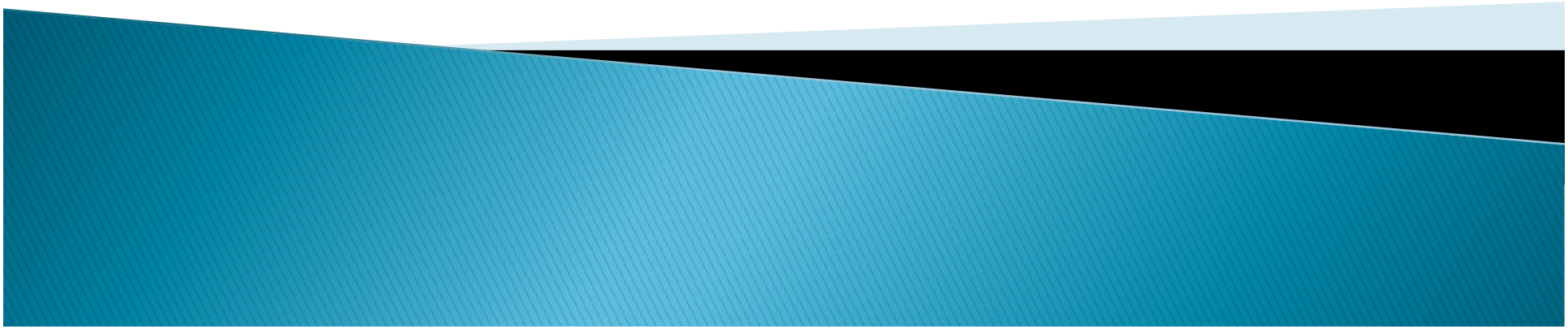
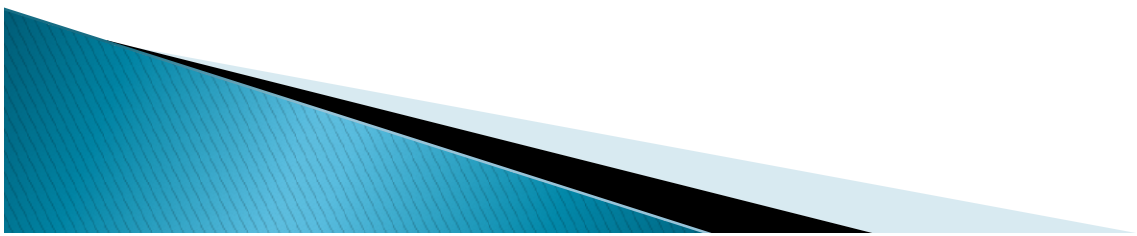
JAVA PROGRAMS

Presented by
Reetanjali Panda
Lecturer,UCPES,Bam



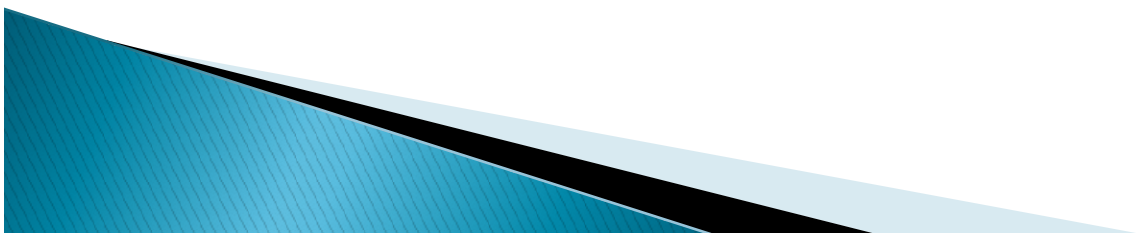
Type of Java Programs

1. Application Program: It is nothing but a program that runs on your computer, under the operating system of your computer.
2. Applets or Intelligent Program: These are mainly used for internet applications. These programs run on a web page and require Java enabled web browser or Applet viewer. These are the Java programs that appear to be embedded in a web document.
3. Servlets: It is a mini server side program similar to an applet. It enables to extend the functionality of web servers. It is particularly used for producing dynamic web contents.

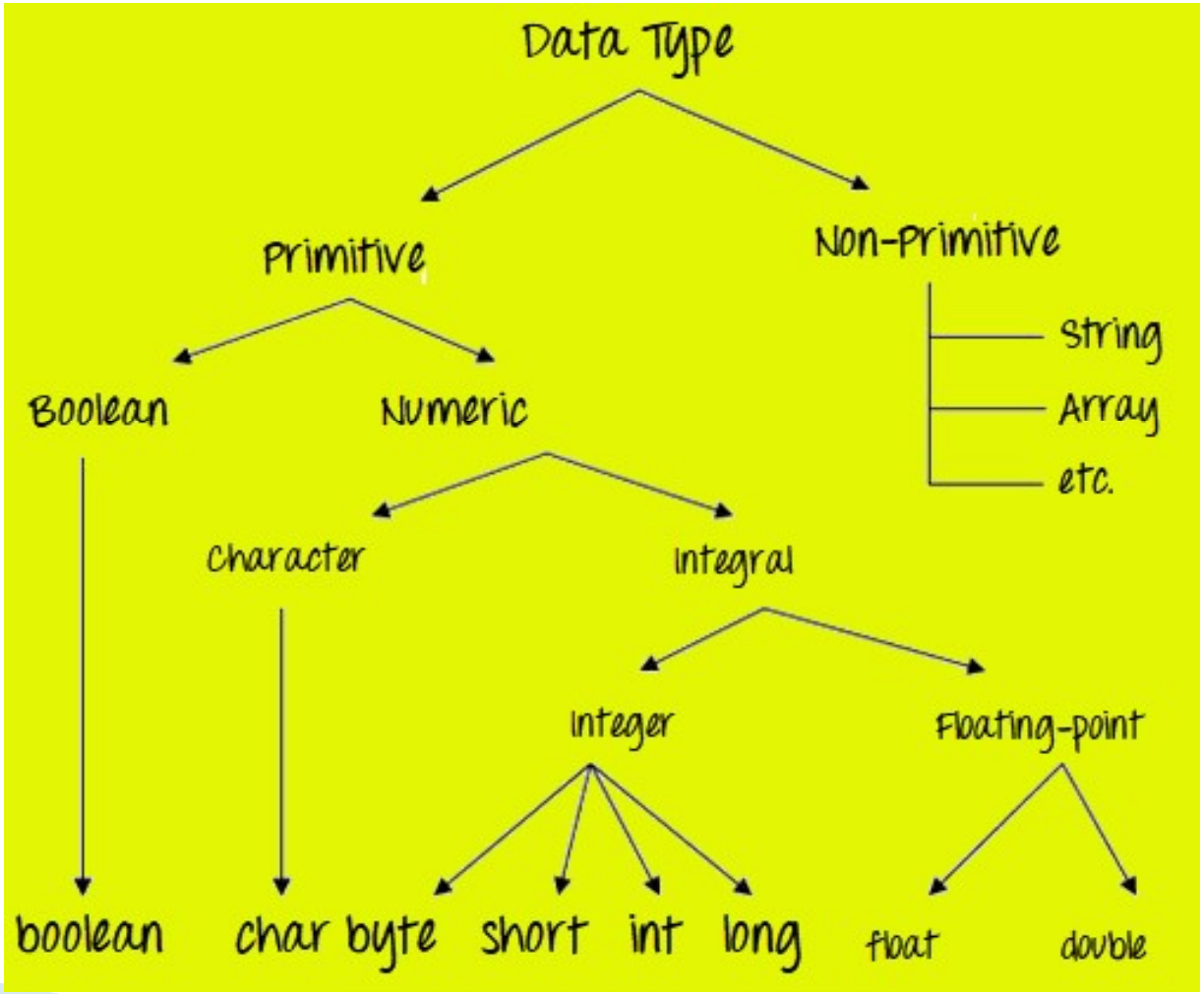


Data Types in Java

- Data types specify size and the type of values that can be stored in an identifier.
- In java, data types are classified into two catagories :
- **Primitive Data Types** :- These are also known as standard data type or built in data type. The java compiler contains detailed instructions on each legal operations supported by the data type. They include integer, character, boolean, and float etc.
- **Non-primitive Data Types** :- These are also known as derived data types or reference datatypes which are built on primitive datatypes. They include classes, arrays and interfaces.

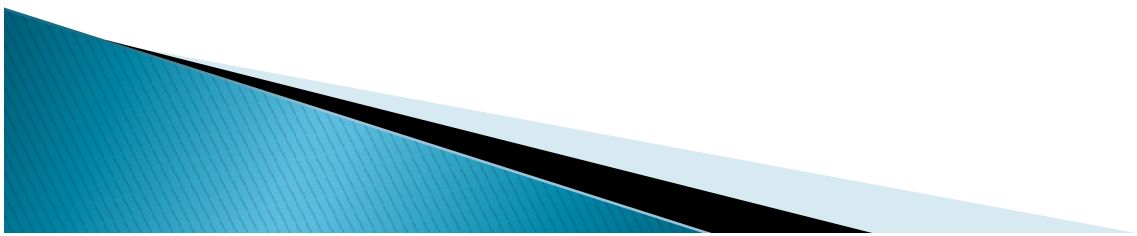


Data Types in Java(Cont.)



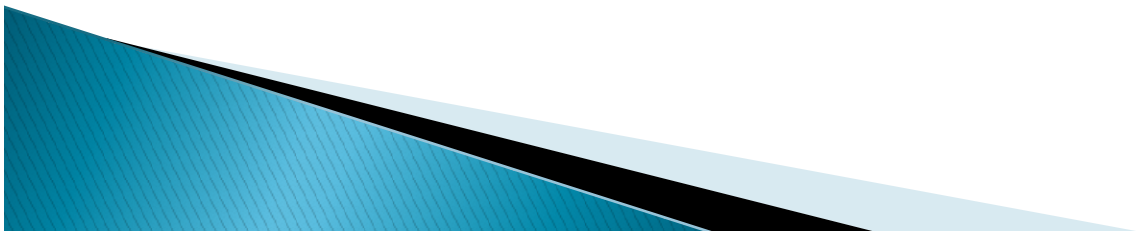
Data Types in Java(Cont.)

- **Primitive Data Types**
- Primitive Data Types are predefined and available within the Java language. Primitive values do not share state with other primitive values.
- There are 8 primitive types: byte, short, int, long, char, float, double, and boolean.



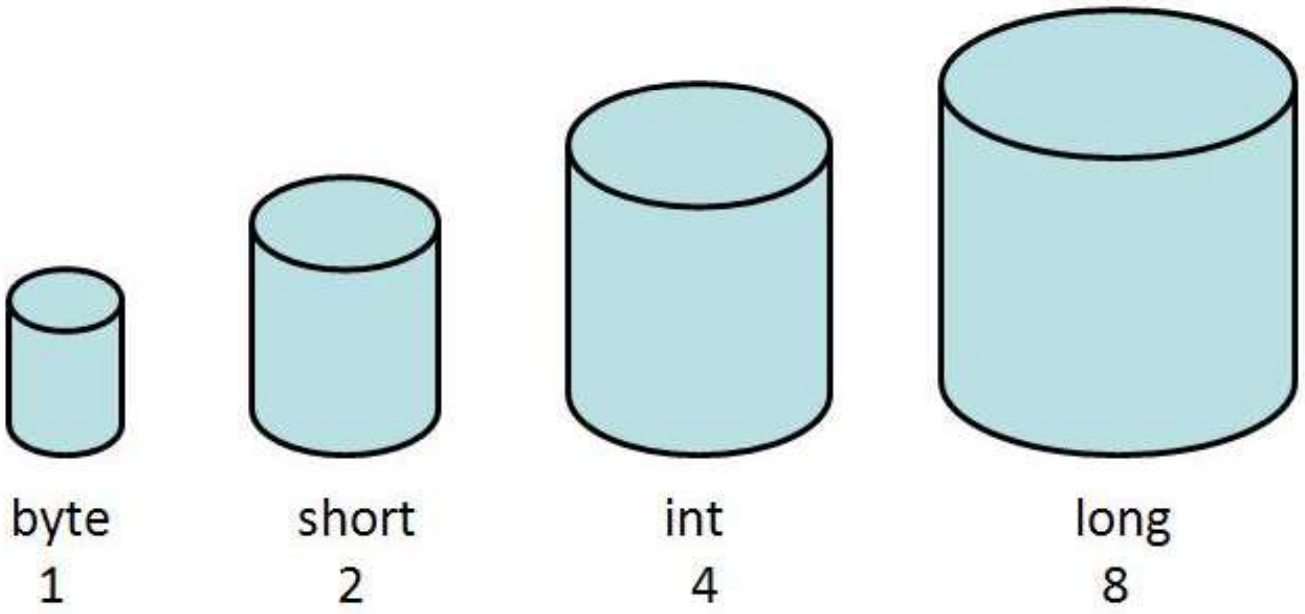
Data Types in Java(Cont.)

Java Data Types		
Data Type	**Default Value**	**Default size**
byte	0	1 byte
short	0	2 bytes
int	0	4 bytes
long	0L	8 bytes
float	0.0f	4 bytes
double	0.0d	8 bytes
boolean	false	1 bit
char	'\u0000'	2 bytes



Data Types in Java(Cont.)

- Integer data types



Data Types in Java(Cont.)

- This group includes byte, short, int, long datatypes
- **byte** : It is 8 bit integer data type. Value range from -128 to 127. Default value is zero. example: byte b=10;
- **short** : It is 16 bit integer data type. Value range from -32768 to 32767. Default value is zero. example: short s=11;
- **int** : It is 32 bit integer data type. Value range from -2147483648 to 2147483647. Default value is zero. example: int i=10;
- **long** : It is 64 bit integer data type. Value range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Default value zero. example: long l=100012;

Data Types in Java(Cont.)

