger; otherwise, num2 is assigned to larger
- ✓ The conditional operator is *ternary* because it requires three operands

Operator Precedence

Highest

1.	( )	[]	.	
2.	++	--	~	!
3.	*	/	%	
4.	+	-		
5.	>>	>>>	<<	
6.	>	>=	<	<=
7.	= =	!=		
8.	&			
9.	^			
10.	|			
11.	&&			
12.	||			
13.	?:			
14.	=	op=		

Lowest

Decision Making, Branching and Looping

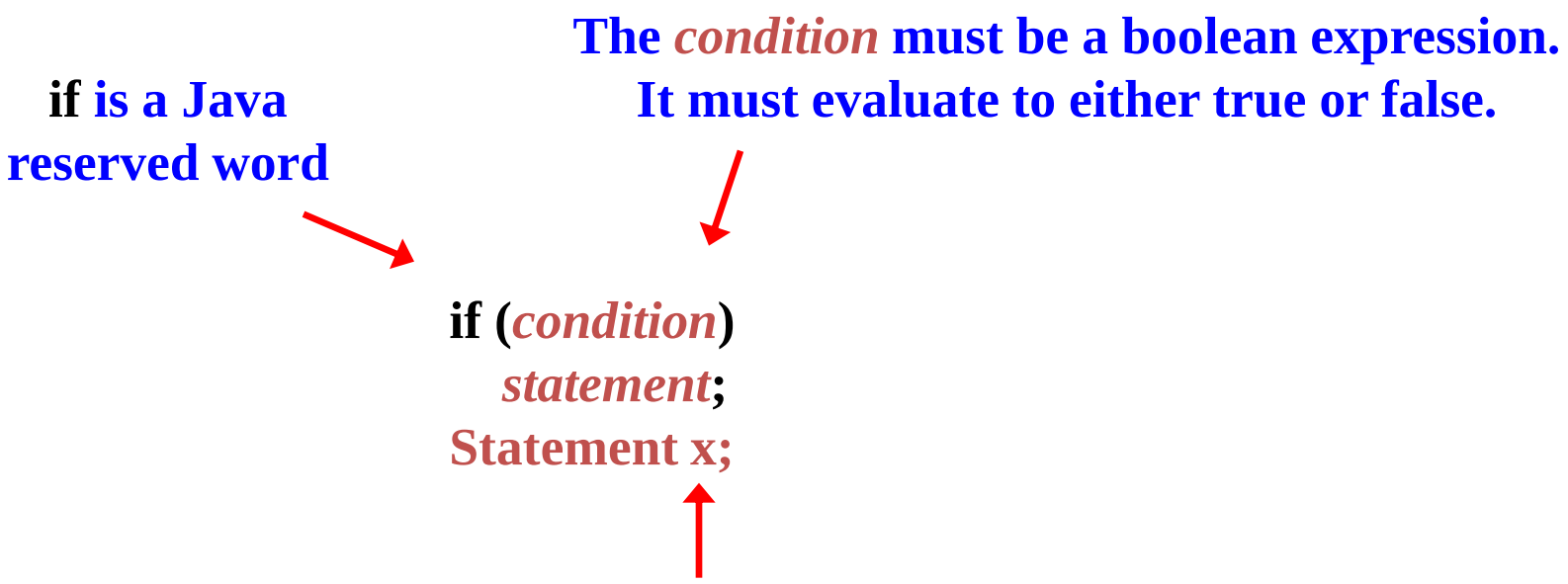
Decision Making and Branching

- ✓ When a program breaks the sequential flow and jumps to another part of the code, it is known as branching. When branching is done on a condition it is known as conditional branching.
- ✓ Three decision making statements:

  1. if statement
  2. switch statement
  3. conditional operator statement

The if Statement

✓ The *if statement* has the following syntax:



The if-else Statement

- ✓ An *else clause* can be added to an if statement to make an *if-else statement*

```
if ( condition )
   statement1;
else
   statement2;
Statement x;
```

- ✓ If the `condition` is true, `statement1` is executed; if the condition is false, `statement2` is executed

- ✓ One or the other will be executed, but not both

Nested if….Else Statements

- ✓ The if..else statement can be contained in another if or else statement.

```
if (test condition1)
{
        if (test condition2)
                statement-1;
        else
                statement-2;
}
else
        statement-3;

statement-x;
```

Nested if….Else Statements

- ✓ An else clause is matched to the last unmatched if (no matter what the indentation implies!)

