# ADVANCED PROGRAMMING

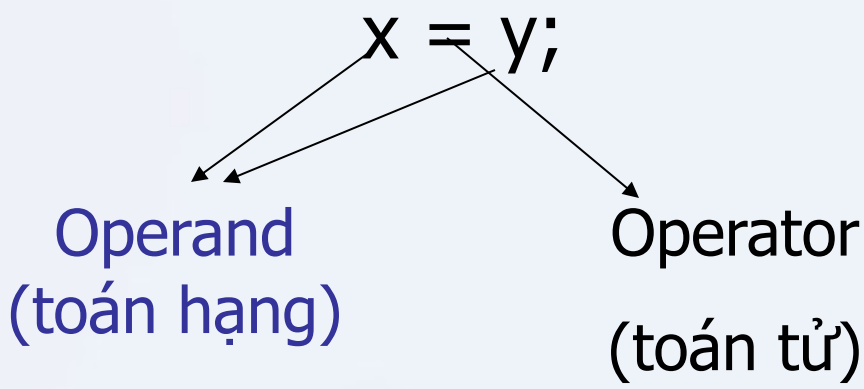
## OPERATORS AND ASSIGNMENTS

# Outline

- Understanding Operations on Data
- Operator Classification
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Arithmetic Promotion(**RAISE)**
- Assignment Operators
- Advanced Operators
- Equality of Two Objects or Two Primitives

# Understanding Operations on Data

- How can we manipulate data ?
  - Java offers operations
- The piece of data (represented by a variable) that is being operated on is called an operand



# Operator Classification

- Unary operators: Require only one operand
  - a++
- Binary operators: Require **two** operands
  - a + b
- **Ternary** operators: Operate on **three** operands
  - (a > 2) ? a : 2

# Operators

- The unary operators support either **prefix** or **postfix** notation.
  - *Prefix notation* means that the operator appears *before* its operand.
    operator op //prefix notation
  - *Postfix notation* means that the operator appears *after* its operand.
    op operator //postfix notation
- All the binary operators use ***infix*** *notation*, which means that the operator appears *between* its operands.
  op1 operator op2 //infix notation
- The ternary operator is also infix; each component of the operator appears between operands.
  op1 ? op2 : op3 //infix notation

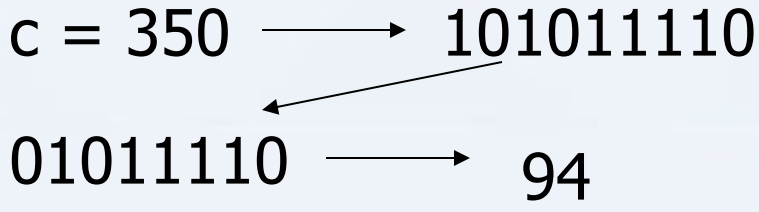
# Arithmetic Operators

- `+` additive operator (also used for String concatenation)
- `-` subtraction operator
- `*` multiplication operator
- `/` division operator
- `%` remainder operator

# Basic Arithmetic Operators (cont.)

- The accuracy of the results is limited to the type
  - If the result of the operations on two variables is larger than what the type can hold, the higher bits are dropped

```
byte a = 70;
byte b = 5;
byte c =  (byte) (a*b);
```




  - Questions:
    c = (byte) a*b; **//Error?**
  - int x=10,y=0,z=x/y; **// Error?**
  - double x=10,y=0,z=x/y; **// Error?**

# Basic Arithmetic Operators (cont.)

- Should be careful about accuracy while dividing two integers
  - The result of dividing an integer by another integer will be an integer
  - 66 divided by 7 would be 9, and not 9.43
- in case of integer types (**char, byte, short, int, and long**), division by zero is **not allowed**

```
int x = 2;
int y =0;
int z = x/y;
```

→ ArithmeticException in execution

# Basic Arithmetic Operators (cont.)

- Division by zero in case of float and double types does **not generate an error**
  - it would generate **POSITIVE_INFINITY** or **NEGATIVE_INFINITY**
- The square root of a negative number of float or double type would generate an **NaN (Not a Number)** value, and will not generate an exception

# Basic Arithmetic Operators (cont.)

- An NaN value indicates that the calculation has no meaningful result
- Two NaN values are defined in the java.lang package: **Float.NaN**, and **Double.NaN**

```
double x = 7.0/0.0;
```

```
x <  Double.NaN
x <=  Double.NaN
x >  Double.NaN
x >=  Double.NaN
x ==  Double.NaN
```

**false**

```
x != Double.NaN
```

**true**

# Arithmetic Operators

```
int a = 10;
int b = 3;
int c = a/b
double d = a/b
// d = ?
d = (float) a/b;
// d = ?
d = (float) (a/b)
// d = ?
```

# Unary operators

| Operator | Use | Description |
|---|---|---|
| ++ | op++ | Increments op by 1; evaluates to the value of op before it was incremented |
| ++ | ++op | Increments op by 1; evaluates to the value of op after it was incremented |
| -- | op-- | Decrements op by 1; evaluates to the value of op before it was decremented |
| -- | --op | Decrements op by 1; evaluates to the value of op after it was decremented |
| + | +op | Unary plus operator; indicates positive value (numbers are positive without this, however) |
| - | -op | Unary minus operator; negates an expression |
| ! | !op | Logical complement operator; inverts the value of a boolean |