- **Floating-Point Number**
- This group includes float, double datatypes.
- **float** : It is 32 bit float data type. Default value 0.0f. example: float ff=10.3f;
- **double** : It is 64 bit float data type. Default value 0.0d. example: double db=11.123;

Characters

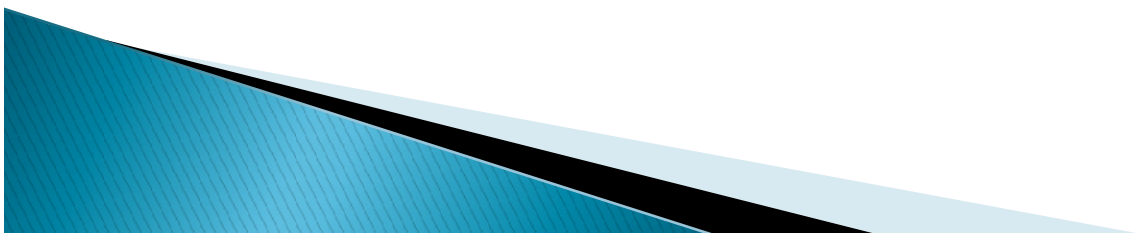
- This group represent char, which represent symbols in a character set, like letters and numbers.
- **char** : It is 16 bit unsigned unicode character. Range 0 to 65,535. example: char c='a';

Boolean

- This group represent boolean, which is a special type for representing true/false values. They are defined constant of the language. example: boolean b=true;

Type Casting

- Type Casting
- Assigning a value of one type to a variable of another type is known as Type Casting.
- Example :
- Int x = 10;
- byte y = (byte)x;
- In Java, type casting is classified into two types,
- Widening Casting(Implicit)
- Narrowing Casting(Explicitly done)



Type Casting (Cont.)

Widening conversion

- Example 1:
  - double x;
  - int y = 10;
  - x = y;
- Example 2:
  - int x;
  - short y =2;
  - x= y;

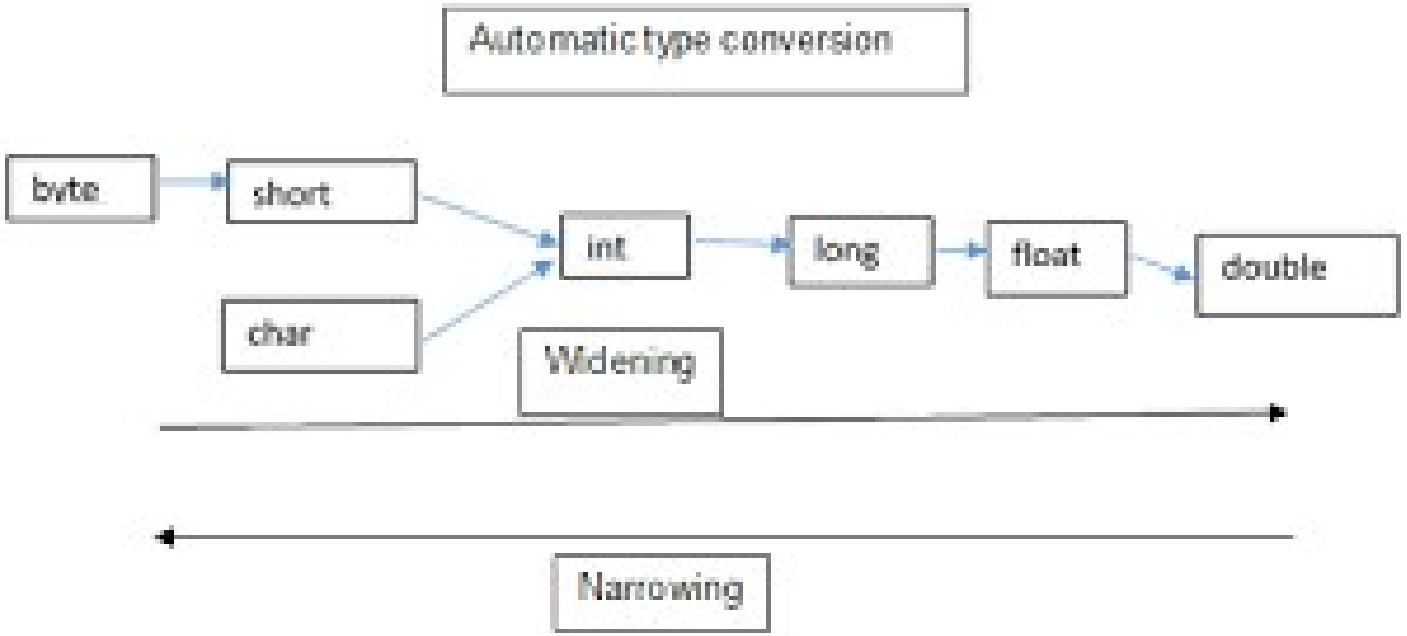
Type Casting (Cont.)

```
double d ;
int i = 10;
d = i;
```

```
double d   = 10;
int i;
i = (int) d
```

Type Cast Operator

Type Casting (Cont.)



Type Casting (Cont.)

- **Widening or Automatic type conversion**: Automatic Type casting take place when the two types are compatible. The target type is larger than the source type.

Example :

```
public class Test
{
    public static void main(String[] args)
    {
      int i = 100;
     long l = i;      //no explicit type casting required
      float f = l;    //no explicit type casting required
      System.out.println("Int value "+i);
      System.out.println("Long value "+l);
      System.out.println("Float value "+f);
    }
}
```

Type Casting (Cont.)

- **Output :**

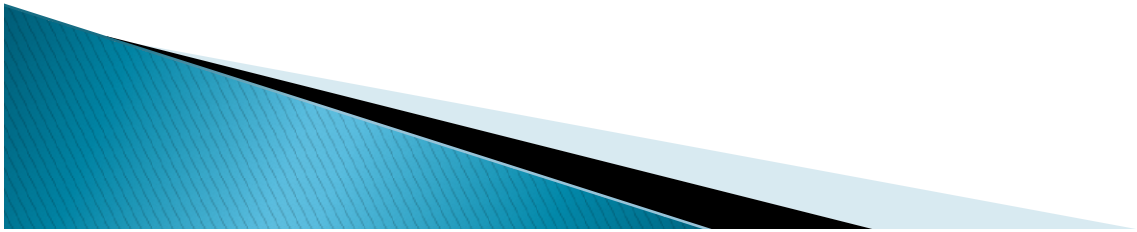
Int value 100

Long value 100

Float value 100.0

- **Narrowing or Explicit type conversion**: When y ou

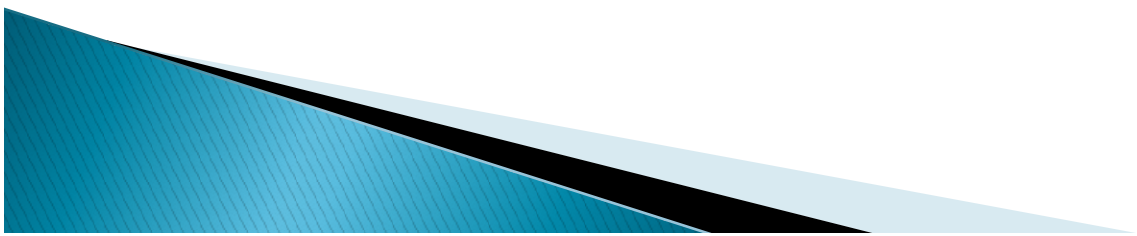
  are assigning a larger type value to a variable of smaller type, then you need to perform ex plicit type casting.



Type Casting (Cont.)

- **Example :**

```
public class Test
{
   public static void main(String[] args)
   {
      double d = 100.04;
      long l = (long)d;  //explicit type casting required
      int i = (int)l;          //explicit type casting required

      System.out.println("Double value "+d);
     System.out.println("Long value "+l);
      System.out.println("Int value "+i);

     }
     }
```



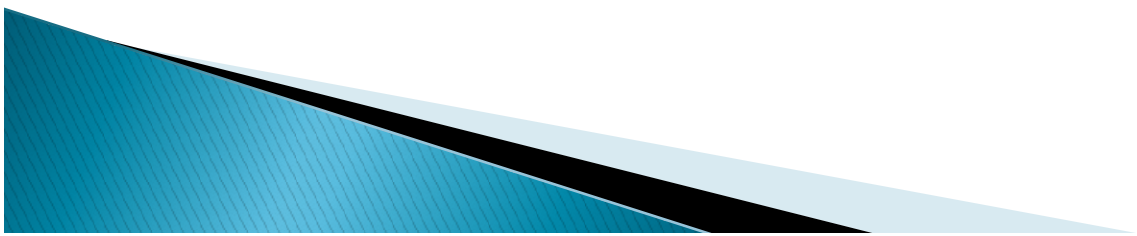
Type Casting (Cont.)

- **Output :**

Double value 100.04

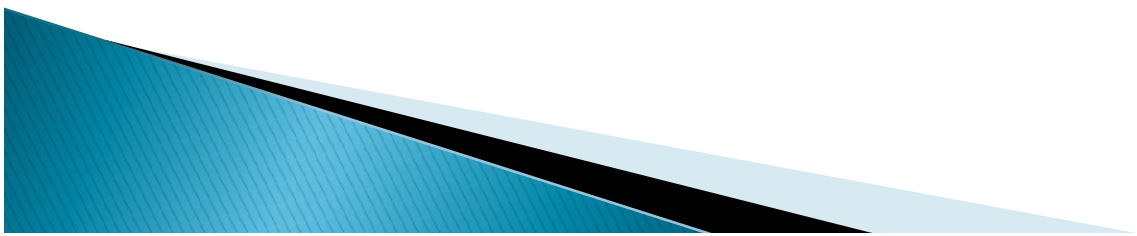
Long value 100

Int value 100



Type Casting (Cont.)

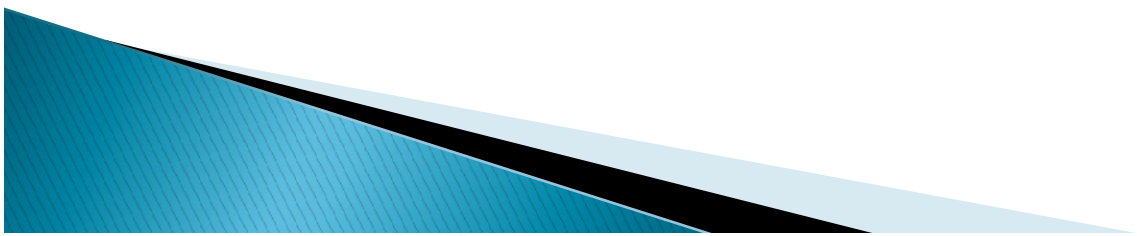
```
 class Demo
 { public static void main(String args[])
{  byte x;
    int a = 270;
    double b = 128.128;
     System.out.println("int converted to byte");
     x = (byte) a;
    System.out.println("a and x " + a + " " + x);
    System.out.println("double converted to int");
     a = (int) b;
     System.out.println("b and a " + b + " " + a);
     System.out.println("double converted to byte");
     x = (byte)b;
      System.out.println("b and x " + b + " " + x);
}
}
```



Type Casting (Cont.)

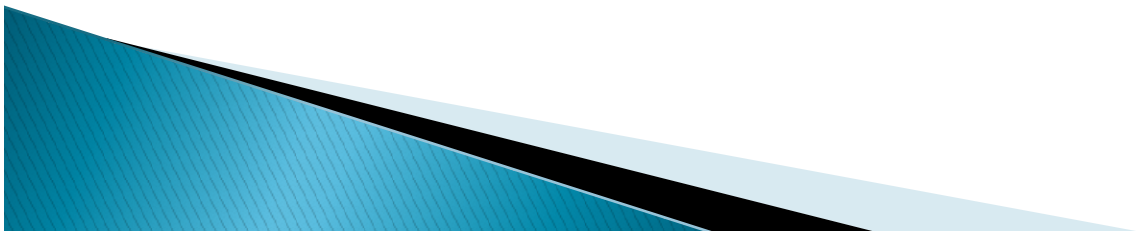
- Output:

  int converted to byte
  a and x 270 14
  double converted to int
  b and a 128.128 128
  double converted to byte
  b and x 128.128 -128



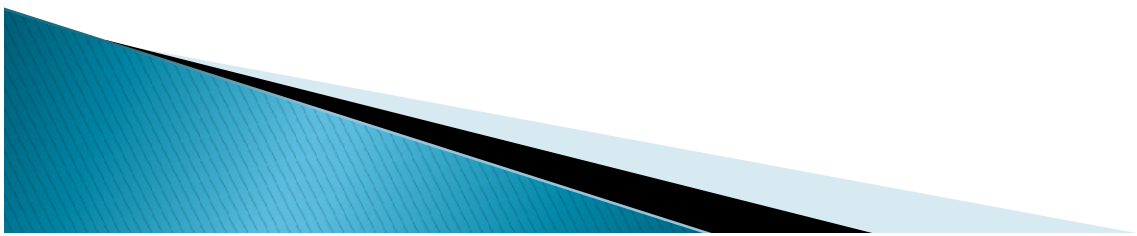
Java Character Set

- A character set is a set of textual and graphic symbols, each of which is mapped to a set of non-negative integers. The first character set used in computing was US-ASCII. It is limited in that it can represent only American English. US-ASCII contains uppercase and lowercase latin alphabets, numerals, punctuations, a set of control codes, and a few miscellaneous symbols. But, Java uses the UNICODE Character Set.
- Unicode defines a standardized, universal character set, used for representing characters and symbols as integers. Unlike ASCII, which uses 7 bits for each character, Unicode uses 16 bits, which means that it can represent 65,536 unique characters. Unicode character set represent the characters '\u0000' to '\uffff' in hexa decimal representation. The \u indicate a Unicode value.



