**JDK 1.4 Features**

**JDK 1.4** is one of the major versions released in February, 2002. The work started on the name of "**Project Merlin**" and includes the developments to support XML processing, **java.nio** package addition, Security restrictions, Logging API, JDBC 3.0 API, Assertions, Chained exceptions, Regular expressions and Drag and Drop. Two of these features, Assertions and Regular expressions are discussed here and the remaining are illustrated in the respective topics.

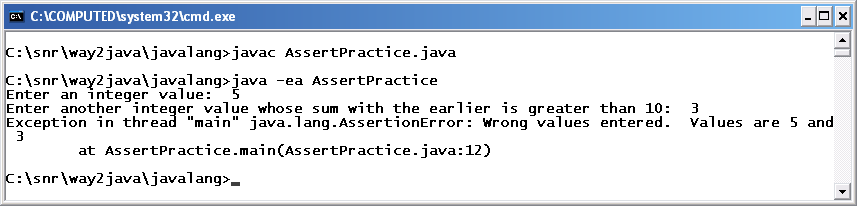
**1. Assertions (of JDK 1.4 Features)**

"**assertion statement**" evaluates to a boolean result where the output is assumed to be true always by the programmer. If the output of evaluation results in false, the assertion statement throws an exception **java.lang.AssertionError** at runtime. The assertion facility is more useful to check the validity of data. This is earlier achieved with if-else statements and exceptions.

"**assert**" is a keyword from JDK 1.4. The **assert** statement is used to know whether the user's input of two numbers evaluates to a sum greater than 10 or not. If the sum is not greater then 10, the assert statement throws exception. Following program illustrates.

Java

|  |
| --- |
| ***import java.util.Scanner;***  ***public class AssertPractice***  ***{***  ***public static void main(String args[])***  ***{***  ***Scanner input = new Scanner(System.in);***  ***System.out.print("Enter an integer value:  ");***  ***int first = input.nextInt();***  ***System.out.print("Enter another integer value whose sum with the earlier is greater than 10:  ");***  ***int second = input.nextInt();***    ***assert(first + second  > 10) : "Wrong values entered.  Values are " + first + " and " + second;***    ***input.close();***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2011/01/ss21.bmp)  
Assertions are the best practices for debugging and knowing logical errors crept unknowingly into the code. By default, the assert action is disabled in the JVM to increase the performance in regular practices. It must be enabled as follows while executing the program. Compilation is as usual.

**java -ea AssertPractice**

**2. Regular Expressions (of JDK 1.4 Features)**

**Regular expressions** is a concept first introduced with **PERL** (Practical Extraction and Report Language) language. For its importance, later many languages like **VBScript** and **Java** from JDK 1.4 supported this concept. Regular expressions are abridged form of code for evaluating and manipulating strings. String validations are easy but coding is very complex, sometimes, to write and understand. It is useful to validate passwords, social security numbers and email addresses etc. entered by the user. Java placed all the supporting classes in **java.util.regex** package.

**Package java.util.regex**

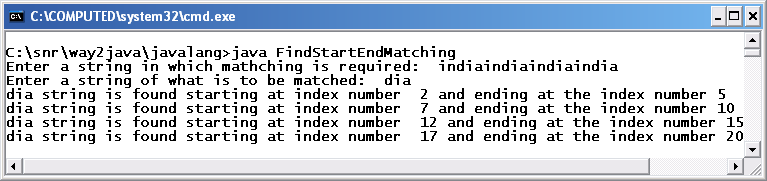
The three classes, available to support, in this package are **Pattern**, **Matcher** and **PatternSyntaxException**.

**Finding the Occurrences of Matching Substrings**

The following program finds the matching of characters in a given string. All matches starting and ending indexes are given.

Java

|  |
| --- |
| ***import java.util.\*;                      // for Scanner***  ***import java.util.regex.\*;           // for Pattern and Matcher***    ***public class FindStartEndMatching***  ***{***  ***public static void main(String args[])***  ***{***  ***Scanner input = new Scanner(System.in);***  ***System.out.print("Enter a string in which mathching is required:  ");***  ***String originalString = input.next();***  ***System.out.print("Enter a string of what is to be matched:  ");***  ***String matchString = input.next();***    ***Pattern pat1 = Pattern.compile(matchString);***  ***Matcher mat1 = pat1.matcher(originalString);***    ***boolean available = true;***  ***while(mat1.find())***  ***{***  ***System.out.println(mat1.group() + " string is found starting at index number  " + mat1.start() + " and ending at the index number " + mat1.end());***  ***available = false;***  ***}***    ***if(available)***  ***{***  ***System.out.println("Matching did not exist.");***  ***}***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2011/01/ss22.bmp)

***Pattern pat1 = Pattern.compile(matchString);  
Matcher mat1 = pat1.matcher(originalString);***

**compile()** is a static method of **Pattern** class that takes a search string as parameter. The search string is converted and returned as a **Pattern** object. **matcher()** is a method of **Pattern** class that takes the original string as parameter in which matching is to be performed. This original string is returned as **Matcher** object.

***mat1.group() + " string is found starting at index number " + mat1.start() + " and ending at the index number " + mat1.end()***

The **group()**, **start()** and **end()** methods are defined in **Matcher** class. The end() method returns the ending character matched plus one. The group() method returns the total matching words (sequences) in the original string.

**3. Splitting a String (of JDK 1.4 Features)**

The JDK 1.4 version adds a new method to String class – **split()**. This method splits the string into independent words like a **StringTokenizer**. Following program uses split() method.

Java

|  |
| --- |
| ***public class SplitIntoTokens***  ***{***  ***public static void main(String args[])***  ***{***  ***String originalString1 ="Determination in work fetches";***  ***String tokens1[] = originalString1.split(" ");***  ***for(int i = 0; i < tokens1.length; i++)***  ***{***  ***System.out.print(tokens1[i] + ", ");***  ***}***    ***System.out.println();***  ***String originalString2 ="Raju/Ramu/Rao";***  ***String tokens2[] = originalString2.split("/");***  ***for(int i = 0; i < tokens2.length; i++)***  ***{***  ***System.out.print(tokens2[i] + " ");***  ***}***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2011/01/ss23.bmp)

***String tokens1[] = originalString1.split(" ");  
String tokens2[] = originalString2.split("/");***

The **split()** method of String class takes a string parameter comprising the delimiter. In the first statement, the delimiter is an empty space and in the second statement it is a forward slash (/). The method returns an array of strings where each element represents a word.

**4. Exception chaining or exception wrapping (of JDK 1.4 Features)**

Each runtime problem is represented by an exception class in Java. One problem may lead to another; that is, one exception may lead to another. This is known as **Exception chaining**. **Exception chaining** is useful to know the original cause of problem and is more useful to programmers for debugging and code maintenance.

***JDK 1.5 Features***

Every version adds new packages and classes. JDK 1.5 version started its work under the code name **Project Tiger** and the version was released on September, 2004. JDK 1.5 version adds the following features to Java language. JDK 1.5 Features are programmatically very important, as often used.

**Following list gives the JDK 1.5 Features**

1. **Autoboxing**
2. **Generics**
3. **Enhanced for loop**
4. **Varargs**
5. **Enums**
6. **Static imports**
7. **C-lang printf()**
8. **StringBuilder**
9. **Metadata**

**Autoboxing – Automatic Conversion**

Upto JDK 1.4, all the data structures of Java stores only objects and when retrieved returns objects. The problem is, even simple data types are to be converted into objects (using wrapper classes) and stored. The retrieved objects are to be converted back to data types to use in arithmetic operations in coding. This is a big nuisance to the programmer. This is overcome in JDK 1.5 with the introduction of **autoboxing** concept. Autoboxing permits to store data types directly in DS and retrieve back data types. Autoboxing is discussed clearly in **data structures** topic.

