Language Basics

Variables

- Instance Variables (Non-Static Fields)
- Class Variables (Static Fields)
- Local Variables
- Parameters

Instance Variables (Non-Static Fields)

- Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the static keyword.
- Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words).
- The current Speed of one bicycle is independent from the current Speed of another.

Class Variables (Static Fields)

- A *class variable* is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.
- A field defining the number of gears for a particular kind of bicycle could be marked as static since conceptually the same number of gears will apply to all instances.
- The code static int numGears = 6; would create such a static field.
- Additionally, the keyword final could be added to indicate that the number of gears will never change.

Local Variables

- Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*.
- The syntax for declaring a local variable is similar to declaring a field (for example, int count = 0;).
- There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method.
- As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

Parameters

- Recall that the signature for the main method is public static void main(String[] args). Here, the args variable is the parameter to this method.
- The important thing to remember is that parameters are always classified as "variables" not "fields".
- This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) .

Naming

- Variable names are case-sensitive.
- A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "_".
- The convention, however, is to always begin your variable names with a letter, not "$" or "_". Additionally, the dollar sign character, by convention, is never used at all.
- A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "_", this practice is discouraged.
- White space is not permitted.

Naming

- Subsequent characters may be letters, digits, dollar signs, or underscore characters.
- When choosing a name for your variables, use full words instead of cryptic abbreviations.
- In many cases it will also make your code self-documenting; fields named cadence, speed, and gear, for example, are much more intuitive than abbreviated versions, such as s, c, and g.
- Also keep in mind that the name you choose must not be a keyword or reserved word.

Naming

- If the name you choose consists of only one word, spell that word in all lowercase letters.
- If it consists of more than one word, capitalize the first letter of each subsequent word. The names gearRatio and currentGear are prime examples of this convention.
- If your variable stores a constant value, such as static final int NUM_GEARS = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character.
- By convention, the underscore character is never used elsewhere.

Java Language Keywords

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

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

Primitive Data Types-byte

- The byte data type is an 8-bit signed two's complement integer.
- It has a minimum value of -128 and a maximum value of 127 (inclusive).
- The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters.
- They can also be used in place of int where their limits help to clarify your code.

Primitive Data Types-short

- The short data type is a 16-bit signed two's complement integer.
- It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive).
- As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.

Primitive Data Types-int

- By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of $-2^{31}$and a maximum value of $2^{31}-1$.
- In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$.
- Use the Integer class to use int data type as an unsigned integer.
- Static methods like compareUnsigned,divideUnsigned etc have been added to the Integer class to support the arithmetic operations for unsigned integers.

Primitive Data Types-long

- The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of $2^{63}-1$.
- In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$.
- The unsigned long has a minimum value of 0 and maximum value of $2^{64}-1$. Use this data type when you need a range of values wider than those provided by int.
- The Long class also contains methods like compareUnsigned divideUnsigned etc to support arithmetic operations for unsigned long.

Primitive Data Types-float

- The float data type is a single-precision 32-bit IEEE 754 floating point.
- As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers.
- This data type should never be used for precise values, such as currency.
- For that, you will need to use the java.math.BigDecimal class instead.
- Numbers and Strings covers BigDecimal and other useful classes provided by the Java platform.

Primitive Data Types- double

- The double data type is a double-precision 64-bit IEEE 754 floating point.
- For decimal values, this data type is generally the default choice.
- As mentioned above, this data type should never be used for precise values, such as currency.

Primitive Data Types- boolean

- The boolean data type has only two possible values: true and false.
- Use this data type for simple flags that track true/false conditions.
- This data type represents one bit of information, but its "size" isn't something that's precisely defined.

Primitive Data Types- char

- The char data type is a single 16-bit Unicode character.
- It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

String

- In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the java.lang.String class.
- Enclosing your character string within double quotes will automatically create a new String object; for example, String s = "this is a string";.
- String objects are *immutable*, which means that once created, their values cannot be changed.
- The String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such.

