

GRAB THE TIGER BY THE TAIL!

JAVA 5 NEW FEATURES

John Jay King, King Training Resources

Introduction to Java 5.0

Java has become the dominant development language for web-based systems. Since its release to the public in 1995 (Java 1.0) Java has matured improving functionality and performance with each new release. In September 2004 Sun released Java 5.0 (code-name "Tiger"). This session discusses and demonstrates what's new in Java syntax as well as some internal improvements that make this the best release of Java ever. Specific topics include: generics, enhanced for loop iteration, autoboxing/unboxing, typesafe enumerations, static imports, annotations, formatted (ala printf) output, variable argument lists, input scanner, and other features.

Numbering change

The current release's actual name is Java 2 Standard Edition 5.0, commonly referred to by Sun as Java 5. Sun has changed the version numbering scheme so that what would have been J2SE 1.5 is now J2SE 5.0 (the leading "1" was dropped from the name). As the Sun website says, "...to better reflect the level of maturity, scalability, and security ..." the name is now Java 5. The code name during product development was "Tiger" hence the name of this paper (Tiger was a pretty popular product code name in 2005 with other vendors too).

To complicate things just a little, the software returns "1.5" if you run the "java -version" command and most Sun documentation including Javadoc references 1.5 rather than 5.0 when specifying the new version. Java 2 Version 5.0 (1.5) still uses "Java 2" to denote the second generation of Java and to illustrate the family nature of J2SE, J2EE, and J2ME.

The "1.5" version numbering will show up in places where developers (not users) will see them. In all cases where Sun's software or documentation refers to Java 1.5 it means the same product as the Java 5.0 version (1.5 and 5.0 are identical).

Java 1.5 show up when using the java and javac executables:

```
java -version (returns java version "1.5.0")
java -fullversion (returns java full version "1.5.0-b64")
javac -source 1.5 (javac -source 5 also works)
```

Java 1.5 shows up in the system properties java.version and java.vm.version as well as in the "@since" Javadoc entries for new classes and members. The installation directories for both the JDK and JRE are "jdk1.5.0" and "jre1.5.0" respectively. And finally two URLs take you to the same place at Sun regarding the new version:

<http://java.sun.com/j2se/1.5.0>
<http://java.sun.com/j2se/5.0>

JDK/SDK nomenclature

Another change you may notice is that Sun has officially resurrected the name "JDK" as in "Java 2 JDK" rather than using the "SDK" moniker adopted by Java 1.2, 1.3, and 1.4. Sun's website explains that this is due to the overwhelming tendency of the development body to call it the JDK anyway. They have also returned to calling the runtime environment "JRE" rather than "J2RE" (again the development community is stubborn).

Here are the "official" names from Sun:

- Java™ 2 Platform Standard Edition 5.0 J2SE™ 5.0
- J2SE™ Development Kit 5.0 JDK™ 5.0
- J2SE™ Runtime Environment 5.0 JRE 5.0

Where to get Java 5.0

You can download the current version of J2SE Version 5 from the following URL:

<http://java.sun.com/j2se/1.5.0/index.jsp>

Potential conflicts

A problem that presents itself in our web-ready world is that eager developers can find out about new language features and obtain snippets of code to download easily. The developers then *upgrade their development systems* and proceed to carefully test some new software or changed software using a new feature. Of course the new features work perfectly in the test environment. However, when they try to deploy it doesn't work! The Java version supported by the local web server, application server, and/or database server frequently limits the ability to use newer features. While Java is generally backward compatible, older JVMs are not likely to recognize newer features.

New syntax features

Java 5/1.5 provides many new features improving the Java environment itself and offering many enhancements to the development environment. As with most significant Java releases there is a performance improvement across the board. The Garbage Collection mechanism has been made quicker and more effective internally. Many new classes have been added to the Java standard libraries. Here is a comparison of Java library size from version to version:

- Java 1.2 1500+ Classes and Interfaces
- Java 1.3.1 1800+ Classes and Interfaces
- Java 1.4 2700+ Classes and Interfaces
- Java 5 (Java 1.5) 3200+ Classes and Interfaces

Finally, Java's syntax has had several improvements to make programming simpler and to introduce type-safety to many object types. Topics covered specifically in this paper include: generics, enhanced for loop, auto boxing/unboxing, typesafe enumerations, static import, metadata via annotations, formatted output, variable argument lists, simplified input processing via scanner, and improved synchronization.