**Generics – Type-Safe Addition**

A data structure in Java accepts any data type or object as input. If only whole numbers are required to be stored, they must be validated before added to the DS. This requires extra validation code of users input. This extra code is avoided in JDK 1.5 and named as **Generics**. Generics allow adding one type of data only; compiler raises error if other types are added. **Generics** is discussed clearly in DS.

**Enhanced for Loop**

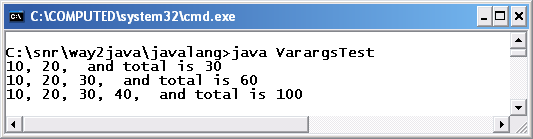
Generally to print the values, we take a **for** loop. The **for** loop includes initialization, test condition and incrementing/decrementing. These are avoided in **enhanced for loop** and this loop works in arrays and DS only. **enhanced for loop** is illustrated in data structures topic.

**Varargs – Variable Number of Arguments**

The **Varargs** concept permits the user to pass any number of arguments to a method depending on the requirement at runtime. The arguments are stored as an array internally. Following program illustrates.

Java

|  |
| --- |
| ***public class VarargsTest***  ***{***  ***public static void add(int... marks)***  ***{***  ***int total = 0;***  ***for(int x : marks)***  ***{***  ***total += x;***  ***System.out.print(x + ", ");***  ***}***  ***System.out.print(" and total is " + total + "\n");***  ***}***  ***public static void main(String args[])***  ***{***  ***add(10, 20);***  ***add(10, 20, 30);***  ***add(10, 20, 30, 40);***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2011/01/ss24.bmp)

***public static void add(int… marks)***

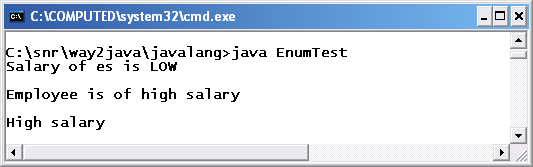
Observe the special syntax of parameter of **add()** method. It takes **three dots**. Internally, the parameters are stored in an **int** array. Enhanced **for** loop is used to print the arguments and their total.

**Enums**

**enum** is a keyword from JDK 1.5. **enum** is a different flavor of a class; enum replaces class prefix. Enums are **type-safe** as they are by default static and final integer values. Generally enum values are written in uppercase, by convention, as they are final.

Java

|  |
| --- |
| ***enum EmployeeSalaries***  ***{***  ***HIGH, MEDIUM, LOW, POOR***  ***}***  ***public class EnumTest***  ***{***  ***public static void main(String args[])***  ***{***  ***EmployeeSalaries es;***  ***es = EmployeeSalaries.LOW;***  ***// knowing enum value***  ***System.out.println("Salary of es is " + es + "\n");***  ***// one more value can be assigned***  ***es = EmployeeSalaries.HIGH;***  ***// comparing two values***  ***if(es == es.HIGH)***  ***{***  ***System.out.println("Employee is of high salary\n");***  ***}***    ***switch(es)***  ***{***  ***case HIGH:***  ***System.out.println("High salary"); break;***  ***case MEDIUM:***  ***System.out.println("Medium salary");  break;***  ***case LOW:***  ***System.out.println("Low salary");  break;***  ***case POOR:***  ***System.out.println("Poor salary");***  ***}***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2011/01/ss25.bmp)

***switch(es)***

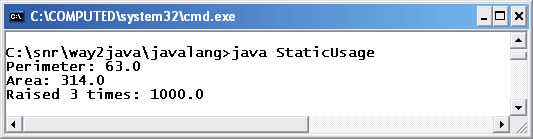
As **enum** represents an integer value always, it can be used with **switch** statement.

**Static Imports**

Many methods of classes like **Math** and **Character** are static. If the variables and methods of these classes are used very often, all are must be prefixed with Math or Character which is tedious. To overcome this, the JDK 1.5 comes with **static imports**. With **static imports**, the **static** keyword need not be used in coding as in the following program.

Java

|  |
| --- |
| ***import static java.lang.Math.\*;***  ***public class StaticUsage***  ***{***  ***public static void main(String args[])***  ***{***  ***int radius = 10;***  ***System.out.println("Perimeter: " + ceil(2 \* PI \* radius));***  ***System.out.println("Area: " + floor(PI \* pow(radius, 2)));***  ***System.out.println("Raised 3 times: " + pow(radius, 3));***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2011/01/ss26.bmp)

***import static java.lang.Math.\*;***

Normal **import** statement imports all classes and interfaces of a package. But **static import** imports only static members of a single class. It avoids usage of the class name multiple times in coding. The above statement avoids **Math** class name to prefix every variable or method used.

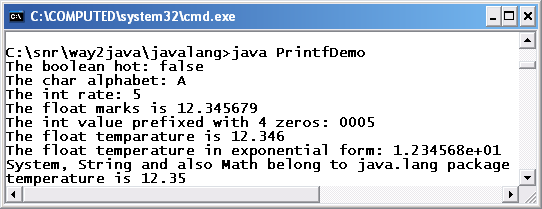
**ceil()**, **floor()**, **pow** are the static methods of Math class and PI is a static variable. All these are used without using prefix Math name.

**Supporting C-lang printf()**

**printf()** is an extra method added to **PrintStream** class from JDK 1.5 version. **printf()** is used to print at command-prompt. The printf() uses **java.util.Formatter** class internally. Following program illustrates a few ways.

Java

|  |
| --- |
| ***public class PrintfDemo***  ***{***  ***public static void main(String args[])***  ***{***  ***boolean hot = false;***  ***char alphabet = 'A';***  ***int rate = 5;***  ***float temperature = 12.3456789f;***    ***System.out.printf("The boolean hot: %b", hot);                         // place holder b for boolean***  ***System.out.printf("\nThe char alphabet: %c", alphabet);                // c for char***  ***System.out.printf("\nThe int rate: %d", rate);                         // d for int***  ***System.out.printf("\nThe float marks is %f", temperature);             // f for float***  ***System.out.printf("\nThe int value prefixed with 4 zeros: %04d", rate);// filling with zeros***  ***System.out.printf("\nThe float temparature is %.3f", temperature);     // precision to three decimal values***  ***System.out.printf("\nThe float temperature in exponential form: %e", temperature);***    ***System.out.printf("\n%s, %s and also %s belong to java.lang package", "System","String","Math");***  ***System.out.printf("\ntemperature is %4.2f", temperature); // width is 4 and precision to 2 decimal points***  ***}***  ***}*** |

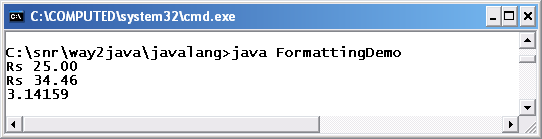
[](http://way2java.com/wp-content/uploads/2011/01/ss27.bmp)  
The terminology used to print the values is quiet familiar to C/C++ programmers.

**More on Number Formatting**

Java’s capability of number formatting is far superior to C/C++. The following program illustrates.

Java

|  |
| --- |
| ***import java.text.\*;                    // for DecimalFormat and NumberFormat classes***  ***public class FormattingDemo***  ***{***  ***public static void main(String args[])***  ***{***  ***DecimalFormat df1 = new DecimalFormat("Rs 0.00");***  ***int rate = 25;***  ***System.out.println(df1.format(rate));          //  Rs 25.00***  ***System.out.println(df1.format(34.45682));                  //  Rs 34.46 (rounded)***    ***NumberFormat nf1 = NumberFormat.getInstance();***  ***nf1.setMinimumFractionDigits(2);***  ***// gives minimum two decimal points; 0 gives no decimal part***  ***nf1.setMaximumFractionDigits(5);***  ***// gives maximum 5 decimal points***  ***System.out.println(nf1.format(Math.PI));                     // prints 3.14159***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2011/01/ss28.bmp)  
The classes **DecimalFormat** and **NumberFormat** from **java.text** package gives more flexibility in number formatting.

***NumberFormat nf1 = NumberFormat.getInstance();***

**NumberFormat** being an abstract class cannot be instantiated directly. The static **getInstance()** method returns an object of NumberFormat.