Default Values

- It's not always necessary to assign a value when a field is declared.
- Fields that are declared but not initialized will be set to a reasonable default by the compiler.
- Generally speaking, this default will be zero or null, depending on the data type.
- Relying on such default values, however, is generally considered bad programming style.

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Default Values

- Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable.
- If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it.
- Accessing an uninitialized local variable will result in a compile-time error.

```
public void method() {
    int i;
    int j = i+5 ;
    double d = 3.14;
    Dog dog;
    dog = new Dog(22,7,1964);
}
```

Literals

- A *literal* is the source code representation of a fixed value;
- literals are represented directly in your code without requiring computation.
- As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals

- An integer literal is of type long if it ends with the letter L or l; otherwise it is of type int.
- It is recommended that you use the upper case letter L because the lower case letter l is hard to distinguish from the digit 1.
- Values of the integral types byte, short, int, and long can be created from int literals.
- Values of type long that exceed the range of int can be created from long literals.
- Integer literals can be expressed by these number systems:
  - Decimal: Base 10, whose digits consists of the numbers 0 through 9
  - Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
  - Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

Integer Literals

- For general-purpose programming, the decimal system is likely to be the only number system you'll ever use.
- However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicates binary:

```
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating-Point Literals

- A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.
- The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
 // same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

Character and String Literals

- Literals of types char and String may contain any Unicode (UTF-16) characters.
- If your editor and file system allow it, you can use such characters directly in your code.
- If not, you can use a "Unicode escape" such as '\u0108' (capital C with circumflex), or "S\u00ED Se\u00F1or" (Sí Señor in Spanish).
- Always use 'single quotes' for char literals and "double quotes" for String literals.
- Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char or String literals.

Character and String Literals

- The Java programming language also supports a few special escape sequences for char and String literals:
  - \b (backspace),
  - \t (tab),
  - \n (line feed),
  - \f (form feed),
  - \r (carriage return),
  - \" (double quote),
  - \' (single quote),
  - and \\ (backslash).

Character and String Literals

- There's also a special *null* literal that can be used as a value for any reference type. null may be assigned to any variable, except variables of primitive types.
- There's little you can do with a *null* value beyond testing for its presence.
- Therefore, *null* is often used in programs as a marker to indicate that some object is unavailable.

Character and String Literals

- Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending ".class"; for example, String.class.
- This refers to the object (of type Class) that represents the type itself.

Using Underscore Characters in Numeric Literals

- In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal.
- This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.
- For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Using Underscore Characters in Numeric Literals

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

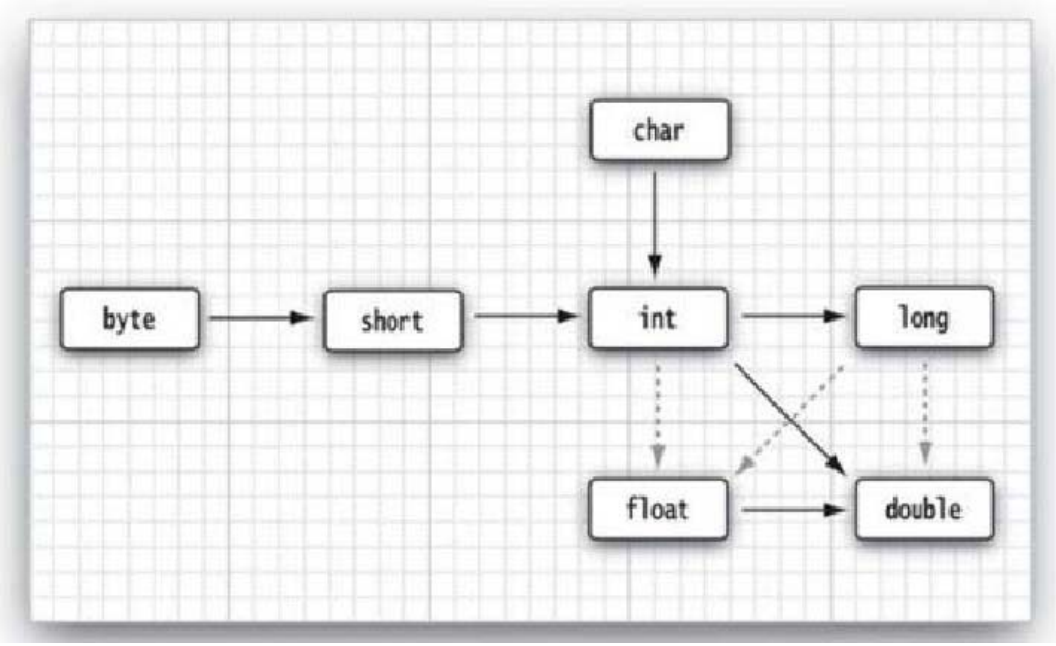
Using Underscore Characters in Numeric Literals

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

Conversions between Numeric Types



Legal conversion

- The six solid arrows denote conversion without information loss.
- The three doted arrows denoted conversions that may lose precision.

