

## Class Libraries

### Type Conversion and Casting

It is often necessary to convert from one numeric type to another.

If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable.

However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**.

Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.

### Java's Automatic Conversions

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

**The two types are compatible.**

**The destination type is larger than the source type.**

### Conversions between Numeric Types

```
int a= 123456789;
```

```
float f = 1.23456789E8
```

### Casting Incompatible Types

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form: (target-type) value

Example: double x= 2.34d;

```
int a = (int) x; // a= 2;
```

```
double x = 9.997;
```

```
int nx = (int) Math.round(x); //10
```

## Class Libraries

### Automatic Type Promotion

in Expressions

```
byte a = 40;  
byte b = 50;  
byte c = 100;  
int d = a * b / c;
```

The result of the intermediate term  $a * b$  easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression. This means that the sub expression  $a*b$  is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression,  $50 * 40$ , is legal even though a and b are both specified as type byte.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;  
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store  $50 * 2$ , a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

## Class Libraries

```
byte b = 50;  
b = (byte)b * 2;
```

which yields the correct value of 100.

### The Type Promotion Rules:

First, all byte, short, and char values are promoted to int, as just described. Then, if one operand is long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands are double, the result is double.

### Object Wrappers and Autoboxing:

Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the java.lang package, which is imported by default into all Java programs. The wrapper classes in java serves two primary purposes.

To provide a mechanism to 'wrap' primitive values in an object so that primitives can do activities reserved for the objects like being added to ArrayList, HashSet, HashMap etc. collection.

To provide an assortment of utility functions for primitives like converting primitive types to and from string objects, converting to various bases like binary, octal or hexadecimal, or comparing various objects. The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int x = 25;  
Integer y = new Integer(33);
```

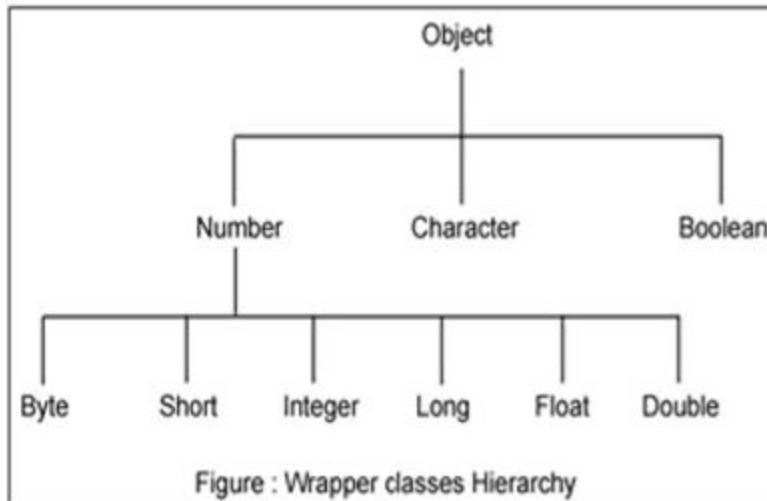
## Class Libraries

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y. Below table lists wrapper classes in Java API with constructor details

| Primitive | Wrapper Class | Constructor Argument    |
|-----------|---------------|-------------------------|
| boolean   | Boolean       | Boolean or String       |
| byte      | Byte          | byte or String          |
| char      | Character     | char                    |
| int       | Integer       | int or String           |
| float     | Float         | float, double or String |
| double    | Double        | double or String        |
| long      | Long          | long or String          |
| short     | Short         | short or String         |

Below is wrapper class hierarchy as per Java API

## Class Libraries



As explained in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get `NumberFormatException` if we try to assign invalid argument in the constructor.

For example to create Integer object we can have following syntax.

```
Integer intObj = new Integer (25);
```

```
Integer intObj2 = new Integer ("25");
```

Here in we can provide any number as string argument but not the words etc. Below statement will throw a runtime exception (`NumberFormatException`)

```
Integer intObj3 = new Integer ("Two");
```

The following discussion focuses on the Integer wrapper class, but applies in a general sense to all eight wrapper classes. The most common methods of the Integer wrapper class are summarized in below table. Similar methods for the other wrapper classes are found in the Java API documentation.

## Class Libraries

| Method        | Purpose   |
|---------------|---|
| parseInt(s)   | returns a signed decimal integer value equivalent to string s |
| toString(i)   | returns a new String object representing the integer i        |
| byteValue()   | returns the value of this Integer as a byte                   |
| doubleValue() | returns the value of this Integer as an double                |

|  |  |
|--|--|
| floatValue()                           | returns the value of this Integer as a float   |
| intValue()                             | returns the value of this Integer as an int  |
| shortValue()                           | returns the value of this Integer as a short   |
| longValue()                            | returns the value of this Integer as a long  |
| int compareTo(int i)                   | Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value. |
| static int compare(int num1, int num2) | Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.  |
| boolean equals(Object intObj)          | Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.  |

## Autoboxing

Beginning with JDK 5, Java added two important features: autoboxing and autounboxing.

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object.

## Class Libraries

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as `intValue()` or `doubleValue()`.