***nf1.setMinimumFractionDigits(2);  
nf1.setMaximumFractionDigits(5);***

The first statement gives two minimum decimal points and the second statement gives five maximum decimal points. They are given if required only.

**High-performance StringBuilder**

We know String is **immutable** and StringBuffer is **mutable**. When a string is to be manipulated in the program, it is advised to choose StringBuffer for performance reasons. But StringBuffer comes with its own demerits. StringBuffer methods are **synchronized** and thus allowing **thread-safe** operations. To overcome this, designers introduced **StringBuilder** with JDK 1.5 where the methods are not synchronized. Same methods of StringBuffer work with StringBuilder also. For a program and more discussion refer String topic.

**Metadata and Annotations**

**Metadata** replaces the usage of templates. Metadata is declarative programming. **Annotations** are useful for tool developers to generate documentation of a project.

**JDK 1.6 JDK 1.7 Versions**

The work for JDK 1.6 started with the name **Project Mustang** and released on December, 2006. This version is known as **Java SE 6** instead of J2SE. This version includes lot of features in GUI execution speed etc. Following list gives.

**JDK 1.6 Features (of JDK 1.6 JDK 1.7 Versions)**

**GUI Desktop:** The package **java.awt** is reinforced with more classes useful to integrate GUI applications with other GUI technologies.

**More support for Internationalization:** We know that Java is known as **i18n** language as it supports characters of many international languages like Telugu, Tamil, Malayalam, Thai, Latin etc. The **internationalization** supports the user inputs and outputs in his own mother-tongue.

**Improvements in Swing: javax.swing** components are lightweight. Still to integrate the swing components with the underlying OS, the designers introduced more classes. These classes also eliminate gray-area affect and allow table sorting and filtering etc.

**Adding Icons: System tray** capability is introduced to add icons and tooltips etc.

**Compiler:** New type of compilers are introduced. Now a Java program can be compiled programmatically (need not from command-prompt).

**Debugger:** Debugger architecture of Java platform is changed.

**Memory Leaks:** New options are added to JVM to run script when the heap is full. It avoids memory leaks.

**Security support:** The added extra security features allows integration with **GSS/Kerberos** security systems.

**JDBC 4.0:** JDBC is added with more features.

***JDK 1.7 Features***

**JDK 1.7 (Java SE 7 or Java 7)**, code named as **Dolphin** was released on July 28, 2011.

As on today (Sept 1, 2013) the latest update 25 of Java 7 was released on June 18, 2013.

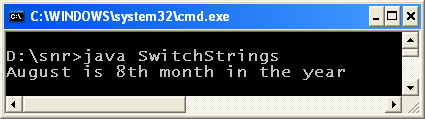
**Following is the list JDK 1.7 Features added, of daily importance.**

**1. String in Switch Expression  
2. Underscores Between Digits in Numeric Literals  
3. Integral Types as Binary Literals  
4. Handling multiple exceptions in a single catch block  
5. Try-with-resources Statement  
6. Automatic Type Inference in Generic object instantiation**

**1. String in Switch Expression**

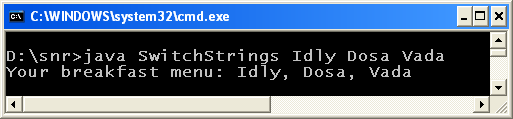
Earlier to **JDK 1.**7, **switch** expression takes **int** values or convertible to **int**. From JDK 1.7, switch accepts string objects also as expression.

|  |
| --- |
| ***public class SwitchStrings{***  ***public static void main(String args[])  {***  ***String str1 = "August";***  ***String str2 = "";***  ***switch(str1)    {***  ***case "January": str2 = "1st";  break;***  ***case "February":   str2 = "2nd";  break;***  ***case "March":   str2 = "3rd";  break;***  ***case "April":   str2 = "4th";  break;***  ***case "May":   str2 = "5th";  break;***  ***case "June":   str2 = "6th";  break;***  ***case "July":   str2 = "7th";  break;***  ***case "August":   str2 = "8th";  break;***  ***case "September":   str2 = "9th";  break;***  ***case "October":   str2 = "10h";  break;***  ***case "November":   str2 = "11th"; break;***  ***case "December":   str2 = "12th"; break;***  ***}***  ***System.out.println(str1 + " is " + str2 + " month in the year");***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/07/image6.png)

**The switch can be used also as follows while accessing** [**command-line arguments**](http://way2java.com/arrays/command-line-arguments/)**.**

|  |
| --- |
| ***public class SwitchStrings{***  ***public static void main(String args[])  {***  ***StringBuffer  sb = new StringBuffer();***  ***for(String str : args)    {***  ***switch(str)      {***  ***case "Idly": sb.append(str +", ");   break;***  ***case "Dosa": sb.append(str +", ");   break;***  ***case "Puri": sb.append(str +", ");   break;***  ***case "Vada": sb.append(str);         break;***  ***default:        sb.append("No breakfast menu");***  ***}***  ***}***  ***System.out.println("Your breakfast menu: " + sb);***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/07/image7.png)Switch does case-sensitive comparison with case statements. Switch gives a more efficient and cleaner code than if-else if-else code.

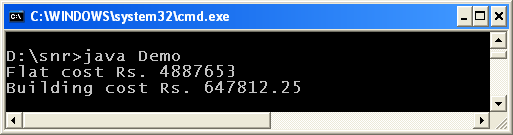
**2. Underscores Between Digits in Numeric Literals**

**Underscores** are permitted in numeric literals. You can place underscores where you feel required to increase readability; like between hundreds and thousands, thousands and lakhs etc.

This is used to group numbers in a bigger literal value (especially of long data type).

**Note:** Do not place underscore at the beginning or ending of the literal value.

|  |
| --- |
| ***public class Demo{***  ***public static void main(String args[])  {***  ***int flatCost = 48\_87\_653;***  ***float buildingCost = 6\_47\_812.25\_67f;***  ***System.out.println("Flat cost Rs. " + flatCost);***  ***System.out.println("Building cost Rs. " + buildingCost);***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/07/image9.png)  
**Note:** But following does not work (cannot parse with underscores).

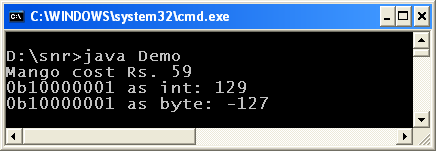
|  |
| --- |
| ***String str ="12\_34";***  ***int x = Integer.parseInt(str);*** |

The above parsing raises [NumberFormatException](http://way2java.com/exceptions/numberformatexception/).

**3. Integral Types as Binary Literals**

With **JDK 1.7**, the integer whole numbers like **byte**, **short**, **int** and **long** can be expressed in binary format also with a prefix of **0b** or **0B**. Earlier, we have **0 prefix for octal** and **0x prefix for hexa** and no prefix for binary. **JDK 1.7, introduced 0b to represent binary literals.**

|  |
| --- |
| ***public class Demo{***  ***public static void main(String args[])  {***  ***int mangoCost = 0b00111011;***  ***System.out.println("Mango cost Rs. " + mangoCost);***  ***int intValue = 0b100\_000\_01;***  ***byte byteValue = (byte) 0b10000001;***  ***System.out.println("0b10000001 as int:   " + intValue);***  ***System.out.println("0b10000001 as byte: " + byteValue);***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/07/image14.png)  
The mango cost statement can be written as follows also with underscores.

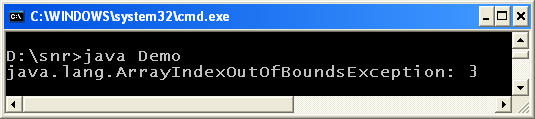
**int mangoCost = 0b001\_110\_11;**

**4. Handling multiple exceptions in a single catch block**

Before JDK 1.7, it is required to write [multiple catches](http://way2java.com/exceptions/rules-of-exceptions-in-multiple-catch-blocks/) to handle different exceptions raised in the code. Now in a single catch statement, multiple exceptions can be included separated by **pipe ( | )** symbol.