```
int n  = 123456789
float f  =  n; //f is 1.23456792E8
```

Conversions between Numeric Types

When two values are combined with a bianry operator(such as n+f where n is an integer and f is a floating-point value), both operands are converted to a common type before the operation is carried out.

1. If either of the operands is of type double, the other one will be converted to a double.
2. Ohterwise, if either of the operands is of type float, the other one will be converted to a float.
3. Ohterwise, if either of the operands is of type long, the other one will be converted to a long.
4. Ohterwise, both operands will be converted to an int.

Casts

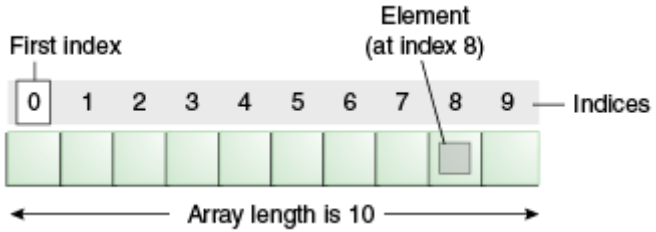
double x = 9.997;

int nx = (int) x;

```
void public method(){
    int i=1,j;
    float f1=0.1;  float f2=123;
    long l1 = 12345678,l2=8888888888;
    double d1 = 2e20,d2=124;
    byte b1 = 1,b2 = 2,b3 = 129;
    j = j+10;
    i = i/10;
    i = i*0.1;
    char c1='a',c2=125;
    byte b = b1-b2;
    char c = c1+c2-1;
    float f3 = f1+f2;
    float f4 = f1+f2*0.1;
    double d = d1*i+j;
    float f = (float)(d1*5+d2);
}
```

Arrays

- An *array* is a container object that holds a fixed number of values of a single type.
- The length of an array is established when the array is created. After creation, its length is fixed.



- Each item in an array is called an *element*, and each element is accessed by its numerical *index*.

Arrays

```
class ArrayDemo {
    public static void main(String[] args) {
        // declares an array of integers
        int[] anArray;

        // allocates memory for 10 integers
        anArray = new int[10];

        // initialize first element
        anArray[0] = 100;
        // initialize second element
        anArray[1] = 200;
        // and so forth
        anArray[2] = 300; anArray[3] = 400;
        anArray[4] = 500; anArray[5] = 600;
        anArray[6] = 700; anArray[7] = 800;
        anArray[8] = 900; anArray[9] = 1000;
        System.out.println("Element at index 0: " + anArray[0]);
        System.out.println("Element at index 1: " + anArray[1]);
        System.out.println("Element at index 2: " + anArray[2]);
        System.out.println("Element at index 3: " + anArray[3]);
        System.out.println("Element at index 4: " + anArray[4]);
        System.out.println("Element at index 5: " + anArray[5]);
        System.out.println("Element at index 6: " + anArray[6]);
        System.out.println("Element at index 7: " + anArray[7]);
        System.out.println("Element at index 8: " + anArray[8]);
        System.out.println("Element at index 9: " + anArray[9]);
    }
}
```

Arrays

The output from this program is:

Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000

Declaring a Variable to Refer to an Array