# Example for Unary operators

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is ?, m is ?
int b = 2 * n++; // now b is ?, n is ?
```

# Example

int a, b;

b=a%2 + a/2 + --a;

Với a = 17 Kết quả:

a = ?; b = ?

Với a = 3 Kết quả:

a = ?; b = ?

---

int a, b;

b=a/3 + a--;

Với a = 8 Kết quả: a = ?; b = ?

Với a =21 Kết quả: a = ?; b = ?

# Example

- Cho biết kết quả đoạn chương trình sau:

```
int n;
(n%2==0)? n ++ : n --;
```

nếu n = 10 thì giá trị n = ?

nếu n = 21 thì giá trị n = ?

- Cho biết kết quả đoạn chương trình sau:

```
int k;
m = (k%3==0)?k++: k--;
```

Với k =10 Kết quả: m = ?

Với k =15 Kết quả: m = ?

# Relational (Comparison) Operators

- Relational operators in Java are **binary** operators
- A relational operators, also called a comparison operators
- **A relational operator** compares the values of two operands and returns a boolean value: true or false
- The operand could be any of the numeric operands
- The comparison operators are commonly used to define conditions in statements such as if

# Relational Operators

| Operator | Use | Description |
|---|---|---|
| `>` | `op1 > op2` | Returns `true` if `op1` is greater than `op2` |
| `>=` | `op1 >= op2` | Returns `true` if `op1` is greater than or equal to `op2` |
| `<` | `op1 < op2` | Returns `true` if `op1` is less than `op2` |
| `<=` | `op1 <= op2` | Returns `true` if `op1` is less than or equal to `op2` |
| `==` | `op1 == op2` | Returns `true` if `op1` and `op2` are equal |
| `!=` | `op1 != op2` | Returns `true` if `op1` and `op2` are not equal |

# Logical Operators

- Logical operators are used to combine more than one condition that may be true or false
- Logical operators deal with connecting the boolean values
- A boolean value is a binary value: true or false that can be represented by a bit 1 or 0
  - Logical operators can operate at bit level
- Java offers two kinds of logical operators
  - **bitwise logical operators (&)**
  - **short-circuit logical operators (&&)**

# Short-Circuit Logical Operators

- The outcome of these operators is, of course, a boolean true or false
- The short-circuit logical operators may be used to build powerful conditions based on compound comparison

| Operator | Name | Usage | Outcome |
|---|---|---|---|
| && | Short-circuit logical AND | op1 && op2 | true if op1 and op2 are both true, otherwise false. Conditionally evaluates op2. |
| \|\| | Short-circuit logical OR | op1 \|\| op2 | true if either op1 or op2 is true, otherwise false. Conditionally evaluates op2. |

# Short-Circuit Logical Operators

- In case of short-circuit logical AND and OR operations, the second operand is only evaluated if the outcome of the overall operation cannot be determined from the evaluation of the first operand
- Example: if(5>2 && 1>2){….return ?????.}
- if(5>2 || 1>2){….return ?????.}

# Short-Circuit Logical AND: &&

Rule

| op1 | op2 | op1 && op2 |
|---|---|---|
| **true** | **true** | **true** |
| true | false | false |
| False | true | false |
| false | false | false |

# Short-Circuit Logical OR: | |

Rule

| op1 | op2 | op1 \|\| op2 |
|---|---|---|
| true | true | true |
| true | false | true |
| False | true | True |
| **false** | **false** | **false** |

# Combining Operators

```
1.  int i = 5;
2.  int j = 10
3.  int k = 15;
4.  if ( (i < j) || ( k++ > j) ) {
5.     System.out.println("First if, value of k: " + k);
6.  }
7.  if ( (i < j) && ( k++ < j) ) {
8.     System.out.println("Second if, value of k: " + k);
9.  }
10.System.out.println("Out of if, k:" + k);
```

**First if, value of k: 15**
**Out of if, value of k: 16**

# Bitwise Logical Operators

- bitwise operators that are used to manipulate the bits of an integer (**byte, short, char, int, long**) value
- These operators perform the boolean logic on a bit-by-bit basis

| Operator | Use | Operation |
|---|---|---|
| & | op1 & op2 | AND |
| \| | op1 \| op2 | OR |
| ^ | op1 ^ op2 | XOR |
| ~ | ~op | Bitwise inversion |
| ! | !op | NOT (Boolean inversion) |

# AND Operator: &

- Rule

| op1 | op2 | op1 & op2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Example:**

```
byte x = 117;
byte y = 89;
byte z = (byte) (x&y);
```

```
   01110101
&  01011001
-----------------
   01010001  = 81
```

# OR Operator: |

- Rule

| op1 | op2 | op1 \| op2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Example:**

