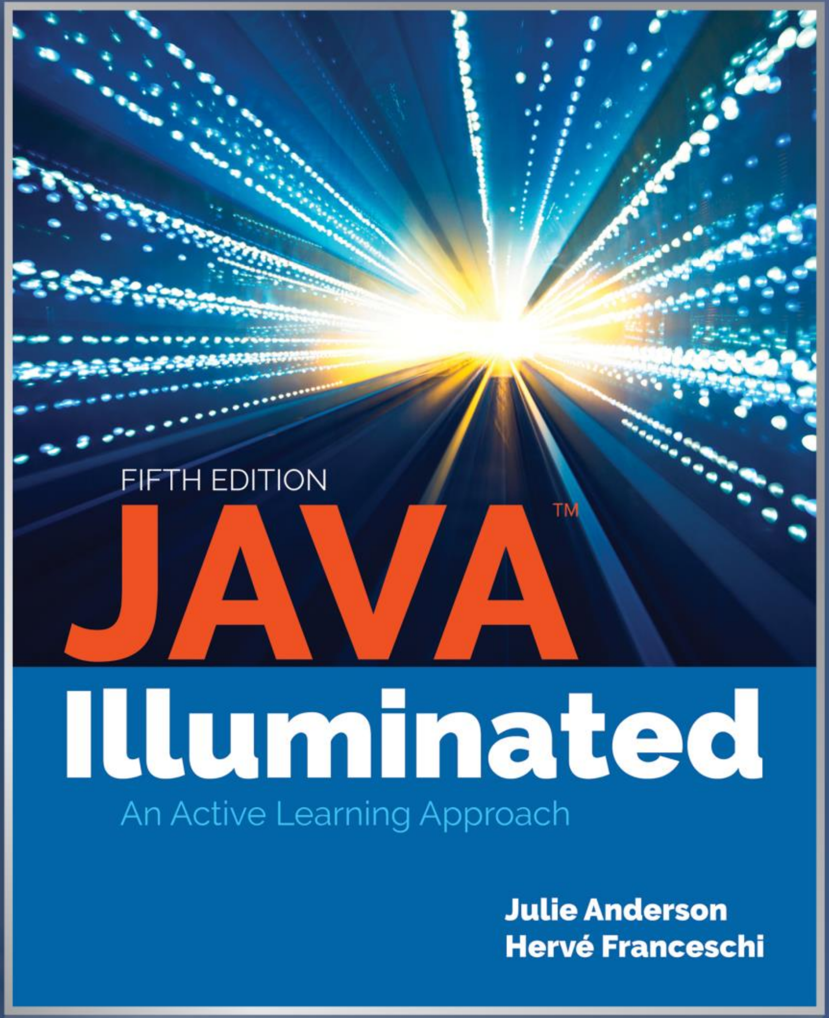
Chapter 3

Object-Oriented Programming, Part 1: Using Classes



Topics

- Class Basics and Benefits
- Creating Objects Using Constructors
- Calling Methods
- Using Object References
- Calling Static Methods and Using Static Class Variables
- Using Predefined Java Classes

Object-Oriented Programming

- Classes combine data and the methods (code) to manipulate the data.
- Classes are a template used to create specific objects.
- All Java programs consist of at least one class.
- Two types of classes:
  - Application classes
  - Service classes

Why Use Classes?

- Usually, the data for a program is not simply one item.
- Often we need to manage entities like students, books, flights, etc.
- We need to be able to manipulate such entities as a unit.
- Classes allow us to separate the data for each object, while using common code to manipulate each object.

Example

- *Student* class
  - Data: name, year, and grade point average
  - Methods: store/get the value of the data, promote to next year, etc.
- *Student* object

  Object name: *student1*

  Data: *Maria Gonzales*, *Sophomore*, *3.5*

Objects

- Object reference
  - An identifier of the object
- Instantiating an object
  - Creating an object of a class; assigns initial values to the object data
  - Objects need to be instantiated before being used.
- Instance of the class
  - An object after instantiation

Class Members

- Members of a class
  - The class's fields and methods
- Fields
  - Instance variables and *static* variables (we'll define *static* later)
  - Instance variables
    - Variables defined in the class and given a value in each object

  Fields can be:
  - Any primitive data type (*int*, *double*, etc.)
  - Objects of the same or another class
- Methods
  - The code to manipulate the object data

Encapsulation

- Instance variables are usually declared to be *private*, which means that they cannot be accessed directly outside the class.
- Users (clients) of the class can only reference the *private* data of an object by calling methods of the class.
- Thus the methods provide a protective shell around the data. We call this encapsulation.
- Benefit: The class's methods can ensure that the object data is always valid.

Naming Conventions

- Class names: start with a capital letter.
- Object references: start with a lowercase letter.
- In both cases, internal words start with a capital letter.
- Example: classes: *Student*, *PetAnimal*

  objects: *marcusAustin*, *myBoaConstrictor*

Reusability

- Reuse
  - Class code is already written and tested.
  - New applications can be built faster.
  - The applications will be more reliable.

Example: A *Date* class could be used in a calendar program, appointment-scheduling program, online shopping program, etc.

How to Reuse a Class

- We don't need to know how the class is written or see the code of the class.
- We do need to know the application programming interface (API) of the class.
- The API is published and tells us:
  - How to create objects
  - What methods are available
  - How to call the methods

1. Declare an Object Reference

- An object reference holds the address of the object.
- Syntax to declare an object reference:

```
ClassName objectReference;
```

or

```
ClassName objectRef1, objectRef2…;
```

- Example:

```
SimpleDate d1;
```

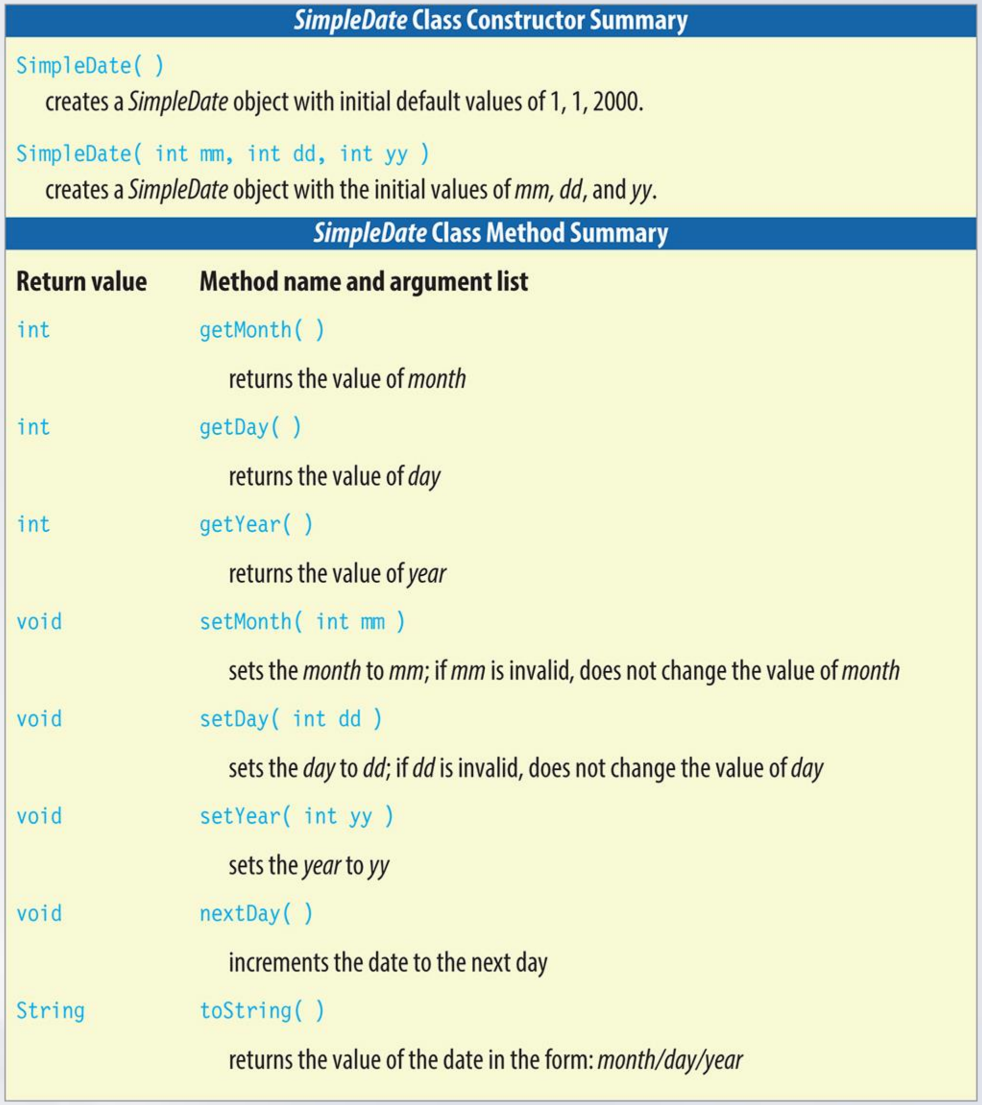
2. Instantiate an Object

- Objects must be instantiated before they can be used
- Call a constructor using the *new* keyword
  - Constructor: a special method that creates an object and assigns initial values to data
- A constructor has the same name as the class.
- Syntax:

```
objectReference =
                 new ClassName( arg list );
```

- *Arg list* (argument list) is comma-separated list of initial values to assign to the object data

SimpleDate Class API



SimpleDate Class Constructor Summary

`SimpleDate( )`

creates a *SimpleDate* object with initial default values of 1, 1, 2000.