```
// declares an array of integers
int[] anArray;
```

- Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name.
- An array's type is written as *type*[], where *type* is the data type of the contained elements;
- The brackets are special symbols indicating that this variable holds an array.

Declaring a Variable to Refer to an Array

- The size of the array is not part of its type (which is why the brackets are empty).
- An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the naming section.
- As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.
- You can also place the brackets after the array's name:

```
// this form is discouraged
float anArrayOfFloats[];
```

- However, convention discourages this form; the brackets identify the array type and should appear with the type designation

Creating, Initializing, and Accessing an Array

- One way to create an array is with the *new* operator. The next statement in the ArrayDemo program allocates an array with enough memory for 10 integer elements and assigns the array to the anArray variable.

```
// create an array of integers
anArray = new int[10];
```

- If this statement is missing, then the compiler prints an error like the following, and compilation fails:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

- The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
```

- Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
```

Creating, Initializing, and Accessing an Array

- Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = { 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 };
```

- Here the length of the array is determined by the number of values provided between braces and separated by commas.

Multidimensional Arrays

- In the Java programming language, a multidimensional array is an array whose components are themselves arrays.
- This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {
            {"Mr. ", "Mrs. ", "Ms. "},
            {"Smith", "Jones"}
        };
        // Mr. Smith
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

- The output from this program is:

  Mr. Smith

  Ms. Jones

Arrays

- Finally, you can use the built-in length property to determine the size of any array.
- The following code prints the array's size to standard output:

```
System.out.println(anArray.length);
```

Copying Arrays

- The System class has an arraycopy() method that you can use to efficiently copy data from one array into another:

  public static void arraycopy(Object src, int srcPos,
  Object dest, int destPos, int length)

- Demo

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

- Output

  caffein

Array Manipulations

Some useful operations provided by methods in the java.util.Arrays class, are:

- Searching an array for a specific value to get the index at which it is placed (the binarySearch() method).
- Comparing two arrays to determine if they are equal or not (the equals() method).
- Filling an array to place a specific value at each index (the fill() method).
- Sorting an array into ascending order. This can be done either sequentially, using the sort() method, or concurrently, using the parallelSort() method introduced in Java SE 8.

Summary of Variables

- The Java programming language uses both "fields" and "variables" as part of its terminology.
- Instance variables (non-static fields) are unique to each instance of a class.
- Class variables (static fields) are fields declared with the static modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated.
- Local variables store temporary state inside a method.
- Parameters are variables that provide extra information to a method; both local variables and parameters are always classified as "variables" (not "fields").
- When naming your fields or variables, there are rules and conventions that you should (or must) follow.

Summary of Variables

- The eight primitive data types are: byte, short, int, long, float, double, boolean, and char.
- The java.lang.String class represents character strings.
- The compiler will assign a reasonable default value for fields of the above types; for local variables, a default value is never assigned.
- A literal is the source code representation of a fixed value.
- An array is a container object that holds a fixed number of values of a single type.
- The length of an array is established when the array is created. After creation, its length is fixed.

Operators

Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

Operator Precedence

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

The Simple Assignment Operator

- One of the most common operators that you'll encounter is the simple assignment operator "=".

```
int cadence = 0;
```

- This operator can also be used on objects to assign *object references*.

The Arithmetic Operators

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

ArithmeticDemo

```
class ArithmeticDemo {

    public static void main (String[] args){

        // result is now 3
        int result = 1 + 2;
        System.out.println(result);

        // result is now 2
        result = result - 1;
        System.out.println(result);

        // result is now 4
        result = result * 2;
        System.out.println(result);

        // result is now 2
        result = result / 2;
        System.out.println(result);

        // result is now 10
        result = result + 8;
        // result is now 3
        result = result % 7;
        System.out.println(result);
    }
}
```

You can also combine the arithmetic operators with the simple assignment operator to create *compound assignments*.