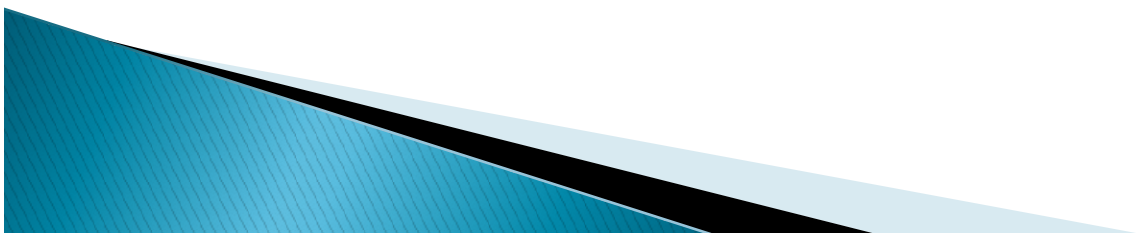
Java Tokens

- Tokens are the Java program‘s elements which are identified by the compiler. A token is the smallest element of a program that is meaningful to the compiler. Tokens supported in Java includes; *keywords, identifiers, literals, punctuators, operators*, etc.



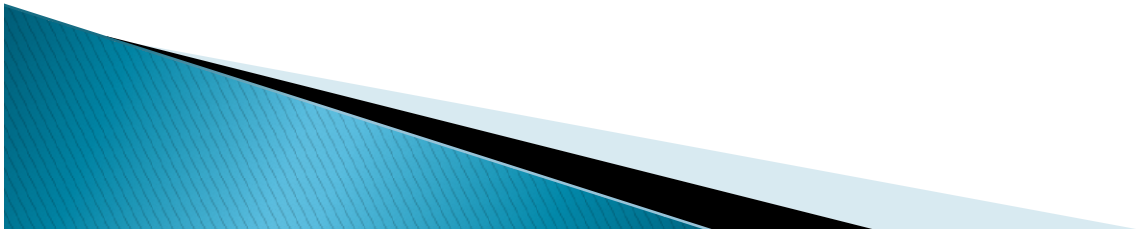
Reserved words or Keywords

- Keywords are those reserved words that convey a special meaning to the compiler. These keywords have pre-defined functions. These keywords can not be used as names for a variable,constant,class or method.



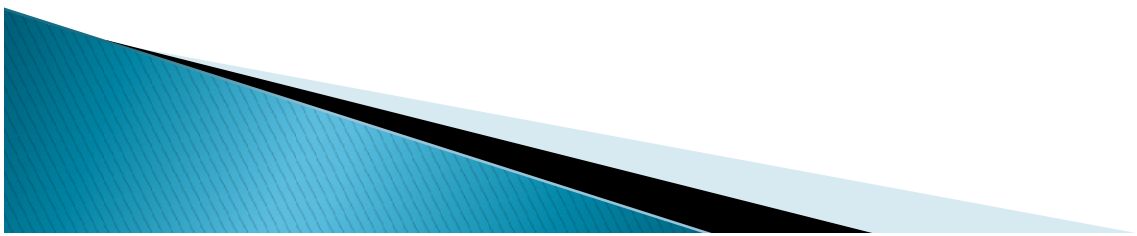
Reserved words or Keywords

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



Literals

- Literals are those data items whose value does not change during the program execution. They are also known as **constants.**
- *Java supports different types of literals which are*
  - Integer literal
  - Character literal
  - Floating-point literal
  - Boolean literal
  - String literal



Integer Literals

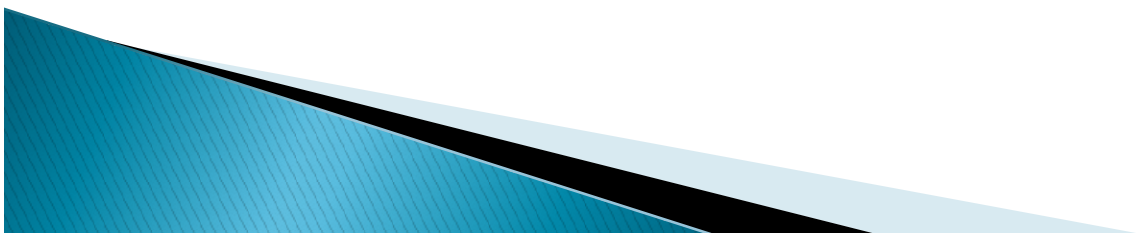
- These are the primary literals used in Java. They are of three types-

  1. decimal (base 10)
  2. hexadecimal (base 16)
  3. octal (base 8

(i)**Decimal Interger Literals**- Whose digits consists of the numbers 0 to 9.

(ii)**Hexadecimal Interger Literals**-Whose digits consists of the numbers 0 to 9 and letters A to F.

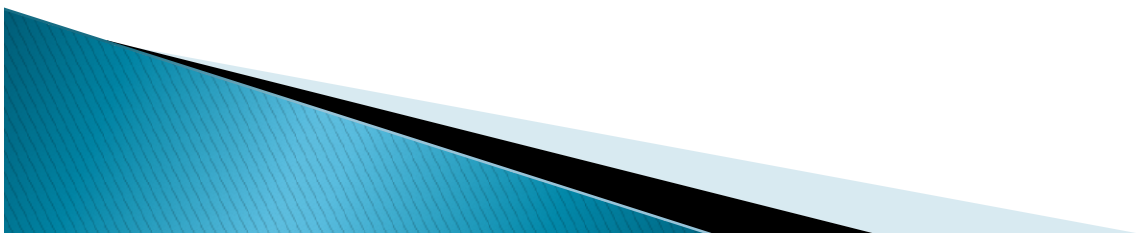
(iii) **Octal Integer Literals**- Whose digits consists of the numbers 0 to 7 only.



Integer Literals

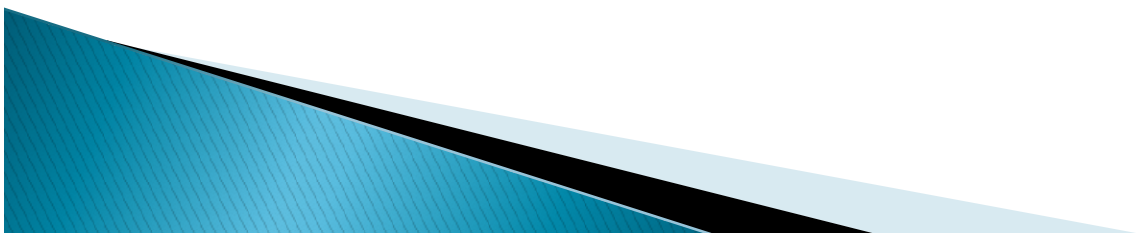
- *Some rules for integer literals are given below*
  - It must have at least one digit and can‘t use a decimal digit.
  - It must have a positive or negative sign, if the number does appear without any sign, it is assumed to be a positive number.
  - Hexadecimal literals appear with a leading Ox (zero, x). Octal literals appear with a leading 0 (zero) in front of its digits. While decimal literals appears as ordinary numbers with no special notation.

- *For example,* an decimal literal for the number 10 is represented as 10 in decimal, OxA in hexadecimal and 012 in octal.



Character Literals

- These literals represent a single unicode character and appear within a pair of single quotation marks. Like : ‘a’, ‘x’ etc.
- There are some character literals which are not readily printable through a keyboard such as backspace, tabs, etc. These type of characters are represented by using escape sequences (\).



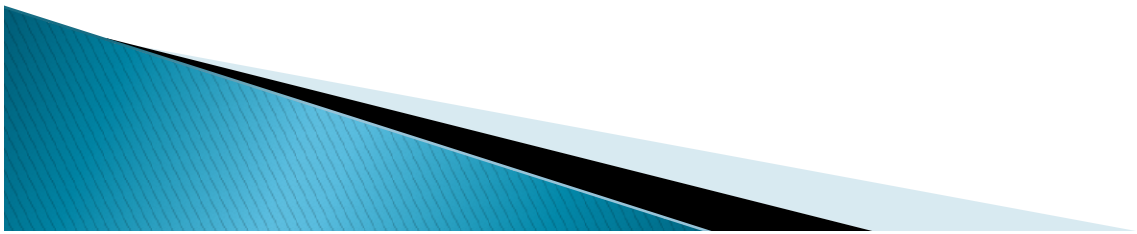
Escape sequences

Escape Sequences

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a formfeed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

Floating-Point Literals

- Floating-point numbers are like real numbers in mathematics. *For*
- *example,* 4.13179, -0.0001. Java has two kinds of floating-point number: **float** and **double.** The default type when you write a floating point literal is double. Float is of 32 bits, where as double is of 64 bits.
- A floating-point literal can be either of two data types float or double type. Floating point constants default to double precision. We have to add a suffix to the floating point literal as D, d, F or f (D or d for double and F or f for float). There are two ways of representing floating point constants.
- 1. Standard Decimal Notation: It consists of a whole number followed by a decimal point and fractional component.
- Example: 0.375, 2.576
- 2. Scientific notation or Exponent form: **Syntax:** Mantissa E Exponent
- It consists of two parts: Mantissa part which can either be decimal or fractional and exponent part which is always a whole number represented by E or e .
- Example : 0.173 E +123 , 341 e -7



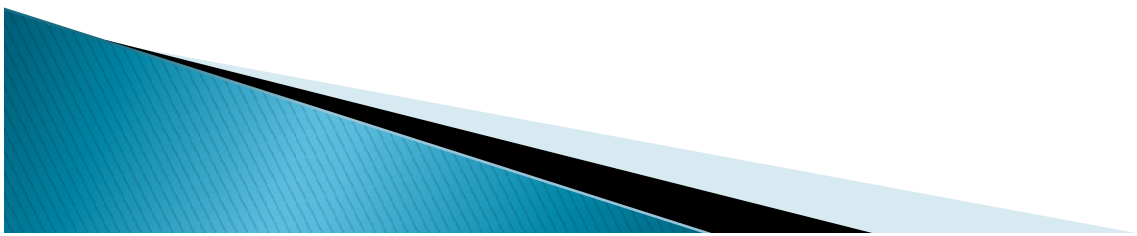
Boolean Literals

There are two Boolean literals true and false. True represents a true value and false represents a false value.

Example: boolean flag;

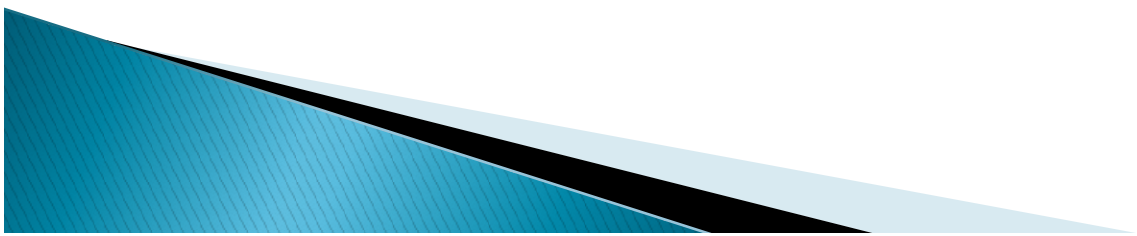
flag= false;

Literals true or false should not be represented by the quotation marks around it. Java compiler will take it as a string of characters, if it is represented in quotation marks.



String Literals

- It is a sequence of characters between a pair of double quotes. The characters may be alphabets,digits,special characters or blank space.
- Example: “1937” , “ welcome”, “Berhampur”

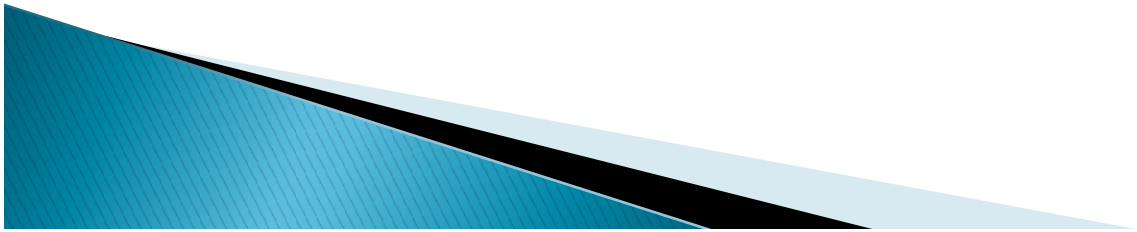


Identifiers

- An identifier is a name of fundamental building blocks of a program such as class, object, interface, method, variable etc.
- *Some of the rules to define a Java identifier are*
  - Identifiers can contain alphabets,digits, underscore or dollar sign character.
  - They must not begin with a digit.
  - They can be of any length and contains upper-case as well as lower- case letters.
  - They cannot be a keyword, boolean literals or null character.
- Identifiers should be meaningful, short enough to be quickly and easily typed and long enough to be descriptive and easily readable.

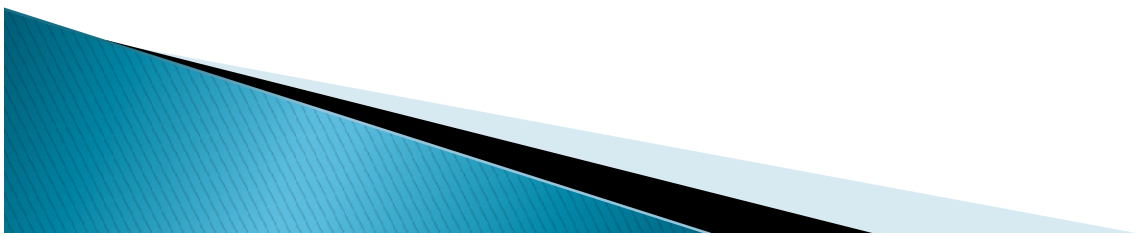
Identifiers

- **Rules for Naming Identifiers**
- Name of instance variables and public methods should start with lowercase letter. *For example, age,*total, percentage.
- When multiple words are used in a name, the second and subsequent words should start with an uppercase letter.
- *For example,* collegeTeam, totalMarks.
- Private and local variables use only lowercase letters together with underscores.
- *For example,* class_exam.
- All uppercase letters and underscores between words are used for constant values.
- *For example,* S_MARKS, SALARY_INCR.
- All classes and interfaces start with a leading uppercase letter.
- *For example,* HelloJava MetroCity



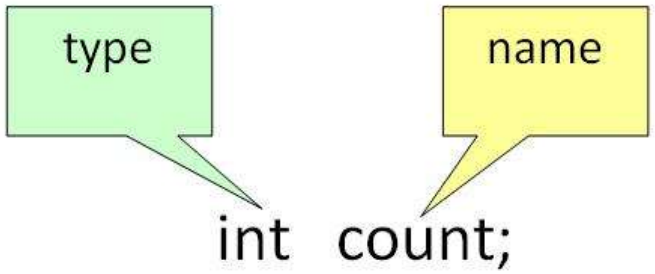
What is a Variable in Java?

