Java and J++ Reference Notes

Learning Java Operators, Data Types and Control Flow Structures by Comparing to VB

Introduction

The tables given below can be used to translate VB expressions and statements into equivalent Java statements and expressions. By using your extensive knowledge of VB in conjunction with the translation guide given below, it should not take you very long to learn how to write simple Java programs.

"**pow**" is ***not*** an operator. It is a mathematical function found in `java.lang.Math`.

Operators in VB and Java

Operator	VB	VB Example	Java	Java Ex.
Arithmetic Operators				
Unary Plus	+	A = +2.35E23	+	a = +2.35e23;
Unary Minus	–	A = –2.35E23	–	a = –2.35e23;
Exponent	^	A = B ^ C	**pow**	a = pow (b, c);
Multiplication	*	A = B * C	*	a = b * c;
Division	/	A = B / C	/	a = b / c;
Integer Division	\	A = B \ C	/	a = b / c;
Remainder (mod)	**Mod**	A = B **Mod** C	%	a = b % c;
Addition	+	A = B + C	+	a = b + c;
Subtraction	–	A = B – C	–	a = b – c;
Shortcut Increment and Decrement Operators				
Postfix Increment	N/A	A(I) = 3 I = I + 1	++	a[i++] = 3;
Prefix Increment	N/A	I = I + 1 A(I) = 3	++	a[++i] = 3;
Postfix Decrement	N/A	A(I) = 3 I = I – 1	– –	a[i– –] = 3;
Prefix Decrement	N/A	I = I – 1 A(I) = 3	– –	a[– –i] = 3;
Assignment Operator				
Assignment	=	A = B + C	=	a = b + c;
Shortcut Assignment Operators				
	N/A	X= X + Y	+=	x += y;
	N/A	X= X – Y	– =	x – = y;
	N/A	X= X * Y	*=	x *= y;
	N/A	X= X / Y	/=	x /= y;
	N/A	X= X **Mod** Y	%=	x %= ;y
	N/A	X= X **And** Y	&=	x &= y;
	N/A	X= X **Or** Y	|=	x |= y;
	N/A	X= X **Xor** Y	^=	x ^= y;
	N/A	N/A	<<=	x <<= y;
	N/A	N/A	>>=	x >>= y;
	N/A	N/A	>>>=	x >>>= y;

Operator	VB	VB Example	Java	Java Ex.
Comparison (Relational) Operators				
Greater than	>	**If** X > 2 **Then**	>	**if** (x > 2)
Less than	<	**If** X < 2 **Then**	<	**if** (x < 2)
Greater than or Equal to	>=	**If** X >= 2 **Then**	>=	**if** (x >= 2)
Less than or Equal to	<=	**If** X <= 2 **Then**	<=	**if** (x <= 2)
Equal to	=	**If** X = 2 **Then**	==	**if** (x == 2)
Not Equal to	< >	**If** X <> 2 **Then**	!=	**if** (x != 2)
Boolean (aka Conditional or Logical)Operators				
Boolean AND	**And**	**If** X>2 **And** Y=1 _ **Then**	&	**if** (x>2 **&** y=1)
Boolean OR	**Or**	**If** X>2 **Or** Y=1 **Then**	|	**if** (x>2 | y=1)
Boolean NOT	**Not**	**If Not** Sorted **Then**	!	**if** (!sorted)
Boolean Exclusive OR	**Xor**	**If** X>2 **Xor** Y=1 _ **Then**	^	**if** (x>2 ^ y=1)
Conditional Boolean AND	N/A	N/A	&&	**if** (x>2 **&&** y=1)
Conditional Boolean OR	N/A	N/A	||	**if** (x>2 || y=1)
Boolean XNOR	**Eqv**	**If** X>2 **Eqv** Y=1 _ **Then**	N/A	**if** (!(x>2 | y=1))
Logical Implication	**Imp**	**If** X>2 **Imp** Y=1 _ **Then**	N/A	**if** (!(x>2) | x>2 & y=1)
Bitwise and Shift Operators				
Bitwise AND	**And**	X = Y **And** Z	&	x = y & z;
Bitwise OR	**Or**	X = Y **Or** Z	|	x = y | z;
Bitwise XOR	**Xor**	X = Y **Xor** Z	^	x = y ^ z;
Bitwise Complement	**Not**	X = **Not** Y	~	x = ~y;
Bitwise XNOR	**Eqv**	X = Y **Eqv** Z	N/A	x = ~(y | z);
Bitwise Logical Implication	**Imp**	X = Y **Imp** Z	N/A	x = ~y | y & z;
Bitwise Left Shift	N/A	N/A	<<	x = y << z;
Signed Bitwise Right Shift	N/A	N/A	>>	x = y >> z;
Unsigned Bitwise Right Shift	N/A	N/A	>>>	x = y >>> z;
Other Operators				
Cast	In VB, type conversions are done with intrinsic functions.		( )	x = (**double**) y;
Ternary Conditional Operator	N/A		?:	x=(y<z) ? y : z;

Note: There are a few other operators in Java that will be included in a separate table.

Summary of Java Operators (from http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opsummary.html)

Summary of Arithmetic Operators

The following table lists the basic arithmetic operators provided by the Java programming language.			These short cut operators increment or decrement a number by one.		
Operator	**Use**	**Description**	**Operator**	**Use**	**Description**
+	*op1* + *op2*	Adds *op1* and *op2* (numeric). Concatenates *op1* and *op2* (string).	++	*op*++	Increments *op* by 1; evaluates to the value of op before *op* is incremented
–	*op1* – *op2*	Subtracts *op2* from *op1*	++	++*op*	Increments *op* by 1; evaluates to the value of *op* after it was incremented
*	*op1* * *op2*	Multiplies *op1* by *op2*	– –	*op*– –	Decrements *op* by 1; evaluates to the value of op before it was decremented
/	*op1* / *op2*	Divides *op1* by *op2*	– –	– –*op*	Decrements *op* by 1; evaluates to the value of *op* after it was decremented
%	*op1* % *op2*	Computes the remainder of dividing *op1* by *op2*			

Here are the Java programming language's other arithmetic operators.

Operator	**Use**	**Description**
+	+*op*	Promotes *op* to **int** if it's a **byte**, **short**, or **char**.
–	–*op*	Arithmetically negates *op*.