For example, x+=1; and x=x+1; both increment the value of x by 1.

The Arithmetic Operators

- The + operator can also be used for concatenating (joining) two strings together, as shown below:

```
class ConcatDemo {
    public static void main(String[] args){
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString+secondString;
        System.out.println(thirdString);
    }
}
```

- By the end of this program, the variable thirdString contains "This is a concatenated string.", which gets printed to standard output.

The Unary Operators

- + Unary plus operator; indicates positive value (numbers are positive without this, however)
- - Unary minus operator; negates an expression
- ++ Increment operator; increments a value by 1
- -- Decrement operator; decrements a value by 1
- ! Logical complement operator; inverts the value of a boolean

The Equality and Relational Operators

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

Comparison Demo

```
class ComparisonDemo {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}
```

The Conditional Operators

- The && and || operators perform *Conditional-AND* and *Conditional-OR* operations on two boolean expressions.
- These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.
  - && Conditional-AND || Conditional-OR

```
public class Test{
   public static void main(String args[]) {
      int i=1,j=2;
      boolean flag1 = (i>3)&&((i+j)>5);
      boolean flag2 = (i<2)||((i+j)<6);
   }
}
```

- Another conditional operator is ?:, which can be thought of as shorthand for an if-then-else statement.
- This operator is also known as the *ternary operator* because it uses three operands.

The Type Comparison Operator instanceof

- The instanceof operator compares an object to a specified type.
- You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.
- When using the instanceof operator, keep in mind that null is not an instance of anything.

Demo

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

        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child));
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }
}

class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}
```

Output:

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

Bitwise and Bit Shift Operators

- ~ Unary bitwise complement
- << Signed left shift
- >> Signed right shift
- >>> Unsigned right shift
- & Bitwise AND
- ^ Bitwise exclusive OR
- | Bitwise inclusive OR

Expressions

- An *expression* is a construct made up of variables, operators, and method invocations, which evaluates to a single value.
- The data type of the value returned by an expression depends on the elements used in the expression.

```
int cadence = 0;
```

- When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first.

```
x + y / 100 // ambiguous
(x + y) / 100 // unambiguous, recommended
x + (y / 100) // unambiguous, recommended
```

Statements

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

Expression Statements

```
// assignment statement
    aValue = 8933.234;
// increment statement
    aValue++;
// method invocation statement
    System.out.println("Hello World!");
// object creation statement
    Bicycle myBike = new Bicycle();
```

Declaration Statements

```
// declaration statement
   double aValue = 8933.234;
```

Control Flow Statements

Control flow statements break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code.

- decision-making statements (if-then, if-then-else, switch),
- looping statements (for, while, do-while)
- branching statements (break, continue, return)

The if-then Statement

- The if-then statement is the most basic of all the control flow statements.
- It tells your program to execute a certain section of code *only if* a particular test evaluates to true.

```
void applyBrakes() {
    // the "if" clause: bicycle must be moving
    if (isMoving){
        // the "then" clause: decrease current speed
        currentSpeed--;
    }
}
```

```
void applyBrakes() {
    // same as above, but without braces
    if (isMoving)
        currentSpeed--;
}
```

The if-then-else Statement

- The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.