- **Variable in Java**: Variables are symbolic names of memory locations. They are used for storing values used in programs. Every variable is assigned data type which designates the type and quantity of value it can hold. In many programming languages including Java, before a variable can be used, it has to be declared so that its name is known and proper space in memory is allocated.
- In order to use a variable in a program you to need to perform two steps
- Variable Declaration
- Variable Initialization



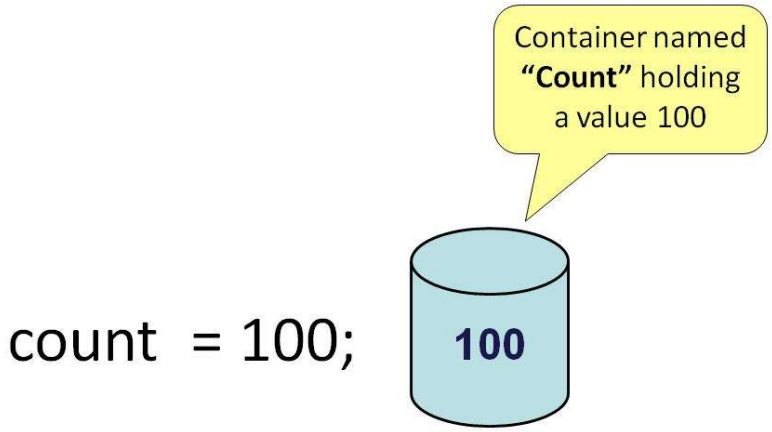
Variable Declaration

- To declare a variable, you must specify the data type & give the variable a unique name. For Example:
- int x; double y;
- Here a variable x is created for storing an int (integer) value and a variable y is created for storing a double (double- precision floating point) value.



Variable Initialization

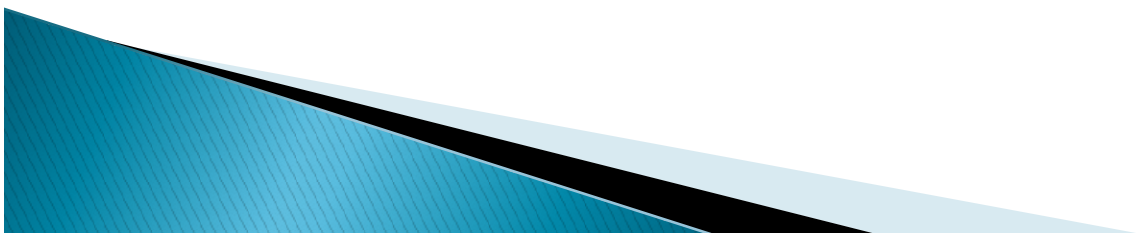
- To initialize a variable, you must assign it a valid value. Variables are normally used with the *assignment operator* (=), which assign the value on the right to the variable on the left.



Variable Initialization(Cont.)

- Example of other Valid Initializations are
- pi =3.14f; d =20.22d; a='v'; You can combine variable declaration and initialization as follows:

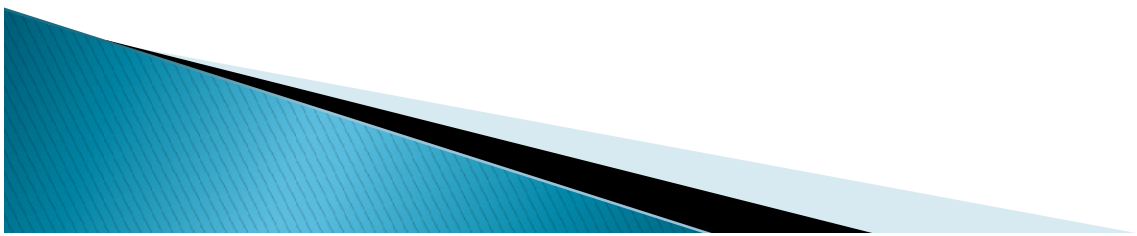
- Example :
- int a=2,b=4,c=6; float pi=3.14f; double d=20.22d; char a='v';



Naming Rules and Styles

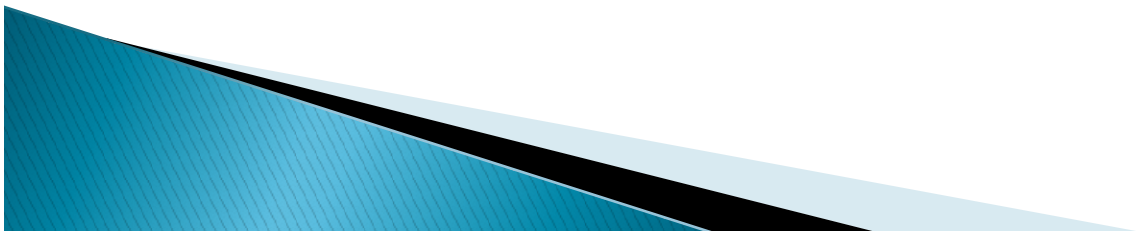
- There are certain rules for the naming of Java identifiers. Valid Java identifier must be consistent with the following rules.

  An identifier cannot be a Java reserve word.
  - An identifier must begin with an alphabetic letter, underscore (_), or a dollar sign ($).
  - If there are any characters subsequent to the first one, those characters must be alphabetic letters, digits, underscores (_), or dollar signs ($).
  - Whitespace cannot be used in a valid identifier.
  - An identifier name must be unique.
  - An identifier must not be longer than 65,535 characters.
  - Java is case sensitive , so upper case and lower case letters are distinct.



Naming Rules and Styles(Cont.)

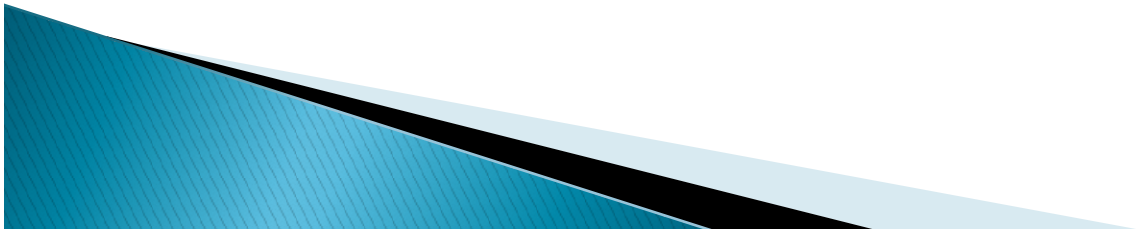
- Also, there are certain styles that programmers widely use in naming variables, classes and methods in Java. Here are some of them.
-
  - Use lowercase letter for the first character of variables' and methods' names.
  - Use uppercase letter for the first character of class names.
  - Use meaningful names.
  - Compound words or short phrases are fine, but use uppercase letter for the first character of the words subsequent to the first. Do not use underscore to separate words.
  - Use uppercase letter for all characters in a constant. Use underscore to separate words.
  - Apart from the mentioned cases, always start with a lowercase letter.
  - Use verbs for methods' names followed by nouns.
-



Naming Rules and Styles(Cont.)

- Here are some examples for good Java identifiers.
-
  - Variables: height, speed, filename, tempInCelcius, incomingMsg, textToShow.
  - Constant: SOUND_SPEED, KM_PER_MILE, BLOCK_SIZE.
  - Class names: Account, DictionaryItem, FileUtility, Article.
  - Method names: locate, sortItem, findMinValue, checkForError.

Invalid variables: 47123, #phone, basic pay, if

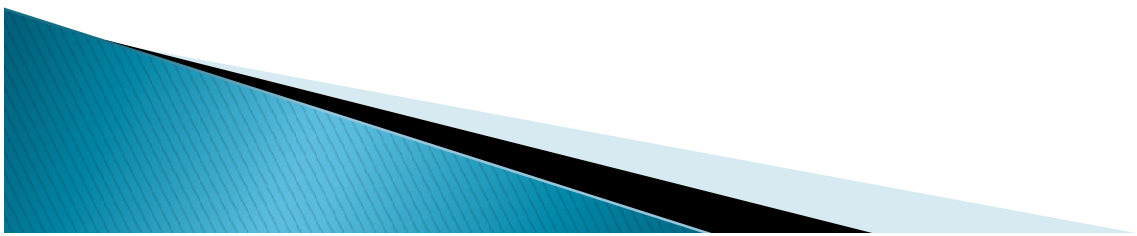


Symbolic constants in Java

- There are several values which never get changed. For example, a day will always have 24 hours, the value of PI (up to three decimal places) will always be 3.141. These are fixed values and always remain constant. In context of programming, it is convenient to represent these values in the same way (declare them as constant). These are known as symbolic constants or named constants which we can refer by its name but its value remains constant through out the program.
- In Java, symbolic constants are declared by the use of final keyword.Final is a reserved keyword and tells the compiler that the value will remain unchanged.
- *For example,* int hours = 24;
- Here, we know this value will remain unchanged as a day always has 24 hours so the final keyword can be used.
- final int hours = 24;

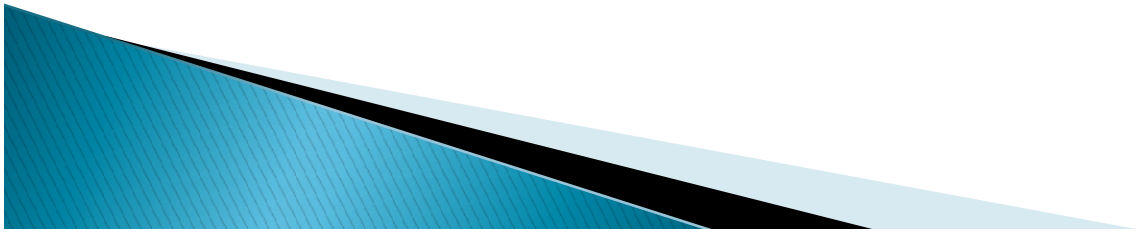
Advantages of using Symbolic Constants

- By making variables final, the values of variables can never be changed accidently.
- You have to declare them only once in a program.
- If you want to change their value, you have to change it only at one place (at the time of declaration).



Types of variables

- In Java, there are three types of variables:
- Local Variables
- Instance Variables
- Static Variables
- 1) Local Variables
- Local Variables are variables that are declared inside the body of a method.

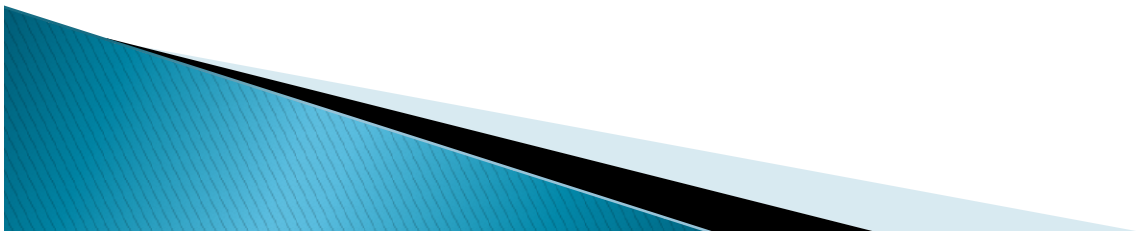


Types of variables(Cont.)

- Local variables are declared in method constructor or blocks. Local variables are initialized when method or constructor block start and will be destroyed once it ends. Access modifiers are not used for local variable.

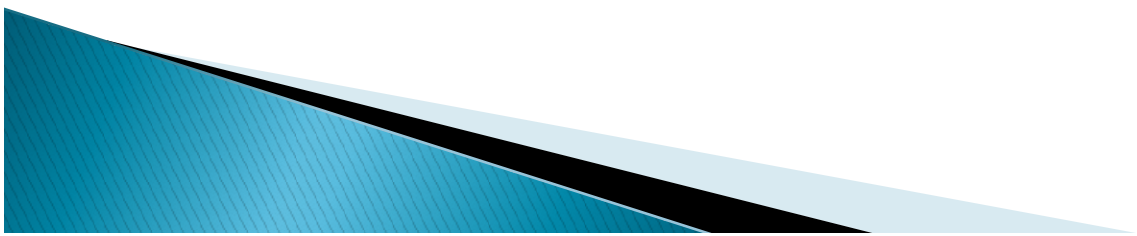
```
float getDiscount(int  price)
{
 float discount;
 discount=price*(20/100);
 return discount;
}
```

Here **discount** is a local variable.



Types of variables(Cont.)

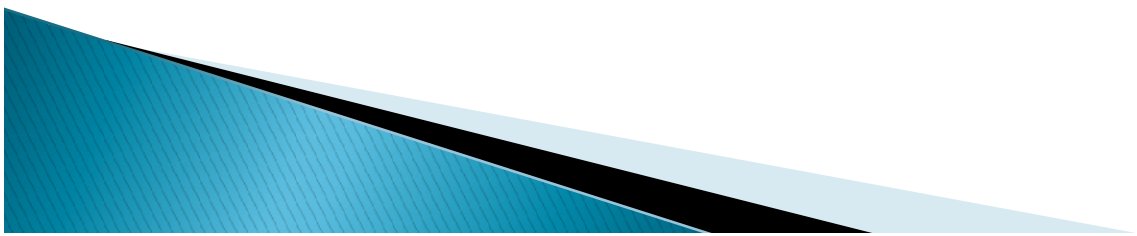
- 2) Instance Variables
- Instance variables are variables that are declared inside a class but outside any method,constructor or block. Instance variable are also variable of object commonly known as data members.
- They are Object specific and are known as instance variables.