Generics

At first glance Java's Generics feels familiar to those of us who have used C++ templates, but Java Generics are so much more. Java provides many opportunities for manipulating objects where the actual object type is stripped and must be re-supplied using a cast when the object is used later. For instance, look at the code below:

```
ArrayList oldStyle = new ArrayList();
oldStyle.add(new String("Hello"));
oldStyle.add(new String("there"));
oldStyle.add(new Integer(12)); // ok the old way
oldStyle.add(new String("whoops"));
// following loop raise runtime error
for (Iterator i = oldStyle.iterator();i.hasNext();) {
    System.out.println("Entry = " + (String) i.next());
}
```

This code generates the runtime error "ClassCastException" since the third item being retrieved from the ArrayList is not the expected data type. Using Generics allows us to specify the allowable data type for a Java object so that the compiler will catch errors like this and keep us from deploying code with this type of flaw. The syntax calls for specifying the data type inside less-than "<" and greater-than ">" symbols when defining the object as follows:

```
ArrayList<String> newStyle = new ArrayList<String>();
```

In this example I show the String class but the class name within the brackets may be any class available in your CLASSPATH. The compiler will now use the data type specified to restrict what may be placed into the object at compile time. To be precise, any iterator defined to process a list created with Generics must also specify a datatype.

```
Iterator<String> i = newStyle.iterator();
```

So the rewritten example looks something like this:

```
ArrayList<String> newStyle = new ArrayList<String>();
newStyle.add(new String("Hello"));
newStyle.add(new String("there"));
// following line raises compile error so it is commented out!
//newStyle.add(new Integer(12)); // compile error
newStyle.add(new String("whoops"));
for (Iterator<String> i = newStyle.iterator();i.hasNext();) {
    System.out.println("Entry = " + (String) i.next());
}
```

Java still allows polymorphism with Generics. Objects using subclass types of a superclass used in a generic are allowed by the compiler so polymorphic activity acts as you would expect. In the following code example, "Room" is the superclass and "Office" is the subclass:

```
ArrayList<Room> roomArrayList = new ArrayList<Room>();
roomArrayList.add(new Room());
roomArrayList.add(new Room(10,20,"Kitchen"));
roomArrayList.add(new Office());
roomArrayList.add(new Office(1,15,20,"Conference"));
Iterator<Room> myIt = roomArrayList.iterator();
Room currRoom = null;
while(myIt.hasNext()) {
    currRoom = myIt.next();
    System.out.println("Room " + currRoom.getRoomName());
}
```

Wildcards may be specified in the cases where you wish to truly allow any data type but do not want the compiler to throw warning messages.

```
public void printIt(List<?> inList, PrintStream out)
```

Additional generic features allow the inheritance of certain behaviors using <MyNumber extends Number> which might also use the wild-card "?" indicator. Consult the Java 5 documentation if you want more information.

Lint is not just in your navel

The "lint" command has long been used in the Unix world to verify C program syntax, data type use, and portability of code. The Java 5 (1.5) compiler adds a new "-Xlint" switch allowing the compiler to flag potential errors that are otherwise syntactically correct. Add lint to the compile execution as follows:

```
javac UsingGenerics.java -Xlint:unchecked -deprecation
```

Options for Xlint include:

- all Get all lint warnings
- deprecation Warns about deprecated API use (similar to -deprecation)
- fallthrough Flags cases in a switch statement that "fall through" to the next case
- finally Indicates an "finally" block that cannot reach completion
- path Warns of path directories specified on command line that do not exist
- serial Indicates that one or more Serializable classes do not have serialVersionUID defined
- unchecked Warns about unchecked use of generic types

When compiled using the -Xlint:unchecked switch, the code illustrated earlier:

```
ArrayList oldStyle = new ArrayList();
oldStyle.add(new String("Hello"));
oldStyle.add(new String("there"));
oldStyle.add(new Integer(12)); // ok the old way
oldStyle.add(new String("whoops"));
// following loop raise runtime error
for (Iterator i = oldStyle.iterator();i.hasNext();) {
    System.out.println("Entry = " + (String) i.next());
}
```

Results in the following compile warnings for all four "add" statement similar to the following::

```
C:\JavaTiger\src\samples\UsingGenerics.java:30: warning: [unchecked] unchecked call to add(E) as a
member of the raw type java.util.ArrayList
    oldStyle.add(new Integer(12)); // ok the old way
                   ^
C:\JavaTiger\src\samples\UsingGenerics.java:31: warning: [unchecked] unchecked call to add(E) as a
member of the raw type java.util.ArrayList
    oldStyle.add(new String("whoops"));
                   ^
```

4 warnings

Anytime you can catch an error at compile time rather than runtime you have done well.