```
byte x = 117;
byte y = 89;
byte z = (byte) (x|y);
```

```
  01110101
| 01011001
----------
  01111101  = 125
```

# XOR Operator: ^

- Rule

| op1 | op2 | op1 ^op2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Example:**

```
byte x = 117;
byte y = 89;
byte z = (byte) (x^y);
```

```
   01110101
^  01011001
----------------
   00101100  = 44
```

# The Bitwise Inversion Operator: ~

- This unary operator inverts the value of each bit of the operand

**Example:**

```
byte x = 117;
byte z = (byte) (~x);
```

```
~   01110101
-----------------
    10001010 =  -118
```

# The Boolean Inversion Operator: !

- This unary operator operates on a boolean operand and the outcome is the inversion of the value of the operand
- Note: Java doesn't like C, C++ using number as boolean value

**Example:**

```
boolean a = true;
boolean b = !a;
```

**b = false;**

# Bitwise operators - Sumary

| Operator | Use | Operation |
|---|---|---|
| & | op1 & op2 | Bitwise AND if both operands are numbers; |
| \| | op1 \| op2 | Bitwise OR if both operands are numbers;<br>Bitwise inclusive OR |
| ^ | op1 ^ op2 | Bitwise exclusive OR (XOR) |
| ~ | ~op | Bitwise complement |