`SimpleDate( int mm, int dd, int yy )`

creates a *SimpleDate* object with the initial values of *mm, dd*, and *yy*.

SimpleDate Class Method Summary

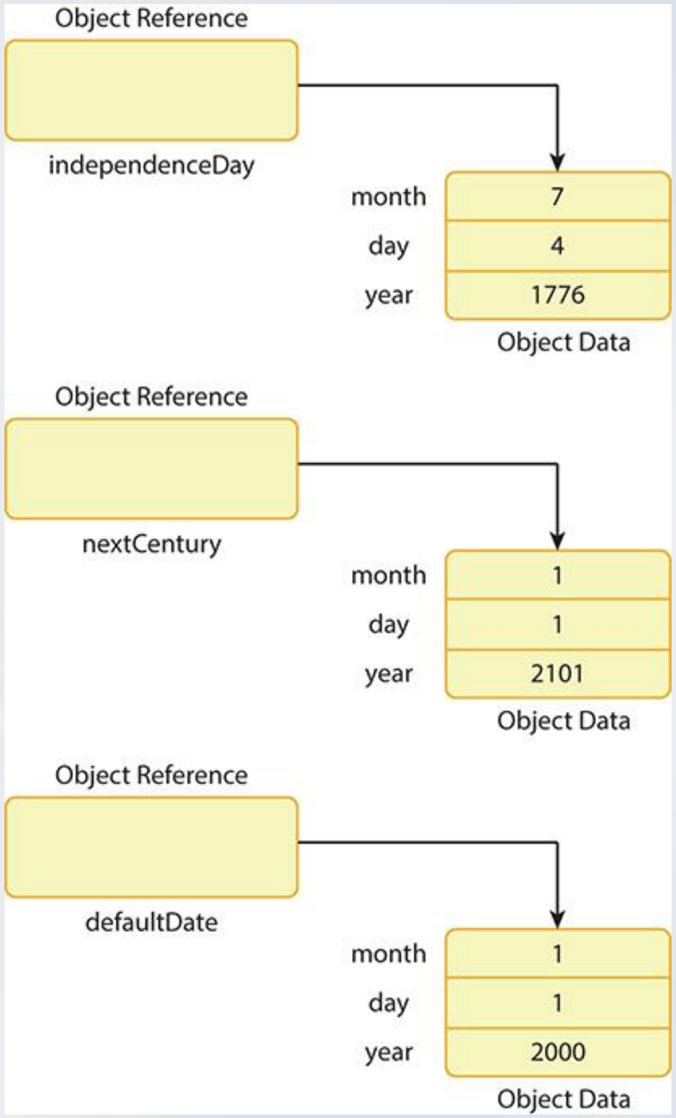
Return value	Method name and argument list
`int`	`getMonth( )` returns the value of *month*
`int`	`getDay( )` returns the value of *day*
`int`	`getYear( )` returns the value of *year*
`void`	`setMonth( int mm )` sets the *month* to *mm*; if *mm* is invalid, does not change the value of *month*
`void`	`setDay( int dd )` sets the *day* to *dd*; if *dd* is invalid, does not change the value of *day*
`void`	`setYear( int yy )` sets the *year* to *yy*
`void`	`nextDay( )` increments the date to the next day
`String`	`toString( )` returns the value of the date in the form: *month/day/year*

Instantiation Examples

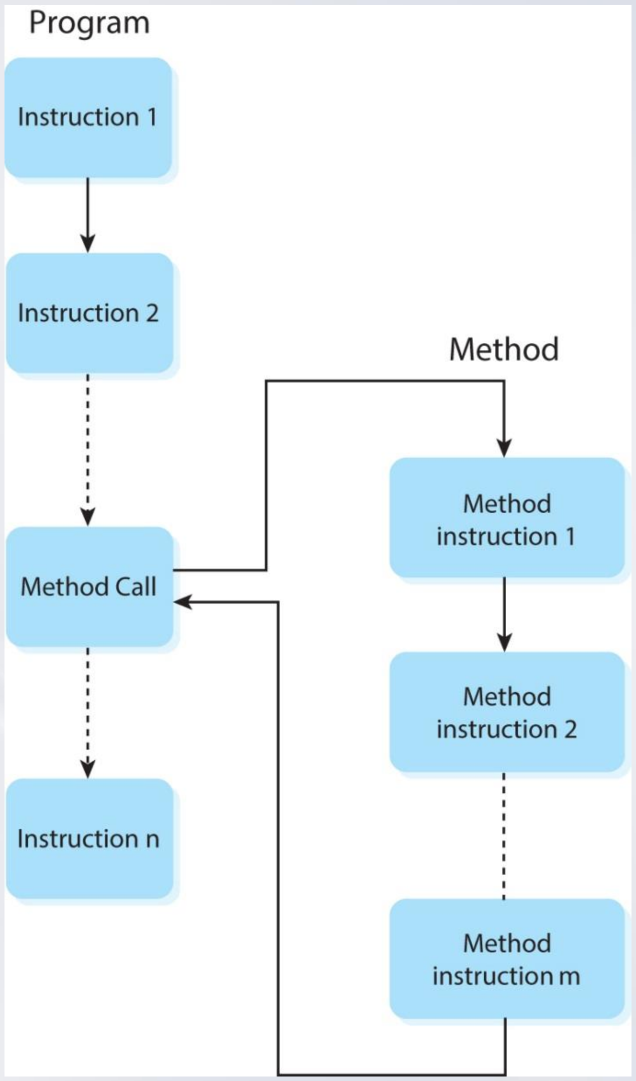
```
SimpleDate independenceDay;
independenceDay =
     new SimpleDate( 7, 4, 1776 );

SimpleDate nextCentury =
     new SimpleDate( 1, 1, 2101 );

SimpleDate defaultDate =
     new SimpleDate( );
```

See Example 3.1 Constructors.java

Objects After Instantiation



Calling a Method



Method Classifications

- Accessor methods
  - `“getter” methods`
  - Return the current values of object data

- Mutator methods
  - `“setter” methods`
  - Change the values of object data

Dot Notation

When calling a method, you need to specify which object the method should use.

Syntax:

```
objectReference.methodName( arg1,
                            arg2, … )
```

Method Return Values

- Can be a primitive data type, class type, or *void*
- Value-returning method
  - The method returns a value.
  - Thus the method call is used in an expression.
  - When the expression is evaluated, the return value of the method replaces the method call.
- Methods with a *void* return type:
  - Do not return a value
  - Method call is a complete statement (ends with ;).