Summary of Relational and Conditional Operators

Use these *relational operators* to determine the relationship between two values.			You can use the following *conditional operators* to form multi-part decisions.		
Operator	**Use**	**Returns true if**	**Operator**	**Use**	**Returns true if**
>	*op1* > *op2*	*op1* is greater than *op2*	&&	*op1* && *op2*	*op1* and *op2* are both true, conditionally evaluates *op2*
>=	*op1* >= *op2*	*op1* is greater than or equal to *op2*	||	*op1* || *op2*	either *op1* or *op2* is true, conditionally evaluates *op2*
<	*op1* < *op2*	*op1* is less than *op2*	!	! *op*	*op* is false
<=	*op1* <= *op2*	*op1* is less than or equal to *op2*	&	*op1* & *op2*	*op1* and *op2* are both true, always evaluates *op1* and *op2*
= =	*op1* = = *op2*	*op1* and *op2* are equal	|	*op1* | *op2*	either *op1* or *op2* is true, always evaluates *op1* and *op2*
!=	*op1* != *op2*	*op1* and *op2* are not equal	^	*op1* ^ *op2*	if *op1* and *op2* are different, that is, if one or the other of the operands is true but not both

Summary of Shift and Logical Operators

Each shift operator shifts the bits of the left-hand operand over by the number of positions indicated by the right-hand operand. The shift occurs in the direction indicated by the operator itself.			These operators perform logical functions on their operands.		
Operator	**Use**	**Operation**	**Operator**	**Use**	**Operation**
>>	*op1* >> *op2*	shift bits of *op1* right by distance *op2*	&	*op1* & *op2*	bitwise and
<<	*op1* << *op2*	shift bits of *op1* left by distance *op2* (signed)	|	*op1* | *op2*	bitwise or
>>>	*op1* >>> *op2*	shift bits of *op1* right by distance *op2* (unsigned)	^	*op1* ^ *op2*	bitwise xor
			~	~*op2*	bitwise complement

Summary of Assignment Operators

The basic assignment operator looks as follows and assigns the value of op2 to op1: **op1 = op2;**

In addition to the basic assignment operation, the Java programming language defines these shortcut assignment operators that perform an operation and an assignment using one operator.

Operator	**Use**	**Equivalent to**	**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*	*op1* = *op1* % *op2*	>>>=	*op1* >>>= *op2*	*op1* = *op1* >>> op2
&=	*op1* &= *op2*	*op1* = *op1* & *op2*			

Summary of Other Operators

The Java programming language also supports these operators.

Operator	**Use**	**Description**
?:	*op1* ? *op2* : *op3*	If *op1* is **true**, returns *op2*. Otherwise, returns *op3*.
[ ]	type [ ]	Declares an array of unknown length, which contains elements of type **type**.
[ ]	type[*op1*]	Creates an array with *op1* elements. This must be used with the **new** operator.
[ ]	*op1*[*op2*]	Accesses the element at *op2* index within the array *op1*. Indices begin at 0 and extend through the length of the array minus one.
.	*op1*.*op2*	Is a reference to the *op2*, member of *op1*.
( )	*op1*(params)	Declares or calls the method named *op1* with the specified parameters. The list of parameters can be an empty list. The list is "comma-separated."
(*type*)	(*type*) *op1*	Casts (converts) *op1* to *type*. An exception will be thrown if the type of *op1* is incompatible with *type*.
new	**new** *op1*	Creates a new object or array. Note that *op1* is either a call to a constructor or an array specification.
instanceof	*op1* **instanceof** *op2*	Returns true if *op1* is an instance of *op2*.

Primitive Data Types in VB and Java

Visual Basic			*Java*		
Data Type	***Storage***	***Range***	***Data Type***	***Storage***	***Range***
Byte	1 byte (8 bits)	0 to 255 (0 to $2^8 - 1$)	**byte**	1 byte (8 bits)	–128 to 127 (-2^7 to $2^7 - 1$)
Integer	2 bytes (16 bits)	-32,768 to 32,767 (-2^{15} to $2^{15} - 1$)	**short**	2 bytes (16 bits)	–32,768 to 32,767 (-2^{15} to $2^{15} - 1$)
Long	4 bytes (32 bits)	-2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31} - 1$)	**int**	4 bytes (32 bits)	–2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31} - 1$)
N/A	N/A	N/A	**long**	8 bytes (64 bits)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (-2^{63} to $2^{63} - 1$)
Single	4 bytes (32 bits)	–3.402823E38 to –1.401298E-45 for negative values; 1.401298E–45 to 3.402823E38 for positive values	**float**	4 bytes (32 bits)	–3.402823E38 to –1.401298E-45 for negative values; 1.401298E–45 to 3.402823E38 for positive values
Double	8 bytes (64 bits)	–1.79769313486232E308 to –4.94065645841247E-324 for negative values; 4.94065645841247E–324 to 1.79769313486232E308 for positive values	**double**	8 bytes (64 bits)	–1.79769313486232E308 to –4.94065645841247E-324 for negative values; 4.94065645841247E–324 to 1.79769313486232E308 for positive values
Currency	8 bytes (64 bits)	–922,337,203,685,477.5808 to 922,337,203,685,477.5807 (Used to store money values.)	**N/A**	N/A	N/A
Decimal*	14 bytes (112 bits)	+/–79,228,162,514,264,337,593,543,950,335 with no decimal point; +/–7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest non-zero number is +/–0.0000000000000000000000000001	**N/A**	N/A	N/A
String (variable-length)	10 bytes (80 bits) + string length	0 to approximately 2 billion characters	**N/A**	N/A	N/A
String (fixed-length)	Length of string	1 to approximately 65,400 characters	**N/A**	N/A	N/A
N/A	N/A	N/A	**char**	16-bit Unicode character	0 to 65535 (64K possible values, usually used for Unicode characters but can also be used for integers)
Boolean	2 bytes (16 bits)	**True** or **False**	**boolean**	1 bit	**true** or **false**
Date	8 bytes (64 bits)	January 1, 100 to December 31, 9999	**N/A**	N/A	N/A
Object	4 bytes (32 bits)	Any Object reference	**N/A**	N/A	N/A
Variant (with numbers)	16 bytes (128 bits)	Any numeric value up to the range of a **Double**	**N/A**	N/A	N/A
Variant (with characters)	22 bytes (176 bits) + string length	Same range as for variable-length **String**	**N/A**	N/A	N/A

Variable Declarations in VB and Java

Variable Declarations in VB

VB Declaration Keywords

The following keywords are used to ***declare variables***:

Dim, **ReDim, Public**, **Private, Static**

- **Dim** is used to declare variables at the ***module level*** (global to a form module or code module) or at the ***procedure level*** (local to a **Sub** or **Function** procedure). Variables declared at the ***module level*** using the **Dim** keyword are by default **Private** (see below).
- **ReDim** is used at the ***procedure level*** (local to a **Sub** or **Function** procedure) to change the size of a ***global*** array that has already been declared with **Dim**.
- **Public** is used to declare variables at the ***module level*** (global to a form module or code module). A variable declared as **Public** can be accessed by other form modules or code modules.
- **Private** is used to declare variables at the ***module level*** (global to a form module or code module). A variable declared as **Private** ***cannot*** be accessed by other form modules or code modules. Variables declared at the ***module level*** using the **Dim** keyword are by default **Private**.
- **Static** is used to declare variables at the ***procedure level*** (local to a **Sub** or **Function** procedure). Unlike local variables declared using **Dim**, the value of a **Static** variable is ***retained*** once the **Sub** or **Function** returns. (This means that the value of such a variable will be saved for the next call of the **Sub** or **Function**.)

Exercise

Write a few VB variable declarations. Make sure that you use each of the keywords listed above ***at least once.***

Variable Declarations in Java

Note

In Java, there are no special keywords that are used to declare variables. Instead, the primitive data type keywords are used in declarations.

Simplest Form of Declaration

The simplest variable declarations in Java take the following form:

primitiveDataType var1, var2, var3, …;

where *primitiveDataType* is one of **byte**, **char**, **short**, **int**, **long**, **float**, **double** or **boolean**.

For example, the statement

int x, y, z;

declares three variables, x, y, and z of type **int**.