Additions to Arrays Class

The `java.util.Arrays` class has had additions that improve usability and convenience. One of the more useful features is the addition of a `toString()` method. `Arrays.toString()` prints the contents of any array/collection in tabular form. This is certainly something you could do manually, but its nice that they've done the drudge work for us.

```
int[] anArray = { 1, 3, 5, 7, 9, 11, 13, 15, 16, 20 };
System.out.println(Arrays.toString(anArray));
```

Generates the following output:

```
[1, 3, 5, 7, 9, 11, 13, 15, 16, 20]
```

Another method `Arrays.deepToString()` displays the contents of a multi-dimensional array. Given the following array:

```
int[][] apartment = new int[5][4];
```

The results of using `Arrays.toString()` and `Arrays.deepToString()` are illustrated below.

First, using `Arrays.toString()` the contents of the first level show as addresses (Windows PC used for example):

```
[[I@10b62c9, [I@82ba41, [I@923e30, [I@130c19b, [I@1f6a7b9]
```

Next, using `Arrays.deepToString()` the contents of the array are listed:

```
[[0, 0, 0, 0], [0, 1, 2, 3], [0, 4, 5, 6], [0, 7, 8, 9], [0, 10, 11, 12]]
```

`Arrays` also added a new `Arrays.deepEquals(array1,array1)` to compare contents of multi-dimensional arrays. In addition Java has added `Arrays.hashCode()` and `Arrays.deepHashCode()` so support this activity.

- `Arrays.hashCode(anArray)` -1796951359
- `Arrays.hashCode(apartment)` -1374565195
- `Arrays.deepHashCode(apartment)` 915682214

Enhanced for loop

Beginning with Java 5 (Java 1.5) a new style of “for” loop is available specifically for looping through each item in a simple array/collection object. The new for loops are sometimes called “for-in” loops even though the word “in” is not part of the syntax. Consider the code below:

```
public class ForEachLoop {
    public static void main ( String[] args ) {
        String[] lastName = new String[5];
        lastName[0] = "Winklemann";
        lastName[1] = "Ruiz";
        lastName[2] = "Gandhi";
        lastName[3] = "Yakazuki";
        lastName[4] = "Jones";
        for (String thisName : lastName) {
            System.out.println("Name is " + thisName);
        }
    } // end main
} // end ForEachLoop class
```

This code loops through each entry in an array or collection object returning one value at a time for processing. In other words an Iterator is used without an Iterator being defined! The new “for” construct is equally at home with either a traditional array or an object of some collection type. The syntax specifies:

```
for (String thisName : lastName) {
    System.out.println("Name is " + thisName);
}
```

- Data type of one item in array/collection (String above)
- Local name used for returned item in for loop (thisName above)
- : (colon, think of it as the word “in”)
- Name of array or collection object that implements the new `java.lang.Iterable` interface (lastName above)

Autoboxing and Unboxing

Before Java 5 (Java 1.5) using wrapper classes was sometimes clumsy, you could not directly move data between wrapper class objects and primitives without invoking code from the wrapper class. Though it makes sense, the following code will not compile in Java 1.4 or earlier due to “incompatible types” since the primitive int and Integer type are not the same. With Java 5 (Java 1.5) and later the compiler inserts code to automatically “box” and “unbox” values. Perhaps the best thing of all is that boxing and unboxing are automatic!