```
Integer iOb = 100; // autobox an int ( new Integer(100) )
int i = iOb; // auto-unbox ( auto call iOb.intValue() )
```

```
// Autobox/unbox a char. Character ch = 'x'; // box a char char ch2 = ch; // unbox a char
```

## Converting Numbers to Strings and String to Numbers

The `Byte`, `Short`, `Integer`, and `Long` classes provide the `parseByte()`, `parseShort()`, `parseInt()`, and `parseLong()` methods, respectively. These methods return the byte, short, int, or long equivalent of the numeric string with which they are called. (Similar methods also exist for the `Float` and `Double` classes.)

```
String str = "5"; int i = Integer.parseInt(str);
```

The `Integer` and `Long` classes also provide the methods `toBinaryString()`, `toHexString()`, and `toOctalString()`, which converts a value into a binary, hexadecimal, or octal string, respectively.

```
int num = 19648;
System.out.println(num + " in binary: " + Integer.toBinaryString(num));
System.out.println(num + " in octal: " + Integer.toOctalString(num));
System.out.println(num + " in hexadecimal: " + Integer.toHexString(num));
```

The output of this program is shown here:

```
19648 in binary: 100110011000000
```

```
19648 in octal: 46300
```

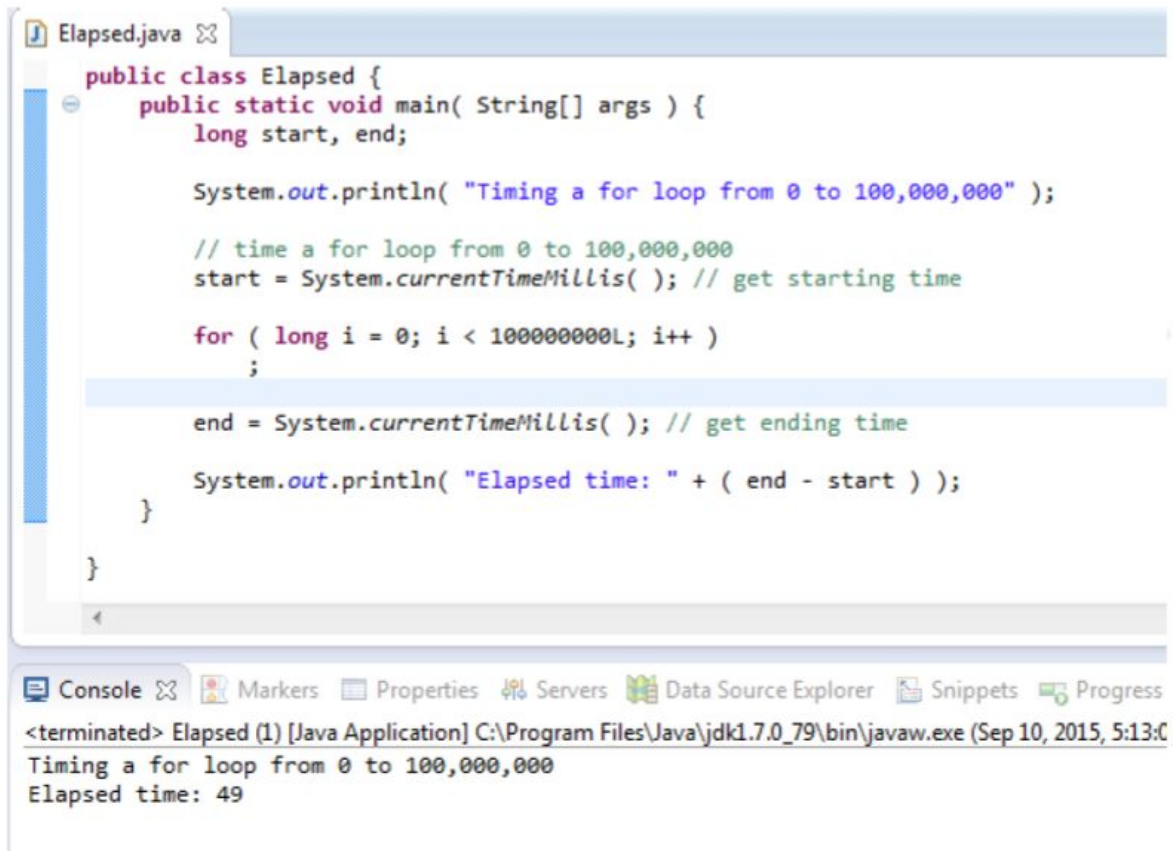
```
19648 in hexadecimal: 4cc0
```

## Class Libraries

### System Class :

The **System** class holds a collection of static methods and variables. The standard input, output, and error output of the Java runtime are stored in the **in**, **out**, and **err** variables.

Using **currentTimeMillis()** to Time Program Execution



```
Elapsed.java
public class Elapsed {
    public static void main( String[] args ) {
        long start, end;

        System.out.println( "Timing a for loop from 0 to 100,000,000" );

        // time a for loop from 0 to 100,000,000
        start = System.currentTimeMillis( ); // get starting time

        for ( long i = 0; i < 100000000L; i++ )
            ;

        end = System.currentTimeMillis( ); // get ending time

        System.out.println( "Elapsed time: " + ( end - start ) );
    }
}
```

Console

<terminated> Elapsed (1) [Java Application] C:\Program Files\Java\jdk1.7.0\_79\bin\javaw.exe (Sep 10, 2015, 5:13:00)  
Timing a for loop from 0 to 100,000,000  
Elapsed time: 49



## Class Libraries

**The three streams** System.in, System.out, and System.err

### System.in

System.in is an InputStream which is typically connected to keyboard input of console programs.