Variables can also be ***initialized*** in declarations as shown in the following example:

int x=2, y=3, z=4;

Java Declaration Modifiers

The following Java keywords are used to ***modify*** variable and method declarations:

public, **private**, **protected**, **static**, **final**, **volatile**, **transient**

- **public** can be used to modify variable (i.e. data field) and method declarations at the ***class level*** (global to a class). A **public** variable (i.e. data field) or method ***is visible*** (can be accessed) everywhere its class is visible.
- **private** can be used to modify variable (data field) and method declarations at the ***class level*** (global to a class). A **private** variable (i.e. data field) or method ***is not visible*** (cannot be accessed) outside its class.
- **protected** can be used to modify variable (data field) and method declarations at the ***class level*** (global to a class). A protected variable or method is ***only visible*** (can only be accessed) within its class, within its subclasses or within the class package.
- **static** can be used to modify variable and method declarations at the ***class level*** (global to a class). No matter how many instances of a given class are created, only a ***single copy*** of each static data field will exist.
- **final** is used along with **static** to declare ***constants***
- **volatile** and **transient** are used for more advanced applications and will not be discussed here (see MSDN or Sun's Java Language Specification.)

Examples

```
public int initialVelocity=100;
private int finalVelocity=100;
private static int averageVelocity=100;
public static final float PI=3.14159;  //Public constant
private static final float PI=3.14159;  //Private constant
```

Naming Conventions in Java

Variable Names, Object Names and Method Names

Variable names, object names and method names should begin with a lowercase letter. All other letters in the variable name should also be in lowercase ***except*** for the first letter of each "word" in the variable name (in the case that the variable name consists of two or more words).

e.g. `surname, givenName, dayOfWeek, buttonQuit, editName, setText, getText`

Class Names and Constructor Method Names

The same conventions for naming variables and objects should be used for class and constructor method names ***except*** that the first character of a class name should always be an uppercase letter. ***Note that the constructor methods for a class must always have the same name as the class!***

e.g. `String, Button, FormRomanConverter, Edit`

Constant Names

Constant names should consist entirely of uppercase letters and underscores.

e.g. `PI, SECS_IN_DAY, HOURS_IN_DAY, DAYS_IN_YEAR`

What's the Difference between…

A Class and an Object?

You should think of a class as a ***blueprint*** or ***template*** for creating ***objects***. A class contains all the ***methods*** and ***data fields*** needed to cause an object to behave in the desired manner. Just as a single set of blueprints can be used to ***construct*** as many houses as you like, a single class can be used to create as many objects as you like. The constructor methods of a class are analogous to the contractors who build a house.

In many ways, a class is similar to a primitive data type. ***Primitive data types*** are used to ***create variables*** while ***classes*** are used to ***create objects***.

What's the difference between an object and a variable? Read on!

An Object and a Variable?

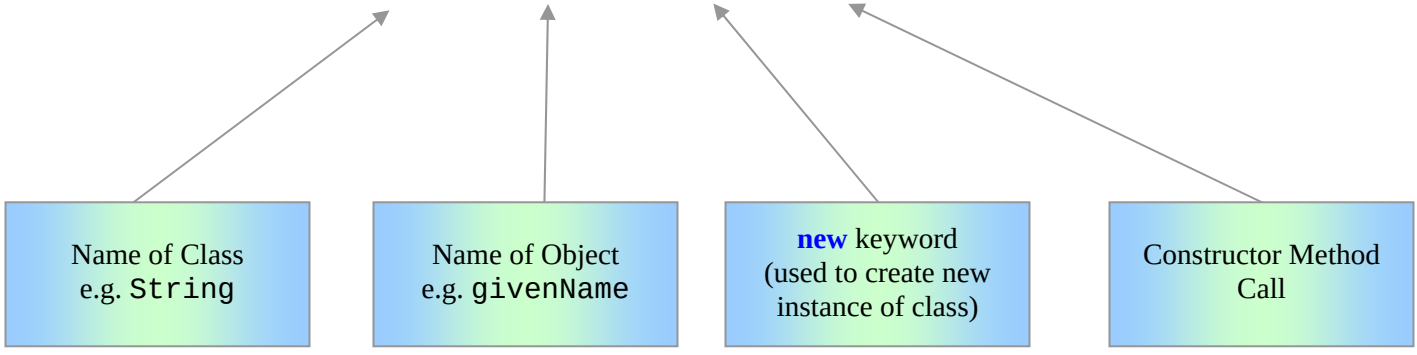
Variables and objects are closely related. In fact, it is possible to think of a ***variable*** as an ***object containing only one data field***. The main difference between the two is that objects have a much richer structure. This is analogous to atoms and molecules. While variables (the "atoms") can only store a single value, ***objects*** (the "molecules") consist of ***data fields*** (variables) and ***methods*** (functions).

For example, a `String` object consists of much more than just a particular string's value. A `String` object also contains a large number of ***methods*** for manipulating the string value.

Class Instantiations

When you ***instantiate*** a class, you create an ***object***, that is, a concrete ***instance*** of the class. Most class instantiations in Java take the following form:

```
ClassName objectName = new ClassName(parameters);
```



e.g. `String givenName = new String("Joe");`

Several Examples of Array Declarations

```
double[] temperature;
                    or
double temperature[];
```

In this example, a variable of array type is declared but no array object is created nor is any storage space allocated for the elements (*components*) of the array.

Number of elements in the array.

```
String[] name=new String[4];
                    or
String name[]=new String[4];
```

Index	0	1	2	3
Data	""	""	""	""

In Java, arrays are implemented as ***objects***. Therefore, you need to use the **new** keyword in the declaration of an array to create a new array object.

```
int[] height={160, 175, 182, 191};
                    or
int height[]={160, 175, 182, 191};
```

Index	0	1	2	3
Data	160	175	182	191

Arrays can be declared, created and initialized on the same line. In such cases, the **new** keyword should not be used. The array object is automatically created when a list of initializers is included.

```
float[][] distance= new float[2][3];
                    or
float distance[][]= new float[2][3];
                    or
float[] distance[]= new float[2][3];
```

	0	1	2
0	-	-	-
1	-	-	-

The statements shown at the left can be used to declare and create a ***two-dimensional*** array of **float** values. The row indices run from 0 to 1 and the column indices run from 0 to 2. Without any assignment statements, however, the two-dimensional array is empty (i.e. the elements have no value).

```
distance[0][0] = 0
distance[0][1] = 10.7
distance[0][2] = 25.3
distance[1][0] = 10.7
distance[1][1] = 0
distance[1][2] = 16.3
```

	0	1	2
0	0	10.7	25.3
1	10.7	0	16.3

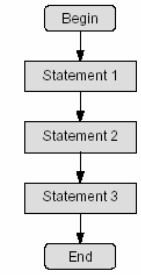
Once the assignment statements at the left are executed, the two-dimensional array (***matrix***) will contain the values shown above.