Types of variables(Cont.)

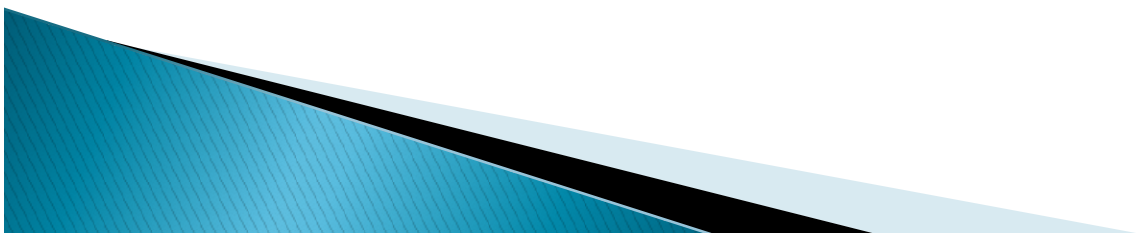
```
class Student
{
 String name;
 int age;
}
```

- Here **name** and **age** are instance variable of Student class.



Types of variables(Cont.)

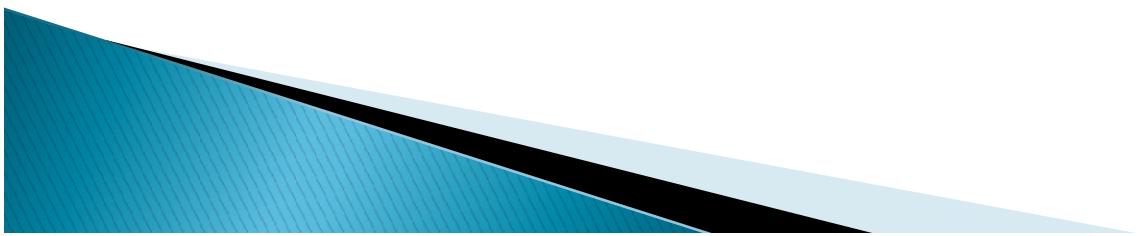
- 3) Static Variables
- Static Variables are class variables declared with static keyword. These variables can be accessed using the name of the class rather than the name of the object. Static variables are initialized only once at the start of the program execution and the same copy of the static variables is accessible to all the objects. These variables should be initialized first, before the initialization of any instance variables.



Types of variables(Cont.)

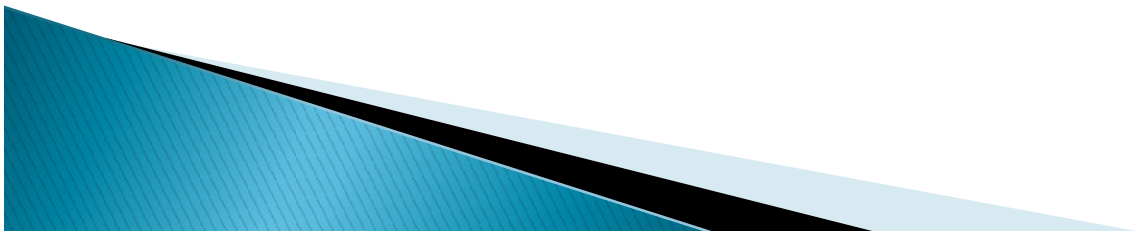
```
class Student
{
 String name;
 int age;
 static int code=1101;
}
```

- Here **code** is a static variable. Each object of Student class will share the code property.



Example: Types of Variables in Java

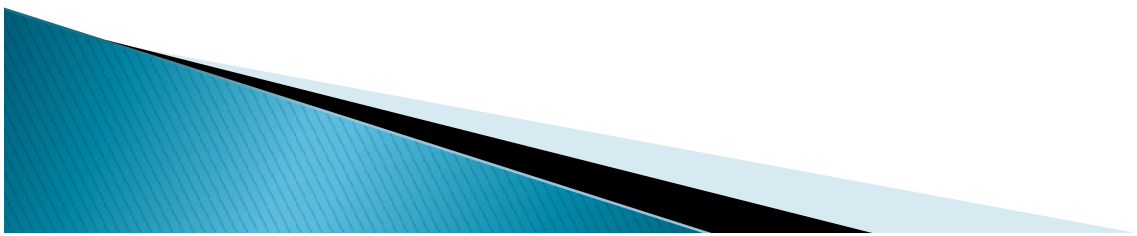
```
class Sample
 {    static int a = 1;     // static variable
      int data = 99;       // instance variable
       void method()
{
   int b = 90;     //local variable
}
}
```



Java Operators

- Java provides a rich operator environment. Most of its operators can be divided into the following categories:
- Increment/Decrement Operator
- Arithmetic Operator
- Relational Operator
- Logical Operator
- Bitwise Operator
- Shift Operator
- Assignment Operator
- Conditional Operator
- Instance of Operator
- New Operator
- Member selection Operator

- Each operator performs a specific task it is designed for.



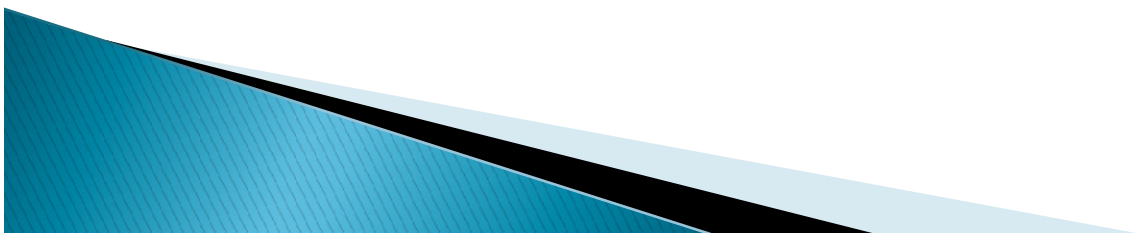
Java Increment/Decrement Operator

- *Increment and Decrement operators*
- Aside from the basic arithmetic operators, Java also includes a unary increment operator(++) and 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++;



Java Increment/Decrement Operator(Cont.)

Operator	*Use*	*Description*
++	op++	Increments op by 1; evaluates to the value of op before it was incremented (Post Increment Operator)
++	++op	Increments op by 1; evaluates to the value of op after it was incremented(Pre Increment Operator)
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented(Post Decrement Operator)
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented(Pre Decrement Operator)

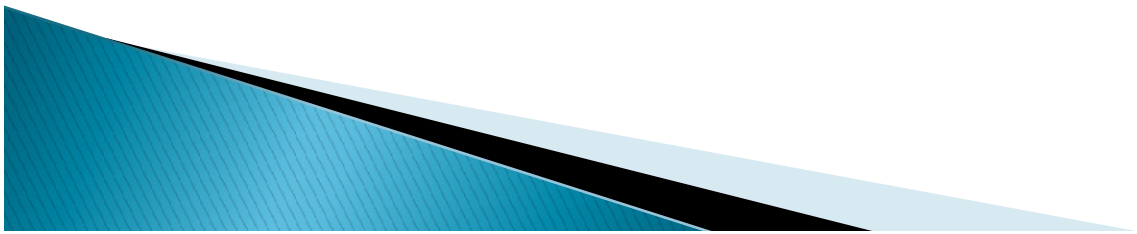
Java Increment/Decrement Operator(Cont.)

- 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
```

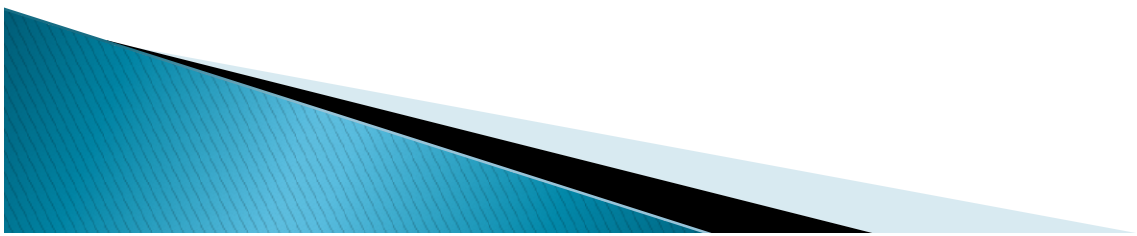


Java Increment/Decrement Operator(Cont.)

```
class OperatorExample{
public static void main(String args[])
{
int x=10;
System.out.println(x++);
System.out.println(++x);
System.out.println(x--);
System.out.println(--x);
}}
```

Output:

```
10
12
12
 10
```



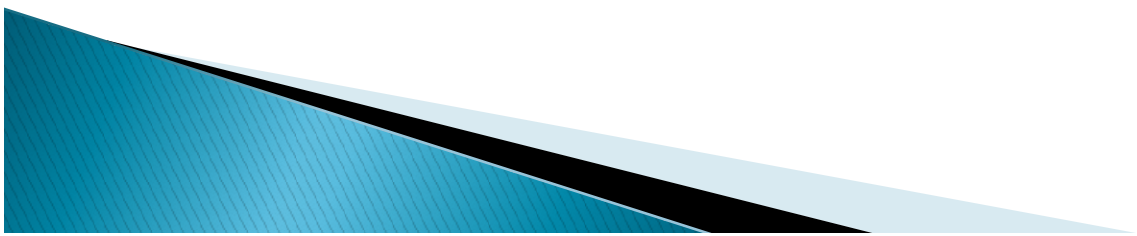
Java Increment/Decrement Operator(Cont.)

```
class OperatorExample{
public static void main(String args[])
{
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21

}}
```

Output:

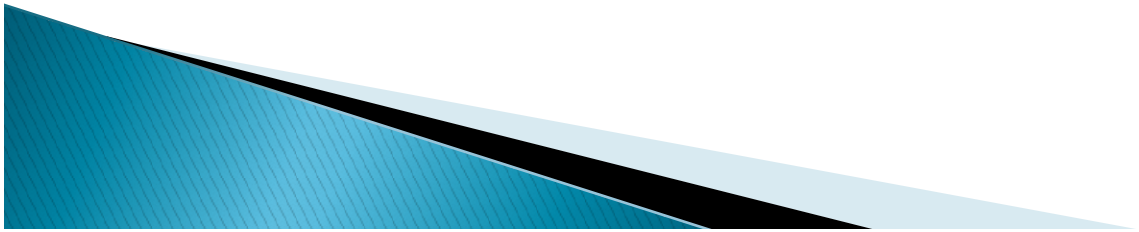
22

21



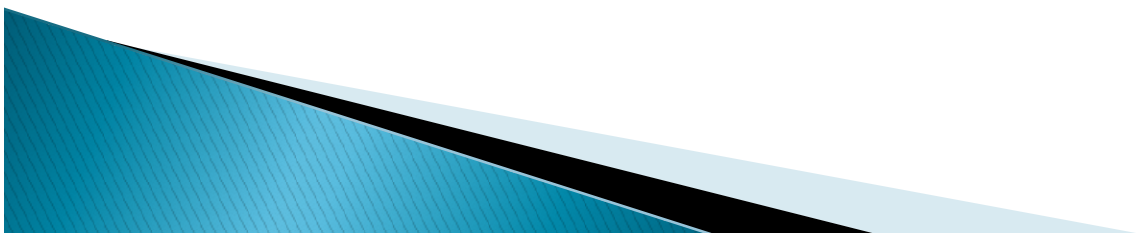
Arithmetic operators

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



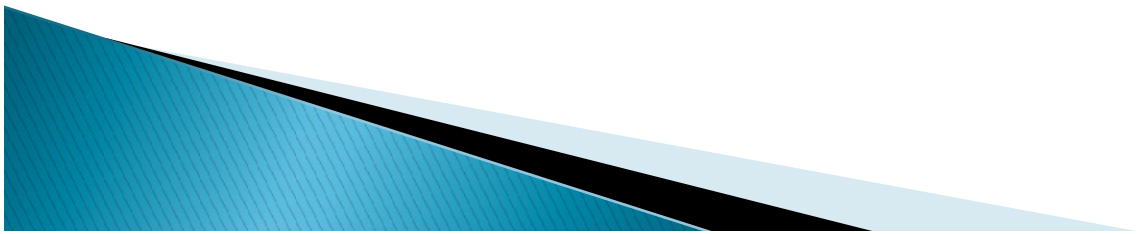
Arithmetic operators(Cont.)

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

//a few numbers
     int i = 37;
     int j = 42;
    double x = 27.475;
     double y = 7.22;
    System.out.println("Variable values...");
    System.out.println("i = " + i);
    System.out.println("j = " + j);
    System.out.println("x = " + x);
    System.out.println("y = " + y); //adding
```



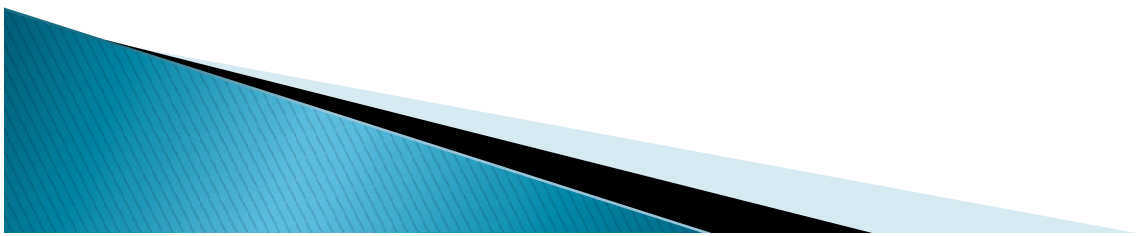
Arithmetic operators(Cont.)

```
System.out.println("Adding...");
 System.out.println(" i + j = " + (i + j));
 System.out.println(" x + y = " + (x + y));

//subtracting numbers
 System.out.println("Subtracting...");
 System.out.println("   i - j = " + (i - j));
System.out.println("    x - y = " + (x - y));
```



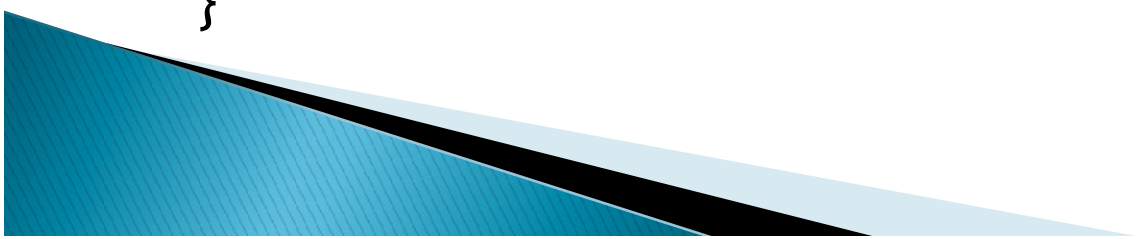
Arithmetic operators(Cont.)

```
//multiplying numbers
System.out.println("Multiplying...");
System.out.println("     i * j = " + (i * j));
 System.out.println("    x * y = " + (x * y));
//dividing numbers
System.out.println("Dividing...");
System.out.println("     i / j = " + (i / j));
 System.out.println("    x / y = " + (x / y));



}
```



Arithmetic operators(Cont.)