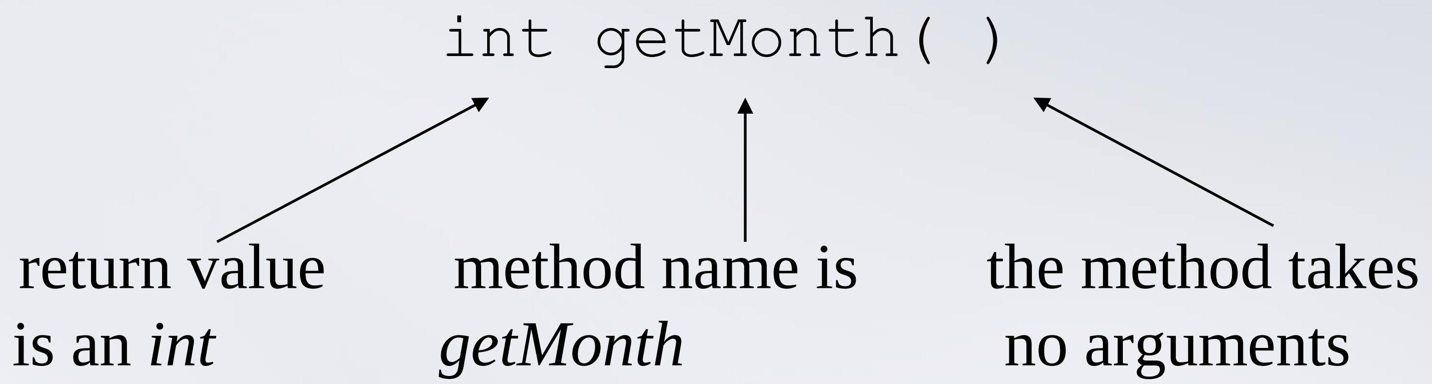
The Argument List in an API

- Comma-separated pairs of

```
dataType variableName
```

- The argument list specifies:
  - The order of the arguments
  - The data type of each argument
- Arguments can be:
  - Any expression that evaluates to the specified data type

Reading an API: Value-Returning Method



```
int getMonth( )
```

Example method calls (assuming *d1* is a *SimpleDate* object):

```
int m = d1.getMonth( );
```

or

```
System.out.println( d1.getMonth( ) );
```

Reading an API: *void* Method

```
void setMonth( int newMonth )
```

method does not return a value

method name is *setMonth*

the method takes one argument: an *int*

Example method call (assuming *d1* is a *SimpleDate* object):

```
d1.setMonth( 12 );
```

See Example 3.2 Methods.java

Common Error Trap

When calling a method, include only expressions in your argument list. Including data types in your argument list will cause a compiler error.

If the method takes no arguments, remember to include the empty parentheses after the method's name. The parentheses are required even if there are no arguments.

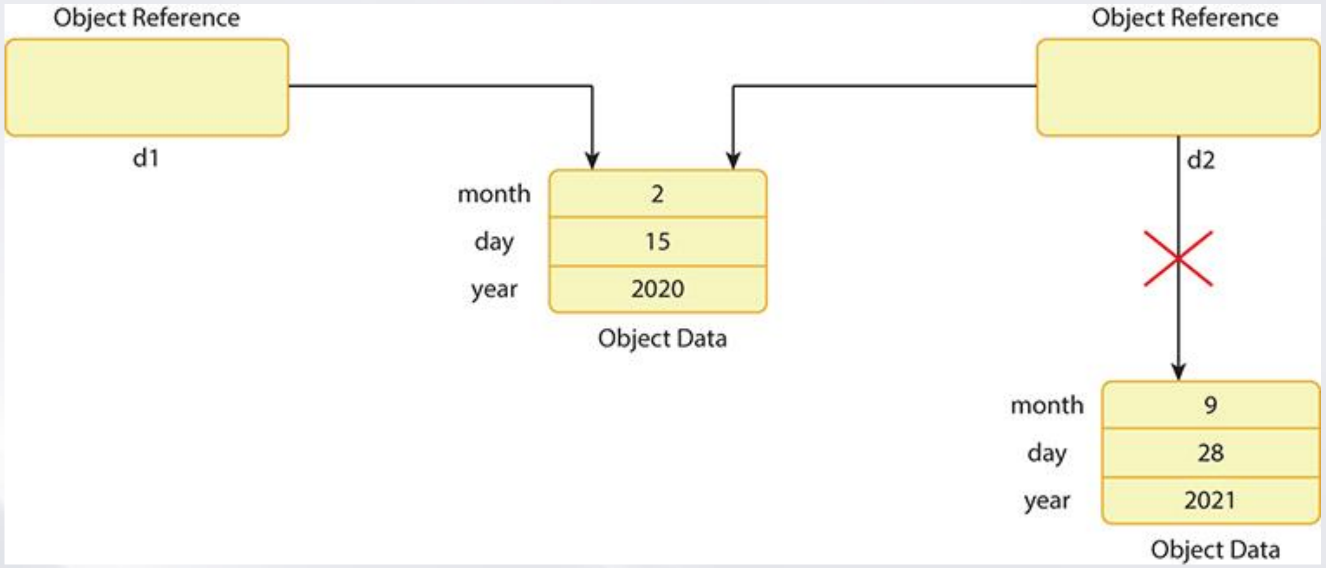
Object Reference vs. Object Data

- Object references point to the memory location of object data.
- An object can have multiple object references pointing to it.
- Or, an object can have no object references pointing to it. If so, the garbage collector will free the object's memory

See Example 3.3 ObjectReferenceAssignment.java

Two References to an Object

After Example 3.3 runs, two object references point to the same object.



null Object References

- An object reference can point to no object. In that case, the object reference has the value *null*.
- Object references have the value *null* when they have been declared but have not been used to instantiate an object.
- Attempting to use a *null* object reference causes a *NullPointerException* at run time.

See Example 3.4 NullReference.java and *Example 3.5 NullReference2.java*

Using Java Predefined Classes

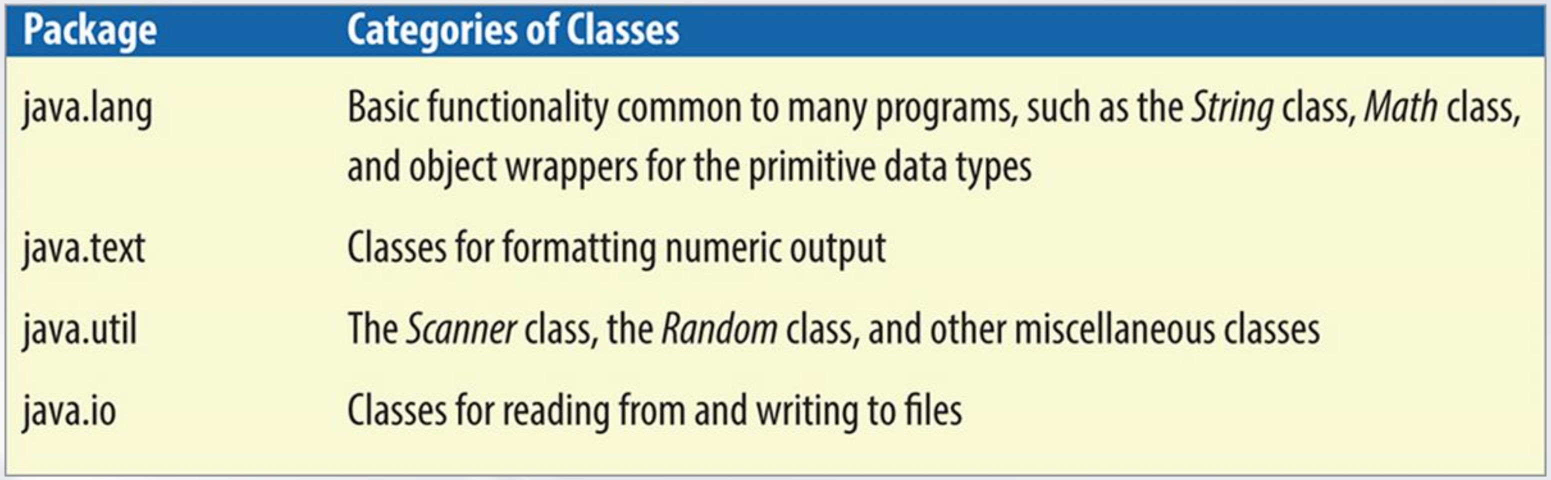
- Java Packages
- The *String* Class
- Formatting Output Using *DecimalFormat*
- The *Random* Class
- Console Input Using the *Scanner* Class
- Using *Static* Methods and Data
- Using *System* and the *PrintStream* Class
- The *Math* Class
- Formatting Output Using *NumberFormat*
- The Wrapper Classes

Java Predefined Classes

- The Java SDK provides more than 2,000 classes that can be used to add functionality to our programs.
- APIs for Java classes are published on Oracle's website: http://www.oracle.com/technetwork/java/

Java Packages

Classes are grouped in packages according to functionality.