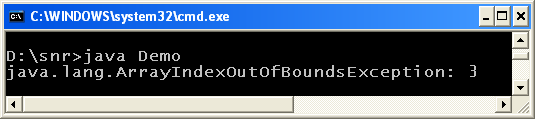
Following is the earlier to **JDK 1.7** code which you are well familiar with.

|  |
| --- |
| ***public class Demo{***  ***public static void main(String args[])  {***  ***int b =0, x[] = { 10, 20, 30 };***  ***try    {***  ***int c = x[3]/b;***  ***}***  ***catch(ArithmeticException e)    {***  ***System.out.println(e);***  ***}***  ***catch(ArrayIndexOutOfBoundsException e)    {***  ***System.out.println(e);***  ***}***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/07/image10.png)

The above multiple catches can be replaced with single catch as follows in JDK 1.7.

|  |
| --- |
| ***public class Demo{***  ***public static void main(String args[])  {***  ***int b =0, x[] = { 10, 20, 30 };***  ***try    {***  ***int c = x[3]/b;***  ***}***  ***catch(ArithmeticException | ArrayIndexOutOfBoundsException e)    {***  ***System.out.println(e);***  ***}***  ***}***  ***}*** |

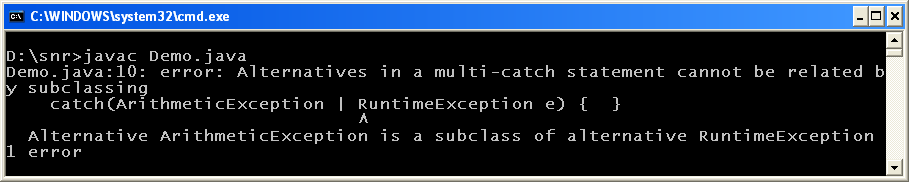
[](http://way2java.com/wp-content/uploads/2013/07/image11.png)

Observe the pipe symbol.

***catch(ArithmeticException | ArrayIndexOutOfBoundsException e)***

**Note:** Following does not work:

|  |
| --- |
| ***try{***  ***int x = 10/0;***  ***}***  ***catch(ArithmeticException | RuntimeException e) {  }*** |

[](http://way2java.com/wp-content/uploads/2013/07/image12.png)

Super class exception must be caught separately (it is a constraint).

**5. Try-with-resources Statement**  
(try statement defining resources)

With **JDK 1.7**, no [finally block](http://way2java.com/exceptions/try-%e2%80%93-catch-%e2%80%93-finally/) is required to close (with close() methods) the resources of files or sockets or JDBC handles (objects) etc. The **resources** (say objects) opened in try block automatically close when the execution control passes out try block (say, at the close brace of try block).

**Your Earlier code:**

|  |
| --- |
| ***import java.io.\*;***  ***public class Demo{***  ***public static void main(String args[])  {***  ***FileReader fr = null;***  ***FileWriter fw = null;***  ***try    {***  ***fr = new FileReader("abc.txt");***  ***fw = new FileWriter("def.txt");***  ***// some file copying code***  ***}***  ***catch(IOException e)    {***  ***e.printStackTrace();***  ***}***  ***finally    {***  ***try      {***  ***if(fr != null) fr.close();***  ***if(fw != null) fw.close();***  ***}***  ***catch(IOException e)      {***  ***e.printStackTrace();***  ***}***  ***}***  ***}***  ***}*** |

**Your Present code with JDK 1.7:**

|  |
| --- |
| ***import java.io.\*;***  ***public class Demo{***  ***public static void main(String args[])  {***  ***try (***  ***FileReader fr = new FileReader("abc.txt");***  ***FileWriter fw = new FileWriter("def.txt");***  ***) {***  ***// some file copying code***  ***}            // at this point fr and fw are closed***  ***catch (IOException e)  {***  ***e.printStackTrace();***  ***}***  ***}***  ***}*** |

A new **interface AutoCloseable** is introduced with JDK 1.7 for this purpose.

|  |
| --- |
| ***public interface java.lang.AutoCloseable***  ***{***  ***public abstract void close() throws java.lang.Exception;***  ***}*** |

Any resource (class) that implements interface "java.lang.AutoCloseable" is eligible as a resource statement to write in try block. From JDK 1.7, many classes implement AutoCloseable interface, like [BufferedReader](http://way2java.com/io/buffered-character-streams-%e2%80%93-bufferedreader-bufferedwriter/), [PrintStream](http://way2java.com/io/printstream/), [Scanner](http://way2java.com/io/keyboard-input-java-util-scanner-%e2%80%93-no-parsing/), [Socket](http://way2java.com/networking/client-to-server-one-way-communication/) etc.

The **close()** method of **AutoCloseable** is called implicitly to close the handles. Observe, the **close()** method throws Exception which you may or may not catch if your code does not demand, or replace with problem specific exception class. In the above code, I used [IOException](http://way2java.com/exceptions/filenotfoundexception-ioexception/). Note that close() method [java.lang.Closeable](http://way2java.com/io/closeable-and-flushable-interfaces/) interface is very different from this.

**JDBC 4.1** can make use of this try-catch-resource management. Following is the snippet of code.

|  |
| --- |
| ***try (Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");***  ***PreparedStatement pst = con.prepareStatement("update Employee set empsal=empsal+500 where empid=?");) {***  ***pst.setInt(1, 100);***  ***int x = pst.executeUpdate();***  ***System.out.println("No. of records inserted: " + x);***  ***}***  ***}***  ***catch (SQLException ex)***  ***{***  ***ex.printStackTrace();***  ***}*** |

**6. Automatic Type Inference in Generic object instantiation**  
(Diamond operator, <> , in collection classes)

In **JDK 1.7**, empty angle brackets (known as **diamond operator**), **<>**, can be used in specifying generic type instead of writing the exact one. But remember, the compiler should be able to judge the type from the generics statement you write.

|  |
| --- |
| ***import java.util.\*;***  ***public class Demo{***  ***public static void main(String args[])  {***  ***// List<String> namesList = new ArrayList<String>();***  ***// Earlier style before JDK 1.7***  ***List<String> namesList = new ArrayList<>();       // JDK 1.7 style***  ***namesList.add("India S N Rao");***  ***namesList.add("Canada Jyostna");***  ***namesList.add("USA Jyothi");***  ***for (String name: namesList)***  ***{***  ***System.out.println(name);***  ***}***  ***}***  ***}*** |

Other improvements are in the fields of java.nio, Phasers and TransferQueue in Concurrency API, Pools and Tasks in Fork Join Framework, java.util.ThreadLocalRandom, ClassLoader Improvements – Deadlock Avoidance, URLClassLoader Improvements, Unicode 6.0 to include thousands of new characters to support Japanese Telcos and Indic scripts, Extensible Currency Codes, NumericShaper for number shaper enhancements, Locale enhancement – Categories, Nimbus Look and Feel for Swing components (better than earlier Metal), Standardize JLayer Component for easy enriching Swing components, Standardize Translucent Windows etc. I have not discussed the above as they are are not for daily usage in coding.

**7. Static blocks**

Earlier to JDK 1.7, to print [static blocks](http://way2java.com/java-introduction/static-blocks-%e2%80%93-static-initialization/) no main() method is required. But from JDK 1.7, if no main() exists, static blocks will not be executed.

***JDK 1.7 Features Java 7***

The latest version of Java, as on today (September 1, 2013) is **Java 7** released on July 28, 2011 by Oracle Corporation.

The latest update to Java 7 is 25 and was released on June 18, 2013.

Always two ideas basically remain in Java designers’ minds while designing the language; while adding new features to every new Java version released.

1. How to make the language simple to use by the developer, by decreasing the coding part?
2. How to increase the performance?

**JDK 1.7 Features**

Following is the list of new features added, of daily importance.