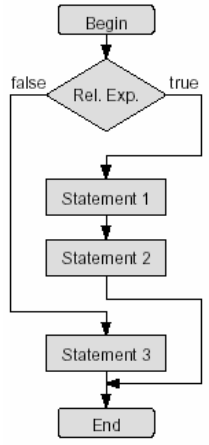
```
float[][] distance={{0, 10.7, 25.3},
                    {10.7, 0, 16.3}};
```

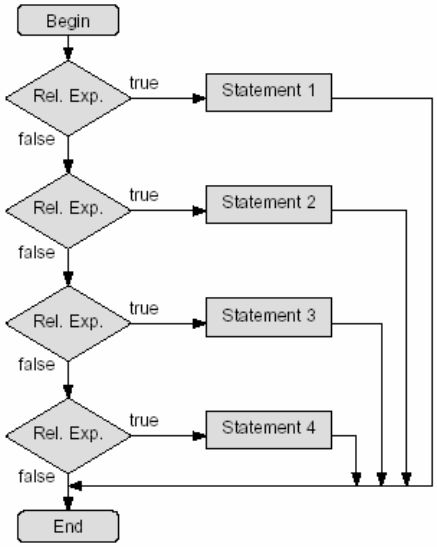
This statement is an alternative (and probably preferable) method of declaring, creating and initializing the two-dimensional array shown above. Each row of the matrix is enclosed in braces and listed in the desired order.

Program Control Flow – Sequence, Selection and Repetition

Essential Selection Structures in VB and Java

Control Flow Structure	VB	Java
Sequence 	*statement1* *statement2* *statement3*	*statement1;* *statement2;* *statement3;*

Selection (Example 1)
 
	Structure
	If *relationalExpression* **Then**
	statement1
	statement2
	Else
	statement3
	End If
	
	Example
	If X > 3 **Then**
	X = X - 4
	Y = Y + X
	Else
	X = X + 4
	End If
		Structure
		if (*relationalExpression*)
		{
		statement1;
		statement2;
		}
		else
		statement3;
		
		Example
		if (x > 3)
		{
		x = x -4;
		y = y + x;
		}
		else
		x = x + 4;

Selection (Example 2)
 
	Structure
	If *relationalExpression1* **Then**
	statement1
	ElseIf *relationalExpression2* **Then**
	Statement2
	ElseIf *relationalExpression3* **Then**
	Statement3
	ElseIf *relationalExpression4* **Then**
	Statement4
	.
	.
	.
	End If
	
	Example
	If X > 3 **Then**
	X = X - 4
	ElseIf X >= 0 **And** X < 3 **Then**
	X = X + 4
	ElseIf X >= -3 **And** X < 0 **Then**
	X = X – 8
	ElseIf X >= -9 **And** X < -3 **Then**
	X = X + 8
	End If
		Structure
		if (*relationalExpression1*)
		statement1;
		else if (*relationalExpression2*)
		statement2;
		else if (*relationalExpression3*)
		statement3;
		else if (*relationalExpression4*)
		statement4;
		.
		.
		.
		
		Example
		if (x > 3)
		x = x -4;
		else if (x >= 0 & x < 3)
		x = x + 4;
		else if (x >= -3 & x < 0)
		x = x - 4;
		else if (x >= -9 & x < -3)
		x = x + 8;

Please Note: In Java, C and C++, brace brackets (i.e. **"{"** and **"}"**) are used to group one or more statements to create a ***compound statement***. Brace brackets are used in "if" statements, loops, method definitions, class definitions and a few other structures. The braces ***must be used to group two or more statements.*** Braces are ***optional*** whenever a group ***consists of only one statement***.

Advanced Selection Structures in VB and Java

VB

Structure

```
Select Case testExpression
    Case expressionList1
        Statements
    Case expressionList2
        Statements
    Case expressionList3
        Statements
    .
    .
    .
    Case Else
       statements
End Select
```

Example

```
Select Case Number
   Case 1 To 5
      Debug.Print "Between 1 and 5"
   Case 6, 7, 8
      Debug.Print "Between 6 and 8"
   Case 9 To 10
      Debug.Print "Greater than 8"
   Case Else
      Debug.Print "Not between 1 and 10"
End Select
```

Equivalent To

```
If Number >= 1 And Number <= 5 Then
    Debug.Print "Between 1 and 5"
ElseIf Number >= 6 And Number <= 8 Then
    Debug.Print "Between 6 and 5"
ElseIf Number >= 9 And Number <= 10 Then
    Debug.Print "Greater than 8"
Else
   Debug.Print "Not between 1 and 10"
End If
```

Note

The "**Select** … **Case**" statement in VB is much more flexible than the "**switch**" statement in Java. See msdn.microsoft.com for more details.

Java

Structure

```
switch (integerExpression)
{
    case 0:
        statements
        break;
    case 1:
        statements
        break;
    case 2:
        statements
        break;
    .
    .
    .
    default:
        statements
}
```

Example

```
int bracket = Integer.parseInt(stdin.readLine());
switch (bracket)
{
   case 1:
      System.out.println("Pay no taxes");
      break;
   case 2:
      System.out.println("Pay 20% taxes");
      break;
   case 3:
      System.out.println("Pay 30% taxes");
      break;
   default:
      System.out.println("Error: bad input");
}
```

Equivalent To

```
int bracket = Integer.parseInt(stdin.readLine());
if (bracket == 1)
    System.out.println("Pay no taxes");
else if (bracket == 2)
    System.out.println("Pay 20% taxes");
else if (bracket == 3)
    System.out.println("Pay 30% taxes");
else
    System.out.println("Error: bad input");
```

N/A

Structure

```
rel_exp ? exp1 : exp2
```

Example

```
z = a < b ? a : b;
```

Equivalent To

```
if (a < b)
    z = a;
else
    z = b;
```

Essential Repetition Structures in VB and Java

VB

Structure

```
Do While RelationalExpression
    statement1
    statement2
    .
    .
    .
Loop
```

Example

```
FactorsOfTwo = 0
Num = Num \ 2
Do While Num > 0
    FactorsOfTwo = FactorsOfTwo + 1
    Num = Num \ 2
Loop
```

Note

In VB, a "**Do**" loop can be ended prematurely by using the "**Exit Do**" statement. A "**For**" loop can be ended prematurely using the "**Exit For**" statement. Although it is often extremely convenient to exit from loops and other structures prematurely, programs that do so are almost impossible to verify for correctness and can be extremely difficult to debug. Therefore, such statements should be used sparingly, if at all.

Structure

```
Do
    statement1
    statement2
    .
    .
    .
Loop While RelationalExpression
```

Example

```
NumDivisionsByTwo = 0
Do
   NumDivisionsByTwo = NumDivisionsByTwo + 1
   Num = Num \ 2
Loop While Num > 0
```

Java

Structure

```
/* "while" loop with a single statement
    Braces are not needed. */
while (relationalExpression)
   statement;

/* "while" loop with a compound statement
    Braces are needed. */
while (relationalExpression)
{
   statement1;
   statement2;
   .
   .
   .
}
```

Examples

```
while (x > 10)
   x = x - 1;

factorsOfTwo = 0;
num = num / 2;
while (num > 0)
{
   factorsOfTwo = factorsOfTwo + 1;
   num = num / 2;
}
```

Note

In Java, a loop can be ended prematurely by using the "**break**;" statement (as was the case with the "**switch**" statement).