Package	Categories of Classes
java.lang	Basic functionality common to many programs, such as the *String* class, *Math* class, and object wrappers for the primitive data types
java.text	Classes for formatting numeric output
java.util	The *Scanner* class, the *Random* class, and other miscellaneous classes
java.io	Classes for reading from and writing to files

Using a Class From a Package

- Classes in *java.lang* are automatically available.
- Classes in other packages need to be "imported" using this syntax:

```
import package.ClassName;
```

or

```
import package.*;
```

(imports all required classes from package)

- Example:

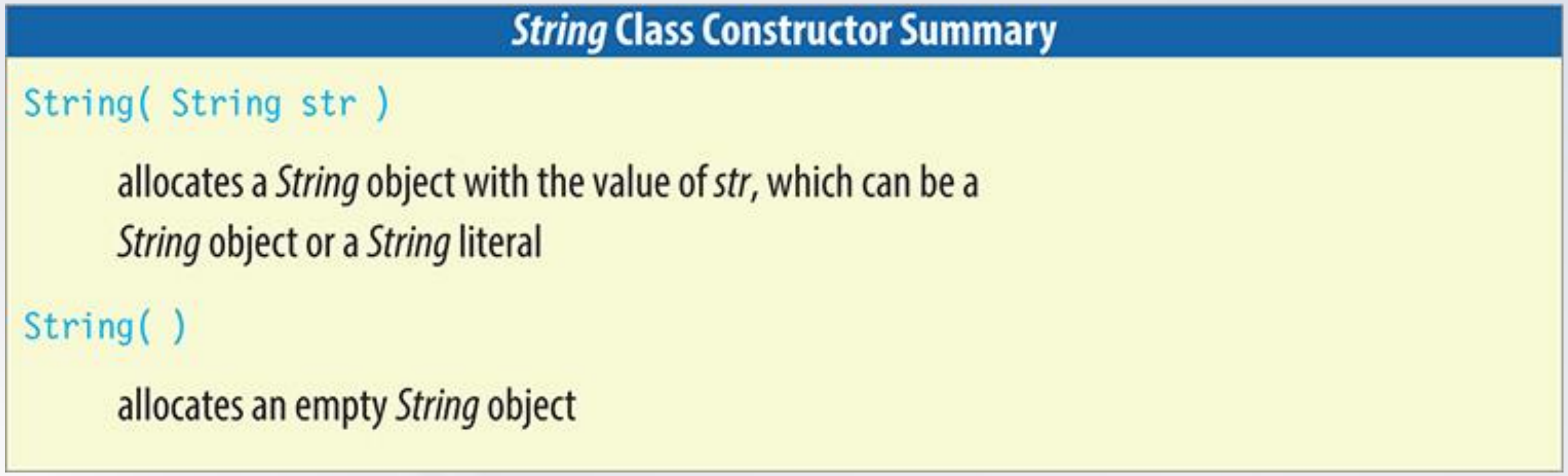
```
import java.text.DecimalFormat;
```

or

```
import java.text.*;
```

The *String* Class

Represents a sequence of characters



String Class Constructor Summary
`String( String str )`
allocates a *String* object with the value of *str*, which can be a *String* object or a *String* literal
`String( )`
allocates an empty *String* object

Examples:

```
String greeting = new String( "Hello" );
String empty = new String( );
```

String Concatenation Operators

+ appends a *String* to another *String*. At least one operand must be a *String*.

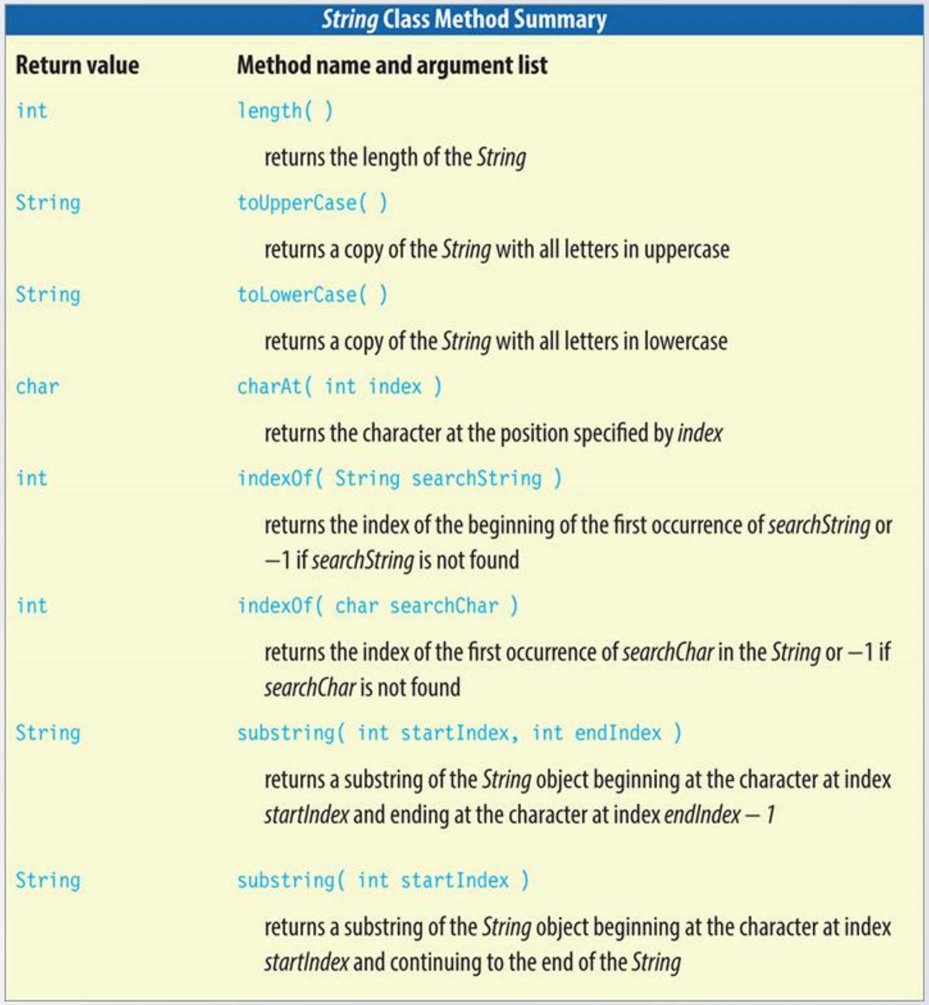
+= shortcut *String* concatenation operator

Example:

```
String s1 = new String( "Hello " );
String s2 = "there";
String s3 = s1 + s2; // s3 is:
                     // Hello there
String s3 += "!";    // s3 is now:
                     // Hello there!
```

The *String* Class

Useful Methods of the String Class



String Class Method Summary

Return value	Method name and argument list
`int`	`length( )`
	returns the length of the *String*
`String`	`toUpperCase( )`
	returns a copy of the *String* with all letters in uppercase
`String`	`toLowerCase( )`
	returns a copy of the *String* with all letters in lowercase
`char`	`charAt( int index )`
	returns the character at the position specified by *index*
`int`	`indexOf( String searchString )`
	returns the index of the beginning of the first occurrence of *searchString* or −1 if *searchString* is not found
`int`	`indexOf( char searchChar )`
	returns the index of the first occurrence of *searchChar* in the *String* or −1 if *searchChar* is not found
`String`	`substring( int startIndex, int endIndex )`
	returns a substring of the *String* object beginning at the character at index *startIndex* and ending at the character at index *endIndex − 1*
`String`	`substring( int startIndex )`
	returns a substring of the *String* object beginning at the character at index *startIndex* and continuing to the end of the *String*

The *length* Method

Example:

```
String greeting = "Hello";
int len = greeting.length( );
```

The value of *len* is 5.

toUpperCase and *toLowerCase* Methods

Example:

```
String greeting = "Hello";
greeting = greeting.toUpperCase( );
```

The value of *greeting* is "HELLO"

The *indexOf* Methods

The index of the first character of a *String* is 0.

Example:

```
String greeting = "Hello";
int index = greeting.indexOf( 'e' );
```

The value of *index* is 1.

The *charAt* Method

Example:

```
String greeting = "Hello";
char firstChar = greeting.charAt( 0 );
```

The value of *firstChar* is 'H'

The *substring* Method

Example:

```
String greeting = "Hello";
String endOfHello = greeting.substring( 3,
                               hello.length( ) );
```

The value of *endOfHello* is "lo"

Another *substring* Method

Example:

```
String greeting = "Hello";
String endOfHello = greeting.substring( 3 );
```

The value of *endOfHello* is “lo”

See Example 3.6 StringDemo.java

Common Error Trap

Specifying a negative start index or a start index past the last character of the *String* will generate a
StringIndexOutOfBoundsException

Specifying a negative end index or an end index greater than the length of the *String* also will generate a
StringIndexOutOfBoundsException

More *String* Processing (1 of 3)

Often, parsing a *String* means using several of the *String* methods together.

If a *String* has the format

```
<lastname>, <firstname>
```

how do we retrieve the first name (and the last name)?

Example: the *String* could be: `lincoln, abraham`

We want the create a *String* in this format:

```
    Abraham Lincoln
```

More *String* Processing (2 of 3)

We know that the last name is followed by a comma, a space, and the first name.

