

The allowed operations on String object, however are the set of methods in the java.lang.String class, which includes length(), toString(), toLowerCase(), toUpperCase(), compareTo(String anotherString) and more... .

In addition, String objects also inherit the set of operations from the base class that String extends from, which is java.lang.Object. These operations include methods such as equals(), hashCode(), wait(), notifyAll(), and getClass().

```
private String name = "Marry Brown";
```

In the above example the *name* object reference's attributes are:

- **Type is** : String
- The **referenced object** is also : String

Both the java.lang.String class methods and java.lang.Object class methods are available for the object reference *name*.

```
private Object name = "Marry Brown";
```

In the above example the *name* object reference's attributes are:

- **Type is** : Object
- The **referenced object** is : String

Only the java.lang.Object class methods are available for the object reference *name*.

Array Types

Arrays in Java are represented as a built-in Array object. As with object types, they behave as a reference but have a few differences allowing them to allow easy access to sub elements.

When one declares an array, the data type is changed to include square brackets. (While you can instead place these square brackets next to the variable name, this is not recommended.) To create an array, you will need to use the new operator to have it create the Array with the specified number of elements:

```
/* Declares an array named data, and has it assigned to an array with  
25 elements. */  
private int[] data = new int[25];
```

Arrays are described in more detail in the Arrays section.

Primitive Data Types

The Java primitive data types contain pure values, no operations. It is basically data types similar to what other non-object-oriented languages have.

There are arrays of primitive types in Java; but because they are not objects, primitive values can not be put in a collection.

For this reason object wrappers are defined in JDK 'java.lang.*' package for all the primitive types.

	Size (bits)	Example	Minimum Value	Maximum Value	Wrapper Class
Integer types					
byte	8	byte b = 65;	-128	127	Byte
char	16	char c = 'A'; char c = 65;	0	$2^{16}-1$	Character
short	16	short s = 65;	-2^{15}	$2^{15}-1$	Short
int	32	int i = 65;	-2^{31}	$2^{31}-1$	Integer
long	64	long l = 65L;	-2^{63}	$2^{63}-1$	Long
Floating-point types					
float	32	float f = 65f;			Float
double	64	double d = 65.55;			Double
Other					
boolean	1	boolean b = true;	--	--	Boolean
void	--	--	--	--	Void

The types **short**, **int**, **long**, **float**, and **double** are usually used in arithmetic operations; **byte** and **char** can also be used, but less commonly.

The character type **char** is used for text processing. The type **byte** is commonly used in binary file input output operations.

String objects representing literal character strings in Java, in the `java.lang.*` package. `java.lang.String` is not a primitive type, but instead is a special class built into the Java language. For further info, see `String`.

Data Conversion (Casting)

Data conversion (casting) can happen between two primitive types. There are two kinds:

- Implicit : casting operation is not required; the magnitude of the numeric value is always preserved. However, *precision* may be lost when converting from integer to floating point types
- Explicit : casting operation required; the magnitude of the numeric value may not be preserved

Example for implicit casting:

```
int i = 65;
long l = i; // --- int is converted to long, casting is not needed
```

Example for explicit casting:

```
long l = 656666L;
int i = (int) l; // --- long is converted to int, casting is needed
```

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" );
}
```