

The following table shows the conversions between primitive types, it shows the casting operation for explicit conversions:

	from byte	from char	from short	from int	from long	from float	from double	from boolean
to byte	-	(byte)	(byte)	(byte)	(byte)	(byte)	(byte)	N/A
to char		-	(char)	(char)	(char)	(char)	(char)	N/A
to short		(short)	-	(short)	(short)	(short)	(short)	N/A
to int				-	(int)	(int)	(int)	N/A
to long					-	(long)	(long)	N/A
to float						-	(float)	N/A
to double							-	N/A
to boolean	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-

Autoboxing/unboxing

Autoboxing/unboxing

Autoboxing and unboxing, language features since Java 1.5, make the programmer's life much easier when it comes to working with the primitive wrapper types. Consider this code fragment:

```
int age = 23;
Integer ageObject = new Integer(age);
```

Primitive wrapper objects were Java's way of allowing one to treat primitive data types as though they were objects. Consequently, one was expected to 'wrap' one's primitive data type with the corresponding primitive wrapper object, as shown above.

Autoboxing

Since Java 1.5, one may write as below and the compiler will automatically create the wrap object. The extra step of wrapping the primitive is no longer required. It has been 'automatically boxed up' on your behalf.



Points to ponder

Keep in mind that the compiler still creates the missing wrapper code, so one doesn't really gain anything performance-wise. Consider this feature a programmer convenience, not a performance booster.

```
int age = 23;
Integer ageObject = age;
```

Unboxing

Uses the same process in reverse. Study the following code for a moment. The **if** statement requires a **boolean** primitive value, yet it was given a Boolean wrapper object. No problem! Java 1.5 will automatically 'unbox' this.

```
Boolean canMove = new Boolean(true);

if ( canMove )
{
    System.out.println( "This code is legal in Java 1.5" );
}
```

```
}
```

java.lang.String

Navigate **Language Fundamentals** topic:

java.lang.String

String is a special class built into the Java language defined in the java.lang package.

The String class represents character strings. String literals in Java programs, such as "abc", are implemented as instances of this class. Strings are *immutable*; that is, they cannot be modified once created.

For example:

```
String str = "This is string literal";
```

On the right hand side a String object is created represented by the string literal. Its object reference is assigned to the str variable.

The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. For example:

```
String str = "First part" + " second part";  
// --- Is the same as:  
String str = "First part second part";
```

Integers will also be converted to String after the (+) operator:

```
String str = "Age=" + 25;
```

Each Java object has the String toString() inherited from the Object class. This method provides a way to convert objects into Strings. Most classes override the default behavior to provide more specific (and more useful) data in the returned String.

The String class provides a nice set of methods for string manipulation. Since String objects are immutable, all methods return a new String object. For example:

```
name = name.trim();
```

The trim() method returns a copy of the string with leading and trailing whitespace removed. Note that the following would do nothing useful:

```
name.trim();    // wrong!
```

This would create a new trimmed string and then throw it away. Study the String class and its methods carefully. Strings are ubiquitous in Java; it will serve you well to know how to manipulate them skillfully.

Using StringBuffer/StringBuilder to concatenate strings

Remember that String objects are immutable objects. Once a String is created, it can not be modified, takes up memory until garbage collected. Be careful of writing a method like this :

```
public String convertToString( Collection coll )
{
    String str = "";
    Iterator iter = coll.iterator();
    while ( iter.hasNext() ) // loops through every element in coll
    {
        String oneElem = (String) iter.next();
        str = str + oneElem + " ";
    }
    return str;
}
```

On the (+) operation a new String object is created at each iteration. Suppose coll contains the elements ["Foo", "Bar", "Bam", "Baz"]. The method creates five Strings ("", "Foo ", "Foo Bar ", "Foo Bar Bam ", and "Foo Bar Bam Baz") even though only last one is actually useful.

Instead use StringBuffer, as shown below, where only one StringBuilder object is created:

To avoid unnecessary memory use like this, use the StringBuffer or StringBuilder class. They provide similar functionality to Strings, but store their data in a mutable way. Also because object creation is time consuming, using StringBuffer or StringBuilder produces much faster code.

```
public String convertToString( Collection coll )
{
    StringBuilder buf = new StringBuilder();
    Iterator iter = coll.iterator();
    while ( iter.hasNext() ) // loops through every element in coll
    {
        String oneElem = (String) iter.next();
        buf.append( oneElem );
        buf.append( " " );
    }
    return buf.toString();
}
```

StringBuilder was introduced in Java 5. Unlike StringBuffer, StringBuilder isn't thread safe, so you can't use it in more than one thread (see the chapter on Concurrency). However, because it doesn't have to worry about synchronization, StringBuilders are faster.

Comparing Strings

Comparing strings is not as easy as it may first seem. We cannot just use a simple equality statement such as:

```
if(myString == "Hello World!")           //Can't Use this.
{
    System.out.println("Match Found");
}
```