We can use the *indexOf* method to retrieve the position of the comma:

```
String invertedName = "lincoln, abraham";
int comma = invertedName.indexOf( ',' );
```

Now we can use the *substring* method to retrieve the last name:

```
String lastName
  = invertedName.substring( 0, comma );
```

More *String* Processing (3 of 3)

The substring between index 0 (included) and the index of the comma (not included) contains the last name.

```
String lastName
   = invertedName.substring( 0, comma );
```

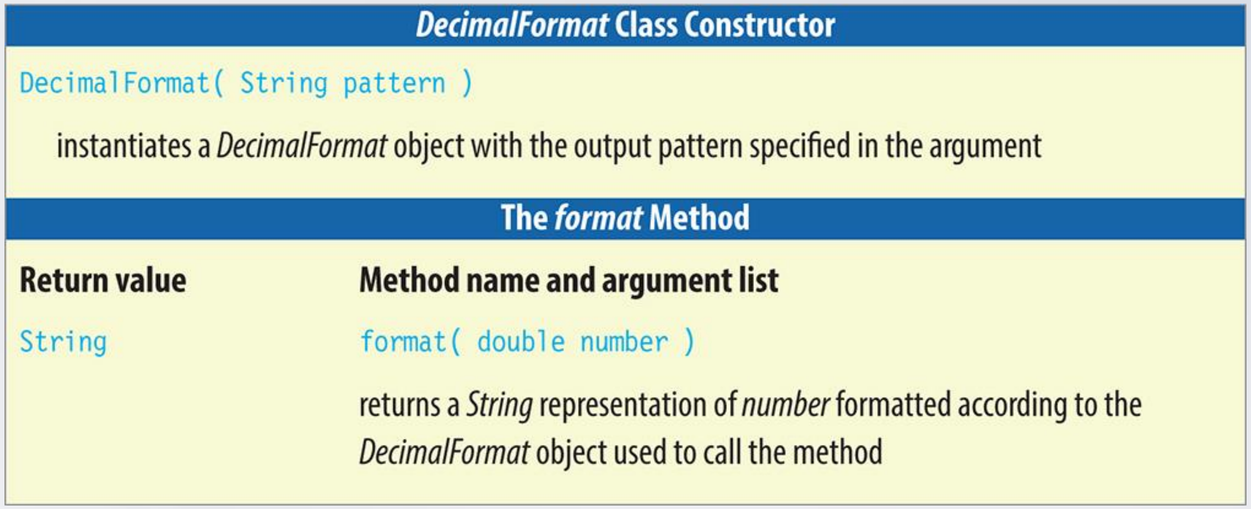
See Example 3.7 StringProcessing.java

Formatting Numeric Output

The *DecimalFormat* class and the *NumberFormat* class allow you to specify the number of digits for printing and to add dollar signs and percent signs to your output.

- Both classes are in the *java.text* package.
- We will cover the *NumberFormat* class later.

The *DecimalFormat* Class



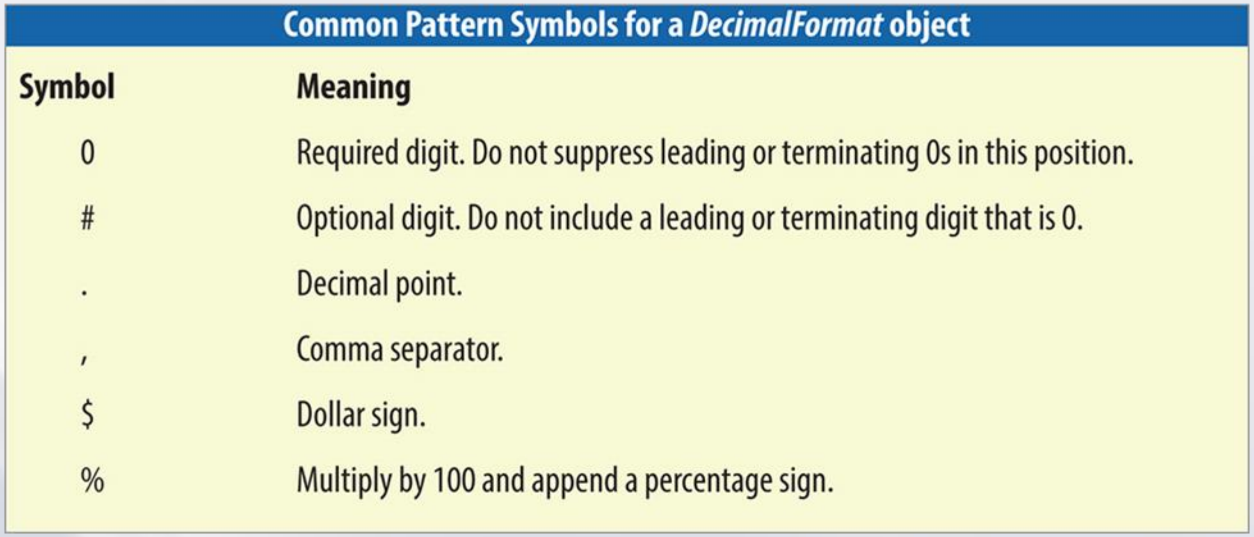
DecimalFormat Class Constructor

```
DecimalFormat( String pattern )
```

instantiates a *DecimalFormat* object with the output pattern specified in the argument

The _format_ Method

Return value	Method name and argument list
`String`	`format( double number )` returns a *String* representation of *number* formatted according to the *DecimalFormat* object used to call the method



Common Pattern Symbols for a _DecimalFormat_ object

Symbol	Meaning
0	Required digit. Do not suppress leading or terminating 0s in this position.
#	Optional digit. Do not include a leading or terminating digit that is 0.
.	Decimal point.
,	Comma separator.
$	Dollar sign.
%	Multiply by 100 and append a percentage sign.

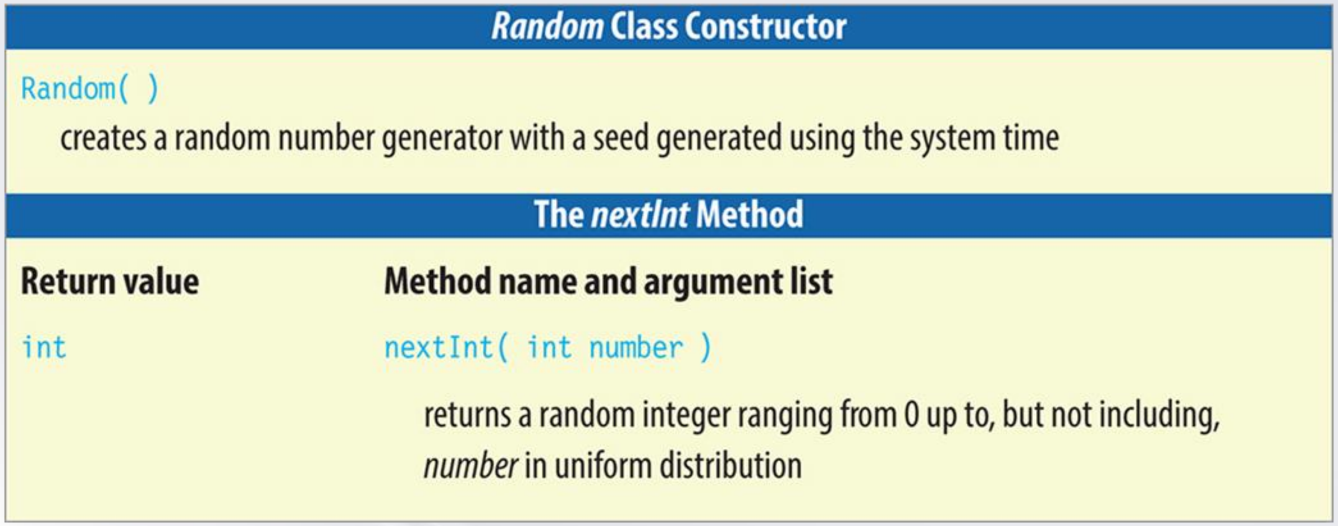
The *DecimalFormat format* Method

- To format a numeric value for output, call the *format* method.

See Example 3.8 DemoDecimalFormat.java

The *Random* Class

Generates a pseudorandom number (appearing to be random, but mathematically calculated)



***Random* Class Constructor**

`Random( )`

creates a random number generator with a seed generated using the system time

The *nextInt* Method

Return value	Method name and argument list
`int`	`nextInt( int number )`
	returns a random integer ranging from 0 up to, but not including, *number* in uniform distribution

The *Random* class is in the *java.util* package.