```
public class ScannerEx {  
    public static void main(String[] args) {  
        System.out.print("Write Something in console window: ");  
        Scanner sc = new Scanner(System.in);  
        String inputText = sc.next();  
        System.out.println(inputText);  
    }  
}
```

### System.out

public static final PrintStream out

The "standard" output stream. This stream is already open and ready to accept output data. Typically this stream corresponds to display output or another output destination specified by the host environment or user. For simple stand-alone Java applications, a typical way to write a line of output data is:

```
System.out.println(data)
```

System.out.print is a standard output function used in java. where System specifies the package name, out specifies the class name and print is a function in that class

### System.err

System.err is a PrintStream. System.err works like System.out except it is normally only used to output error texts. Some programs (like Eclipse) will show the output to System.err in red text, to make it more obvious that it is error text.

## Executing Other Programs(java.lang.Runtime. exec)

Several forms of the exec( ) method allows you to name the program you want to run as well as its input parameters. The exec( ) method returns a Process object, which can then be used to control how your Java program interacts with this new running process.

## Class Libraries

The following example uses `exec()` to launch notepad/calculator.

```
public class ExecDemo {  
    public static void main(String[] args) {  
        try {  
            Runtime r = Runtime.getRuntime();  
            r.exec("calc");  
        } catch (Exception e) {  
            System.out.println("Error executing calculator.");  
        }  
    }  
}
```

## Java Math class

Java Math class provides several methods to work on math calculations like `min()`, `max()`, `avg()`, `sin()`, `cos()`, `tan()`, `round()`, `ceil()`, `floor()`, `abs()` etc.

Unlike some of the `StrictMath` class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit the same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is `int` or `long` and the results overflow the range of value, the methods `addExact()`, `subtractExact()`, `multiplyExact()`, and `toIntExact()` throw an `ArithmeticException`.

## Java Math Methods

## Class Libraries

The **java.lang.Math** class contains various methods for performing basic numeric operations such as the logarithm, cube root, and trigonometric functions etc. The various java math methods are as follows:

### Basic Math methods

| Method                         | Description  |
|--------------------------------|--|
| <b>Math.abs()</b>              | It will return the Absolute value of the given value.  |
| <b>Math.max()</b>              | It returns the Largest of two values.  |
| <b>Math.min()</b>              | It is used to return the Smallest of two values.   |
| <b>Math.round()</b>            | It is used to round of the decimal numbers to the nearest value.   |
| <b>Math.sqrt()</b>             | It is used to return the square root of a number.  |
| <b>Math.cbrt()</b>             | It is used to return the cube root of a number.  |
| <b>Math.pow()</b>              | It returns the value of first argument raised to the power to second argument.                                       |
| <b>Math.signum()</b>           | It is used to find the sign of a given value.  |
| <b>Math.ceil()</b>             | It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer. |
| <b>Math.copySign()<br/>( )</b> | It is used to find the Absolute value of first argument along with sign specified in second argument.                |

## Class Libraries

|                                 |  |
|---------------------------------|--|
| <code>Math.nextAfter()</code>   | It is used to return the floating-point number adjacent to the first argument in the direction of the second argument.                               |
| <code>Math.nextUp()</code>      | It returns the floating-point value adjacent to d in the direction of positive infinity.   |
| <code>Math.nextDown()</code>    | It returns the floating-point value adjacent to d in the direction of negative infinity.   |
| <code>Math.floor()</code>       | It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value. |
| <code>Math.floorDiv()</code>    | It is used to find the largest integer value that is less than or equal to the algebraic quotient.   |
| <code>Math.random()</code>      | It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.  |
| <code>Math rint()</code>        | It returns the double value that is closest to the given argument and equal to mathematical integer.   |
| <code>Math.hypot()</code>       | It returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.  |
| <code>Math.ulp()</code>         | It returns the size of an ulp of the argument.   |
| <code>Math.getExponent()</code> | It is used to return the unbiased exponent used in the representation of a value.  |

## Class Libraries

|                                    |  |
|------------------------------------|--|
| <code>Math.IEEEremainder()</code>  | It is used to calculate the remainder operation on two arguments as prescribed by the IEEE 754 standard and returns value. |
| <code>Math.addExact()</code>       | It is used to return the sum of its arguments, throwing an exception if the result overflows an int or long.               |
| <code>Math.subtractExact()</code>  | It returns the difference of the arguments, throwing an exception if the result overflows an int.                          |
| <code>Math.multiplyExact()</code>  | It is used to return the product of the arguments, throwing an exception if the result overflows an int or long.           |
| <code>Math.incrementExact()</code> | It returns the argument incremented by one, throwing an exception if the result overflows an int.                          |
| <code>Math.decrementExact()</code> | It is used to return the argument decremented by one, throwing an exception if the result overflows an int or long.        |
| <code>Math.negateExact()</code>    | It is used to return the negation of the argument, throwing an exception if the result overflows an int or long.           |
| <code>Math.toIntExact()</code>     | It returns the value of the long argument, throwing an exception if the value overflows an int.                            |

## Logarithmic Math Methods

| Method                  | Description   |
|-------------------------|---|
| <code>Math.log()</code> | It returns the natural logarithm of a double value. |

## Class Libraries

|                                |   |
|--------------------------------|---|
| <b>Math.log</b><br><b>10()</b> | It is used to return the base 10 logarithm of a double value.   |
| <b>Math.log</b><br><b>1p()</b> | It returns the natural logarithm of the sum of the argument and 1.  |
| <b>Math.ex</b><br><b>p()</b>   | It returns E raised to the power of a double value, where E is Euler's number and it is approximately equal to 2.71828. |
| <b>Math.ex</b><br><b>pm1()</b> | It is used to calculate the power of E and subtract one from it.  |

## Trigonometric Math Methods

| Method             | Description  |
|--------------------|--|
| <b>Math.sin()</b>  | It is used to return the trigonometric Sine value of a Given double value.       |
| <b>Math.cos()</b>  | It is used to return the trigonometric Cosine value of a Given double value.     |
| <b>Math.tan()</b>  | It is used to return the trigonometric Tangent value of a Given double value.    |
| <b>Math.asin()</b> | It is used to return the trigonometric Arc Sine value of a Given double value    |
| <b>Math.acos()</b> | It is used to return the trigonometric Arc Cosine value of a Given double value. |

## Class Libraries

|                          |   |
|--------------------------|---|
| <code>Math.atan()</code> | It is used to return the trigonometric Arc Tangent value of a Given double value. |
|--------------------------|---|

## Hyperbolic Math Methods

| Method                   | Description  |
|--------------------------|--|
| <code>Math.sinh()</code> | It is used to return the trigonometric Hyperbolic Cosine value of a Given double value.  |
| <code>Math.cosh()</code> | It is used to return the trigonometric Hyperbolic Sine value of a Given double value.    |
| <code>Math.tanh()</code> | It is used to return the trigonometric Hyperbolic Tangent value of a Given double value. |

## Angular Math Methods

| Method                        | Description  |
|-------------------------------|--|
| <code>Math.toDegrees()</code> | It is used to convert the specified Radians angle to equivalent angle measured in Degrees. |
| <code>Math.toRadians()</code> | It is used to convert the specified Degrees angle to equivalent angle measured in Radians. |

## Class Libraries

### NumberFormat class

NumberFormat is an abstract base class for all number formats. This class provides the interface for formatting and parsing numbers. NumberFormat also provides methods for determining which [locales](#) (US, India, Italy, etc) have number formats, and what their names are. NumberFormat helps you to format and parse numbers for any locale.

Example: Suppose we have a double type number. But this double type number is represented in different ways in different Countries. To represent a number according to various countries we have to take the help of NumberFormat class like:

**double d = 123456.789;**

**For India, it is represented like 1,23,456.789**

**For US, it is represented like 123,456.789**

**For ITALY, it is represented like 123.456,789**

#### Some important points about NumberFormat class:

- NumberFormat class is present in java.text package and it is an abstract class.
- NumberFormat class implements Serializable, Cloneable.
- NumberFormat is the direct child class of Format class.
- Number formats are generally not synchronized. It is recommended to create separate format instances for each thread. If multiple threads access a format concurrently, it must be synchronized externally.

#### Methods present in NumberFormat class:

- **public static NumberFormat getInstance();** To get the NumberFormat object for default Locale.
- **public static NumberFormat getCurrencyInstance();** To get the NumberFormat object for default Locale to represent in specific Currency.



## Class Libraries

- **public static NumberFormat getPercentInstance();**
- **public static NumberFormat getInstance(Locale l);** To get the NumberFormat object for the specified Locale object.
- **public static format(long l);**To convert java number to locale object.