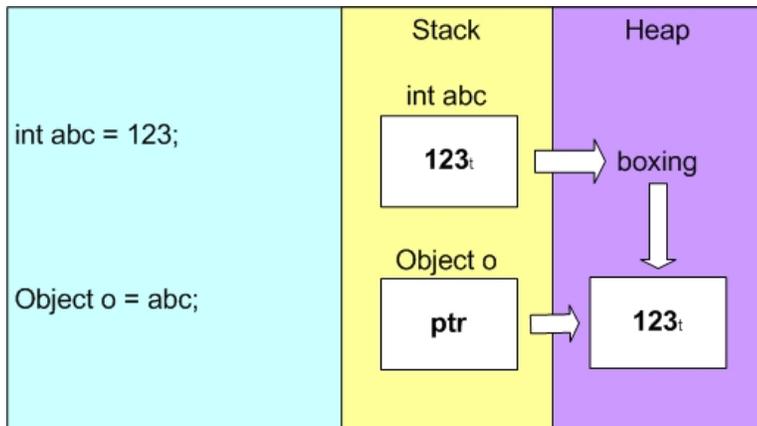
```
Integer intObject = new Integer(123);
int intPrimitive = intObject;           // pre-Java 5 error "incompatible types"
double doublePrimitive = 123.45;
Double doubleObject = doublePrimitive; // pre-Java 5 error "incompatible types"
```

In Java 1.4 (or earlier releases), to properly move data between wrapper class objects and primitives requires the use of wrapper class methods as follows:

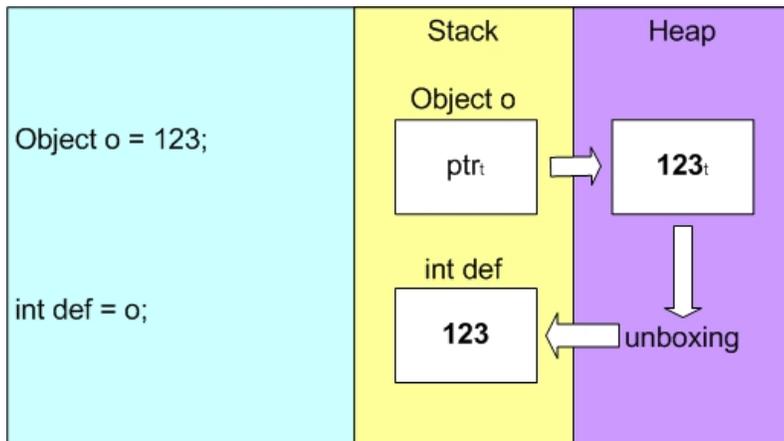
```
Integer intObject = new Integer(123);
int intPrimitive = intObject.intValue();
double doublePrimitive = 123.45;
Double doubleObject = new Double(doublePrimitive);
```

The advent of automatic Boxing and automatic Unboxing greatly simplifies code when using Collection and other types of objects. Boxing and Unboxing is also important since Primitive data and Reference Type data are stored in different places. Primitives representing local variables are stored on the stack while objects are stored in heap memory.

When an integer is assigned to an object, the system, “boxing” makes a copy of the value on the heap and points the object to the new value.



When an object assigned to an integer, “unboxing” copies the value from the heap into the variable's storage in the stack



Typesafe enumerations

Another new Java 5 feature that will look familiar to C programmers is the "Enum" object. Enum allows assignment of a specific set of values to associated type. Here is an example using an enum:

```
package samples;
public class UsingEnums {
    public enum Weekdays {
        Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
    };
    public UsingEnums() {
        Weekdays weekDays;
    }
    public static void main(String[] args) {
        UsingEnums myUE = new UsingEnums();
    }
}
```

Static import

Beginning with Java 5 (Java 1.5) it is possible to reduce coding effort by specifying in an import that the items to be imported are static. Static import is especially useful when the same static methods or variables are used repeatedly. It is considered "best practice" to import each method/variable explicitly rather than using wild-card imports. Be careful, the use of static import may confuse those who read your code, please consider using comments to indicate what is going on (at least for the next couple of years).

```
import static java.lang.Double.parseDouble; // won't compile in Java 1.4
import static java.lang.Integer.parseInt; // won't compile in Java 1.4
public class StaticImportDemo {
    public static void main(String[] args) {
        String intValue = "123";
        String dblValue = "567.89";
        double resultValue = 0;
        try {
            resultValue = parseInt(intValue) + parseDouble(dblValue);
            System.out.println("resultValue is " + resultValue);
        }
        catch (NumberFormatException e) {
            System.out.println("Either intValue or dblValue not numeric");
        }
        return;
    }
}
```

Metadata via Annotations

Java 5 introduces a formal method for adding metadata (data about information) to package and type declarations, methods, constructors, parameters, fields, and variables. Java provides several "Standard Annotation" types, three categories of annotations, and a mechanism for creating custom annotation types.