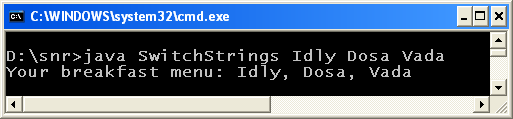
**1. String in Switch Expression  
2. Underscores Between Digits in Numeric Literals  
3. Integral Types as Binary Literals  
4. Handling multiple exceptions in a single catch block  
5. Try-with-resources Statement  
6. Automatic Type Inference in Generic object instantiation**

Only feature (the first one) is shown here as it is very interesting for every one even for C/C++ Programmers.

Generally switch takes a parameter of int data type or convertible to int (remember does not take long data type). But, one of JDK 1.7 Features is switch accepts string as parameter.

**Following is the example on switch with parameter string (of JDK 1.7 Features)**

|  |
| --- |
| ***public class SwitchStrings***  ***{***  ***public static void main(String args[])***  ***{***  ***StringBuffer  sb = new StringBuffer();***    ***for(String str : args)***  ***{***  ***switch(str)***  ***{***  ***case "Idly": sb.append(str +", ");   break;***  ***case "Dosa": sb.append(str +", ");   break;***  ***case "Puri": sb.append(str +", ");   break;***  ***case "Vada": sb.append(str);         break;***  ***default:        sb.append("No breakfast menu");***  ***}***  ***}***  ***System.out.println("Your breakfast menu: " + sb);***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/07/image7.png)Switch does case-sensitive comparison with case statements. Switch gives a more efficient and cleaner code than if-else if-else code.

**JDK 1.8 Features**

**Following are JDK 1.8 Features**

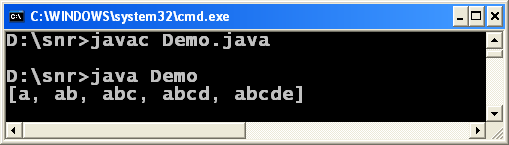
Read one-by-one as they are very innteresting.

**1. Method References**

**a) With static method:**

Java

|  |
| --- |
| ***import java.util.Arrays;***  ***class Test {***  ***public static int matchStringLength(String str1, String str2)  {***  ***return str1.length() - str2.length();***  ***}***  ***}***  ***public class Demo{***  ***public void printByStringLength()   {***  ***String names[] = {"abcde","abc","ab","abcd", "a"};***  ***Arrays.sort(names, Test::matchStringLength);***  ***System.out.println(Arrays.toString(names));***  ***}***  ***public static void main(String args[])***  ***{***  ***new Demo().printByStringLength();***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/08/image11.png)

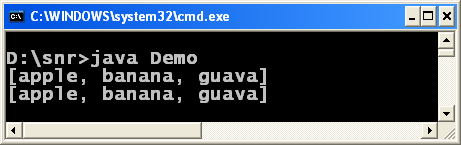
***Arrays.sort(names, Test::matchStringLength);***

You can call the methods with their names (references) with **::** operator.

**2. Parallel Sort**

Java

|  |
| --- |
| ***import java.util.Arrays;***  ***public class Demo {***  ***public static void main(String args[])  {***  ***String fruits[] = {"guava","apple","banana"};***  ***String fruits1[] = {"guava","apple","banana"};***  ***// before JDK 1.8, using Arrays.sort()***  ***Arrays.sort(fruits);***  ***System.out.println(Arrays.toString(fruits));***  ***// with JDK 1.8, using Arrays.parallelSort()***  ***Arrays.parallelSort(fruits1);***  ***System.out.println(Arrays.toString(fruits1));***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/08/image12.png)

Both **sort()** and **parallelSort()** sorts the array. The performance with **parallelSort()** can be seen when the number of arrays to sort are very many. Same sorting can be done also using [List](http://way2java.com/collections/list-fundamentals/) or [Set](http://way2java.com/collections/interface-set-tutorial/) instead of [arrays](http://way2java.com/arrays/arrays-introduction-properties/).

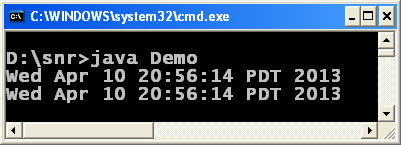
**parallelSort() is overloaded as follows:**

parallelSort(char[] a)  
parallelSort(char[] a, int fromIndex, int toIndex)  
parallelSort(byte[] a)  
parallelSort(byte[] a, int fromIndex, int toIndex)  
parallelSort(short[] a)  
parallelSort(short[] a, int fromIndex, int toIndex)  
parallelSort(int[] a)  
parallelSort(int[] a, int fromIndex, int toIndex)  
parallelSort(long[] a)  
parallelSort(long[] a, int fromIndex, int toIndex)  
parallelSort(float[] a)  
parallelSort(float[] a, int fromIndex, int toIndex)  
parallelSort(double[] a)  
parallelSort(double[] a, int fromIndex, int toIndex)  
parallelSort(T[] a)  
parallelSort(T[] a, Comparator c)  
parallelSort(T[] a, int fromIndex, int toIndex)  
parallelSort(T[] a, int fromIndex, int toIndex, Comparator c)

**3) Addition of Calendar.Builder**

Before **JDK 1.8**, each [date](http://way2java.com/java-util/class-date/) field is set separately with individual methods. Each set method added as a separate statement. See the [**calendar**](http://way2java.com/java-util/class-calendar-timezone-and-locale/) instance in the following code.Java

|  |
| --- |
| ***import java.util.Calendar;                 // for Calendar class***  ***import static java.util.Calendar.\*;        // for static methods of Calendar class***  ***public class Demo {***  ***public static void main(String args[])   {***  ***Calendar calendar = Calendar.getInstance();***  ***// populate calendar with individual set methods one-by-one as a separate statement***  ***calendar.set(YEAR, 2013);***  ***calendar.set(MONTH, APRIL);***  ***calendar.set(DATE, 10);***  ***calendar.set(HOUR, 8);***  ***calendar.set(MINUTE, 56);***  ***calendar.set(SECOND, 14);***  ***calendar.set(AM\_PM, PM);***  ***System.out.println(calendar.getTime());***  ***// let us see the JFK 1.8 style***  ***// all set methods are set as one statement***  ***Calendar calendar1 = new Calendar.Builder()***  ***.set(YEAR, 2013)***  ***.set(MONTH, APRIL)***  ***.set(DATE, 10)***  ***.set(HOUR, 8)***  ***.set(MINUTE, 56)***  ***.set(SECOND, 14)***  ***.set(AM\_PM, PM)***  ***.build();         // one semicolon here***  ***System.out.println(calendar1.getTime());***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/08/image13.png)

In JDK **1.8**, **Calendar.Builder** is used to instantiate **calendar1** instance and all set methods are used as a single statement. **Semicolon** is given only one after **build()** method.

**4) Introduction of Functional Interfaces**

**An interface containing only one abstract method is known as functional interface.** For example, the [java.lang.Runnable](http://way2java.com/multithreading/creating-and-spawning-threads/) interface is a functional interface as it contains only one abstract method **run()**.

A new annotation, **@FunctionalInterface**, is introduced to raise compilation error if an interface marked as **@FunctionalInterface** contains more than one abstract method.

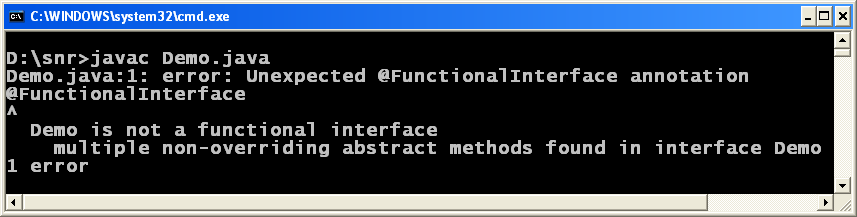
Java

|  |
| --- |
| ***@FunctionalInterface***  ***public interface Demo***  ***{***  ***public abstract void display();***  ***}*** |

The above interface compiles fine. The following does not compile, observe.