- ✓ Example:

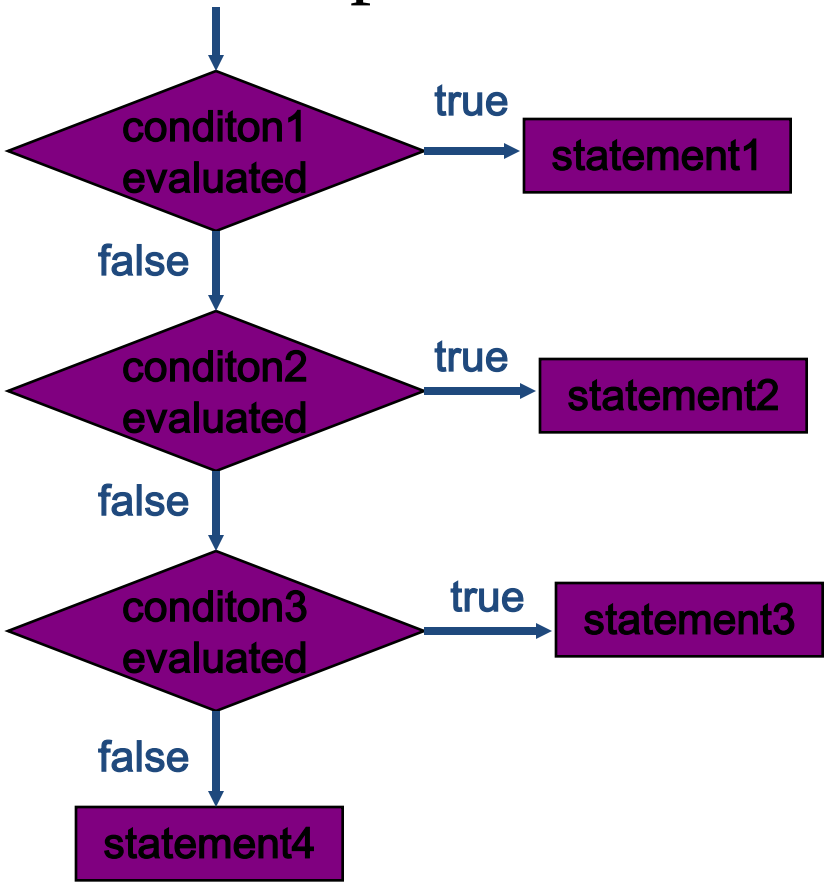
```
if(female)
        if(bal>5000)
                bon = 0.05 * bal;
else
        bon = 0.02 * bal;
bal = bal + bon;
```

- ✓ Braces can be used to specify the if statement to which an else clause belongs

Multiway Selection: Else if

- ✓ Sometime you want to select one option from several alternatives

```
if (conditon1)
    statement1;
else if (condition2)
    statement2;
else if (condition3)
    statement3;
else
    statement4;
```



Else if example

```
double numberGrade = 83.6;
char letterGrade;

if (numberGrade >= 89.5) {
    letterGrade = ‘A’;
} else if (numberGrade >= 79.5) {
    letterGrade = ‘B’;
} else if (numberGrade >= 69.5) {
    letterGrade = ‘C’;
} else if (numberGrade >= 59.5) {
    letterGrade = ‘D’;
} else {
    letterGrade = ‘F’;
}

System.out.println(“My Grade is ” + numberGrade + “, ” + letterGrade);
```

Output:

```
My Grade is 83.6, B
```

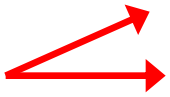
The switch Statement

- ✓ The *switch statement* provides another means to decide which statement to execute next
- ✓ The switch statement evaluates an expression, then attempts to match the result to one of several possible *cases*
- ✓ The expression of a switch statement must result in an *integral type*, meaning an int or a char
- ✓ Each case contains a value and a list of statements
- ✓ The flow of control transfers to statement associated with the first value that matches

The switch Statement

- ✓The general syntax of a switch statement is:

switch and case are reserved words



```
switch (expression) {
   case value1:
      statement-list1
   case value2:
      statement-list2
   case value3:
      statement-list3
}
```

If *expression* matches *value2*, control jumps from here

The switch Statement

- ✓ Often a *break statement* is used as the last statement in each case's statement list
- ✓ A break statement causes control to transfer to the end of the switch statement
- ✓ If a break statement is not used, the flow of control will continue into the next case
- ✓ Sometimes this can be appropriate, but usually we want to execute only the statements associated with one case

The switch Statement

- ✓ A switch statement can have an optional *default case*
- ✓ The default case has no associated value and simply uses the reserved word default
- ✓ If the default case is present, control will transfer to it if no other case value matches
- ✓ If there is no default case, and no other value matches, control falls through to the statement after the switch

Switch example

```
char letter = 'b';

switch (letter) {
    case 'a':
        System.out.println("A");
        break;
    case 'b':
        System.out.println("B");
        break;
    case 'c':
        System.out.println("C");
        break;
    case 'd':
        System.out.println("D");
        break;
    default:
        System.out.println(”?");
}
```

B

```
char letter = 'b';

switch (letter) {
    case 'a':
        System.out.println("A");
    case 'b':
        System.out.println("B");
    case 'c':
        System.out.println("C");
        break;
    case 'd':
        System.out.println("D");
        break;
    default:
        System.out.println(”?");
}
```

B

C

The Conditional Operator

- ✓ Java has a *conditional operator* that evaluates a boolean condition that determines which of two other expressions is evaluated
- ✓ The result of the chosen expression is the result of the entire conditional operator
- ✓ Its syntax is:
- ✓ *condition* ? *expression1* : *expression2*
- ✓ If the *condition* is true, *expression1* is evaluated; if it is false, *expression2* is evaluated