```
int a = 13 & 12;  // a = 12
int a = 13 | 12;  // a = 13;
int a = 13 ^ 12;  // a = 1;
int a = ~(byte)129;// a = 126
```

# Shift Operators

| Operator | Use | Description |
|---|---|---|
| << (left) | op1 << op2 | Shifts bits of op1 left by distance op2; fills with 0 bits on the right side<br>Signed left shift |
| >> (right) | op1 >> op2 | Shifts bits of op1 right by distance op2; fills with highest (sign) bit on the left side<br>Signed right shift |
| >>> (right) | op1 >>> op2 | Shifts bits of op1 right by distance op2; fills with 0 bits on the left side<br>Unsigned right shift |

# Example - Left Shift Operator in Java

- Let us take an example to understand the concept of the Java left shift operator.
- 1. If int x = 20. Calculate x value if x << 3.

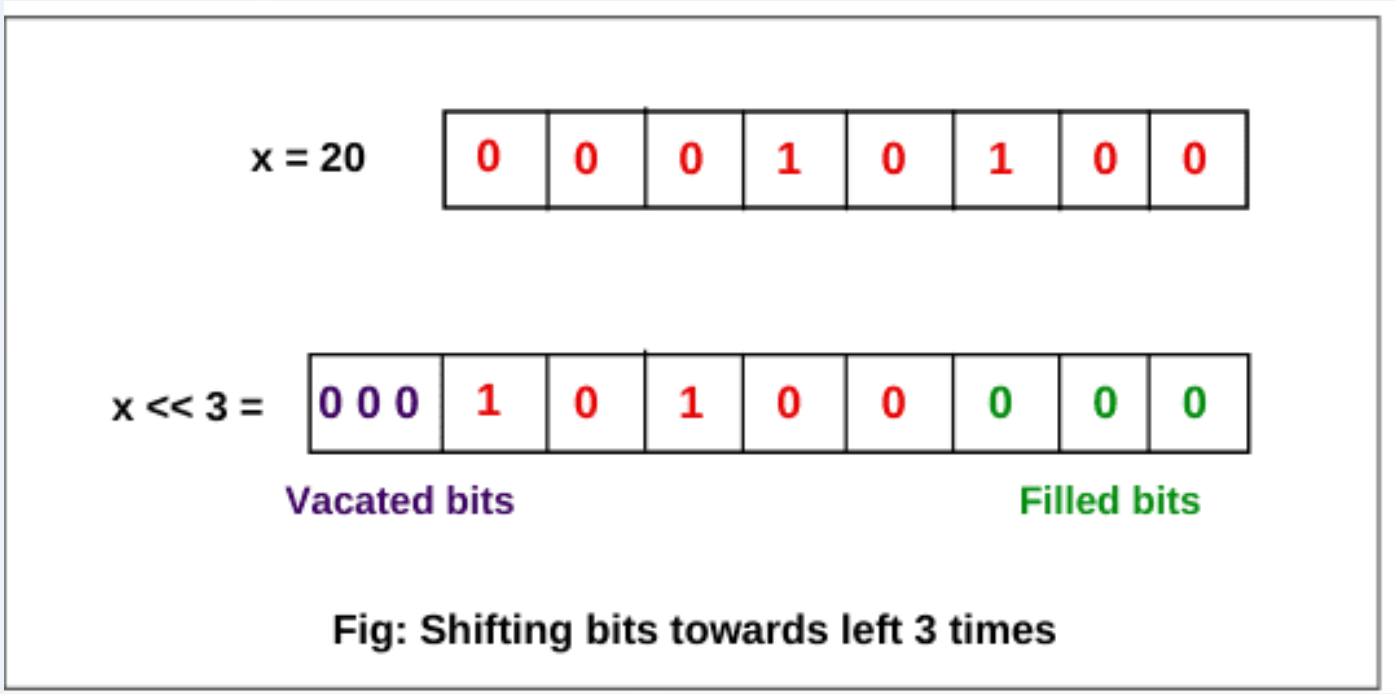



Fig: Shifting bits towards left 3 times

# Example - Right Shift Operator in Java

- Let's understand the concept of right shift operator with the help of an example.
- 1. If int x = 10 then calculate x >> 3 value.

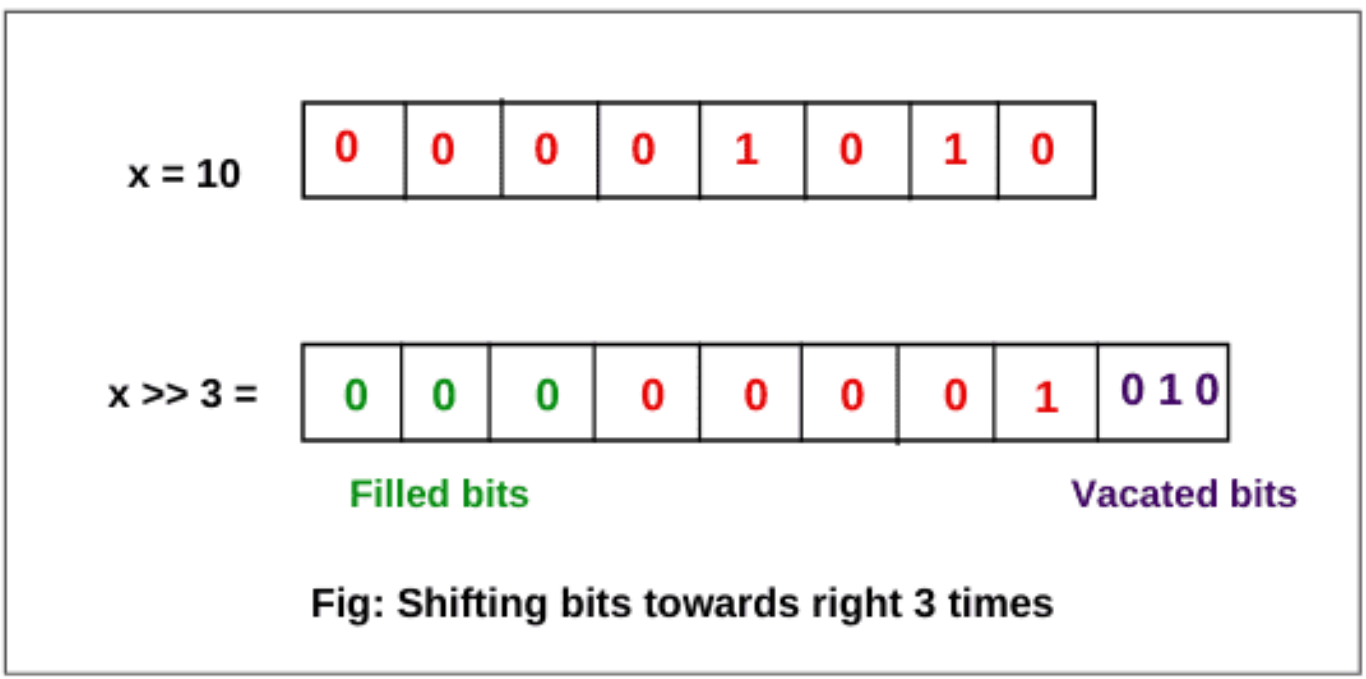


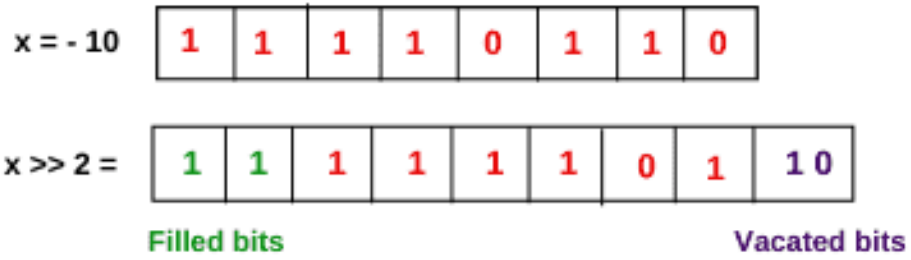

Fig: Shifting bits towards right 3 times

# Example - right shifting on a negative number

- Let's take an example that is based on right shifting on a negative number.
- 2. If int x = -10 then calculate x >> 2 value.

Step 1: +10 in 8-bit form: 0 0 0 0 1 0 1 0
Step 2: 1's complement form: 1 1 1 1 0 1 0 1
Step 3: 2's complement form: +1

2's complement form of -10: 1 1 1 1 0 1 1 0
Step 4:




Step 5: 1 1 1 1 1 1 0 1
- 1
1 1 1 1 1 1 0 0 1's complement form.
Step 6: 0 0 0 0 0 0 1 1 Taking complement of each bit. This is nothing but -3 in decimal form.

# Assignment - Shift Operators

- For examples:

1. int a = -13 >> 1;  // a = -7
2. int a = -13 >>> 1; // a = 2147483641
3. int a =  13 << 1;  // a = 26
4. int a = -13 << 1;  // a = -26

# Assignment