Java

|  |
| --- |
| ***@FunctionalInterface***  ***public interface Demo{***  ***public abstract void display();***  ***public abstract void show();***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/08/image14.png)  
The Demo is marked as a functional interface but contains two abstract method. See the above screen for compilation error.

**5) Lambda Introduction**

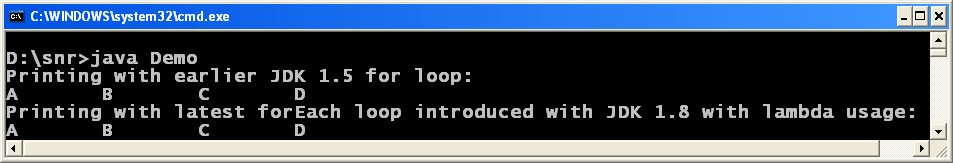
The strength of **JDK 1.8** is introduction of [Lambda function](http://way2java.com/java-versions-2/java-lambda-function/).

**A) Lambdas with forEach loop**

JDK 1.8 introduces a new method **forEach()** to use with lambda expression.

Java

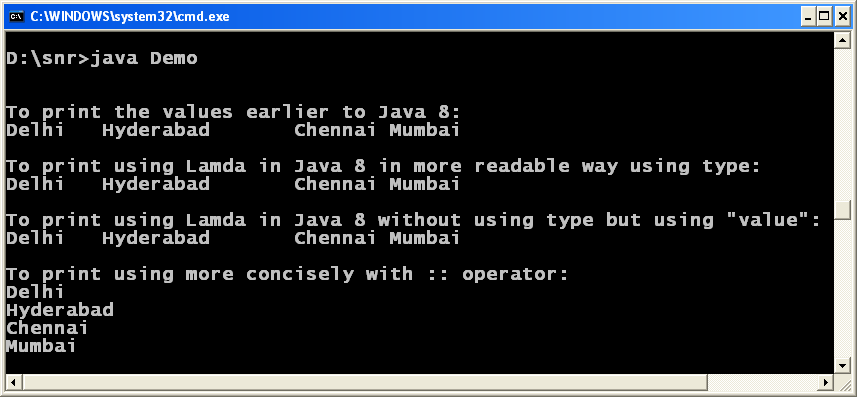
|  |
| --- |
| ***import java.util.\*;***  ***public class Demo{***  ***public static void main(String args[])  {***  ***List<String> alphabets = Arrays.asList("A", "B", "C", "D");***  ***System.out.println("Printing with earlier JDK 1.5 for loop:");***  ***for(String str : alphabets)    {***  ***System.out.print(str + "\t");***  ***}***  ***System.out.println("\nPrinting with latest forEach loop introduced with JDK 1.8 with lambda usage:");***  ***alphabets.forEach(str ->***  ***{***  ***System.out.print(str + "\t");***  ***}***  ***);***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/08/image15.png)

**"-> " is known as lambda expression.**

Java

|  |
| --- |
| ***import java.util.\*;***  ***public class Demo{***  ***public static void main(String args[])  {***  ***List<String> cities = Arrays.asList("Delhi","Hyderabad","Chennai","Mumbai");***  ***System.out.println("\n\nTo print the values earlier to Java 8:");***  ***for(String str : cities)    {***  ***System.out.print(str + "\t");***  ***}***  ***System.out.println("\n\nTo print using Lamda in Java 8 in more readable way using type:");***  ***cities.forEach((String str) -> System.out.print(str + "\t"));***  ***System.out.println("\n\nTo print using Lamda in Java 8 without using type but using \"value\":");***  ***cities.forEach(value -> System.out.print(value + "\t"));***  ***System.out.println("\n\nTo print using more concisely with :: operator:");***  ***cities.forEach(System.out :: println);***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/08/image16.png)

**Java 8** **lambda expressions** helps to write less code in a more readable way.

*System.out.println(“\n\nTo print using Lamda in Java 8 in more readable way using type:”);  
cities.forEach((String str) -> System.out.print(str + “\t”));*

The [**lambda expression**](http://way2java.com/java-versions-2/java-lambda-function/) consists of two parts, left of lambda arrow and right of lambda arrow. Left gives the list of parameters and the right gives the body part. Here, the parameter is **str** and the body contains a single [print()](http://way2java.com/io/printstream/) statement.

***System.out.println(“\n\nTo print using Lamda in Java 8 without using type but using \”value\”:”);  
cities.forEach(value -> System.out.print(value + “\t”));***

The above **forEach()** can be written more concisely with **method reference (using :: operator)**.

***System.out.println(“\n\nTo print using more concisely with :: operator:”);  
cities.forEach(System.out :: println);***

**6. Replacement of PremGen with Metaspace**

JDK 1.8 officially announced the removal of **Permanent Generation (PremGen)** space and in its place introduced **Metaspace**. In HotSpot VM, the **PremGen** used to give **OutOfMemoryError** due to depletion of space, may sometimes be caused by memory leaks while loading and unloading a J2EE applications.

In practice, the **IBM JRE** and **Oracle JRockit** are not using **PremGen** space and are using the **native memory (C-Heap)** to store a class’s metadata. This concept is taken into JDK 1.8 for Java VM.

From JDK 1.8, most of the memory allocation for storing metada is done through native memory. The existing classes used to retrieve the metadata of a class no more works with metaspace. By default, the metaspace allocation memory depends on the native memory availability, OS virtual memory availability and on JVM of 32-bit or 64-bit. A flag introduced to limit the maximum memory allocation for metaspace – **MaxMetaspaceSize**. If this flag is not set, the metaspace will dynamically updated, at intervals, as per the requirement of the application running.

The garbage collector is triggered to go for garbage collection when the metadata usage is more than the size of MaxMetaspaceSize.

Due to removal of **PremGen** space, we cannot configure the space through XX:PermSize & -XX:MaxPermSize

**String Switch support**

Every version of Java brings some new features to make Java coding simple to practice. **JDK 1.7** brought many features and one among is **switch** statement. Switch is a control structure existing in all programming languages where a switch parameter should be an **integer value or convertible to integer**. If you pass a [char](http://way2java.com/java-introduction/primitive-data-types/), it is converted into integer implicitly. If you pass a long data type, it raises compilation error; of course in particular to Java.

In **JDK 1.7** (known as **Java 7**), the **switch statement takes a string as parameter**. Surprising, it is true.

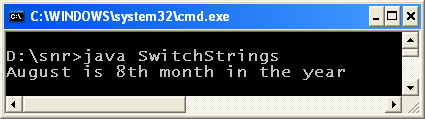
**1. String in Switch Expression**

Earlier to **JDK 1.**7, **switch** expression takes **int** values or convertible to **int**. From JDK 1.7, switch accepts string objects also as expression.

**Example on String Switch support**

Java

|  |
| --- |
| ***public class SwitchStrings{***  ***public static void main(String args[])  {***  ***String str1 = "August";***  ***String str2 = "";***  ***switch(str1)                 // observe, string passed as parameter (String Switch support)***  ***{***  ***case "January": str2 = "1st";  break;***  ***case "February":   str2 = "2nd";  break;***  ***case "March":   str2 = "3rd";  break;***  ***case "April":   str2 = "4th";  break;***  ***case "May":   str2 = "5th";  break;***  ***case "June":   str2 = "6th";  break;***  ***case "July":   str2 = "7th";  break;***  ***case "August":   str2 = "8th";  break;***  ***case "September":   str2 = "9th";  break;***  ***case "October":   str2 = "10h";  break;***  ***case "November":   str2 = "11th"; break;***  ***case "December":   str2 = "12th"; break;***  ***}***  ***System.out.println(str1 + " is " + str2 + " month in the year");***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/07/image6.png)

Try with Resources Example

Every version of Java adds some new features and the latest version JDK 1.7 (as on August, 2013) comes with new additions.

Following are the new additions.