The Conditional Operator

- ✓ The conditional operator is similar to an if-else statement, except that it forms an expression that returns a value
- ✓ For example:
- ✓ larger = ((num1 > num2) ? num1 : num2);
- ✓

```
if (num1 > num2)
    larger = num1;
else
    larger = num2;
```

- ✓ The conditional operator is *ternary* because it requires three operands

The Conditional Operator

- ✓ Another example:

System.out.println ("Your change is " + count +
((count == 1) ? “Rupee" : “Rupees"));

If count equals 1, then "Dime" is printed

If count is anything other than 1, then "Dimes" is printed

Repetition Statements

- ✓ *Repetition statements* allow us to execute a statement multiple times
- ✓ Often they are referred to as *loops*
- ✓ Like conditional statements, they are controlled by boolean expressions
- ✓ Java has three kinds of repetition statements:
  - ✓ the *while loop*
  - ✓ the *do loop*
  - ✓ the *for loop*
- ✓ The programmer should choose the right kind of loop for the situation

The while Statement

✓ The *while statement* has the following syntax:

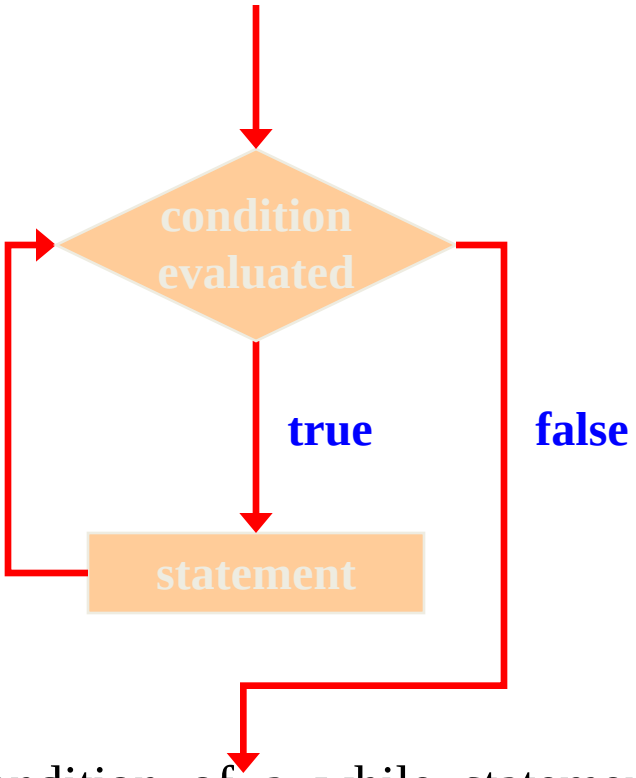
while is a reserved word

while (*condition*)
***statement*;**

If the *condition* is true, the *statement* is executed.
Then the *condition* is evaluated again.

The *statement* is executed repeatedly until the *condition* becomes false.

Logic of a while Loop



Note that if the condition of a while statement is false initially, the statement is never executed. Therefore, the body of a while loop will execute zero or more times

while Loop Example

```
final int LIMIT = 5;
int count = 1;


while (count <= LIMIT) {


    System.out.println(count);
    count += 1;
}
```

Output:

1
2
3
4
5

Infinite Loops

- ✓ The body of a while loop eventually must make the condition false
- ✓ If not, it is an *infinite loop*

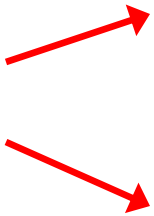
Nested Loops

- ✓ Similar to nested if statements, loops can be nested as well
- ✓ That is, the body of a loop can contain another loop
- ✓ Each time through the outer loop, the inner loop goes through its full set of iterations

The do Statement

✓ The *do statement* has the following syntax:

do and while are reserved words



```
do{
   statement;
} while (condition);
```

The *statement* is executed once initially, and then the *condition* is evaluated

The *statement* is executed repeatedly until the *condition* becomes false

do-while Example

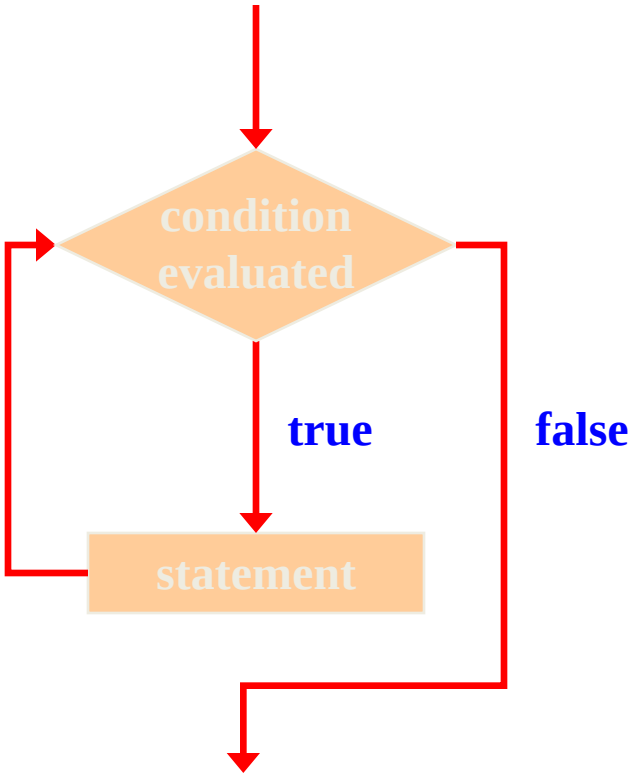
```
final int LIMIT = 5;
int count = 1;


do {
    System.out.println(count);
    count += 1;
} while (count <= LIMIT);
```