```
// Java Program to illustrate NumberFormat class use
class NumberFormatDemo {
    public static void main(String[] args) {
        double d = 123456.789;
        NumberFormat nf = NumberFormat.getInstance(Locale.ITALY);
        System.out.println("ITALY representation of " + d + " : "
                           + nf.format(d));
    }
}
```

### Some Other methods On Number Format

- **getMaximumFractionDigits()** method is a built-in method of the **java.text.NumberFormat** returns the maximum number of digits allowed in the fraction portion of a number of an instance.
- **setMaximumFractionDigits(int n)** method is a built-in method of the **java.text.NumberFormat** which sets the maximum number of digits allowed in the fraction portion of a number.If the new value for maximumFractionDigits is less than the current value of minimumFractionDigits, then minimumFractionDigits will also be set to the new value.
- **getMinimumIntegerDigits()** method is a built-in method of the **java.text.NumberFormat** returns the minimum number of digits allowed in the integer portion of a number of an instance.
- **setMinimumIntegerDigits(int n)** method is a built-in method of the **java.text.NumberFormat** which sets the minimum number of digits allowed in the

## Class Libraries

integer portion of a number. If the new value for `minimumIntegerDigits` is less than the current value of `maximumIntegerDigits`, then `maximumIntegerDigits` will also be set to the new value.

- **`getCurrency()`** method is a built-in method of the **`java.text.NumberFormat`** returns the currency which is used while formatting currency values by this currency. It can be null if there is no valid currency to be determined or if no currency has been set previously.
- **`getAvailableLocales()`** method is a built-in method of the **`java.text.NumberFormat`** returns an array of locales for which localized `NumberFormat` instances are available. All the returned values thus represents the local supported by the JRE
- **`getCurrencyInstance()`** method is a built-in method of the **`java.text.NumberFormat`** returns a currency format for the current default `FORMAT` locale..
- **`getPercentInstance()`** method is a built-in method of the **`java.text.NumberFormat`** returns a percentage format for the current default `FORMAT` locale.
- **`getCurrencyInstance(Locale inLocale)`** method is a built-in method of the **`java.text.NumberFormat`** returns a currency format for any specifies locale.
- **`getIntegerInstance()`** method is a built-in method of the **`java.text.NumberFormat`** returns an integer number format for the current default `FORMAT` locale.
- **`getIntegerIntegerInstance(Locale inLocale)`** method is a built-in method of the **`java.text.NumberFormat`** returns an integer number format for any specified locale.
- **`getNumberInstance()`** method is a built-in method of the **`java.text.NumberFormat`** returns a general purpose number format for the current default `FORMAT` locale.
- **`getNumberInstance(Locale inLocale)`** method is a built-in method of the **`java.text.NumberFormat`** returns a general-purpose number format for any specified locale.

## Class Libraries

### Java DecimalFormat

The `java.text.DecimalFormat` class is used to format numbers using a formatting pattern you specify yourself. This text explains how to use the `DecimalFormat` class to format different types of numbers.

### Creating a DecimalFormat

Creating a `DecimalFormat` instance is done like this:

```
String pattern = "###,###.###";  
DecimalFormat decimalFormat = new DecimalFormat(pattern);
```

The `pattern` parameter passed to the `DecimalFormat` constructor is the number pattern that numbers should be formatted according to.

### `applyPattern()` + `applyLocalizedPattern()`

As you can see, the `DecimalFormat` is created with a formatting pattern. You can change this pattern later using the `applyPattern()` or `applyLocalizedPattern()` method. Here are two examples:

```
decimalFormat.applyPattern("#0.##");
```

```
decimalFormat.applyLocalizedPattern("#0,##");
```

The `applyPattern()` method simply applies a new pattern to the `DecimalFormat` instance as if it were created with that pattern.

The `applyLocalizedPattern()` does the same as `applyPattern()` except it interpretes the characters in the pattern according to the `Locale` the `DecimalFormat` was created with.

## Class Libraries

That means that the characters used to signal where the decimal separator should be etc. are now interpreted according to what characters are used for that Locale, instead of just using the standard marking characters in the pattern.

If you look at the example above, notice that the second pattern uses a comma as decimal (fraction) separator instead of the normal dot. In Danish we use a comma instead of a dot between the integer part and fraction part of numbers. Therefore the localized pattern uses a comma too.

Normally you don't need to use the `applyLocalizedPattern()` method. There are some situations where it makes sense, and that is when you allow the user to type in a formatting pattern. A Danish user would naturally use different symbols in such a pattern than an english user.

Constructing a `DecimalFormat` for a specific Locale is covered later in this text.

## Formatting Numbers

You format a number using the **`format()`** method of the **`DecimalFormat`** instance. Here is an example:

```
String pattern = "###,###.###";
```

```
DecimalFormat decimalFormat = new DecimalFormat(pattern);
```

```
String format = decimalFormat.format(123456789.123);
```

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

The output printed from this code would be:

```
123.456.789,123
```

## Class Libraries

### Creating a DecimalFormat For a Specific Locale

The previous section created a DecimalFormat for the default Locale of the JVM (computer) the code is running on. If you want to create a DecimalFormat instance for a specific Locale, create a NumberFormat and cast it to a DecimalFormat. Here is an example:

```
Locale locale = new Locale("en", "UK");  
String pattern = "###.##";  
  
DecimalFormat decimalFormat = (DecimalFormat)  
    NumberFormat.getNumberInstance(locale);  
decimalFormat.applyPattern(pattern);  
  
String format = decimalFormat.format(123456789.123);  
System.out.println(format);
```

The output printed from this code would be:

**123456789.12**

If you had used a Danish Locale instead, the output would have been:

**123456789,12**

Notice the use of a comma instead of a dot to separate the integer part from the fraction part of the number.

## Class Libraries

### Number Format Pattern Syntax

You can use the following characters in the formatting pattern:

- 0** A digit - always displayed, even if the number has fewer digits (then 0 is displayed)
- #** A digit, leading zeroes are omitted.
- .** Marks decimal separator
- ,** Marks grouping separator (e.g. thousand separator)
- E** Marks separation of mantissa and exponent for exponential formats.
- ;** Separates formats
- Marks the negative number prefix
- %** Multiplies by 100 and shows number as percentage
- ?** Multiplies by 1000 and shows number as per mille
- ¤** Currency sign - replaced by the currency sign for the Locale. Also makes formatting use the monetary decimal separator instead of normal decimal separator. ¤¤ makes formatting use international monetary symbols.
- X** Marks a character to be used in number prefix or suffix
- '** Marks a quote around special characters in prefix or suffix of formatted number.

## Class Libraries

Here are a few examples, formatted using a UK Locale:

| Pattern    | Number     | Formatted String |
|------------|------------|------------------|
| ###.###    | 123.456    | 123.456          |
| ###.#      | 123.456    | 123.5            |
| ###,###.## | 123456.789 | 123,456.79       |
| 000.###    | 9.95       | 009.95           |
| ##0.###    | 0.95       | 0.95             |

Notice that some numbers are rounded, just like with a **NumberFormat** instance.

### DecimalFormatSymbols

You can customize which symbols are used as decimal separator, grouping separator etc. using a DecimalFormatSymbols instance. Here is an example:

```
Locale locale = new Locale("en", "UK");
```

```
DecimalFormatSymbols symbols = new DecimalFormatSymbols(locale);
```

```
symbols.setDecimalSeparator(';');
```

```
symbols.setGroupingSeparator(':');
```

```
String pattern = "#,##0.###";
```

```
DecimalFormat decimalFormat = new DecimalFormat(pattern, symbols);
```

```
String number = decimalFormat.format(123456789.123);
```

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

The output printed from this code would be:

## Class Libraries

**123:456:789;123**

Notice the use of : as a thousand separator, and ; as a decimal separator (fraction separator). These were the symbols set on the DecimalFormatSymbols instance passed to the DecimalFormat constructor.

There are a whole lot more symbols you can set. Here is a list of the methods you can call on a DecimalFormatSymbols instance:

**setDecimalSeparator();**  
**setGroupingSeparator();**  
**setCurrency();**  
**setCurrencySymbol();**  
**setDecimalSeparator();**  
**setDigit();**  
**setExponentSeparator();**  
**setGroupingSeparator();**  
**setInfinity();**  
**setInternationalCurrencySymbol();**  
**setMinusSign();**  
**setMonetaryDecimalSeparator();**  
**setNaN();**  
**setPatternSeparator();**  
**setPercent();**  
**setPerMill();**  
**setZeroDigit();**

For a full explanation of what these methods do, see the JavaDoc for the DecimalFormatSymbols class.



## Class Libraries

### Grouping Digits

The DecimalFormat class has a method called setGroupingSize() which sets how many digits of the integer part to group. Groups are separated by the grouping separator. Here is an example:

```
String pattern = "#,###.###";  
DecimalFormat decimalFormat = new DecimalFormat(pattern);  
decimalFormat.setGroupingSize(4);  
  
String number = decimalFormat.format(123456789.123);  
System.out.println(number);
```

The output printed from this code would be:

**1,2345,6789.123**

Notice how the integer part of the number is now grouped using 4 digits per group instead of the normal 3 digits.

The same effect could have been achieved by just changing the pattern string, like this:

```
String pattern = "####,####.###";  
DecimalFormat decimalFormat = new DecimalFormat(pattern);  
  
String number = decimalFormat.format(123456789.123);  
System.out.println(number);
```

## Class Libraries

# BigInteger Class

BigInteger class is used for mathematical operation which involves very big integer calculations that are outside the limits of all available primitive data types.

For example factorial of 100 contains 158 digits in it so we can't store it in any primitive data type available. We can store as large Integer as we want in it. There is no theoretical limit on the upper bound of the range because memory is allocated dynamically but practically as memory is limited you can store a number which has Integer.MAX\_VALUE number of bits in it which should be sufficient to store mostly all large values.

**Below is an example Java program that uses BigInteger to compute Factorial.**