**1. String in Switch Expression  
2. Underscores Between Digits in Numeric Literals  
3. Integral Types as Binary Literals  
4. Handling multiple exceptions in a single catch block  
5. Try-with-resources Statement  
6. Automatic Type Inference in Generic object instantiation**

**Now let us discuss Try with Resources Example Statement**

With **JDK 1.7**, no [finally block](http://way2java.com/exceptions/try-%e2%80%93-catch-%e2%80%93-finally/) is required to close (with close() methods) the resources of files or sockets or JDBC handles (objects) etc. The **resources** (say objects) opened in try block automatically close when the execution control passes out try block (say, at the close brace of try block).

**Your Earlier code before Java 7:**

Java

|  |
| --- |
| ***import java.io.\*;***  ***public class Demo{***  ***public static void main(String args[])  {***  ***FileReader fr = null;***  ***FileWriter fw = null;***  ***try    {***  ***fr = new FileReader("abc.txt");***  ***fw = new FileWriter("def.txt");***  ***// some file copying code***  ***}***  ***catch(IOException e)    {***  ***e.printStackTrace();***  ***}***  ***finally    {***  ***try***  ***{***  ***if(fr != null) fr.close();      //closing code here.  This is not necessary from JDK 1.7 with Try with Resources Example***  ***if(fw != null) fw.close();***  ***}***  ***catch(IOException e)      {***  ***e.printStackTrace();***  ***}***  ***}***  ***}***  ***}*** |

**Your Present code with JDK 1.7:**

Java

|  |
| --- |
| ***import java.io.\*;***  ***public class Demo{***  ***public static void main(String args[])  {***  ***try (***  ***FileReader fr = new FileReader("abc.txt");   // Try with Resources Example, fr closes implicitly***  ***FileWriter fw = new FileWriter("def.txt");   // Try with Resources Example, fw closes implicitly***  ***)        {***  ***// some file copying code***  ***}            // at this point fr and fw are closed***  ***catch (IOException e)         {***  ***e.printStackTrace();***  ***}***  ***}***  ***}*** |

A new **interface AutoCloseable** is introduced with JDK 1.7 for this purpose.

Java

|  |
| --- |
| ***public interface java.lang.AutoCloseable***  ***{***  ***public abstract void close() throws java.lang.Exception;***  ***}*** |

Any resource (class) that implements interface "java.lang.AutoCloseable" is eligible as a resource statement to write in try block.

The **close()** method of **AutoCloseable** is called implicitly to close the handles. Observe, the **close()** method throws Exception which you may or may not catch if your code does not demand, or replace with problem specific exception class. In the above code, I used [IOException](http://way2java.com/exceptions/filenotfoundexception-ioexception/). Note that close() method [java.lang.Closeable](http://way2java.com/io/closeable-and-flushable-interfaces/) interface is very different from this.

J**DBC 4.1** can make use of this try-catch-resource management.

**Autoboxing Java – Automatic Conversion**

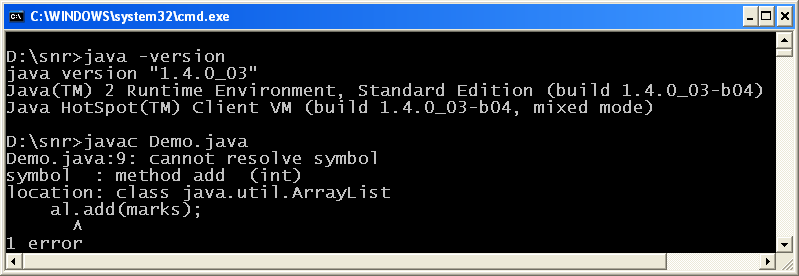
Java **data structures** always (in all the JDK versions), stores only objects and when retrieved return only objects. The problem is, even the simple [primitive data types](http://way2java.com/java-introduction/primitive-data-types/) (used very often) are to be converted into objects (using [wrapper classes](http://way2java.com/java-lang/wrapper-classes/)) and then stored. The retrieved objects from the DS are to be converted back to original data types to be used in arithmetic operations later. This is a big headache to the programmer. This is overcome in [JDK 1.5](http://way2java.com/collections/data-structures-jdk-1-5-features/) with the introduction of **autoboxing** concept. Autoboxing Java permits to store primitive data types directly in DS and retrieve back in data types form. Let us discuss more elaborately.

See the following code. ArrayList was introduced in Java 2 (JDK 1.2). An **ArrayList** object **al** is created and added with an element of **marks**. Nothing more is done.

**Example on Autoboxing Java**

Java

|  |
| --- |
| ***import java.util.ArrayList;***  ***public class Demo{***  ***public static void main( String args[])  {***  ***ArrayList al = new ArrayList();***  ***int marks = 50;***  ***al.add(marks);***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/11/image.png)

Observe the error message raised by the compiler of JDK 1.4. Upto JDK 1.4, primitive data types cannot be added to a DS of Java. They must be explicitly converted into objects using [wrapper classes](http://way2java.com/java-lang/wrapper-classes/) and then added. To work with the above code successfully, following modification should be done.

***ArrayList al = new ArrayList();  
int marks = 50;  
Integer i1 = new Integer(marks);  
al.add(i1);***

Let us see when retrieved what to be done.

Java

|  |
| --- |
| ***import java.util.ArrayList;***  ***public class Demo{***  ***public static void main( String args[])  {                           // addition of an element***  ***ArrayList al = new ArrayList();***  ***int marks = 50;***  ***Integer i1 = new Integer(marks);***  ***al.add(i1);***  ***// retrieval of an element***  ***Object obj = al.get(0);***  ***Integer i2 = (Integer) obj;***  ***int marks1 = i2.intValue();***  ***System.out.println(marks1\*marks1);           // prints 2500***  ***}***  ***}*** |

***Object obj = al.get(0);  
Integer i2 = (Integer) obj;  
int marks1 = i2.intValue();  
System.out.println(marks1\*marks1); // prints 2500***

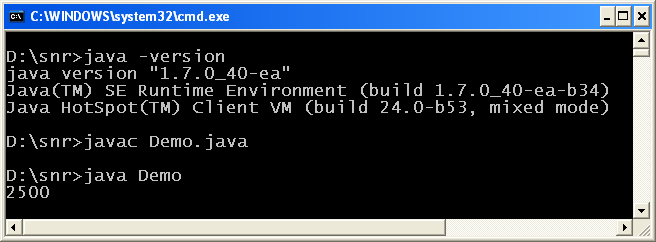
The **get()** method returns an object of [Object](http://way2java.com/java-general/java-object-class/) class(here, it is obj). Explicit cast to Integer, **i2**. Get back primitive data type **marks1** using **intValue()** method. Don’t you feel it is all round about process?

Forget all this laborious process from [JDK 1.5](http://way2java.com/collections/data-structures-jdk-1-5-features/). From JDK 1.5, the compiler itself converts the primitive data type into object and then stores. When retrieved, it returns primitive data type. Converting primitive data type to object implicitly is known as **autoboxing** and again converting object to data type is known as **unboxing**. Autoboxing and unboxing were introduced with JDK 1.5 as a new feature and made programmer life easier.

Let us rewrite the above code and compile with [JDK 1.7](http://way2java.com/java-versions-2/jdk-1-7-features/) version.