```
boolean b = (i > 10 & ((j = 20) > 10));
```

➔ *với i=5, j=15.*

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

→ ***Kết quả in ra là bao nhiêu***

```
boolean b = (i > 10 && ((j = 20) > 10));
System.out.println("b = " + b);
System.out.println("i = " + i);
System.out.println("j = " + j);
```

→ ***Kết quả in ra là bao nhiêu***

# Assignment Operators

- An assignment operator is used to set (or reset) the value of a variable

```
int x = 7;
```

- The assignment operators can be combined with other operators, and the resultant operators are called **shortcut assignment operators**
- The operands on the two sides of an assignment operator **do not have to be of the same type**

# Shortcut Assignment Operators

- Shortcut assignment operators that reduce down to the basic assignment operator =

x = x + y;  =>  x += y;

| Operator | Use | Equivalent To |
|---|---|---|
| += | op1 += op2 | op1 = op1 + op2 |
| -= | op1 -= op2 | op1 = op1 - op2 |
| *= | op1 *= op2 | op1 = op1 * op2 |
| /= | op1 /= op2 | op1 = op1 / op2 |
| %= | op1 %= op2 | op1 = op1 % op2 |
| &= | op1 &= op2 | op1 = op1 & op2 |
| \|= | op1 \|= op2 | op1 = op1 \| op2 |
| ^= | op1 ^= op2 | op1 = op1 ^ op2 |
| <<= | op1 <<= op2 | o1 = op1 << op2 |
| >>= | op1 >>= op2 | o1 = op1 >> op2 |
| >>>= | op1 >>>= op2 | o1 = op1 >>> op2 |

# Assignment Operators

| Operator | Use | Description |
|---|---|---|
| += | op1 += op2 | Equivalent to op1 = op1 + op2 |
| -= | op1 -= op2 | Equivalent to op1 = op1 - op2 |
| *= | op1 *= op2 | Equivalent to op1 = op1 * op2 |
| /= | op1 /= op2 | Equivalent to op1 = op1 / op2 |
| %= | op1 %= op2 | Equivalent to op1 = op1 % op2 |
| &= | op1 &= op2 | Equivalent to op1 = op1 & op2 |
| \|= | op1 \|= op2 | Equivalent to op1 = op1 \| op2 |
| ^= | op1 ^= op2 | Equivalent to op1 = op1 ^ op2 |
| <<= | op1 <<= op2 | Equivalent to op1 = op1 << op2 |
| >>= | op1 >>= op2 | Equivalent to op1 = op1 >> op2 |
| >>>= | op1 >>>= op2 | Equivalent to op1 = op1 >>> op2 |

# Other Operators

| Operator | Use | Description |
|---|---|---|
| `?:` | `op1 ? op2 : op3` | If `op1` is `true`, returns `op2`; otherwise, returns `op3` |
| `[]` | int`[]` array | Used to declare arrays, to create arrays, and to access array elements |
| `.` | System.out.println(“”) | Used to form long names |
| (*params*) | See Defining Methods | Delimits a comma-separated list of parameters |
| (`type`) | (*type*) op | `Casts` (converts) `op` to the specified type; an exception is thrown if the type of `op` is incompatible with *type* |
| *new* | new Aclass() | Creates a new object or array |
| *instanceof* | op1 `instanceof` op2 | Returns `true` if `op1` is an instance of `op2` |

# Operator Precedence

| Operators | Precedence |
|---|---|
| Postfix | `[]  . (params)  x++  x--` |
| Unary | `++x  --x  +x  -x  ~ !` |
| Creation or cast | `new  (type)x` |
| Multiplicative | `* / %` |
| Additive | `+ -` |
| Shift | `<< >> >>>` |
| Relational | `< > <= >= instanceof` |
| Equality | `== !=` |
| Bitwise AND | `&` |
| Bitwise exclusive OR | `^` |
| Bitwise inclusive OR | `\|` |
| Logical AND | `&&` |
| Logical OR | `\|\|` |
| Conditional | `? :` |
| Assignment | `= += -= *= /= %= &= ^= \|= <<= >>= >>>=` |

# Arithmetic Promotion

- What happen when binary operations between operands of different types ?
  - the compiler may convert the type of one operand to the type of the other operand, or the types of both operands to entirely different types
- This conversion, called arithmetic promotion, is performed before any calculation is done

# Arithmetic Promotion (cont.)

- The rules that govern arithmetic promotion in Java:

1. If both the operands are of a type narrower than int (that is **byte**, **short**, or **char**), then both of them are promoted to type **int**
2. If **one** of the operands is of type **double**, then the other operand is converted to **double** as well
3. If **none** of the operands is of type **double**, and **one** of the operands is of type **float**, then the other operand is converted to type **float** as well
4. If **none** of the operands is of type **double** or **float**, and one of the operands is of type **long**, then the other operand is converted to type **long** as well
5. If **none** of the operands is of type **double**, **float**, or **long**, then both the operands are converted to type **int**, if they already are not

# Ex: Shortcut Assignment Operators

```
byte b = 3;
b += 7;
```

**OK**

```
byte b = 3;
b = b + 7;
```

**Error: since b + 7 results in an int**

# Arithmetic Promotion (cont.)