Example:

```
Random rand = new Random( );
```

The *nextInt* Method

Example:

to generate a random integer between 1 and 6:

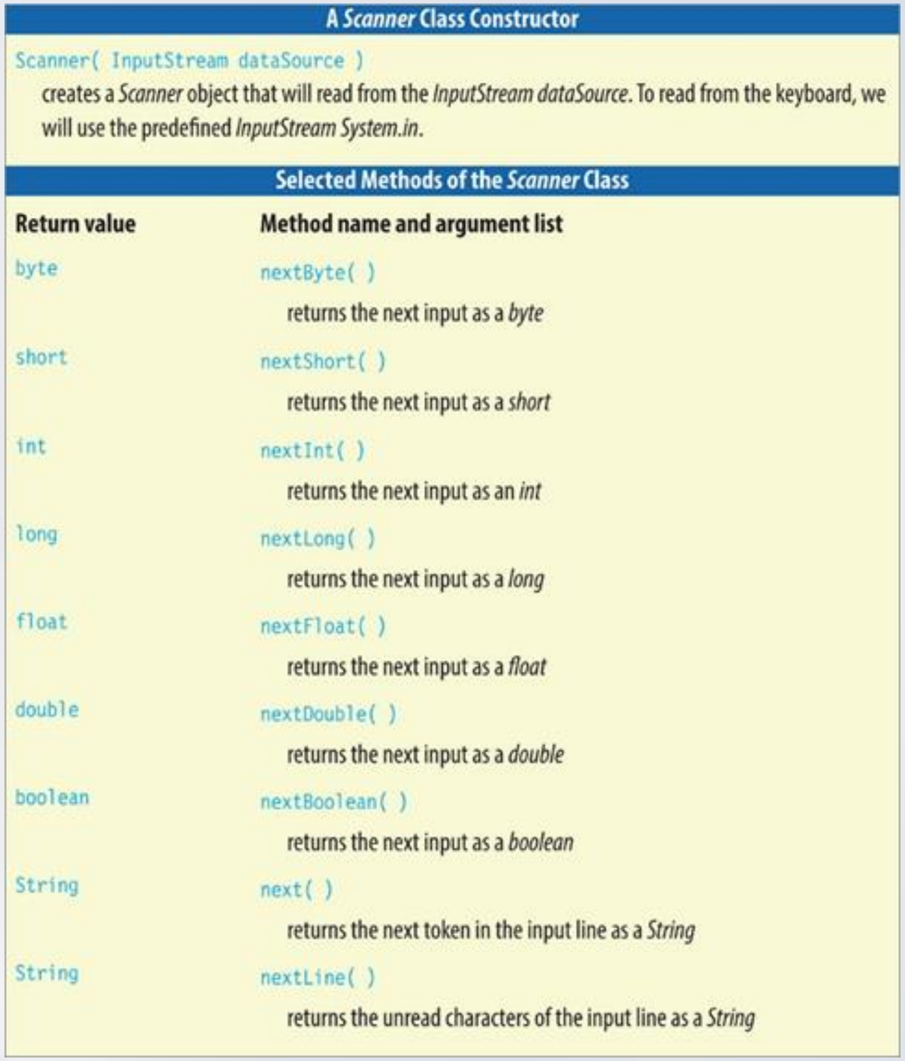
```
int die = rand.nextInt( 6 ) + 1;
```

See Example 3.9 RandomNumbers.java

Input Using the *Scanner* Class

- Provides methods for reading *byte*, *short*, *int*, *long*, *float*, *double*, and *String* data types from the Java console and other sources
- *Scanner* is in the *java.util* package.
- Scanner parses (separates) input into sequences of characters called tokens.
- By default, tokens are separated by standard white space characters (tab, space, *newline*, etc.)

A *Scanner* Constructor & *next*… Methods



A *Scanner* Class Constructor

`Scanner( InputStream dataSource )`

creates a *Scanner* object that will read from the *InputStream dataSource*. To read from the keyboard, we will use the predefined *InputStream System.in*.

Selected Methods of the *Scanner* Class

Return value	Method name and argument list
`byte`	`nextByte( )` returns the next input as a *byte*
`short`	`nextShort( )` returns the next input as a *short*
`int`	`nextInt( )` returns the next input as an *int*
`long`	`nextLong( )` returns the next input as a *long*
`float`	`nextFloat( )` returns the next input as a *float*
`double`	`nextDouble( )` returns the next input as a *double*
`boolean`	`nextBoolean( )` returns the next input as a *boolean*
`String`	`next( )` returns the next token in the input line as a *String*
`String`	`nextLine( )` returns the unread characters of the input line as a *String*

Prompting the User

- The *next*… methods do not prompt the user for an input value.
- Use *System.out.print* to print the prompt, then call the *next*… method.

Example:

```
Scanner scan = new Scanner( System.in );
System.out.print( "Enter your age > " );
int age = scan.nextInt( );
```

See Example 3.10 DataInput.java

SOFTWARE ENGINEERING TIP

End your prompts with an indication that input is expected.

Include a trailing space for readability.

Example:

```
System.out.print( "Enter your age > " );
```

SOFTWARE ENGINEERING TIP

Provide the user with clear prompts for input.

Prompts should use words the user understands and should describe the data requested as well as any restrictions on valid input values.

Example:

```
   Enter your first and last name >
```

or

```
  Enter an integer between 0 and 10 >
```

Character Input

- *Scanner* does not have a *nextChar* method.
- To read a single character, read the input as a *String*, then extract the first character of the *String* into a *char* variable.

Example:

```
System.out.print( "Enter your " +
                  "middle initial > " );
String initialS = scan.next( );
char initial = initialS.charAt( 0 );
```

See Example 3.11 CharacterInput.java

Reading a Whole Line

- The *next* method will read only one word of *String* input because the space is a white space character.
- To read a whole line, including spaces, use the *nextLine* method.

Example:

```
System.out.print( "Enter a sentence > " );
String sentence = scan.nextLine( );
```

See Example 3.12 InputALine.java

static Methods

- Also called class methods.
- Can be called without instantiating an object
- Might provide some quick, one-time functionality—for example, popping up a dialog box or executing a math function
- In the method API, the keyword *static* precedes the return type.

Calling *static* Methods

- Use the dot syntax with the class name instead of the object reference

Syntax:

```
ClassName.methodName( argument list )
```

Example:

```
int absValue = Math.abs( -9 );
```

abs is a *static* method of the *Math* class that returns the absolute value of the argument. (Here, the argument is -9, and the value 9 would be returned from the method and assigned to the variable *absValue*.)

static Class Data

- Syntax:

```
ClassName.staticDataName
```

- Example:

```
Color.BLUE
```

BLUE is a *static* constant of the *Color* class.

The *System* Class

- The *System* class is in the *java.lang* package, so it does not need to be imported.
- *static* constants of the *System* class

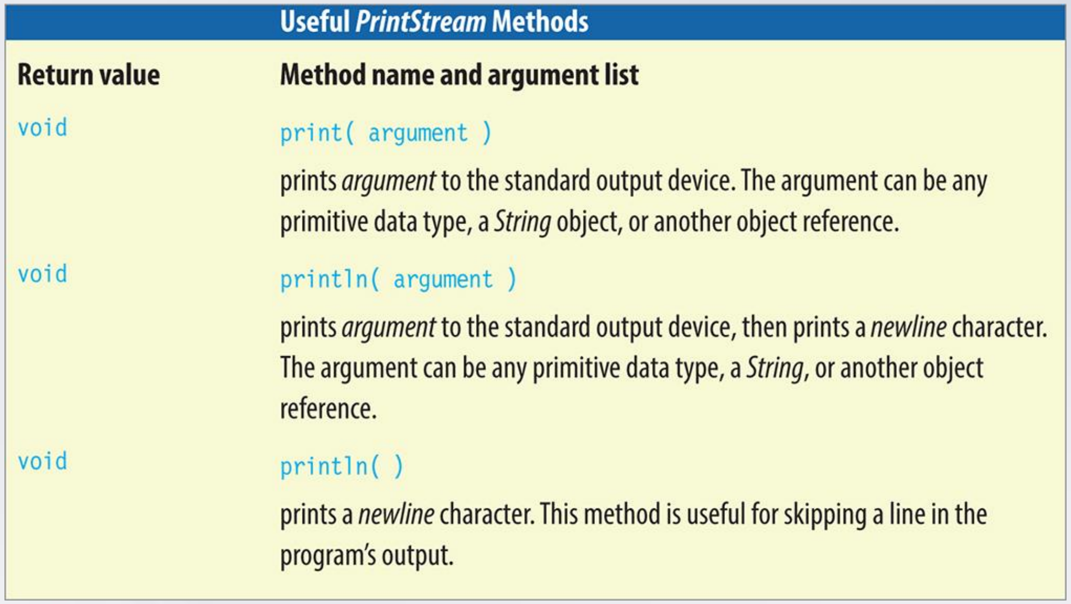
  in represents the standard input device (the keyboard by default). *in* is an object of the *InputStream* class.

  out represents the standard output device (the Java console by default). *out* is an object of the *PrintStream* class.