Structure

```
//"do while" loop with a single statement
do
    statement;
while (relational_expression);

//"do while" loop with a compound statement
do
{
    statement1;
    statement2;
    .
    .
    .
} while (relational_expression);
```

Example

```
numDivisionsByTwo = -1
do
{
   numDivisionsByTwo++;
   num = num / 2;
} while (num > 0)
```

VB	Java

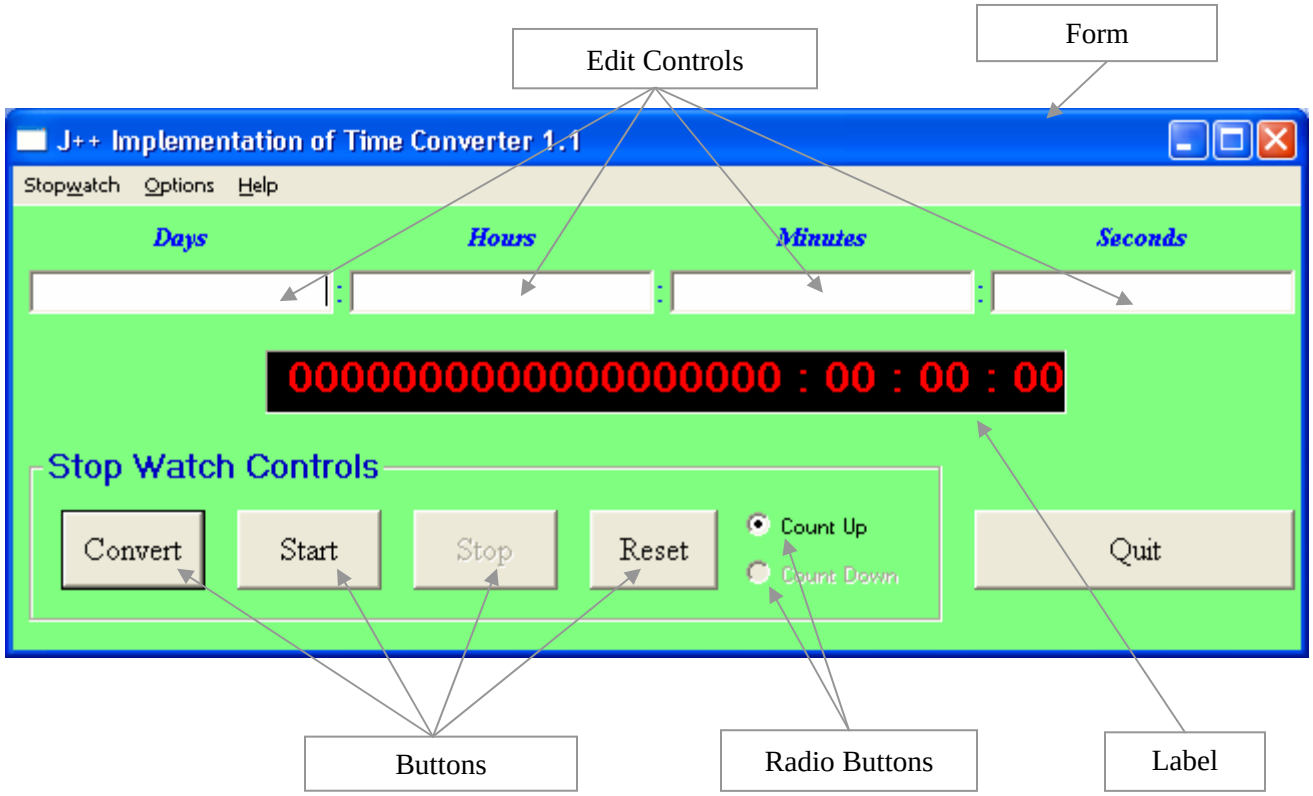
Structure
For *counter* **=** *start* **To** *end* [**Step** *step*]
statement1
statement2
.
.
.
Next [*counter*]
Examples
`'Evaluate 0+1+2+3+4`
`Dim I As Long, Sum As Integer`
`Sum = 0`
`For I = 0 To 4`
`Sum = Sum + I`
`Next I`
 	
`'Important Exercise: Rewrite the strange`
`'Java program segment at the right in VB.`
`'What difficulties do you encounter when`
`'you try to use a "For … Next" loop in your`
`'VB code?`
	Structure
	`/* "for" loop with a single statement`
	`Braces are not needed. */`
	`for (expr1; relExpr; expr2)`
	`statement;`
	`/* "for" loop with a compound statement`
	`Braces are needed. */`
	`for (expr1; relExpr; expr2)`
	`{`
	`statement1;`
	`statement2;`
	`.`
	`.`
	`.`
	`}`
	Examples
	`//Evaluate 0+1+2+3+4`
	`int sum = 0;`
	`for (int i = 0; i < 5; i++)`
	`sum += i;`
	
	`//This strange example shows that the`
	`//"while" condition in a "for" does not`
	`//need to involve the loop counter!`
	`int j = 0, x = 0;`
	`for (int i = 0; x < 1500; i++)`
	`{`
	`x = i + j;`
	`j = i * i * i;`
	`}`

N/A	***Structure***
	
	`//"do while" loop with a compound statement`
	`do`
	`{`
	`statement1;`
	`statement2;`
	`.`
	`.`
	`.`
	`} while (relational_expression);`
	Example
	`factorsOfTwo = -1`
	`do`
	`{`
	`factorsOfTwo = factorsOfTwo + 1`
	`num = num / 2`
	`} while (num > 0)`

Understanding the Organization of Java and J++

What exactly is Object-Oriented Programming?

Object-oriented programming (OOP) is a ***highly organized*** method of coding in which all programming tasks are centred about reusable, neatly packaged items called ***objects***. Often, objects are modelled after ***real-world entities***. Examples of such objects include command buttons (called "buttons" in J++), text boxes (called "edit controls" in J++) and option buttons (called "radio buttons" in J++).



However, objects ***do not need*** to be confined to the realm of real-world objects. They can be patterned after just about anything!

What the Heck are Classes?

Whenever I teach students about classes, I can usually sense a heightened feeling of tension in the air. First, the concept of a class in object-oriented programming is already confusing enough. What really seems to bother the students, however, is the number of times per period that the word "classes" is uttered. It reminds them too much of all their schoolwork. Perhaps the creators of OOP should have thought of a different word.

There are many ways of explaining the concept of a class, but I think that the simplest way is to think of ***classes*** as ***blueprints*** or ***templates*** for creating objects. The class itself contains all the code that is required to make an object function in the desired manner. Every time we create a new object of a certain class, a ***copy*** is made of (most of) the code in the class for the new object. In a sense, the class is able to "reproduce" itself whenever an object of its type is created.

A helpful analogy is to think of a ***class*** as a ***cookie cutter***. The cookie cutter (the class) is used to create any number of cookies (the objects). It is also important to remember that classes should are most often used to model ***real-world objects***. For example, the J++ class "`Button`" can be considered a blueprint for making button objects.