```
byte x = 1;
byte y = 2;
```

What type of x / y ? int

```
byte b = 5;
int i = 3;
double d = b / i;
```

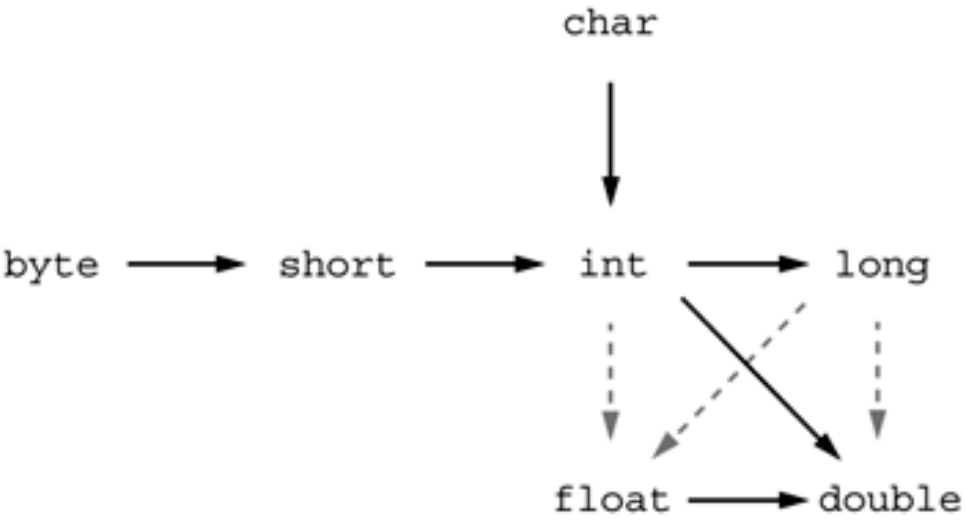
this result is promoted to double

b is promoted to type int

# Arithmetic Promotion (cont.)

- The preceding rules imply that the result of any binary arithmetic operation would be at least of type int
- Not any type can be converted to any other type
  - There will be situations in which you explicitly need to use an operator, called the **cast operator**, to convert one type to another

# Conversions Between Numeric Types




For example, a large integer such as **1234567890** has more digits than the float type can represent. When converting it to a float, it loses some precision

```
class Test {
public static void main(String[] args) {
        int big = 1234567890;
        float approx = big;
        System.out.println(big - (int)approx);
    }
}
```

→-46

# Casts

- The syntax for casting is to give the target type in parentheses, followed by the variable name. For example:
- **double x = 9.997;**
  **int nx = (int)x;**
- Then, the variable nx has the value 9, as casting a floating-point value to an integer discards the fractional part.
- If you want to round a floating-point number to the nearest integer (which is the more useful operation in most cases), use the `Math.round` method:
- **double x = 9.997;**
  **int nx = (int)Math.round(x);**
- Now the variable nx has the value 10. You still need to use the cast (int) when you call round. The reason is that the return value of the round method is a long, and a long can only be assigned to an int with an explicit cast since there is the possibility of information loss.

# The Cast Operator:(<type>)

- The cast operator explicitly converts a value to the specified type