Output:

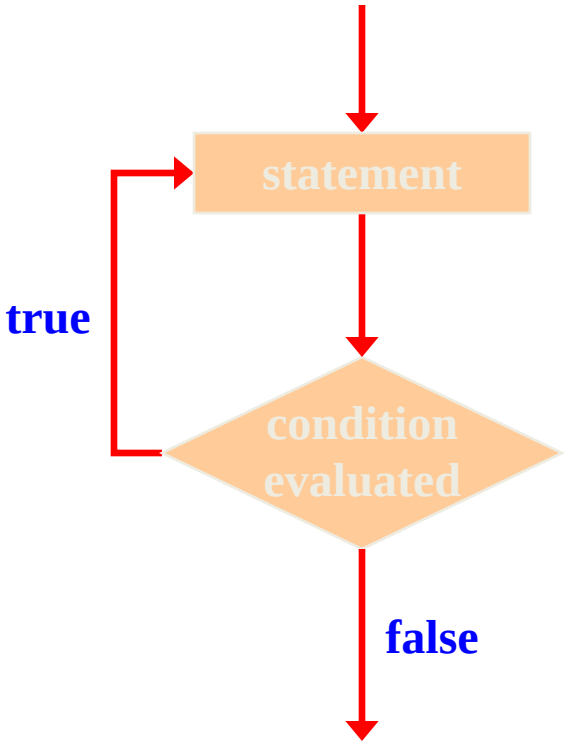
```
1
2
3
4
5
```

Comparing while and do

while loop

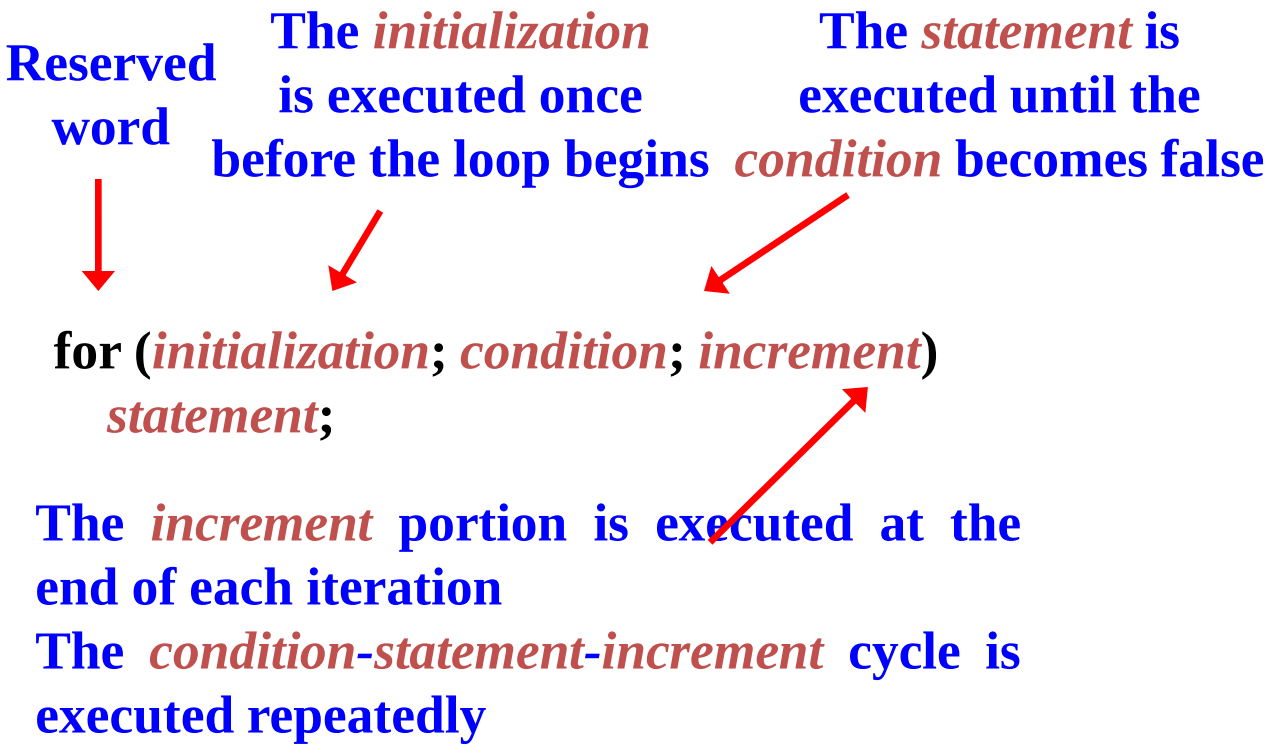


do loop



The for Statement

✓ The *for statement* has the following syntax:

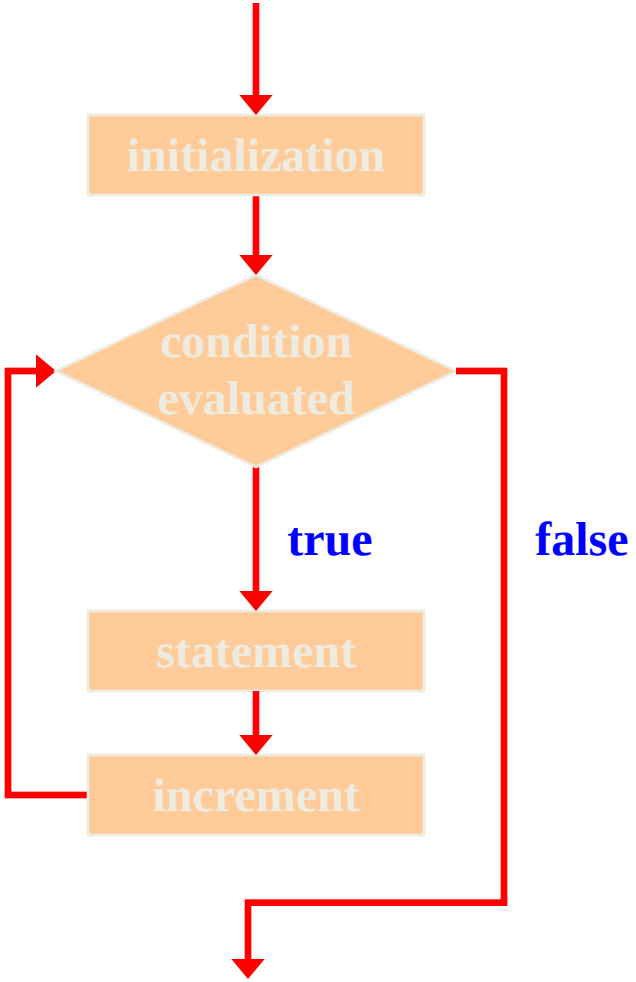


The for Statement

- ✓ A for loop is functionally equivalent to the following while loop structure:

```
initialization;
while (condition) {
    statement;
    increment;
}
```

Logic of a for loop



The for Statement

- ✓ Like a while loop, the condition of a for statement is tested prior to executing the loop body
- ✓ Therefore, the body of a for loop will execute zero or more times
- ✓ It is well suited for executing a loop a specific number of times that can be determined in advance

for Example

```
final int LIMIT = 5;

for (int count = 1; count <= LIMIT; count++) {
    System.out.println(count);
}
```

Output:

```
1
2
3
4
5
```

The for Statement

- ✓ Each expression in the header of a for loop is optional
  - ✓ If the *initialization* is left out, no initialization is performed
  - ✓ If the *condition* is left out, it is always considered to be true, and therefore creates an infinite loop
  - ✓ If the *increment* is left out, no increment operation is performed
- ✓ Both semi-colons are always required in the for loop header

Choosing a Loop Structure

- ✓ When you can't determine how many times you want to execute the loop body, use a while statement or a do statement
  - ✓ If it might be zero or more times, use a while statement
  - ✓ If it will be at least once, use a do statement
- ✓ If you can determine how many times you want to execute the loop body, use a for statement

Import Statement
&
Taking Input

Import Statement

- ✓ A java package is a collection of related classes.
- ✓ In order to access the available classes in the package, the program must specify the complete dot seperated package path.
- ✓ The general format:

  import package-level1.[package-level2.]classname|*
- ✓ Two form of import statement:

  1. import package.class;
  2. import package.*;

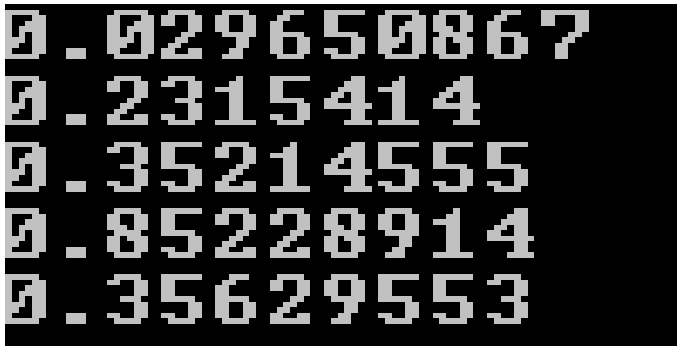
Example:

```
import java.util.Scanner;
import java.util.*;
```

Example

```
import java.util.Random;

class random
{
	public static void main(String args[])
	{
		Random r = new Random();
		int i;
		float v;

		for(i=0;i<5;i++)
		{
			v=r.nextFloat();
			System.out.println(v);
		}

}
```

Output:



- ✓java.lang in automatically imported with every java program.
- ✓System.out.println() belongs to java.lang.