The three Standard Annotation Types are:

- Override (java.lang.Override) Indicates that a method overrides a superclass method

```
@Override
public String accountHtml() {
    // overriding code goes here
}
```

The compiler will throw an error if the method above does not match the name and signature of a method in the superclass thanks to the @Override specification.

- Deprecated (java.lang.Deprecated) Flags method or element as deprecated

```
@Deprecated public class Y2Ktools (
    // deprecated code
}
```

The compiler will throw a warning message when code extends this class when deprecation checking has been specified during compilation. This adds to the functionality of the @deprecated Javadoc tag. Functionality seems to be identical between using the -deprecation compile switch and the -Xlint:deprecation switch.

- SuppressWarnings (java.lang.SuppressWarnings) Turns off compiler warnings

```
@SuppressWarnings("unchecked")
public void printArray(SomeObject objIn) {
    // code goes here
}
```

To suppress multiple warnings use a list (inside brackets) as follows:

```
@SuppressWarnings({"unchecked", "fallthrough"})
```

As illustrated above, Annotations are used with an "@" sign (@) followed by a keyword but the syntax varies depending upon the type of annotation. The Override annotation is an illustration of a "Marker" annotation; the name of the annotation (Override above) is the metadata. The SuppressWarnings annotation illustrates the use of a "Single-value" annotation providing a name for the annotation and a single value associated with it the examples of SuppressWarnings above to not show the option of using the keyword "value" (@SuppressWarnings(value={"unchecked","fallthrough"}). Full annotations are not really a category so much as the full annotation syntax variation allowed with subdivisions as necessary.

It is possible to create your own annotations using @interface and other new code elements. Please consult the Sun documentation for more information.

Formatted Output

Even though Java offers the excellent java.text package classes and methods for formatting of data, people with C backgrounds still miss "printf" and its functionality. The Java 5 team added the java.util.Formatter class adding new capabilities to all of Java's output methods. Formatting may be applied using the locale and may be directed to a "sink" (often a file). The methods "format()" and "printf()" (C programmers say "hooray!") are synonymous. The PrintStream class now includes methods for "format()" and "printf()" using the same formatting characters.

The formatting characters used in Java are similar to those used in C but with a little more sophistication:

'%b', '%B'	If the argument is null, the result is "false", if it is boolean or Boolean the result is the string returned by String.valueOf(), if it is not null or Boolean the result is "true".
'%h', '%H'	Formats boolean output as "true"/"TRUE" ("%h"/"%H"), "false"/"FALSE" ("%h"/"%H"), or "null"
'%s', '%S'	Formats output as String data using argument's formatTo() method (if available) or toString()
'%c', '%C'	Formats Byte, Short, Character, or Integer as a single character
'%d'	Formats Byte, Short, Integer, Long, or BigInteger as an integer (base-10)
'%o'	Formats Byte, Short, Integer, Long, or BigInteger as an octal integer
'%x', '%X'	Value (or its hashCode) formatted as hexadecimal integer
'%e', '%E'	Formats Float, Double, or BigDecimal value using exponential notation
'%f'	Formats Float, Double, or BigDecimal value using floating-point notation
'%g', '%G'	Formats Float, Double, or BigDecimal value with less than six significant digits using floating-point notation
'%a', '%A'	Formats Float, Double, or BigDecimal value with less than six significant digits using floating-point notation with base-16 values for the decimal part and base-10 values for the exponent
'%t', '%T'	Prefix used for date/time conversions (see below)
'%%'	Used to print a literal '%'
'%n'	Platform-specific line separator

Then printing Times, Dates, and Date/Time items additional codes are used.

Times:

- 'H' 2-digit hour using 24-hour clock (leading zero)
- 'T' 2-digit hour using 12-hour clock (leading zero)
- 'k' Hour using 24 hour clock (0-23)
- 'l' Hour using 12-hour clock (1-12)
- 'M' 2-digit minute within hour (leading zero)
- 'S' 2-digit seconds within minute (leading zero)
- 'L' 3-digit millisecond within second (leading zeros)
- 'N' 9-digit nanosecond within second (leading zeros)
- 'p' Locale-specific morning or afternoon marker in lower case for "%tp" (am/pm) upper case for "%Tp" (AM/PM)
- 'z' RFC 822 time zone offset from GMT, e.g. -0800
- 'Z' String representing timezone abbreviation
- 's' Seconds since the beginning of the epoch starting 1 January 1970 00:00:00 UTC (Long value)
- 'Q' Milliseconds since the beginning of the epoch starting 1 January 1970 00:00:00 UTC (Long value)