```
byte x = 1;
byte y = 2;
byte z = x / y;
```

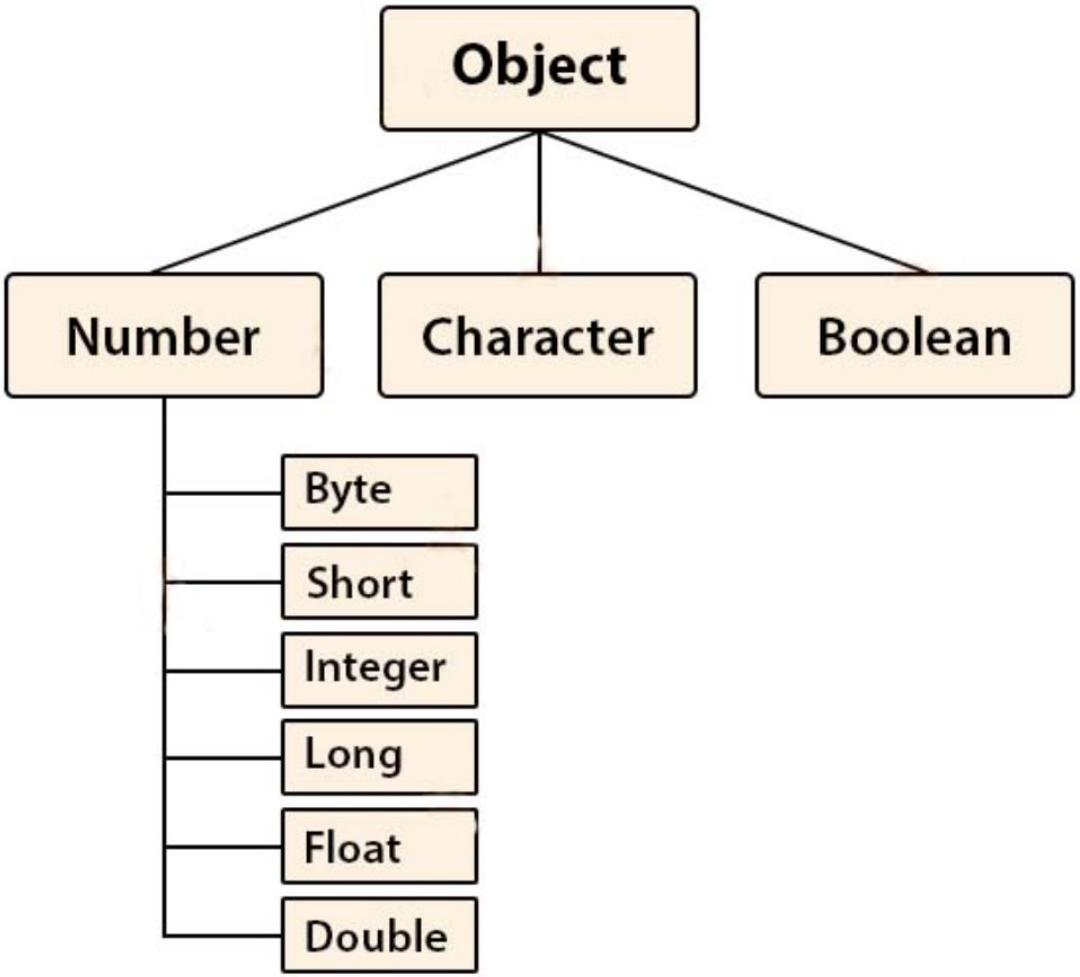
generate a compiler error

x/y is promoted to type int but z type byte

Should write:

**byte z = (byte) (x / y);**

# Wrapper Classes



# Wrapper Classes

- Primitives have no associated methods
- Wrapper classes:
  - Encapsulate primitives
  - Provide methods to work on them
  - Are included as part of the base Java API

| Primitive Type | Wrapper Class |
|---|---|
| boolean | Boolean |
| byte | Byte |
| char | Char |
| double | Double |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |

# Using Wrapper Classes

```
double number = Double.parseDouble("42.76");
```

```
String hex = Integer.toHexString(42);
```

```
double value = new Integer("1234").doubleValue();
```

```
String input = "test 1-2-3";
int output = 0;
for (int index = 0; index < input.length(); index++) {
   char c = input.charAt(index);
   if (Character.isDigit(c))
      output = output * 10 + Character.digit(c, 10);
}
System.out.println(output); // 123
```

# Initialize Wrapper Classes

- All wrapper classes provide two constructors (except the Character class):

```
Float wfloat = new Float("12.34f");

Float yfloat = new Float(12.34f);

Boolean wbool = new Boolean("false");

Boolean ybool =new Boolean(false);

Character c1 = new Character('c');
```

# Convert from String to wrapper classes

- Use static methods
  - valueOf(String s)
  - valueOf(String s, int radix)
- Integer i2 = Integer.valueOf("101011", 2);
  // converts 101011 to 43 and assigns the
  // value 43 to the Integer object i2
- Float f2 = Float.valueOf("3.14f");
  // assigns 3.14 to the Float object f2

# Returns the variable of the primitive data type

- use typeValue() method

```
// make a new wrapper object
        Integer i2 = new Integer(42);
// convert i2's value to a byte primitive
        byte b = i2.byteValue();
// another of Integer's xxxValue methods
        short s = i2.shortValue();
// yet another of Integer's xxxValue methods
        double d = i2.doubleValue();
```

# Convert from String to primitive data types

- Use static methods of the wrapper class

**static <type> parseType(String s)**

```
String s = "123";
//assign an int value of 123 to the int variable i
int i = Integer.parseInt(s);
//assign an short value of 123 to the short variable j
short j = Short.parseShort(s)
```

# The instanceof Operator

- The **instanceof** operator determines if a given object is of the type of a specific class
- the instanceof operator tests whether its first operand is an instance of its second operand
- The test is made at runtime
- The first operand is supposed to be the name of an object or an array element, and the second operand is supposed to be the name of a class, interface, or array type

# The instanceof Operator (cont.)

```
interface  X{}
class A  implements X {}
class B extends A {}
A a = new A();
B b = new B();
```

```
If (b instanceof X)
if (b instanceof B)
If (b instanceof A)
If (a instanceof A)
If (a instanceof X)
```

**true**

# Expressions, Statements

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

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);
int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```

# Expressions, Statements

- Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution.
- The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).
  - Assignment expressions
  - Any use of ++ or --
  - Method invocations
  - Object creation expressions

# The kinds of Statements

- Such statements are called *expression statements*. Here are some examples of expression statements.