Examples:

```
Scanner scan = new Scanner( System.in );
System.out.println( "Hello" );
```

Using *System.out*



Useful *PrintStream* Methods

Return value	**Method name and argument list**
`void`	`print( argument )` prints *argument* to the standard output device. The argument can be any primitive data type, a *String* object, or another object reference.
`void`	`println( argument )` prints *argument* to the standard output device, then prints a *newline* character. The argument can be any primitive data type, a *String*, or another object reference.
`void`	`println( )` prints a *newline* character. This method is useful for skipping a line in the program's output.

Example:

```
System.out.print( "The answer is " );
System.out.println( 3 );
```

The output is:

```
The answer is 3
```

The *exit* Method

The *static exit* method can be useful to stop execution at a place other than the normal end of the program.

The *exit* method's API is:

```
void exit( int exitStatus )
```

Example:

```
System.exit( 0 );
```

The *toString* Method

All classes have a *toString* method. Its purpose is to return a *String* representing the data of the object.

The API of the *toString* method is:

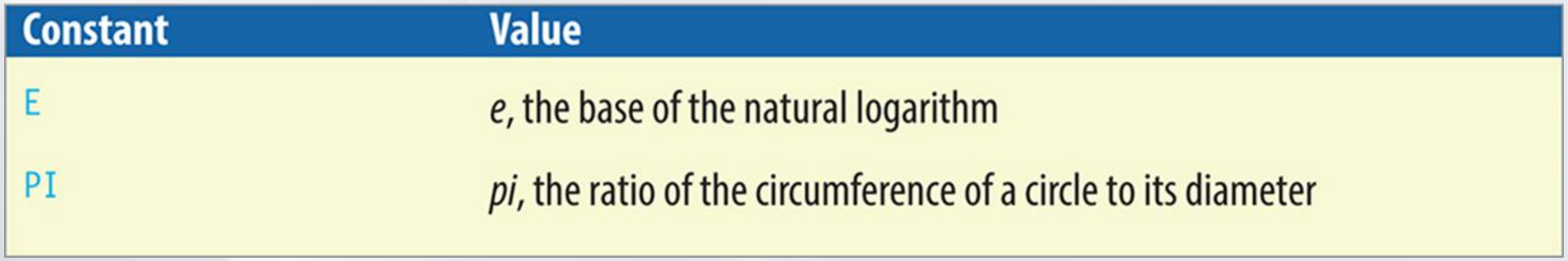
```
String toString( )
```

The *toString* method is called automatically (implicitly) when an object reference is used with the *print* or *println* methods.

See Example 3.13 PrintDemo.java

The *Math* Class

- The *Math* class provides *static* constants and *static* methods for performing common calculations.
- The *Math* class is in the *java.lang* package, so it does not need to be imported.



Constant	Value
`E`	*e*, the base of the natural logarithm
`PI`	*pi*, the ratio of the circumference of a circle to its diameter

The *Math* Class Constants

Two *static* constants

PI – the value of pi

E – the base of the natural logarithm

Example:

```
System.out.println( Math.PI );
System.out.println( Math.E );
```

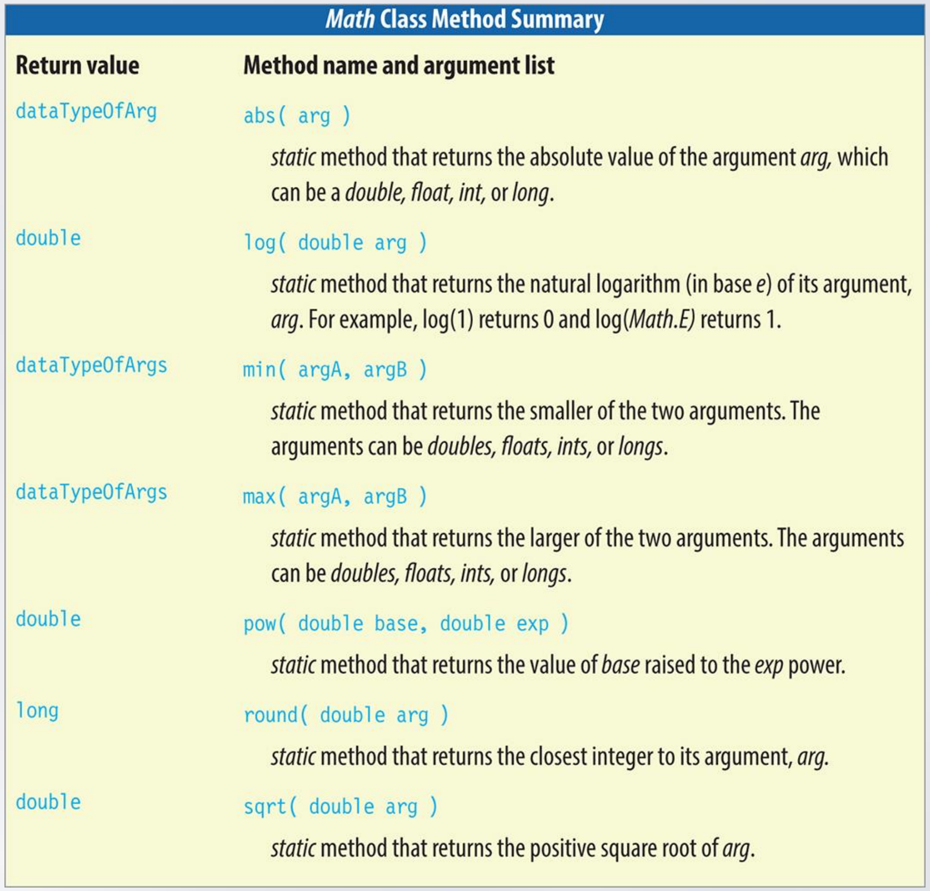
The output is:

```
3.141592653589793
2.718281828459045
```

See Example 3.14 MathConstants.java

Methods of the *Math* Class

All methods are *static*.



***Math* Class Method Summary**

Return value	Method name and argument list
`dataTypeOfArg`	`abs( arg )` *static* method that returns the absolute value of the argument *arg,* which can be a *double, float, int,* or *long*.
`double`	`log( double arg )` *static* method that returns the natural logarithm (in base *e*) of its argument, *arg*. For example, log(1) returns 0 and log(*Math.E)* returns 1.
`dataTypeOfArgs`	`min( argA, argB )` *static* method that returns the smaller of the two arguments. The arguments can be *doubles, floats, ints,* or *longs*.
`dataTypeOfArgs`	`max( argA, argB )` *static* method that returns the larger of the two arguments. The arguments can be *doubles, floats, ints,* or *longs*.
`double`	`pow( double base, double exp )` *static* method that returns the value of *base* raised to the *exp* power.
`long`	`round( double arg )` *static* method that returns the closest integer to its argument, *arg.*
`double`	`sqrt( double arg )` *static* method that returns the positive square root of *arg*.

See Example 3.15 MathMethods.java

The Math *round* Method

Rounding rules:

- Any factional part < .5 is rounded down.
- Any fractional part .5 and above is rounded up.

See Example 3.16 MathRounding.java

The Math *min/max* Methods

Find the smallest of three numbers:

```
int smaller = Math.min( num1, num2 );
int smallest = Math.min( smaller, num3 );
```

See Example 3.17 MathMinMaxMethods.java

The *NumberFormat* Class (1 of 2)

- The *NumberFormat* class (in the *java.text* package), like the *DecimalFormat* class, can be used to format numeric values for output.
- The *NumberFormat* class provides factory methods for creating currency and percentage objects that use predefined formatting patterns.
- These *static* factory methods are called instead of using instantiating an object using the *new* keyword.