Predefined Stream

Stream:

-A stream is an abstraction that either produces or consumes information.

-It is linked to a physical device by a java I/O system.

-They hide the details of the physical device to which they are connected.

System:

-System is a predefined class included in the package **java.lang.** (Imported automatically by all java programs).

-It includes three predefined stream variables:

1. in 2. out. 3. err

The Predefined Streams

1. **System.out:** refers to the standard output stream. It is an object of type PrintStream.
2. **System.in:** refers to the standard input stream. It is an object of type InputStream.
3. **System.err:** refers to the standard error stream. It is an object of type PrintStream.

Note: they are defined as **public** and **static**.

InputStream:

- This class in included in the package **java.io**.
- It has some subclasses that handle the differences between various devices.
- Includes some methods that the subclasses will implement.

Scanner Class

- ✓ Scanner class is used to read input from the keyboard, a file, a string or any source that implements the **Readable** or **ReadByteChannel**.
- ✓ Scanner can be created for a string, an InputStream, or any object that implements **Readable** or **ReadByteChannel** interface.
- ✓ Scanner class is under the package of java.util
- ✓ Added in J2SE 5.

Readable Interface:

- ✓ It is added by J2SE.
- ✓ It is included in Java.lang.
- ✓ It defines one method:

  int read(CharBuffer buf) throws IOException

It reads characters into buf. It returns the number of characters read or -1 if an EOF is encountered.

Taking Input from the Keyboard

- ✓ First, Scanner class is connected to System.in which is an object of type InputStream.
- ✓ Then, it uses it's internal functions to read from System.in

Example:

```
Scanner test = new Scanner(System.in);
```

Calls the constructor Scanner(InputStream)

Take an input from the keyboard-1

```
import java.util.*;
public static void main(String[] args) {
     int value;
   System.out.print(“Enter an Integer number:");
     Scanner tmp = new Scanner(System.in);
     value=tmp.nextInt();
           System.out.println(“You have entered: ”+value);

     else
            System.out.println(“Not an Integer”);
}
```

Taking Input from the KeyBoard-2

```
import java.util.*;
class input
{
    public static void main(String args[])
    {
        Scanner tmp = new Scanner(System.in);
        float i;

        while(tmp.hasNextFloat())
        {
            i=tmp.nextFloat();
            System.out.println("The Number: ",i);
        }

    }
}
```

Scanning Basics

- ✓ A Scanner reads tokens from the underlying source.
- ✓ A token is a portion of input that is delineated by a set of delimiters, which is by default whitespace.
- ✓ A token is read by matching it with a particular regular expression.
- ✓ Scanner follow the procedure below:

  1. Determine if a specific type of input is available by calling one of the **hasNextX** methods.

  2. If input is available, read it by calling one of **nextX** method.

  3. Repeat the process until the input is exhausted.

Note: if nextX() method does not find a matching token, it throws a **NoSuchElementException**.

Important Methods of Scanner Class

public Scanner(InputStream in) // Scanner(): convenience constructor for an InputStream object

boolean hasNext() //Return true if another token of any type is available to be read.

boolean hasNextBoolean() //Return true if a boolean value is available to read.

boolean hasNextByte() //Return true if a byte value is available to read.

boolean hasNextShort() //Return true if a byte value is available to read.

boolean hasNextInt() //Return true if a int value is available to read.

boolean hasNextLong() //Return true if a long value is available to read.

boolean hasNextFloat() //Return true if a float value is available to read.

boolean hasNextDouble() //Return true if a double value is available to read.

Important Methods of Scanner Class

int nextInt() // return next token as int value.

short nextShort() // return next token as short value.

byte nextByte() // return next token as byte value.

long nextLong() // return next token as long value.

double nextDouble() // return next token as double value

float nextFloat() // return next token as float value

String next() //return next token of any type from the input source

String nextLine() // return the next line of input as a string

Homework (Math Library)

- ✓ Suppose you are given the following
  - ✓ double a=56.34, b=6.58334, c=-34.4265;
- ✓ Calculate the following value:
  - ✓ Print a random number.
  - ✓ Find the absolute value of the variable c
  - ✓ Find the square root of a
  - ✓ Find the maximum value between a and b
  - ✓ Calculate the value a^b
  - ✓ Round the number a
  - ✓ Calculate the value of $\sqrt{(a^2+b^2)}$
  - ✓ Find the floor, ceil and round value of b and c
  - ✓ Find the radian value of a.
  - ✓ Find the sin value of a where a represents the degree

Array in Java

One-Dimensional Array

- ✓ Creating an array is a two steps process:

  1. type var_name[];

  2. var_name = new type [size];

- ✓ Example: int month_days [];

  month_days = new int [12];

  - first line declares *month_days* as an array variable, no array actually exists.