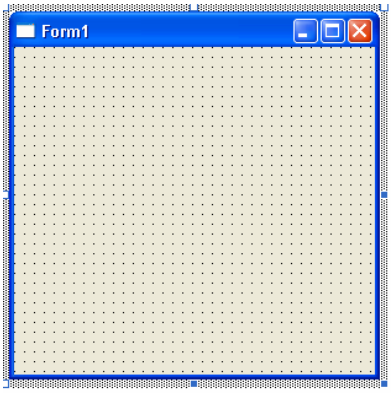
Simple Example of Classes from Visual J++

Visual programming languages are wonderfully convenient because they allow us to create many kinds of objects ***without*** writing any code. One must realize, however, that code is still required to make these objects work! So where does the code come from? The answer is that it is generated automatically by the programming software being used. ***See next page…***

When you begin a new J++ project, a certain amount of code is generated automatically for you. In particular, a portion of the code is displayed with a grey background. It is prefaced with a comment reminding you not to modify the code manually. This code is automatically generated by J++ for all the objects you create visually. At the outset of a new project, there isn't much code.

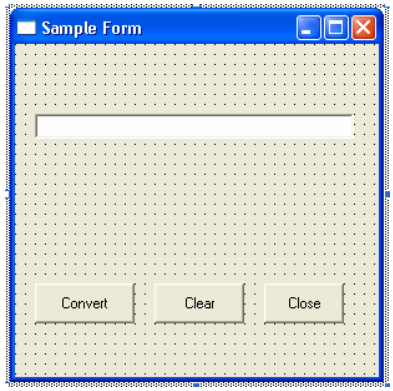
```
/**
* NOTE: The following code is required by the Visual J++
* form designer. It can be modified using the form.
 * editor. Do not modify it using the code editor.
 */
Container components = new Container();

private void initForm()
{
    this.setSize (new Point(300,300));
    this.setText ("Form1");
}
```

The keyword "**this**" is used as a placeholder for the ***name*** of an object. This gives programmers the freedom to choose whatever names they like.



As you add objects to your form, the code becomes more complicated:

```
Container components = new Container();
Button buttonConvert = new Button();
Edit editNumber = new Edit();
Button buttonClose = new Button();
Button buttonClear = new Button();
private void initForm()
{
    this.setSize (new Point(300,300));
    this.setText ("Sample Form");
    this.setAutoScaleBaseSize(new Point(5, 13));
    this.setClientSize(new Point(292, 267));
    buttonConvert.setLocation(new Point(16, 192));
    buttonConvert.setSize(new Point(80, 32));
    buttonConvert.setTabIndex(0);
    buttonConvert.setText("Convert");
    buttonClear.setLocation(new Point(112, 192));
    buttonClear.setSize(new Point(72, 32));
    buttonClear.setTabIndex(1);
    buttonClear.setText("Clear");
    buttonClear.addOnClick(new
                    EventHandler(this.buttonClear_click));
    buttonClose.setLocation(new Point(200, 192));
    buttonClose.setSize(new Point(64, 32));
    buttonClose.setTabIndex(2);
    buttonClose.setText("Close");
    editNumber.setLocation(new Point(16, 56));
    editNumber.setSize(new Point(256, 20));
    editNumber.setTabIndex(3);
    editNumber.setText("");
    this.setNewControls(new Control[] {
                                        editNumber,
                                        buttonClose,
                                        buttonClear,
                                        buttonConvert});
}
```



The "**new**" keyword is used to ***instantiate*** a class. That is, it is used to create an ***instance*** of the class. A more familiar term for ***instance*** is ***object*** of course.

Notice the format of most of these statements. Most of them are of the following form:

objectName.memberName(…)

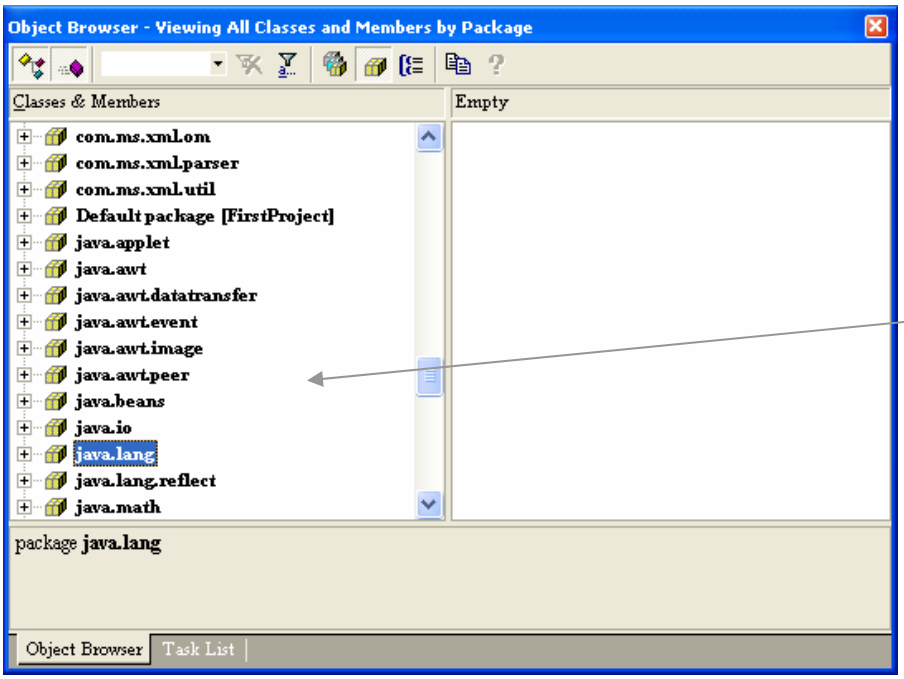
The final line of code (`this.setNewControls(…)`) is used to store the names of all the controls (i.e. objects) on the form in an array. The elements of the array all belong to the class "Control."

Note: By studying Java's class hierarchy, we would notice that the class "Control" only exists in the Microsoft language extensions. It is not part of Sun's Java language specification.

Java is the only Popular Programming Language that is Entirely Object-Oriented

Many people have heard of Java, C++ and Visual Basic. Furthermore, many people also know that these languages are object-oriented. What many people do not realize is that Java is the only language of these three that is ***entirely object-oriented***. C++ has several features inherited from C that are not object-oriented. Nonetheless, C++ is a superbly organized object-oriented language when compared to Visual Basic! VB is a highly disorganized jumble of object-oriented and non-object-oriented features.

The beauty of Java is that it is entirely based on classes. Once you understand classes, you pretty much understand the entire language! The diagrams below show the basic structure of Java.