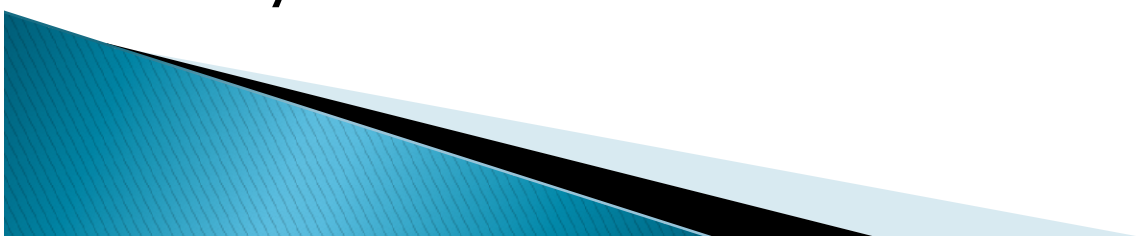
```
//computing the remainder resulting from dividing

   numbers

    System.out.println("Computing the remainder...");
   System.out.println("      i % j = " + (i % j));
   System.out.println("      x % y = " + (x % y));

//mixing types System.out.println("Mixing types...");
    System.out.println("    j + y = " + (j + y));
   System.out.println("      i * x = " + (i * x));

   }
```



Arithmetic operators(Cont.)

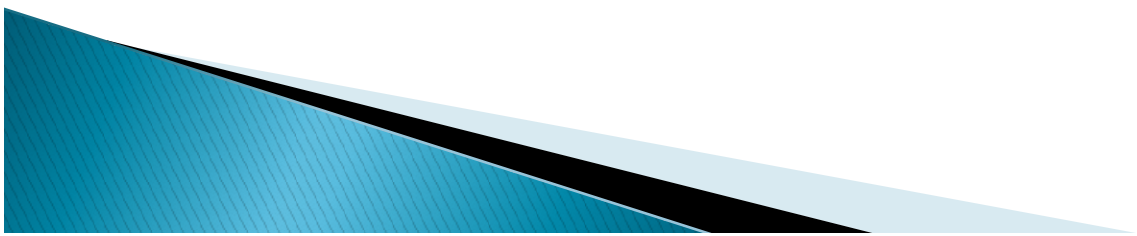
Here is the output of the program,

Variable values... i = 37
j = 42
x = 27.475
y = 7.22
Adding...
i + j = 79
x + y = 34.695
Subtracting... i - j = -5
x - y = 20.255



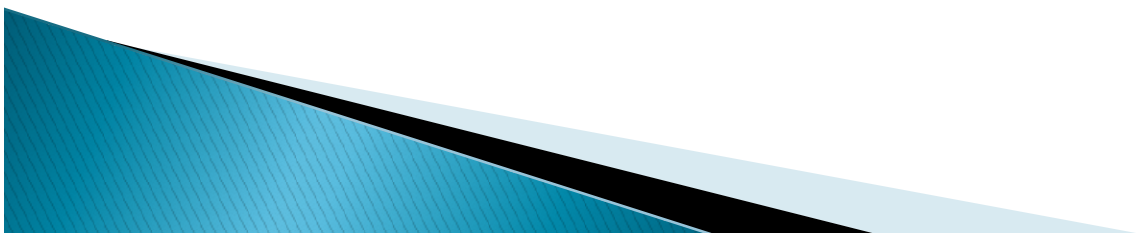
Arithmetic operators(Cont.)

Multiplying...
i * j = 1554 x * y = 198.37
Dividing...
i / j = 0
x / y = 3.8054 Computing the remainder...
i % j = 37
x % y = 5.815
Mixing types...
j + y = 49.22 i * x = 1016.58



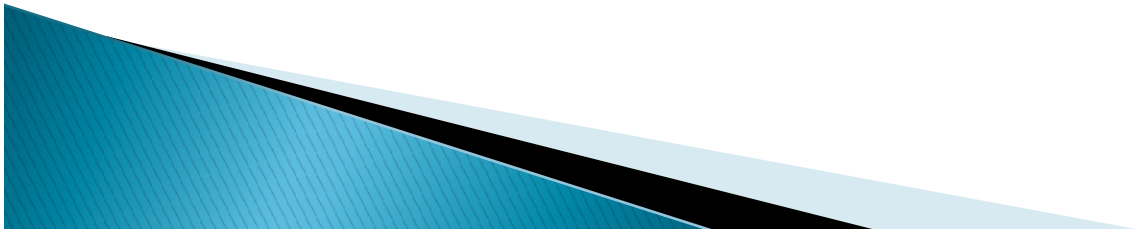
Relational operators

- Relational operators compare two values and determines the relationship between those values. The output of evaluation are the boolean values true or false.



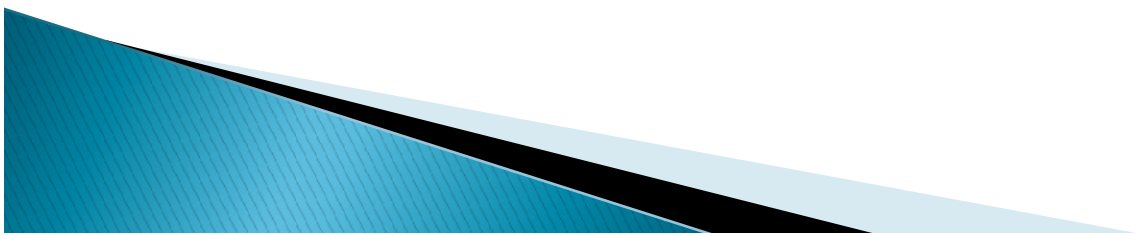
Relational operators (Cont.)

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



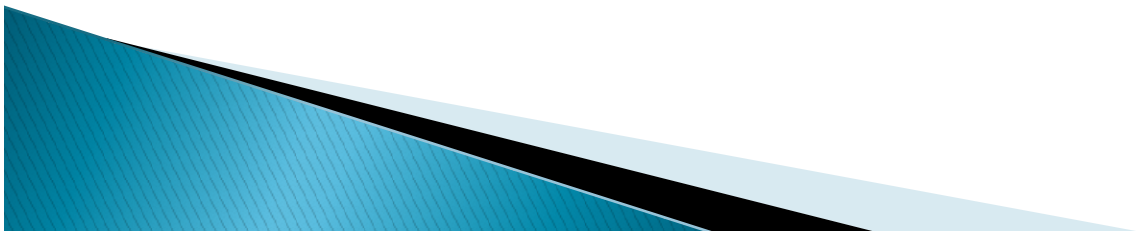
Relational operators(Cont.)

```
public class RelationalDemo
{
public static void main(String[] args) {
//a few numbers
 int i = 37;
int j = 42; int k = 42;
System.out.println("Variable values...");
System.out.println("       i = " + i);
System.out.println("       j = " + j);
System.out.println("       k = " + k);
//greater than
System.out.println("Greater than...");
System.out.println("       i > j = " + (i > j)); //false
System.out.println("       j > i = " + (j > i)); //true
System.out.println("       k > j = " + (k > j)); //false
```



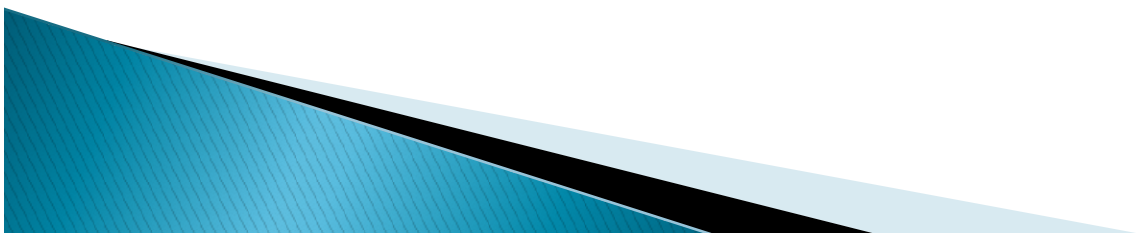
Relational operators (Cont.)

```
//greater than or equal to
System.out.println("Greater than or equal to...");
System.out.println("       i >= j = " + (i >= j)); //false
System.out.println("       j >= i = " + (j >= i)); //true
System.out.println("       k >= j = " + (k >= j)); //true

//less than
System.out.println("Less than...");
 System.out.println("       i < j = " + (i < j)); //true
 System.out.println("       j < i = " + (j < i)); //false
 System.out.println("       k < j = " + (k < j)); //false
//less than or equal to
System.out.println("Less than or equal to...");
System.out.println("       i <= j = " + (i <= j)); //true
System.out.println("       j <= i = " + (j <= i)); //false
System.out.println("       k <= j = " + (k <= j)); //true
```



Relational operators (Cont.)

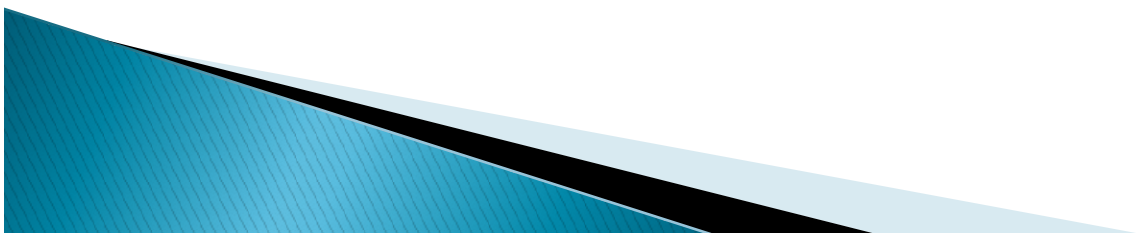
```
//equal to
System.out.println("Equal to...");
System.out.println("     i == j = " + (i == j)); //false
  System.out.println("k == j = " + (k == j)); //true

//not equal to
System.out.println("Not equal to...");
System.out.println("     i != j = " + (i != j)); //true
  System.out.println("k != j = " + (k != j)); //false

}
}
```



Relational operators (Cont.)

output from this program:

Variable values... i = 37
j = 42
k = 42
Greater than...
i> j = false
j > i = true
k > j = false
Greater than or equal to...
i >= j = false
j>= i = true
k >= j = true
Less than...
i< j = true
j < i = false
k < j = false



Relational operators (Cont.)

Less than or equal to...

i <= j = true

j<= i = false

k <= j = true

Equal to...

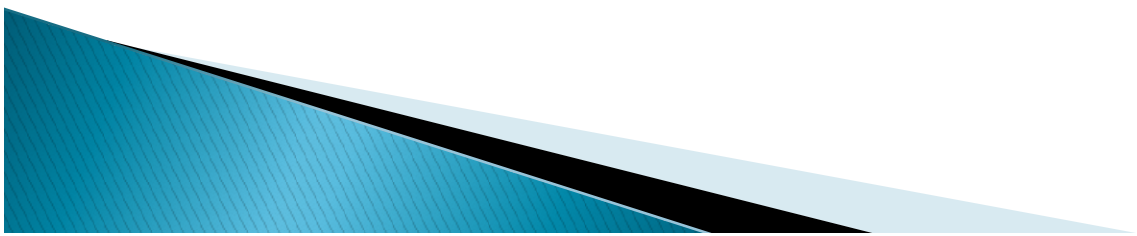
i == j = false

k == j = true

Not equal to...

i != j = true

k != j = false



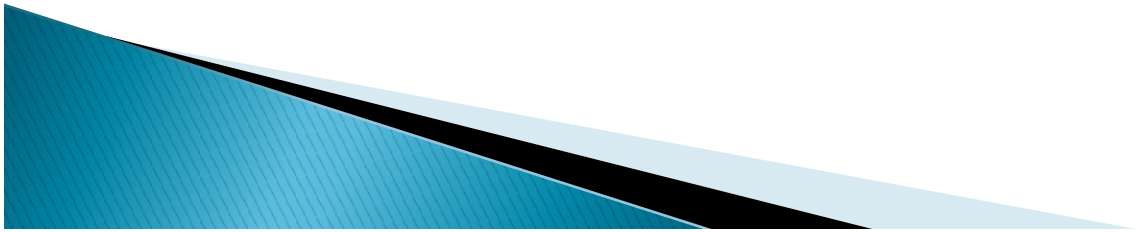
Logical operators Vs Bitwise Operators

- Logical operators have one or two boolean operands that yield a boolean result. There are three logical operators: && (logical AND), || (logical OR),, and ! (logical NOT) while the four bitwise operators are & (AND), | (inclusive OR), ^ (exclusive OR) and ~(NOT) can be used to integer types like long,int,short,char and byte as its operands. It can also be used with assignment form such as &= ,|= , ^= etc.