  - Actual, physical array of integers, is allocated using **new** and assign it to *month_days*.

  -The elements of array are automatically initialized to 0.

- ✓ int month_days [] = new int[12];

One-Dimensional Array

- ✓ Once array is created, a specific element in the array can be accessed by specifying it's index within square brackets.

  Example: month_days[0]=31;

  month_days[1]=28;

- ✓ Arrays can be initialized when they are declared.

  Example:

  int month_days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

Features of Java to Manipulate Array

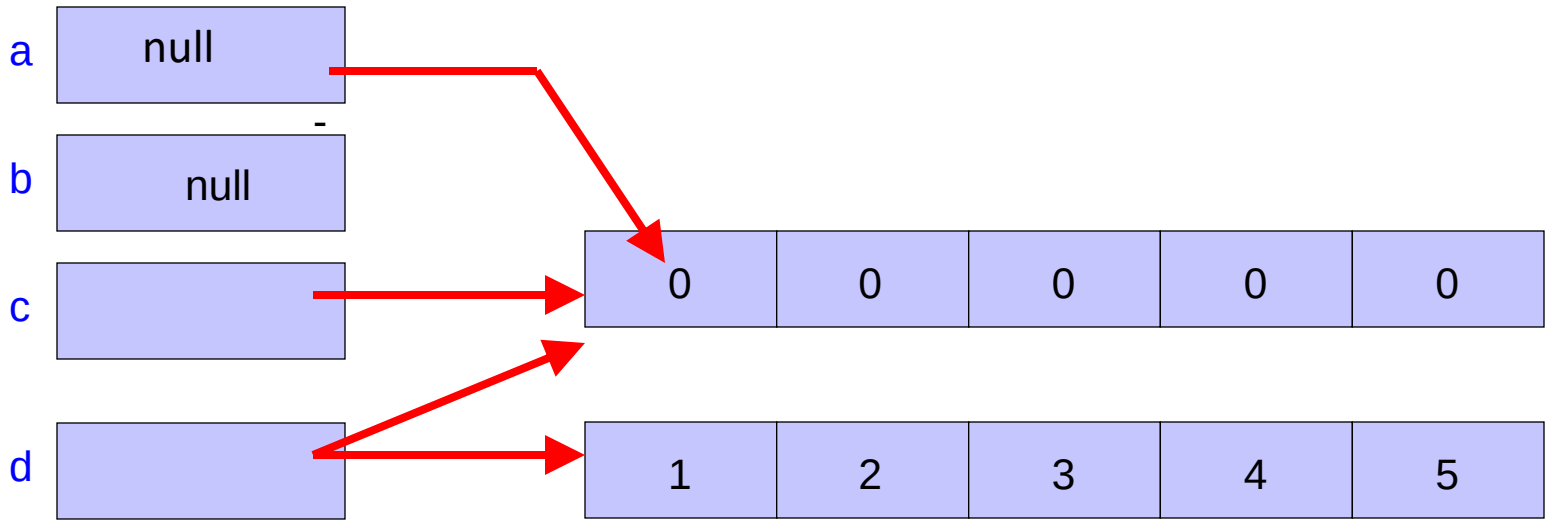
- ✓ All arrays are dynamically allocated.
- ✓ Size of the array can be specified at the runtime.
- ✓ Index type is integer and the index range must be 0 to n-1, where n is the number of elements.
- ✓ Java runtime system will make sure that all array indices are in the correct range. Incorrect reference will generate ArrayIndexOutofBoundsException.

Example

```
import java.util.Scanner;

class array_avg
{
	public static void main(String args[])
	{
		int i,sum=0;
		float avg;

		int a[] = new int[5];

		Scanner test=new Scanner(System.in);

		System.out.println("Enter the input:");

		for(i=0;i<5;i++)
		{
		a[i]=test.nextInt();
		sum=sum+a[i];
		}
		avg=sum/5;

		System.out.println("The average value is: " + avg);

	}
}
```