```
aValue = 8933.234;        //assignment statement
aValue++;                 //increment
System.out.println(aValue); //method invocation
 //object creation statement
Integer integerObject = new Integer(4);
```

- A *declaration statement* declares a variable.

```
double aValue = 8933.234; //declaration stat.
```

- A *control flow statement* regulates the order in which statements get executed. The for loop and the if statement are both examples of control flow statements.

# Statements

- Terminated by a semicolon (;)
- Several statements can be written on one line, or
- Can be split over several lines

```
System.out.println("This is part of the same line");
```

```
a=0; b=1; c=2;
```

# Statements Blocks

- A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed:

```
class BlockDemo {
   public static void main(String[] args) {
      boolean condition = true;
      if (condition) { // begin block 1
         System.out.println("Condition is true.");
      } // end block one
      else { // begin block 2
         System.out.println("Condition is false.");
      } // end block 2
   }
}
```

# Assignment 1

- Write a Java program to count letters, spaces, numbers and other characters in an input string.
  *Expected Output*
- The string is : Aa kiu, I swd skieo 236587. GH kiu: sieo?? 25.33
- letter: 23
- space: 9
- number: 10
- other: 6

# Assignment 2

- Given string **str** of size **N** consisting of lowercase English alphabets, the task is to encode the given string as follows:
  - change every character of that string to another character
  - the distance between the changed character and the current character is the same as the distance between the current character and ‘a’.
  - Also, assume that the character’s array forms a cycle, i.e. after ‘z’ the cycle starts again from ‘a’.
  - Input: str = “geeksforgeeks”
    Output: “miiukkcimiiuk” `miiukkcimiiuk`

# Assignment 3

- Write a Java program to find the k largest elements in a given array. Elements in the array can be in any order.
- *Expected Output:*
  Original Array:
  [1, 4, 17, 7, 25, 3, 100]
  3 largest elements of the said array are:
  100 25 17

# Math and Input / output

# Basic Mathematical Routines

- Static methods in the Math class
  - Call `Math.cos()`, `Math.random()`, etc.
  - Most operate on double precision floating point numbers
- Simple operations:
  - **`pow`** ($x^y$), **`sqrt`** ($\sqrt{x}$), **`cbrt`**, **`exp`** ($e^x$), **`log`** ($\log_e$), **`log10`**
- Trig functions:
  - **`sin`**, **`cos`**, **`tan`**, **`asin`**, **`acos`**, **`atan`**
  - Args are in radians, not degrees, (see **`toDegrees`** and **`toRadians`**)
- Rounding and comparison:
  - **`round/rint`**, **`floor`**, **`ceiling`**, **`abs`**, **`min`**, **`max`**
- Random numbers:
  - **`Math.random()`** returns from 0 inclusive to 1 exclusive

# More Mathematical Routines

- Special constants
  - `Double.POSITIVE_INFINITY`
  - `Double.NEGATIVE_INFINITY`
  - `Double.NAN`
  - `Double.MAX_VALUE`
  - `Double.MIN_VALUE`
- Unlimited precision libraries
  - `BigInteger`, `BigDecimal`
  - Contain the basic operations, plus `BigInteger` has `isPrime`

# Reading Simple Input

- For simple testing, use standard input
  - Convert if you want numbers. Two main options:
    - Use Scanner class

```
Scanner input = new Scanner(System.in);
int i = input.nextInt();
double d = input.nextDouble();
String s = input.nextLine();
```

- In real applications, use a GUI
  - Collect input with textfields, sliders, combo boxes, …
  - Convert to numeric types with `Integer.parseInt`, `Double.parseDouble`, …

# Example: Printing Random Numbers

```
import java.util.*;
public class RandomNums {
  public static void main(String[] args) {
    System.out.print("How many random nums? ");
    Scanner inputScanner = new Scanner(System.in);
    int n = inputScanner.nextInt();
    for (int i=0; i<n; i++) {
      System.out.println("Random num " + i
          + " is " + Math.random());
    }
  }
}
```

```
How many random nums? 5
Random num 0 is 0.2268636967083570 4
Random num 1 is 0 078376852713779 7
Random num 2 is 0.17918121951887145
Random num 3 is 0.3441924454634313
Random num 4 is 0.6131053203170818
```

# Questions

- Consider the following code snippet.
  int i = 10;
  int n = i++%5;
  - What are the values of i and n after the code is executed?
  - What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i))?
- What is the value of i after the following code snippet executes?
  int i = 8; i >>=2;
- What is the value of i after the following code snippet executes?
  int i = 17;
  i >>=1;

# Exercises

- MyDate
- Hãy thiết kế lớp MyDate chỉ có 1 thuộc tính duy nhất date là kiểu số nguyên int, dùng để lưu trữ các thông tin về ngày, tháng và năm.
- Viết constructer và các getter, setter tương ứng
- Mydate(int year, int month, int day)
- void setYear(int year)
- void setMonth(int month)
- void setDay(int day)
- int getYear(), int getMonth(); int getDay();

# Exercises

- Sử dụng toán tử xor viết ứng dụng mã hóa và giải mã đối xứng