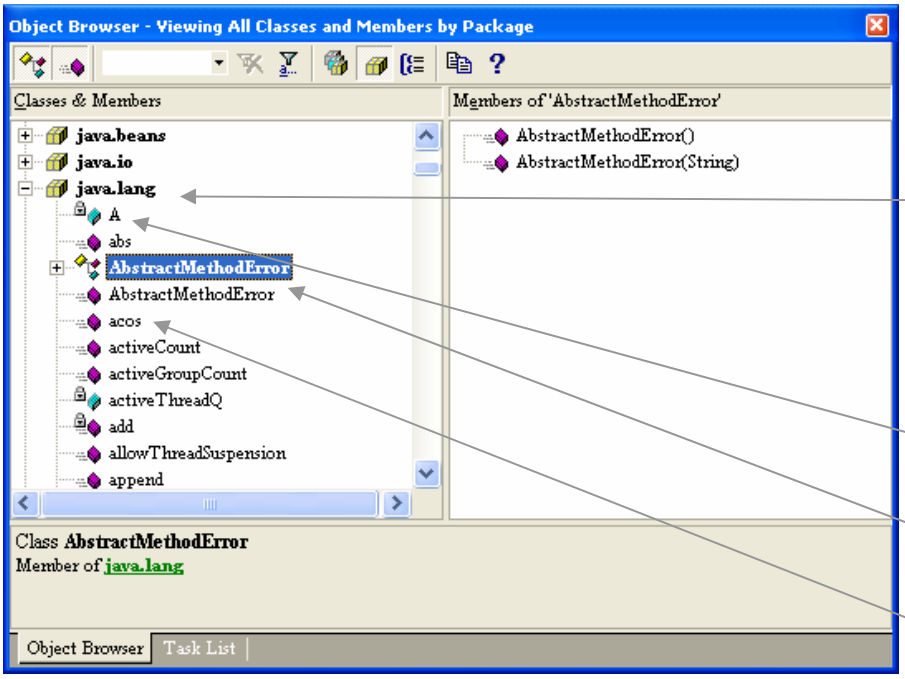


By loading the "Object Browser" in J++, one can view a list of all the available ***packages***.

A ***package*** is a collection of ***classes***, ***interfaces***, ***methods*** (functions) and ***data fields*** (properties).

Notice that most of the package names begin with "java," "sun" or "com." The packages whose names begin with "com" collectively form the ***Microsoft Language Extensions***. They are ***not*** contained in Sun's Java language specification.

We shall not be studying ***interfaces*** in this course. If you are interested in learning about them, consult MSDN and/or any other appropriate sources.



This shows an expanded view of the package "java.lang." You can see that it contains ***data fields*** (the blue-green icons), ***methods*** (the violet icons) and ***classes*** (the multi-coloured icons). If you were to scroll down further in the package, you would also see a few ***interfaces***. You may also notice that certain members have a small padlock icon beside them. The padlock indicates the ***private*** members.

A Data Field

A Class

A Method

Using Strings in Java

Introduction

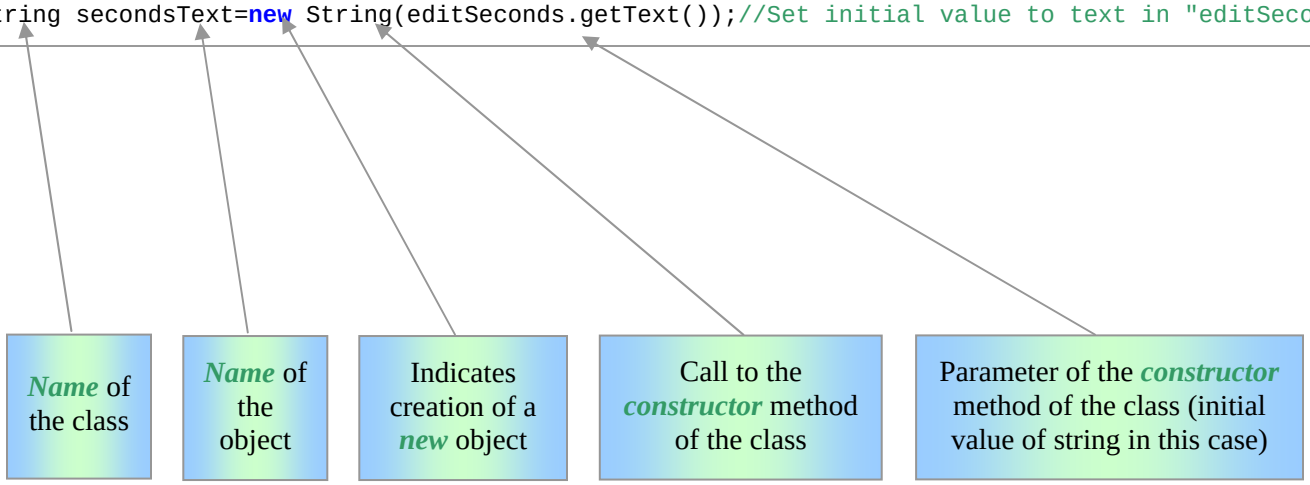
Unlike many other programming languages you may know, Java ***does not*** have a string primitive type. Strings are implemented through ***classes***. There is a ***class*** called "`String`" but there is no ***primitive data type*** called "string."

You probably have noticed that Java has a small number of ***primitive data types***, especially when compared to a language like VB. At first, this seems somewhat annoying, but as our understanding of Java deepens, we begin to appreciate the method behind the madness. You should think of each of these ***primitive data types*** as one of the ***basic elements*** out of which all other data types are constructed. Just as complex molecules are made up of chemical elements and simpler molecules, complex data types like ***strings*** are constructed from primitive data types and simpler classes. In particular, the "`String`" class uses an array of type "**char**" to store strings. (The primitive data type "**char,**" inherited from the C programming language, is used to store single characters. In fact, any variable of type "**char**" stores an integer ranging from 0 to 65535 (0 to $2^{16} - 1$). Although the type "**char**" is intended to store the Unicode code of any Unicode character, it can also be used as if it were intended to store unsigned integers in the range given above.)

Examples

Creating an Object of the "String" Class

```
String surname = new String(); //Create an object of class String and set initial value to ""
String givenName = new String("Joe"); //Create an object of class String and set initial value to "Joe"
String daysText =new String(editDays.getText());//Set initial value to text in "editDays" edit cont.
String hoursText=new String(editHours.getText());//Set initial value to text in "editHours" edit
String minutesText=new String(editMinutes.getText());//Set initial value to text in "editMinutes"
String secondsText=new String(editSeconds.getText());//Set initial value to text in "editSeconds"
```