Java

|  |
| --- |
| ***import java.util.ArrayList;***  ***public class Demo{***  ***public static void main( String args[])***  ***{                            // addition of an element***  ***ArrayList<Integer> al = new ArrayList<Integer>();***  ***int marks = 50;***  ***al.add(marks);                                 // int marks implicitly converted to object and then stores***  ***// known as Autoboxing Java***  ***int marks1 = al.get(0);                        // retrieval of an element***  ***System.out.println(marks1\*marks1);             // prints 2500***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/11/image1.png)

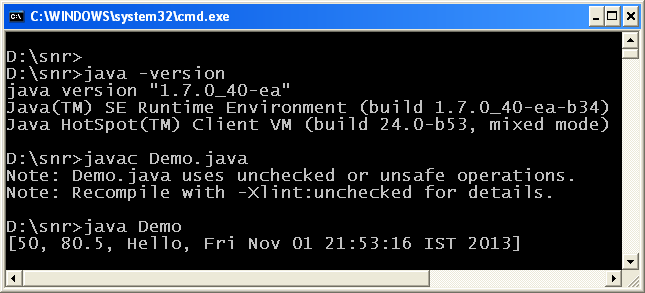
Here, another feature of JDK 1.5 generics is used. Now observe, primitive data type **marks** is added and retrieved as primitive data type **marks1**. **This automatic conversion to object is known as autoboxing and getting back to data type is known as unboxing.**

**Generics Java**

The main difference between arrays and data structures is array accepts only one type of data and data structure accepts elements of dissimilar data types. See the following code of [ArrayList](http://way2java.com/collections/arraylist-introduction/) accepting dissimilar [data type](http://way2java.com/java-introduction/primitive-data-types/) values.

Java

|  |
| --- |
| ***import java.util.ArrayList;***  ***import java.util.Date;***  ***public class Demo{***  ***public static void main( String args[])  {***  ***ArrayList al = new ArrayList();                      // non Generics Java***  ***// ArrayList can store any type of data***  ***al.add(50);                                           // int value***  ***al.add(80.5);                                         // double value***  ***al.add("Hello");                                      // String object***  ***al.add(new Date());                                   // Date object***  ***System.out.println(al);***  ***}***  ***}*** |

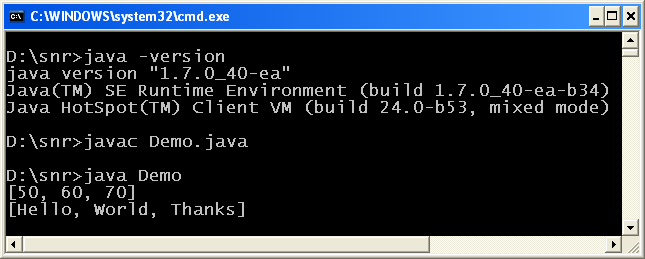
[](http://way2java.com/wp-content/uploads/2013/11/image2.png)

The above compiles and runs successfully and can be seen in the above screen output. But the compiler raises a warning of unchecked and unsafe operations. One more observation is the code is compiled with [JDK 1.7](http://way2java.com/java-versions-2/jdk-1-7-features/).

From [JDK 1.5](http://way2java.com/collections/data-structures-jdk-1-5-features/) onwards, **Java allows a DS to store only one type of data**, say, integers, doubles, String objects, Date objects or Employee objects. This feature added in JDK 1.5 is known as **Generics**. Now observe the following ArrayList objects that allow to add only ints or Strings.

Java

|  |
| --- |
| ***import java.util.ArrayList;***  ***public class Demo{***  ***public static void main( String args[])  {***  ***ArrayList<Integer> al1 = new ArrayList<Integer>();  // stores only int values***  ***al1.add(50);***  ***al1.add(60);***  ***al1.add(70);***  ***System.out.println(al1);***  ***ArrayList<String> al2 = new ArrayList<String>(); // stores only String values***  ***al2.add("Hello");***  ***al2.add("World");***  ***al2.add("Thanks");***  ***System.out.println(al2);***  ***}***  ***}*** |

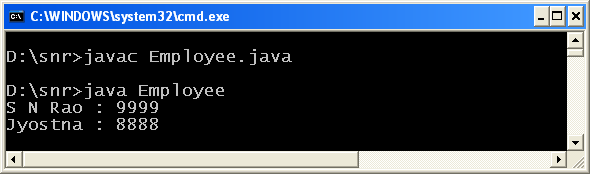
[](http://way2java.com/wp-content/uploads/2013/11/image3.png)

Two **generic objects** of ArrayList **al1** and **al2** are created storing only [int values](http://way2java.com/java-introduction/primitive-data-types/) and [String](http://way2java.com/string-and-stringbuffer/one-stop-destination-for-all-string-and-stringbuffer/) values. Now observe, the compiler does not raise a warning message because we are doing a safe operation of checking the input. This type safe checking operation (known as Generics) helps in realtime a lot. For example, a software asks the teacher to give the roll numbers of students that did not pay the fees. The input for the program is taken from keyboard. If the teacher adds by mistake a roll number of 25.5, the compiler raises error; the same in C/C++, should be checked with extra code.

**Let us go a little bit deeper where ArrayList stores only Employee objects (Generics Java storing Employee objects)**

Java

|  |
| --- |
| ***import java.util.ArrayList;***  ***public class Employee{***  ***String name;***  ***int salary;***  ***public Employee(String name, int salary)  {***  ***this.name = name;***  ***this.salary = salary;***  ***}***  ***public String toString()  {***  ***return name + " : " + salary;***  ***}***  ***public static void main(String args[])  {***  ***Employee emp1 = new Employee("S N Rao", 9999);***  ***Employee emp2 = new Employee("Jyostna", 8888);***  ***ArrayList<Employee> al = new ArrayList<Employee>(); // observe, syntax of Generics Java***  ***al.add(emp1);                                       // while creating generics ArrayList***  ***al.add(emp2);***  ***System.out.println(al.get(0));***  ***System.out.println(al.get(1));***  ***}***  ***}*** |

[](http://way2java.com/wp-content/uploads/2013/11/image4.png)

A **generic ArrayList** object **al** storing only Employee objects is created and added with two Employee objects **emp1** and **emp2**. Later retrieved with get() method overriding [toString()](http://way2java.com/java-general/tostring-java/) method.

**Lambda Function Java**

The strength of [**Java 8**](http://way2java.com/java-versions-2/jdk-1-8-features/) is the introduction of **Lambda expression** like the strength of [Java 5](http://way2java.com/java-versions-2/jdk-1-5-java-se-5-version/) is [annotations](http://way2java.com/annotations/annotations-introduction/).

**1. What is Lambda?**

Lambda, represented as [image](http://way2java.com/wp-content/uploads/2013/08/image18.png)is the 11th letter in in the **Greek** alphabet and having a value of 30 in Greek numerals. Lambda is used by every science and technology to represent something like lambda particle is sub atomic particle, in crystal optics to represent the period of lattice, in chemistry lambda is an isomer, to represent wave length in optical fiber etc.

Lambda refers **anonymous function** in a programming language. See the following JavaScript function to know what is an anonymous function.

***var price = function (k) { };***

The function (function is keyword in JavaScript) does not have a name (like function getMe(k)). This is anonymous function or this is **Lambda function**. The lambda function does not bind with a name (nameless).

From [JDK 1.8](http://way2java.com/java-versions-2/jdk-1-8-features/), Java allows you to create Lambda functions. Lambda functions will become a powerful concept once integrated with Java.

**2. What is Lambda function?**

**Lambda function**, generally known as **Lambda expression**, is a function but without a name. It is very much used in languages like **Python** and **Ruby** (which borrowed from LISP) etc. Now in JDK 1.8, designers are interested to introduce in Java. An anonymous function (lambda function) does not carry name, access specifier, access modifier, parameters etc.

**3. Why lambda is required in Java?**

It is just to simplify the code. Lambda function is very convenient to use in the same place where we write a function. If you would like to use the function only once, **lambda function** is the more convenient way. It reduces typing of much code. Because the function code is written directly where we use the function, the programmer need not go another part of application to see the function code (if required to modify).

**4. What is the symbol we use?**

To write lambda function in Java, we use **->** symbol.

**5. Where we use Lambda functions?**

Lambda function code must be designed to be very short, else, difficult to read. Lambda function can be used anywhere you would like to use a normal function. It is more convenient where a function contains a single statement.

**6. Can we avoid Lambda functions completely?**

Yes, we can. Using them is never compulsory. Did not Java survive without lambda up to JDK 1.7? It is just stylish. Instead of Lambda, you can use a regular normal function