Array in Java

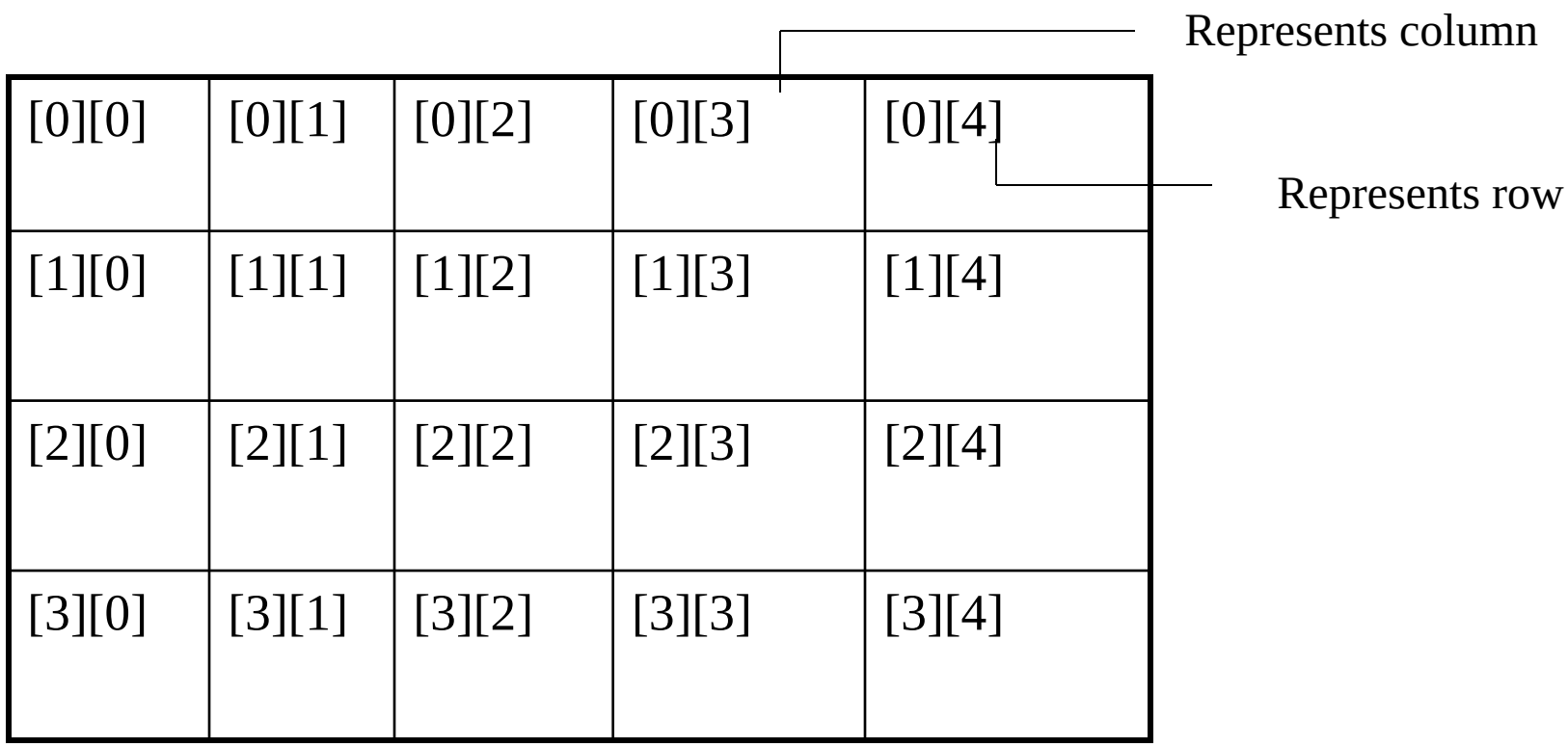
```
int a[];
int b[] = null;
int c[] = new int[5];
int d[] = { 1, 2, 3, 4, 5 };
a = c;
d = c;
```



Multi-Dimensional Arrays

- ✓ A two-dimensional array can be declared as

  int twoD [] [] = new int[4][5];



Example

```
int tmp[][] = new int[3][3];
int i,j;

Scanner test=new Scanner(System.in);

for(i=0;i<3;i++)
        for(j=0;j<3;j++)
        {
                System.out.print("Enter Input:");

                tmp[i][j]=test.nextInt();
        }

for(i=0;i<3;i++)
{
        for(j=0;j<3;j++)
                System.out.print(tmp[i][j] + " ");

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

Multi-dimensional Array

- ✓ int twoD [][] = new int[4][];

  twoD[0] = new int[5];

  twoD[1] = new int[4];

  twoD[2] = new int[3];
- ✓ int m [][]={{1,2},{2,3}};

Example

```
int a[][]=new int[3][];
a[0]=new int[1];
a[1]=new int[2];
a[2]=new int[3];

int i,j,k=0;
for(i=0;i<3;i++)
   for(j=0;j<i+1;j++)
   {
            a[i][j]=k;
            k++;
   }
for(i=0;i<3;i++)
{
   for(j=0;j<i+1;j++)
            System.out.print(a[i][j]+" ");
   System.out.println();
}
```

Output:

0

1 2

3 4 5

Alternative Array Declaration Syntax

- ✓ Type [] var_name;
- ✓ Example: int [] a = new int[3];
- ✓ int [] num1, num2, num3;

  same as

  int num1[], num2[], num3[];

Array as an Object

- ✓ An array is implemented as an object.
- ✓ It has a special attribute: **length** instance variable.
- ✓ Example:

```
int a1[] = new int[10];
int a2[] = {3, 5, 7, 9, 11, 13, 15, 17};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is: "+a1.length);
System.out.println("length of a2 is: "+a2.length);
System.out.println("length of a3 is: "+a3.length);
```

Java.util.Arrays class

- Arrays.sort(A)
- Arrays.toString(A)
- Arrays.equal(A1,A2)
- Arrays.binarySearch(A,item)