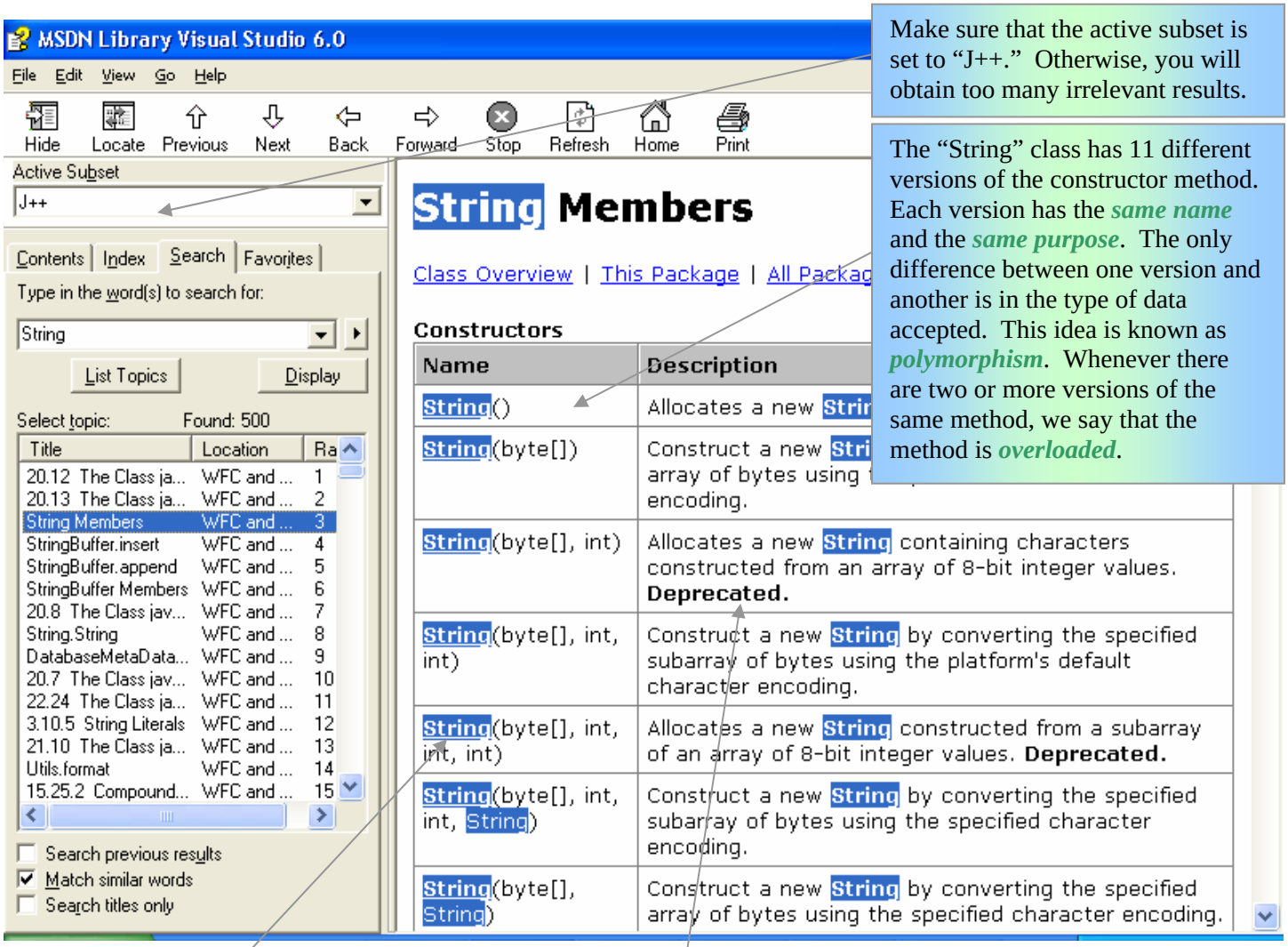


Working with String Objects

Once you have instantiated a string, you can work with it using the "+" string operator and any of the methods in the string class. Since the number of members of each class tends to be large, it is ***not advisable*** to ***memorize*** the names and functions of each member. Instead, you should use the MDSN library to find the information that you need.

Using the MSDN Library to Learn about Class Members

Use the "Search" tab of the MSDN library to search for information on a particular class. For instance, to find information on the "String" class, type "String" in the search field and press "Enter." From the list of topics displayed, double-click on "String Members." You will obtain a list of all the members of the "String" class.



Make sure that the active subset is set to "J++." Otherwise, you will obtain too many irrelevant results.

The "String" class has 11 different versions of the constructor method. Each version has the ***same name*** and the ***same purpose***. The only difference between one version and another is in the type of data accepted. This idea is known as ***polymorphism***. Whenever there are two or more versions of the same method, we say that the method is ***overloaded***.

Click on a link to get more details on how the method is used.

Avoid the use of ***deprecated*** methods. These methods are scheduled to be discontinued in upcoming releases of Java.

Example of a Static (Class) Method

Syntax

public static String valueOf**(** **boolean** *b* **)**

Parameters

b, a **boolean**.

Returns

If the argument is **true**, a string equal to "true" is returned; otherwise, a string equal to "false" is returned.

Description

Returns the string representation of the **boolean** argument.

Example of an Instance (non-static) Method

Syntax

public String trim**( )**

Returns

this string, with white space removed from the beginning and the end.

Description

Removes white space from both ends of **this** string. All characters that have codes less than or equal to '\u0020' (the space character) are considered white space.

What is the Difference between a Static (Class) Method and an Instance Method?

Earlier on, we described the instantiation of a class as a process that involves creating a ***copy of most of the code*** in the class. The ***object*** that results from this process is completely self-contained in that it has a copy of ***most of the code*** that it needs to function in the intended manner. There are some methods, however, that are not copied for every object created, which is why we keep stating "most of the code" instead of "all the code."

Most methods, known as ***instance methods,*** can only exist in the context of an object. A copy of such a method must be made for every ***instance*** (object) of a class because these methods depend directly on the values of the data fields belonging to the object. For example, every edit control has methods called "getText" and "setText." Both of these methods have to be ***instance methods*** because the values they return depend directly on the text stored in the edit control. Thus, each edit control that you create must have its own copy of the "getText" and "setText" methods.

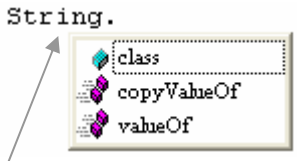
There are some methods, on the other hand, that can exist entirely independently of any objects. Such methods are called ***static*** or ***class*** methods. Class methods are independent of the structure of any particular object and therefore, can be used without instantiation. This means that ***only one copy*** of a static method is needed, a feature that is very helpful in the conservation of memory! For example, every control, including edit controls has a static method called "getMouseButtons." Since this method depends only on which mouse buttons are pressed and does not depend on the state of any particular control, it is best to define it as a **static** method.

Class and Instance Methods of the String Class

Most methods in the String class are ***instance methods***. The only class (static) methods are "valueOf" and "copyValueOf." The table below illustrates the use of both types of methods in the context of the "String" class.

```
int x = 43;
editNumber.setText(String.valueOf(x));
```

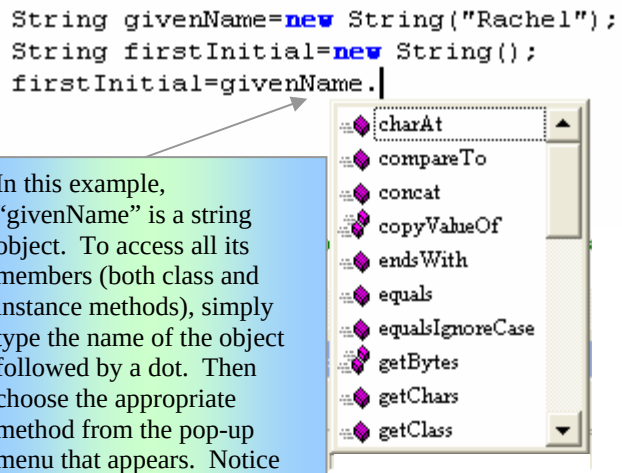
The ***class*** name is used to access ***static (class)*** methods. No object of the given class need exist.



In this example, there is no string object associated with the code. All we want to do is convert an integer value to string form. Simply type the name of the class (String) followed by a dot. The pop-up menu that appears lists the available members. Notice that the list is very short since most method are ***instance*** methods.

```
String givenName=new String("Rachel");
String firstInitial=new String();
firstInitial=givenName.charAt(0);
```

The ***object*** name is used to access ***instance*** methods. The action performed by the method relates directly to the particular object.



In this example, "givenName" is a string object. To access all its members (both class and instance methods), simply type the name of the object followed by a dot. Then choose the appropriate method from the pop-up menu that appears. Notice that the list of available members is very long compared to the class method example at the left.