Dates:

- 'B' Locale-specific full month name ("January")
- 'b' Locale-specific abbreviated month name ("Jan")
- 'h' Same as 'b'.
- 'A' Locale-specific full name of the day of the week ("Sunday")
- 'a' Locale-specific short name of the day of the week ("Sun")
- 'C' 2-digit year (00-99), four-digit year divided by 100 (leading zero)
- 'Y' 4-digit year (0000-9999)
- 'y' Last two digits of the year (leading zeros)
- 'j' 3-digit (Julian) day of year (001-366, leading zeros)
- 'm' 2-digit month (leading zero)
- 'd' 2-digit day of month (leading zero)
- 'e' 2-digit day of month (1-31)

Date/Time:

- 'R' Time formatted for the 24-hour clock as "%tH:%tM"
- 'T' Time formatted for the 24-hour clock as "%tH:%tM:%tS"
- 'r' Time formatted for the 12-hour clock as "%tI:%tM:%tS %Tp" (morning/afternoon marker ('%Tp) location may be locale-dependent)
- 'D' Date formatted as "%tm/%td/%ty"
- 'F' ISO 8601 complete date formatted as "%tY-%tm-%td".

'c' Date and time formatted as "%ta %tb %td %tT %tZ %tY", e.g. "Sun Jul 20 16:17:00 EDT 1969"

Formatting also uses special flags to control print-related functionality like justification, signs, and zero padding.

'-' Right-justified output (all data types)

'#' Left-justified output (numeric data only)

'+' Output includes sign (numeric data only)

' ' Output includes leading-space for positive values (numeric data only)

'0' Output is zero-padded (numeric data only)

',' Output uses group locale-specific group separators (numeric data only)

'(' Output surrounds negative numbers with parentheses (numeric data only)

Here is an example of a numeric value being formatted using System.out.format(), System.out.printf() works identically:

```
double balance = 1234.56;
System.out.format("Balance is $%,6.2f",balance);
Output:
    Balance is $1,234.56
```

Here is an example of a date being formatted using System.out.format():

```
Date today = new Date();
System.out.format("\nToday is %TF %TT", today,today);
Output:
    Today is 2005-06-09 20:15:26
```

The Formatter class may also be used to format String data anytime, the following example shows the use of the Formatter object and locales:

```
Formatter myUSformat = new Formatter();
Formatter myFRformat = new Formatter();
String balUS = myUSformat.format("Balance is $%,6.2f",balance).toString();
String balFR = myFRformat.format(Locale.FRANCE,"Balance is $%,6.2f",balance).toString();
System.out.println("US " + balUS);
System.out.println("FRANCE " + balFR);
Output:
    US Balance is $1,234.56
    FRANCE Balance is $1 234,56
```

Variable argument lists

Java 5's new variable argument list feature fills a void in creating overloaded methods when you don't always know how many objects will be passed to the method. Variable argument lists allow specification of a method that can accept a final parameter of the same time with the number of values to be determined at runtime. Only one variable argument list is allowed per method and it must be the last argument defined for the method. The ellipsis "..." is used to indicate that an argument might appear a variable number of times.

In the class below, the constructor is designed to expect a variable number of options for the specified automobile:

```
public Auto (String year, String make, String model, String... options) { ... }
```

The variable argument list allows specification of multiple cars with varying lists of options as shown below. Though this example is illustrated using String data type, any data/object type may be used.

```
Auto johnsToy = new Auto("1969","Fiat","124 Spider", "5-speed", "disk brakes");
Auto myTruck = new Auto("1997","Ford","Expedition", "Automatic",
    "Four-wheel drive","power windows","power locks",
    "air-conditioning","stereo with cd changer","tinted glass");
```

Simplified input processing via scanner

Console input is not common in production programs, but it is very useful when learning Java or creating test modules. Before Java 5 we frequently used System.in and its "readLine()" method to access the keyboard. Java 5 introduces the java.util.Scanner class designed specifically for this purpose greatly reducing the amount of code needed to communicate with the keyboard.