The basic expression for a logical operation is,

x1 op x2

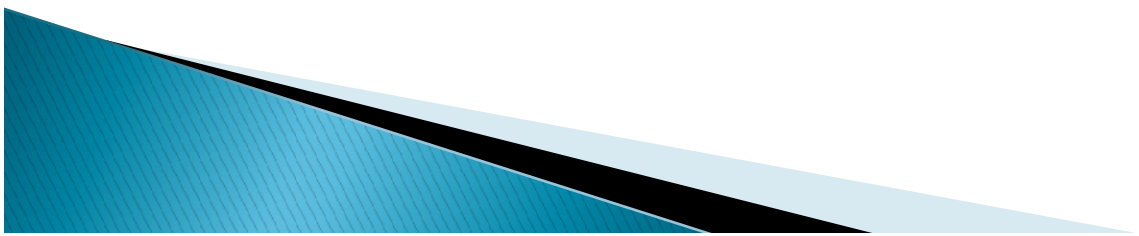
where x1, x2 are the operands, and op is the operator.



&& (logical AND) and & (Bitwise AND)

Truth Table for && and & :

x1	*x2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE



&& (logical AND) and & (Bitwise AND)

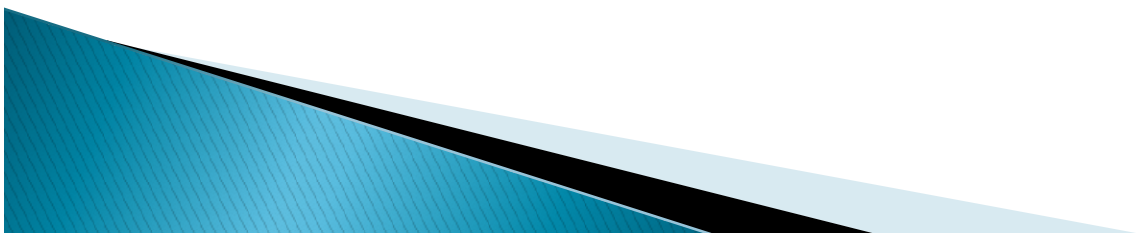
Given an expression,

exp1 && exp2

&& will evaluate the expression exp1, and immediately return a false value if exp1 is false. If exp1 is false, the operator never evaluates exp2 because the result of the operator will be false regardless of the value of exp2. In contrast, the & operator always evaluates both exp1 and exp2 before returning an answer.

&& (logical AND) and & (boolean logical AND)

```
public class TestAND
{
public static void main( String[] args ){
int  i= 0;
int j = 10;
boolean test= false;

//demonstrate &&
test = (i > 10) && (j++ > 9);
System.out.println(i);
System.out.println(j);
System.out.println(test);
```



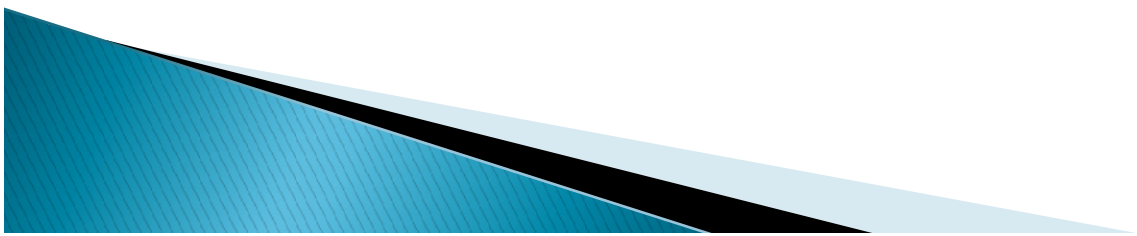
&& (logical AND) and & (Bitwise AND)

```
//demonstrate &
test = (i > 10) & (j++ > 9);
System.out.println(i);
System.out.println(j);
System.out.println(test);
}
}
```

The output of the program is,

```
0
10
False
0
11
false
```

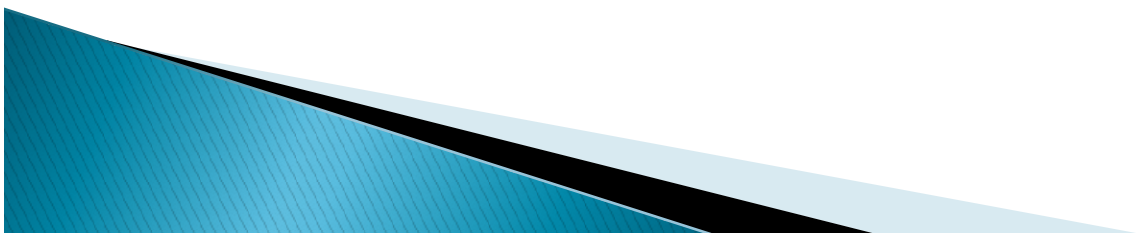
Note, that the j++ on the line containing the && operator is not evaluated since the first expression (i>10) is already equal to false.



|| (logical OR) and | (bitwise inclusive OR)

- The Truth Table for || and | is as follows:

x1	*x2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE



|| (logical OR) and | (bitwise inclusive OR)

Given an expression,

exp1 || exp2

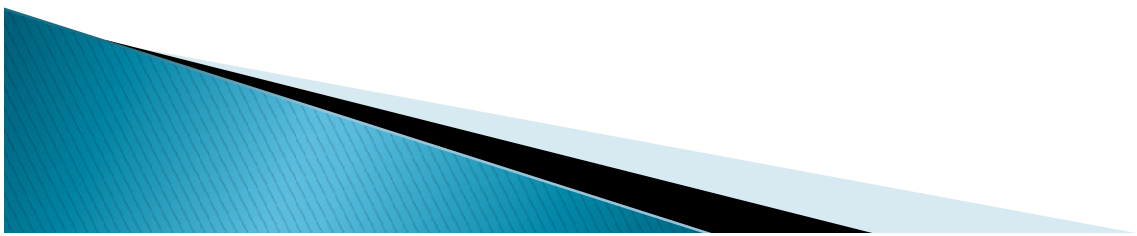
|| will evaluate the expression exp1, and immediately return a true value is exp1 is true. If exp1 is true, the operator never evaluates exp2 because the result of the operator will be true regardless of the value of exp2. In contrast, the | operator always evaluates both exp1 and exp2 before returning an answer.

A Sample Program

```
public class TestOR
{
public static void main( String[] args ){
int  i= 0;
int j = 10;
boolean test= false;
```

|| (logical OR) and | (bitwise inclusive OR)

```
//demonstrate ||
test = (i < 10) || (j++ > 9);
 System.out.println(i);
System.out.println(j);
System.out.println(test);

//demonstrate |
test = (i < 10) | (j++ > 9);
System.out.println(i);
System.out.println(j);
System.out.println(test);
}
}
```

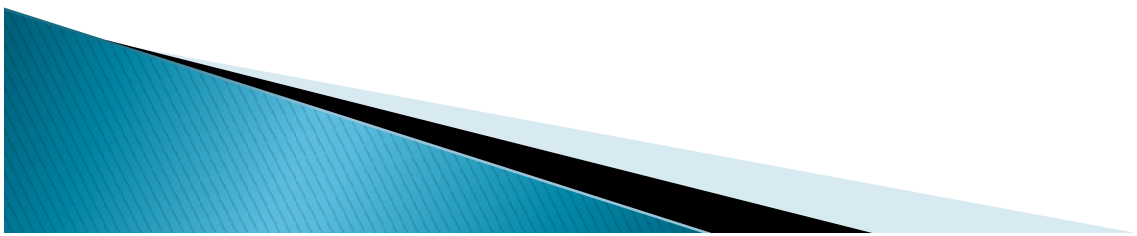


|| (logical OR) and | (bitwise inclusive OR)

The output of the program is,

```
0
10
true
0
11
true
```

Note, that the j++ on the line containing the || operator is not evaluated since the first expression (i<10) is already equal to true.



^ (bitwise exclusive OR)

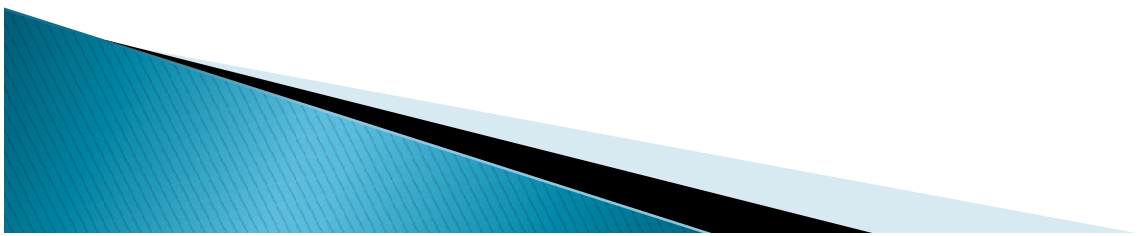
The Truth Table for ^ is as follows:

x1	*x2*	*Result*
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

The ion is TRUE, if and only if one operand is true and the other is false. Note that both operands must always be evaluated in order to calculate the result of an exclusive OR.

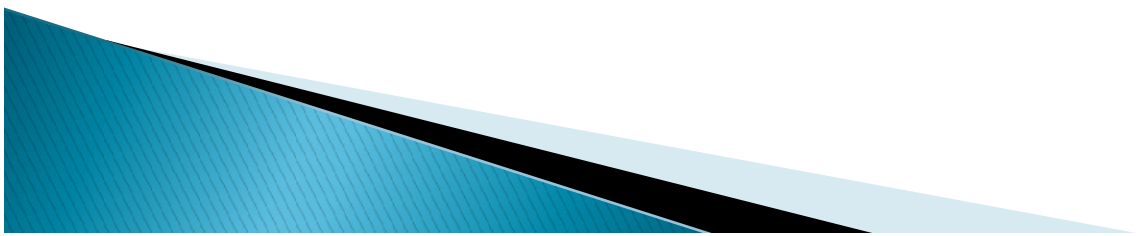
^ (bitwise exclusive OR)

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

boolean val1 = true;
boolean val2 = true;
System.out.println(val1 ^ val2);

val1 = false;
 val2 = true;
System.out.println(val1 ^ val2);
```



^ (bitwise exclusive OR)

```
val1 = false;
val2 = false;
System.out.println(val1 ^ val2);
 val1 = true;
val2 = false;
System.out.println(val1 ^ val2);
}
}
```



^ (bitwise exclusive OR)

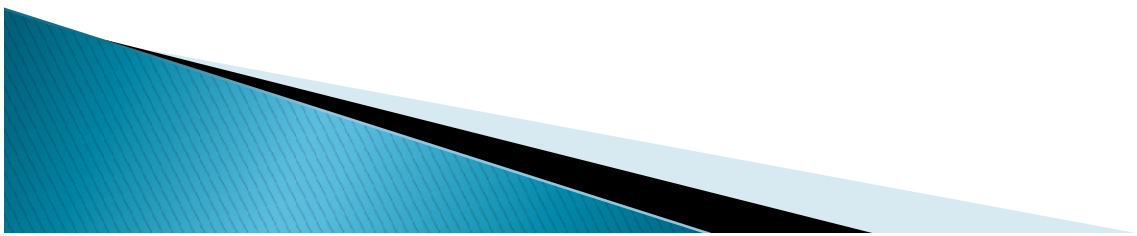
The output of the program is,

False

true

false

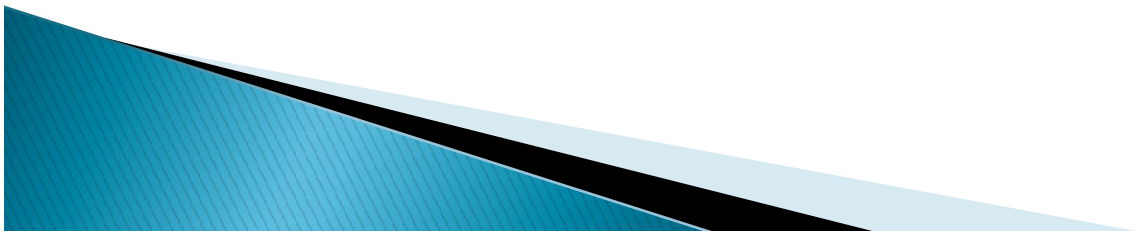
true



! (logical NOT)/~(Bitwise NOT)

The ! logical NOT/~ Bitwise NOT takes in one argument, wherein that argument can be an expression, variable or constant. The truth table for !/~

x1	*Result*
TRUE	FALSE
FALSE	TRUE

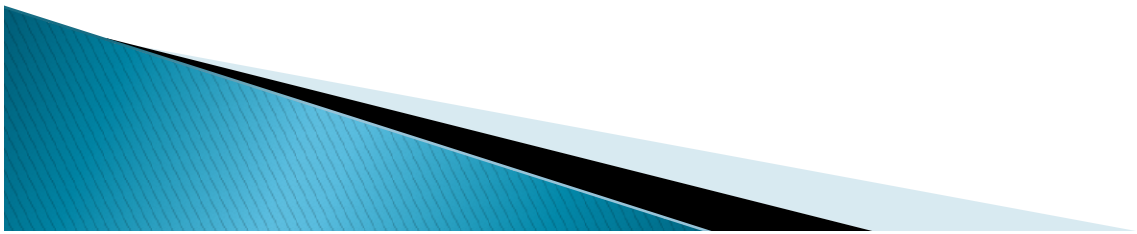


! (logical NOT)/~(Bitwise NOT)

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

boolean val1 = true;
boolean val2 = false;
System.out.println(!val1);
System.out.println(!val2);
}
}
```

The output of the program is,

```
False
 true
```



Bitwise Shift Operators

- The bitwise shift operators shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value are to be shifted. Both operands have the same precedence.
- Different types of shift operators

1.Left shift(<<) op1<<op2 shifts bits at op1 left by distance of op2

2.Right shift(>>) op1>>op2 shifts bits at op1 right by distance of op2

3. 1.Right shift with zero fill or unsigned right shift (>>>)
op1<<op2 shifts bits at op1 right by distance of op2 (unsigned or zero fill)