```
import java.math.BigInteger;
import java.util.Scanner;
public class Example {
    static BigInteger factorial(int N) {
        BigInteger f = new BigInteger("1"); // Or BigInteger.ONE
        for (int i = 2; i <= N; i++)
            f = f.multiply(BigInteger.valueOf(i));
        return f;
    }
    public static void main(String args[]) throws Exception {
        int N = 20;
        System.out.println(factorial(N));
    }
}
```

## Class Libraries

### Methods of BigInteger Class:

1. **BigInteger abs():** This method returns a BigInteger whose value is the absolute value of this BigInteger.
2. **BigInteger add(BigInteger val):** This method returns a BigInteger whose value is (this + val).
3. **BigInteger and(BigInteger val):** This method returns a BigInteger whose value is (this & val).
4. **BigInteger andNot(BigInteger val):** This method returns a BigInteger whose value is (this & ~val).
5. **int bitCount():** This method returns the number of bits in the two's complement representation of this BigInteger that differ from its sign bit.
6. **int bitLength():** This method returns the number of bits in the minimal two's-complement representation of this BigInteger, excluding a sign bit.
7. **byte byteValueExact():** This method converts this BigInteger to a byte, checking for lost information.
8. **BigInteger clearBit(int n):** This method returns a BigInteger whose value is equivalent to this BigInteger with the designated bit cleared.
9. **int compareTo(BigInteger val):** This method compares this BigInteger with the specified BigInteger.
10. **BigInteger divide(BigInteger val):** This method returns a BigInteger whose value is (this / val).
11. **BigInteger[] divideAndRemainder(BigInteger val):** This method returns an array of two BigIntegers containing (this / val) followed by (this % val).
12. **double doubleValue():** This method converts this BigInteger to a double.
13. **boolean equals(Object x):** This method compares this BigInteger with the specified Object for equality.
14. **BigInteger flipBit(int n):** This method returns a BigInteger whose value is equivalent to this BigInteger with the designated bit flipped.

## Class Libraries

15. **float floatValue():** This method converts this BigInteger to a float.
16. **BigInteger gcd(BigInteger val):** This method returns a BigInteger whose value is the greatest common divisor of `abs(this)` and `abs(val)`.
17. **int getLowestSetBit():** This method returns the index of the rightmost (lowest-order) one bit in this BigInteger (the number of zero bits to the right of the rightmost one bit).
18. **int hashCode():** This method returns the hash code for this BigInteger.
19. **int intValue():** This method converts this BigInteger to an int.
20. **int intValueExact():** This method converts this BigInteger to an int, checking for lost information.
21. **boolean isProbablePrime(int certainty):** This method returns true if this BigInteger is probably prime, false if it's definitely composite.
22. **long longValue():** This method converts this BigInteger to a long.
23. **long longValueExact():** This method converts this BigInteger to a long, checking for lost information.
24. **BigInteger max(BigInteger val):** This method returns the maximum of this BigInteger and val.
25. **BigInteger min(BigInteger val):** This method returns the minimum of this BigInteger and val.
26. **BigInteger mod(BigInteger m):** This method returns a BigInteger whose value is  $(this \bmod m)$ .
27. **BigInteger modInverse(BigInteger m):** This method returns a BigInteger whose value is  $(this^{-1} \bmod m)$ .
28. **BigInteger modPow(BigInteger exponent, BigInteger m):** This method returns a BigInteger whose value is  $(this^{\text{exponent}} \bmod m)$ .
29. **BigInteger multiply(BigInteger val):** This method returns a BigInteger whose value is  $(this * val)$ .
30. **BigInteger negate():** This method returns a BigInteger whose value is  $(-this)$ .

## Class Libraries

- 31. **BigInteger nextProbablePrime():** This method returns the first integer greater than this BigInteger that is probably prime.
- 32. **BigInteger not():** This method returns a BigInteger whose value is ( $\sim$ this).
- 33. **BigInteger or(BigInteger val):** This method returns a BigInteger whose value is ( $\text{this} \mid \text{val}$ ).
- 34. **BigInteger pow(int exponent):** This method returns a BigInteger whose value is ( $\text{this}^{\text{exponent}}$ ).
- 35. **static BigInteger probablePrime(int bitLength, Random rnd):** This method returns a positive BigInteger that is probably prime, with the specified bitLength.
- 36. **BigInteger remainder(BigInteger val):** This method returns a BigInteger whose value is ( $\text{this} \% \text{val}$ ).
- 37. **BigInteger setBit(int n):** This method returns a BigInteger whose value is equivalent to this BigInteger with the designated bit set.
- 38. **BigInteger shiftLeft(int n):** This method returns a BigInteger whose value is ( $\text{this} \ll n$ ).
- 39. **BigInteger shiftRight(int n):** This method returns a BigInteger whose value is ( $\text{this} \gg n$ ).
- 40. **short shortValueExact():** This method converts this BigInteger to a short, checking for lost information.
- 41. **int signum():** This method returns the signum function of this BigInteger.
- 42. **BigInteger sqrt():** This method returns the integer square root of this BigInteger.
- 43. **BigInteger[] sqrtAndRemainder():** This method returns an array of two BigIntegers containing the integer square root  $s$  of this and its remainder  $\text{this} - s^2$ , respectively.
- 44. **BigInteger subtract(BigInteger val):** This method returns a BigInteger whose value is ( $\text{this} - \text{val}$ ).

## Class Libraries

- 45. **boolean testBit(int n)**: This method returns true if and only if the designated bit is set.
- 46. **byte[] toByteArray()**: This method returns a byte array containing the two's-complement representation of this BigInteger.
- 47. **String toString()**: This method returns the decimal String representation of this BigInteger.
- 48. **String toString(int radix)**: This method returns the String representation of this BigInteger in the given radix.
- 49. **static BigInteger valueOf(long val)**: This method returns a BigInteger whose value is equal to that of the specified long.
- 50. **BigInteger xor(BigInteger val)**: This method returns a BigInteger whose value is (this ^ val).

## BigDecimal Class

The BigDecimal class provides operations on double numbers for arithmetic, scale handling, rounding, comparison, format conversion and hashing. It can handle very large and very small floating point numbers with great precision but compensating with the time complexity a bit.

A BigDecimal consists of a random precision integer unscaled value and a 32-bit integer scale. If greater than or equal to zero, the scale is the number of digits to the right of the decimal point. If less than zero, the unscaled value of the number is multiplied by  $10^{(-scale)}$ .

**Input : double a=0.03;**

**double b=0.04;**

**double c=b-a;**

**System.out.println(c);**

**Output :0.009999999999999998**

## Class Libraries