Here is the old way of getting user input:

```
String firstName;
InputStreamReader inStream = new InputStreamReader(System.in);
BufferedReader inBuf = new BufferedReader(inStream);
System.out.print("Please enter your first name => "); // stays on line
try {
    firstName = inBuf.readLine();
} // end of first try block
catch (IOException e) {
    System.out.println("Problem reading first name, bye-bye!");
    return;
} // end catch block
```

And here is the much-simpler Scanner version:

```
String lastName;
System.out.print("Please enter your last name => "); // stays on line
Scanner fromkeyboard = new Scanner(System.in);
lastName = fromkeyboard.next();
```

Scanner objects have several input methods, `next()` (shown above) returns the next String token from the input buffer. Scanner offers several variations on `next()` including `next(comparePattern)` or `next(compareString)` where the next String matching a given pattern is returned. In addition, many numeric variations exist: `nextBigDecimal()`, `nextBigInteger()`, `nextBoolean()`, `nextByte()`, `nextDouble()`, `nextFloat()`, `nextInt()`, `nextLine()`, `nextLong()`, and `nextShort()`. In addition the following methods allow specification of a starting position: `nextBigInteger(n)`, `nextByte(n)`, `nextInt(n)`, `nextLong(n)`, and `nextShort(n)`.

Synchronization

Beginning with Java 5 (Java 1.5) the `java.util.concurrent.locks`, `java.util.concurrent`, and `java.util.concurrent.atomic` packages are available providing better locking support than provided by the "synchronized" modifier. However, all existing code still works as before.

The `java.util.concurrent.locks.Lock` interface has several methods including: `lock()` to obtain a lock (blocking if the lock cannot be obtained), `unlock()` to release a lock, `lockInterruptibly()` obtains a lock but allows interruptions, and `tryLock()` attempts to obtain a lock without a wait. The `java.util.concurrent.locks.ReentrantLock` class behaves very much like using `synchronized` does today. The `java.util.concurrent.locks.Condition` interface allows complex and multiple conditional waits. The `java.util.concurrent.locks.ReadWriteLock` interface allows separate locks for reading and writing. If you are interested in knowing more about the new locking features please consult the Sun documentation.

StringBuilder class

New with Java 5.0 (Java 1.5) the `java.lang.StringBuilder` class provides a faster alternative to `StringBuffer`. In most ways `StringBuilder` works exactly the same as `StringBuffer`. `StringBuilder` is faster than `StringBuffer` because it is not `ThreadSafe` and multiple threads should not access `StringBuilder` objects without `Synchronizing`. Developers should use `StringBuilder` when speed is important in a single-thread environment and use `StringBuffer` if multiple threads might require access.

Strings are very useful but immutable, if applications are constantly changing the value of a String, memory is being allocated and reallocated in an inefficient manner. The `StringBuilder` (`java.lang.StringBuilder`) class by contrast is mutable and should be used for strings that will change in size or value frequently. The efficiency difference is significant for applications that "build" output lines by concatenating strings. Here is an example using `StringBuilder`:

```
/** StringBuilderTest.java */
public class StringBuilderTest {
    public static void main(String[] args) {

        StringBuilder myStrBldr = new StringBuilder("How");

        myStrBldr.append(" now");
        myStrBldr.append(" Brown");
        myStrBldr.append(" Cow?");

        System.out.println("StringBuilder = " + myStrBldr);
    }
}
```

Conclusion

Java 5 or Java 1.5, whatever the name is this new version of Java provides many features to make the life of developer richer allowing creation of better and more interesting programs. Generics and the new for loop are probably exciting enough in their own right, but the other features all work together to make this new release the best Java ever.

About the Author

John King is a Partner in King Training Resources, a firm providing instructor-led training since 1988 across the United States and Internationally. John specializes in application development software on a variety of platforms including Web, Unix, Linux, IBM mainframe, and personal computers. John has worked with Oracle products and the database since Version 4 and has been providing training to Oracle application developers since Oracle Version 5. John develops and presents customized courses in a variety of topics including Oracle, DB2, UDB, SOA, Java, XML, C#, and various programming languages. He has presented papers at many industry events including: IOUG-A Live!, UKOUG Conference, EOUG Conference, AUSOUG Conferences, RMOUG Training Days, MAOP-AOTC, NYOUG, and the ODTUG conference.

John Jay King - King Training Resources
6341 South Williams Street
Littleton, CO 80121-2627 U.S.A.
Phone: 1.303.798.5727 1.800.252.0652 (within the U.S.)
Fax: 1.303.730.8542
Email: john@kingtraining.com
Website: www.kingtraining.com