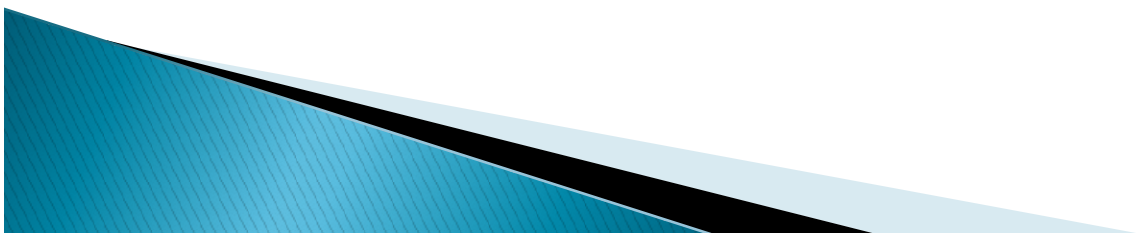
- Example

a = 0001000

b= 2

a << b= 0100000 (8<<2 = 8*2^2=32)

a >> b= 0000010 (8>>2 = 8/2^2=2)



Assignment Operators

operator	description	example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

Conditional Operator (?:)

- The conditional operator ?: is a ternary operator. This means that it takes in three arguments that together form a conditional expression. The structure of an expression using a conditional operator is,

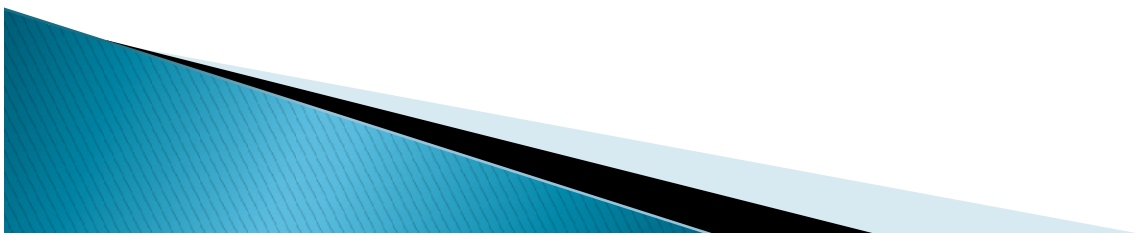
exp1?exp2:exp3

- wherein exp1 is a boolean expression whose result must either be true or false. If exp1 is true, exp2 is the value returned. If it is false, then exp3 is returned.

Conditional or Ternary Operator (?:) in Java

Conditional Operator (?:)

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

String  status = "";
int grade = 80;

//get status of the student
status = (grade >= 60)?"Passed":"Fail";
```

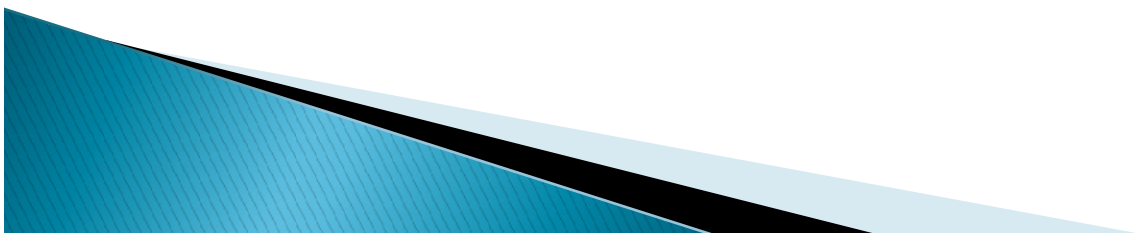


Conditional Operator (?:)

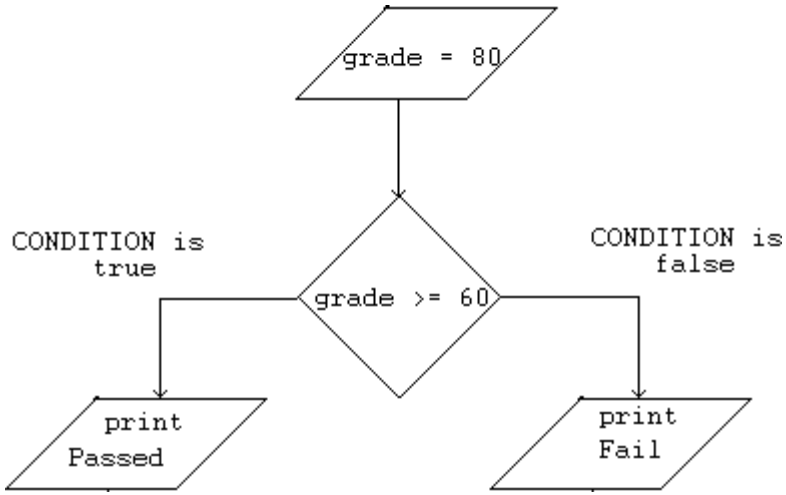
```
//print status
 System.out.println( status );
}
}
```

The output of this program will be,

Passed



Conditional Operator (?:)



Instance of Operator

- Only object reference variables can be used with this operator. The objective of this operator is to check if an object is an instance of an exiting class or interface. The return type is boolean. If object is of the specified type ,then the instance of operator returns true otherwise it returns false. It is also known as runtime operator.

Syntax:

(<object >) instance of(<interface/class>)

Example: rose instance of flower is true if the object rose belongs to the class flower otherwise false.

New() Operator

When we allocate memory to an object by creating an instance of a class, we use the new() operator to allocate memory dynamically at run time.

Case-1: Declare the object and then allocate memory

Syntax:

Classname objectname;

Objectname= new Classname(); //memory allocation

Example:

Flower rose;

rose=new Flower();

Case-2: Declare the object and allocate memory in a single step

Syntax:

Classname objectname= new Classname(); //memory allocation

Example:

Flower rose =new Flower();

Java has automatic garbage collector , so, unlike C++., there is no delete operator in java to dynamically deallocate the memory .

Member Selection Operator(.)

The class consists of data members and member methods. It can be accessed through a member selection operator or dot operator.

Syntax

object.data member

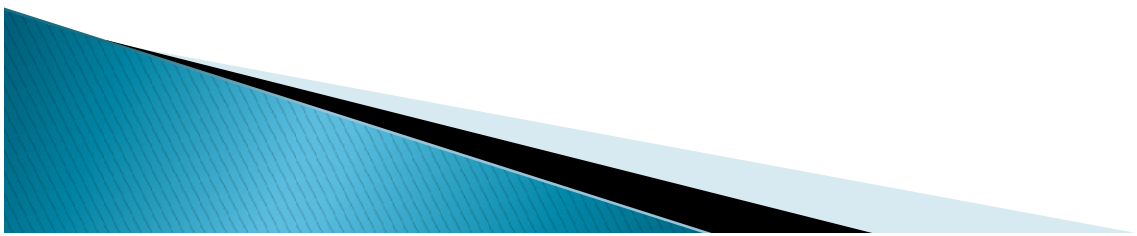
object.member method

Example

Student s1;

s1.rollno;

s1.total();



Unary, binary and ternary operators in Java

- **Unary Operator** These are the operators which work on single operands.
- For example, !, -, ++, —, ( ), (cast) operator, Unary + and Unary – are some examples of unary operators.
- **Binary Operators** These are the mostly used operators. These operators work on two operands. Binary operators include arithmetic operators (+, *, /, % etc.).
- **Ternary Operator** Operator that works on three operands is known as ternary operator. Conditional operator (? :) is the example of ternary operator.

Java Expressions

- An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

```
int x = 3 + 2 * 6;
```

This expression is evaluated to 30 if the addition operator is executed first. However, it is 15 if the multiplication operator is executed first.

Operator Precedence and Associativity

- In fact, Java compiler has no problem with such ambiguity. Order of the operators can be determined using *precedence and association rules.* Each operator is assigned a precedence level. Operators with higher precedence levels are executed before ones with lower precedence levels. Associativity is also assigned to operators with the same precedence level. It indicates whether operators to the left or to the right are to be executed first, in the case of equal precedence levels. Expressions in parentheses () are executed first. In the case of nested parentheses, the expression in the innermost pair is executed first.

Operator Precedence and Associativity

Level	Operators	Description	Associativity
15	() [] .	Function Call Array Subscript Member Selection	Left to Right
14	++ --	Postfix Increment / Decrement	Right to Left
13	++ -- + - ! ~ (type)	Prefix Increment / Decrement Unary plus / minus Logical negation / bitwise complement Casting	Right to Left
12	* / %	Multiplication Division Modulo	Left to Right
11	+ -	Addition / Subtraction	Left to Right
10	<< >> >>>	Bitwise Left Shift Bitwise Right Shift with sign extension Bitwise Right Shift with zero extension	Left to Right
9	< <= > >= instance of	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To Type Comparison for objects	Left to Right
8	== !=	Equality Inequality	Left to Right
7	&	Bitwise AND	Left to Right
6	^	Bitwise XOR	Left to Right
5	|	Bitwise OR	Left to Right
4	&&	Logical AND	Left to Right
3	||	Logical OR	Left to Right
2	?:	Conditional Operator	Right to Left
1	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left

Examples on Operator Precedence and Associativity

Evaluate the following expression by clearly state the order of operations of all operators according to the precedence and associativity rule.

Example-1 : 4*2+20/4

- There are three operators in the above expression. They are *, + and /.The precedence values of * and / are both 12, while the precedence value of + is just 11. Therefore, * and / must be operated prior to +. Since * and / have the same precedence value, we need to look at their associativity which we can see that the one on the left have to be performed first.Therefore, the order of operation from the first to the last is *, / and then+. Consequently, the evaluation of the expression value can take place in the steps and the resulting value is 13.

Examples on *Operator Precedence and Associativity*

4 * 2 + 20 / 4
=8 + 20 / 4
=8 + 5
=13

Example-2 : Evaluate the following expression.

2+2==6-2+0

Considering the precedence values of the four operators appearing in the expression, which are + (the leftmost one), ==, ▪, and + (the rightmost one), we can see that +, and ▪ have the same precedence value of 11 (additive operators) which is higher than the one of ==. Among the three additive operators, we perform the operation from the left to the right according to their associativity. The resulting value of this expression can be evaluated to **true.**

Examples on Operator Precedence and Associativity

2 + 2 == 6 - 2 + 0

4 == 6 - 2 + 0

4 == 4 + 0

4 == 4

True

Example-3 : Evaluate the following expression. Assume that the variable x has already been properly declared as an int variable.

(x=3)==(x=+1-2)&&true

Examples on *Operator Precedence and Associativity*

- First, we perform the expression in the left pair of parentheses. The variable x is assigned with the int value 3 and this is also the resulting value of the expression in this pair of parentheses. Then, the expression x=+1-2 is evaluated due to the fact that it is in the next pair of parentheses. In this expression, we have the assignment operator = (with the precedence value of 1), the unary positive + (with the precedence value of 13), and the binary operator · (with the precedence value of 13).Based on the comparison of their precedence value, the unary positive is performed first. This operator just indicates the positiveness of its operand. Consequently, the value of the right side of the assignment operator is -1 and then

Examples on Operator Precedence and Associativity

assigned to x. Therefore, the new value of x is −1 which is the value of this pair of parentheses too. The next operator to be performed is the equality operator ==. It compares the values of (x=3)and(x==+1−2), which have just been shown that they are not equal.

- Therefore, the resulting value associated with the action of this operator is the boolean value false. Finally, the logical AND (&&) is performed and the final result of the expression in this example is the boolean value false.

Examples on Operator Precedence and Associativity

- Place the grouping operators () into the following expression in order to explicitly determine the order of operations of all operators appearing in the expression. Evaluate the values of every expression involved in steps according to the inserted parentheses.

Example-4 :

-9.0+5.0*3.0-1.0/0.5 >= 5.0%2.0&&6.0+3.0-9.0==0

- By considering the precedence values of all operators appearing in the expression above, we can place parentheses into the expression in order to explicitly determine the order of operation and then evaluate the values of each part.

Examples on *Operator Precedence and Associativity*

((((-9.0)+(5.0*3.0))-(1.0/0.5)) >=
(5.0%2.0))&&(((6.0+3.0)-9.0)==0)

((((-9.0)+15.0)- 2.0) >= 1.0)&&((9.0 -9.0)==0)

((6.0- 2.0) >= 1.0)&&((9.0 -9.0)==0)

(4.0 >= 1.0)&&(0 ==0)

(true)&&(true)

true.

Example-5 : Given an expression, re-write the expression and place parentheses based on operator precedence

6%2*5+4/2+88-10

Answer:

((((6%2)*5)+(4/2))+88)-10;

Punctuators (Separators)

- A punctuator is a type of token that has syntactic and semantic meaning to the compiler, but the exact meaning depends upon the context where we use it.
- Punctuators are used for grouping and separating the numeric and non- numeric data. Some of the Punctuators used in Java are

1. ()

   Parantheses are used to contain a list of parameters in method definition, contains statements for condition etc.

2.{}

   Used to define a code for method and classes.

3.[]

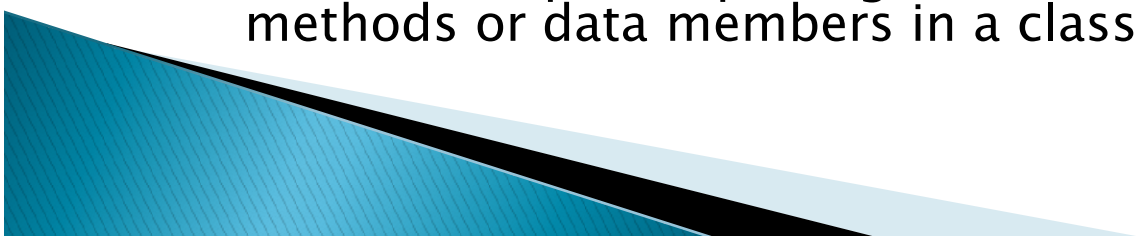
   Brackets are used for declaring array types.

4.Comma (,)

Used for separating identifier in a variable declaration, sometimes in for loop.

5. Period(.)

   Used to separate package names from sub package,referring methods or data members in a class such as dot operator.



THANK YOU