```
Input : BigDecimal _a = new BigDecimal("0.03");  
        BigDecimal _b = new BigDecimal("0.04");  
        BigDecimal _c = _b.subtract(_a);  
        System.out.println(_c);
```

Output :0.01

### Need Of BigDecimal

- The two java primitive types(double and float) are floating point numbers, which is stored as a binary representation of a fraction and an exponent.
- Other primitive types(except boolean) are fixed-point numbers. Unlike fixed point numbers, floating point numbers will most of the times return an answer with a small error (around  $10^{-19}$ ) This is the reason why we end up with 0.0099999999999999998 as the result of 0.04-0.03 in the above example.

But BigDecimal provides us with the exact answer.

### Methods of BigDecimal Class:

- **BigDecimal abs():** This method returns a BigDecimal whose value is the absolute value of this BigDecimal, and whose scale is this.scale().
- **BigDecimal abs(MathContext mc):** This method returns a BigDecimal whose value is the absolute value of this BigDecimal, with rounding according to the context settings.
- **BigDecimal add(BigDecimal augend):** This method returns a BigDecimal whose value is (this + augend), and whose scale is max(this.scale(), augend.scale()).
- **byte byteValueExact():** This method converts this BigDecimal to a byte, checking for lost information.
- **int compareTo(BigDecimal val):** This method compares this BigDecimal with the specified BigDecimal.
- **BigDecimal divide(BigDecimal divisor):** This method returns a BigDecimal whose value is (this / divisor), and whose preferred scale is (this.scale() –

## Class Libraries

`divisor.scale()`); if the exact quotient cannot be represented (because it has a non-terminating decimal expansion) an `ArithmeticException` is thrown.

- **`BigDecimal divide(BigDecimal divisor, int scale, RoundingMode roundingMode)`**: This method returns a `BigDecimal` whose value is  $(\text{this} / \text{divisor})$ , and whose scale is as specified.
- **`BigDecimal divide(BigDecimal divisor, MathContext mc)`**: This method returns a `BigDecimal` whose value is  $(\text{this} / \text{divisor})$ , with rounding according to the context settings.
- **`BigDecimal divide(BigDecimal divisor, RoundingMode roundingMode)`**: This method returns a `BigDecimal` whose value is  $(\text{this} / \text{divisor})$ , and whose scale is `this.scale()`.
- **`BigDecimal[] divideAndRemainder(BigDecimal divisor)`**: This method returns a two-element `BigDecimal` array containing the result of `divideToIntegerValue` followed by the result of remainder on the two operands.
- **`BigDecimal[] divideAndRemainder(BigDecimal divisor, MathContext mc)`**: This method returns a two-element `BigDecimal` array containing the result of `divideToIntegerValue` followed by the result of remainder on the two operands calculated with rounding according to the context settings.
- **`BigDecimal divideToIntegerValue(BigDecimal divisor)`**: This method returns a `BigDecimal` whose value is the integer part of the quotient  $(\text{this} / \text{divisor})$  rounded down.
- **`BigDecimal divideToIntegerValue(BigDecimal divisor, MathContext mc)`**: This method returns a `BigDecimal` whose value is the integer part of  $(\text{this} / \text{divisor})$ .
- **`double doubleValue()`**: This method converts this `BigDecimal` to a `double`.
- **`boolean equals(Object x)`**: This method compares this `BigDecimal` with the specified `Object` for equality.
- **`float floatValue()`**: This method converts this `BigDecimal` to a `float`.
- **`int hashCode()`**: This method returns the hash code for this `BigDecimal`.



## Class Libraries

- **int intValue():** This method converts this BigDecimal to an int.
- **int intValueExact():** This method converts this BigDecimal to an int, checking for lost information.
- **long longValue():** This method converts this BigDecimal to a long.
- **long longValueExact():** This method converts this BigDecimal to a long, checking for lost information.
- **BigDecimal max(BigDecimal val):** This method returns the maximum of this BigDecimal and val.
- **BigDecimal min(BigDecimal val):** This method returns the minimum of this BigDecimal and val.
- **BigDecimal movePointLeft(int n):** This method returns a BigDecimal which is equivalent to this one with the decimal point moved n places to the left.
- **BigDecimal movePointRight(int n):** This method returns a BigDecimal which is equivalent to this one with the decimal point moved n places to the right.
- **BigDecimal multiply(BigDecimal multiplicand):** This method returns a BigDecimal whose value is (this × multiplicand), and whose scale is (this.scale() + multiplicand.scale()).
- **BigDecimal multiply(BigDecimal multiplicand, MathContext mc):** This method returns a BigDecimal whose value is (this × multiplicand), with rounding according to the context settings.
- **BigDecimal negate():** This method returns a BigDecimal whose value is (-this), and whose scale is this.scale().
- **BigDecimal negate(MathContext mc):** This method returns a BigDecimal whose value is (-this), with rounding according to the context settings.
- **BigDecimal plus():** This method returns a BigDecimal whose value is (+this), and whose scale is this.scale().
- **BigDecimal plus(MathContext mc):** This method returns a BigDecimal whose value is (+this), with rounding according to the context settings.

## Class Libraries

- **BigDecimal pow(int n):** This method returns a BigDecimal whose value is (this<sup>n</sup>). The power is computed exactly, to unlimited precision.
- **BigDecimal pow(int n, MathContext mc):** This method returns a BigDecimal whose value is (this<sup>n</sup>).
- **int precision():** This method returns the precision of this BigDecimal.
- **BigDecimal remainder(BigDecimal divisor):** This method returns a BigDecimal whose value is (this % divisor).
- **BigDecimal remainder(BigDecimal divisor, MathContext mc):** This method returns a BigDecimal whose value is (this % divisor), with rounding according to the context settings.
- **BigDecimal round(MathContext mc):** This method returns a BigDecimal rounded according to the MathContext settings.
- **int scale():** This method returns the scale of this BigDecimal.
- **BigDecimal scaleByPowerOfTen(int n):** This method returns a BigDecimal whose numerical value is equal to (this \* 10<sup>n</sup>).
- **BigDecimal setScale(int newScale):** This method returns a BigDecimal whose scale is the specified value, and whose value is numerically equal to this BigDecimal.
- **BigDecimal setScale(int newScale, RoundingMode roundingMode):** This method returns a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.
- **short shortValueExact():** This method converts this BigDecimal to a short, checking for lost information.
- **int signum():** This method returns the signum function of this BigDecimal.
- **BigDecimal sqrt(MathContext mc):** This method returns an approximation to the square root of this with rounding according to the context settings..