```
void applyBrakes() {
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

The if-then-else Statement

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

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

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
     }
}
```

The switch Statement

- Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths.
- A switchworks with the byte, short, char, and int primitive data types.
- It also works with *enumerated types* (discussed in Enum Types), the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer
- Another point of interest is the break statement. Each break statement terminates the enclosing switch statement.

Demo

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

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}
```

```
public class SwitchDemoFallThrough {

    public static void main(String[] args) {
        java.util.ArrayList<String> futureMonths =
            new java.util.ArrayList<String>();

        int month = 8;

        switch (month) {
            case 1:  futureMonths.add("January");
            case 2:  futureMonths.add("February");
            case 3:  futureMonths.add("March");
            case 4:  futureMonths.add("April");
            case 5:  futureMonths.add("May");
            case 6:  futureMonths.add("June");
            case 7:  futureMonths.add("July");
            case 8:  futureMonths.add("August");
            case 9:  futureMonths.add("September");
            case 10: futureMonths.add("October");
            case 11: futureMonths.add("November");
            case 12: futureMonths.add("December");
                     break;
            default: break;
        }

        if (futureMonths.isEmpty()) {
            System.out.println("Invalid month number");
        } else {
            for (String monthName : futureMonths) {
               System.out.println(monthName);
            }
        }
    }
}
```

The switch Statement

- Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing.
- An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Using Strings in switch Statements

- Reading for homework

The while Statement

- The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression)   {
    statement(s)
}
```

- The while statement continues testing the expression and executing its block until the expression evaluates to false.

The do-while **Statement**

```
do {
    statement(s)
} while (expression);
```

- The do-while evaluates its expression at the bottom of the loop instead of the top.
- Therefore, the statements within the do block are always executed at least once,

The for Statement

- The for statement provides a compact way to iterate over a range of values.
- Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied.
- The general form of thefor statement can be expressed as follows:

```
for (initialization; termination; increment) {
  statement(s)
}
```

The for Statement

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The for Statement

```
class ForDemo {
    public static void main(String[] args){
         for(int i=1; i<11; i++){
              System.out.println("Count is: " + i);
         }
    }
}
```

- Notice how the code declares a variable within the initialization expression.
- The scope of this variable extends from its declaration to the end of the block governed by the for statement, so it can be used in the termination and increment expressions as well.
- If the variable that controls a for statement is not needed outside of the loop, it's best to declare the variable in the initialization expression.

The for Statement

The three expressions of the for loop are optional; an infinite loop can be created as follows:

```
// infinite loop
for (  ;  ; ) {
    // your code goes here
}
```

Enhanced for Statement

```
class EnhancedForDemo {
    public static void main(String[] args){
         int[] numbers =
             {1,2,3,4,5,6,7,8,9,10};
         for (int item : numbers) {
             System.out.println("Count is: " + item);
         }
    }
}
```

Branching Statements -break

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

        int[] arrayOfInts =
            { 32, 87, 3, 589,12, 1076, 2000, 8, 622, 127 };
        int searchfor = 12;
        int i;
        boolean foundIt = false;
        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled breakterminates an outer statement.

Branching Statements -break

```
class BreakWithLabelDemo {
    public static void main(String[] args) {
        int[][] arrayOfInts = {
            { 32, 87, 3, 589 }, { 12, 1076, 2000, 8 }, { 622, 127, 77, 955 }
        };
        int searchfor = 12;
        int i;
        int j = 0;
        boolean foundIt = false;
    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length;
                 j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

Branching Statements -continue

- The continue statement skips the current iteration of a for, while , or do-while loop.
- The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop.

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

        String searchMe = "peter piper picked a " + "peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            // interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            // process p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }
}
```

Branching Statements -continue

- A labeled continue statement skips the current iteration of an outer loop marked with the given label.

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

        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;
        int max = searchMe.length() -
                  substring.length();
    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++) != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
                break test;
        }
        System.out.println(foundIt ? "Found it" : "Didn't find it");
    }
}
```

Branching Statements -return

- The return statement exits from the current method, and control flow returns to where the method was invoked.
- The return statement has two forms:
  - one that returns a value,

    return ++count;

    The data type of the returned value must match the type of the method's declared return value.
  - one that doesn't.

    return;

Summary of Control Flow Statements

- The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true.
- The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.
- Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths.
- The while and do-while statements continually execute a block of statements while a particular condition is true.

Summary of Control Flow Statements

- The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top.
- Therefore, the statements within the do block are always executed at least once.
- The for statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.