The *NumberFormat* Class (2 of 2)

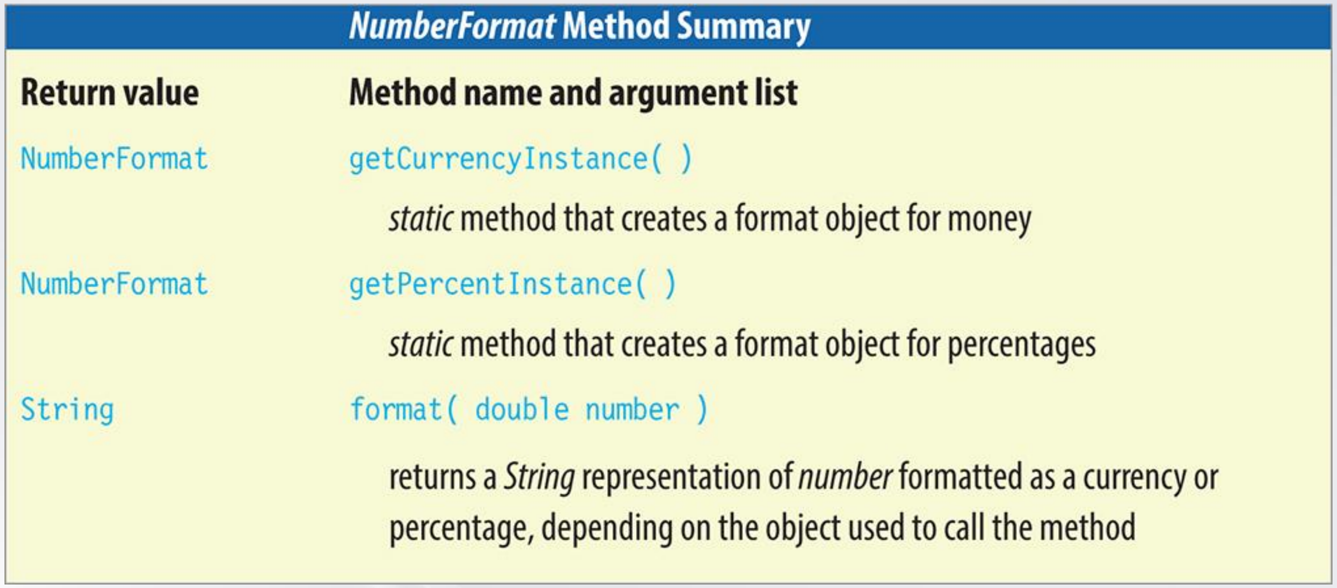


Table: *NumberFormat* Method Summary

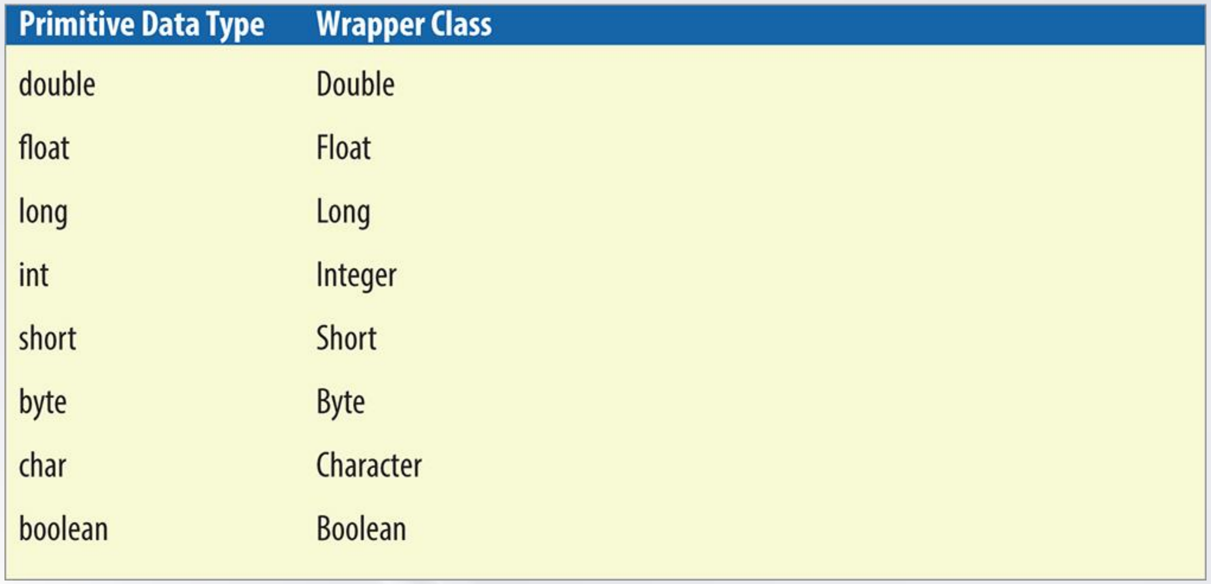
Return value	Method name and argument list
`NumberFormat`	`getCurrencyInstance( )` *static* method that creates a format object for money
`NumberFormat`	`getPercentInstance( )` *static* method that creates a format object for percentages
`String`	`format( double number )` returns a *String* representation of *number* formatted as a currency or percentage, depending on the object used to call the method

See Example 3.18 DemoNumberFormat.java

The Wrapper Classes (1 of 2)

- “Wrap" the value of a primitive data type into an object
- Useful when methods require an object argument
- Also useful for converting *Strings* to an *int* or *double*

The Wrapper Classes (2 of 2)



Primitive Data Type	Wrapper Class
double	Double
float	Float
long	Long
int	Integer
short	Short
byte	Byte
char	Character
boolean	Boolean

Autoboxing and Unboxing

- Autoboxing

  Automatic conversion between a primitive type and a wrapper object when a primitive type is used where an object is expected

```
Integer intObject = 42;
```

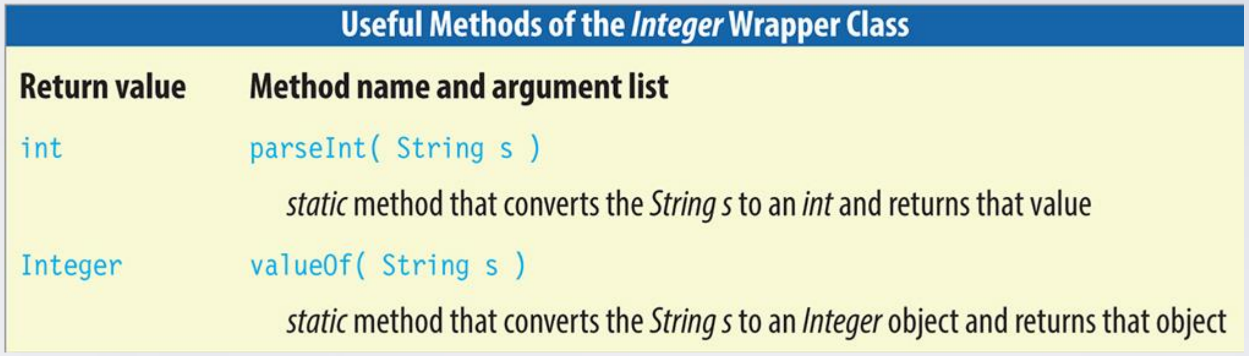
- Unboxing

  Automatic conversion between a wrapper object and a primitive data type when a wrapper object is used where a primitive data type is expected

```
int fortyTwo = intObject;
```

Integer and *Double* Methods

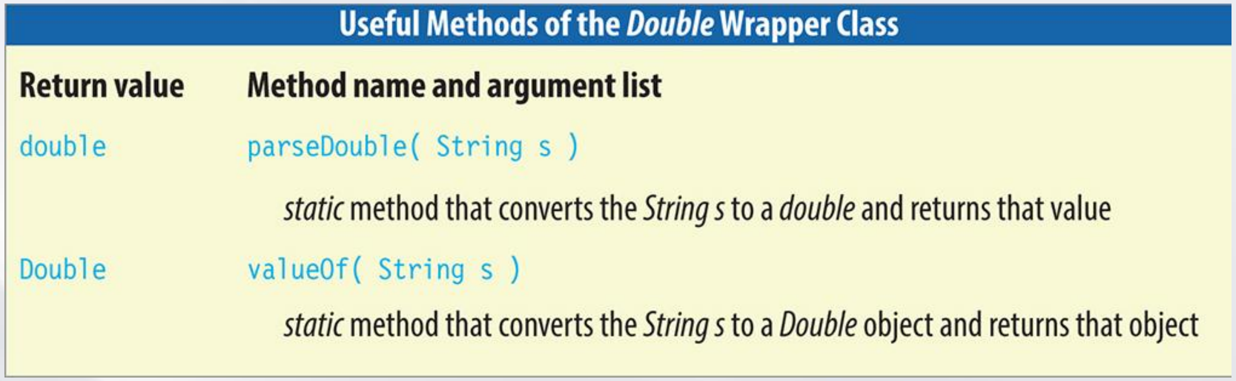
static Integer methods



Useful Methods of the *Integer* Wrapper Class

Return value	Method name and argument list
`int`	`parseInt( String s )`
	static method that converts the *String s* to an *int* and returns that value
`Integer`	`valueOf( String s )`
	static method that converts the *String s* to an *Integer* object and returns that object

static Double methods



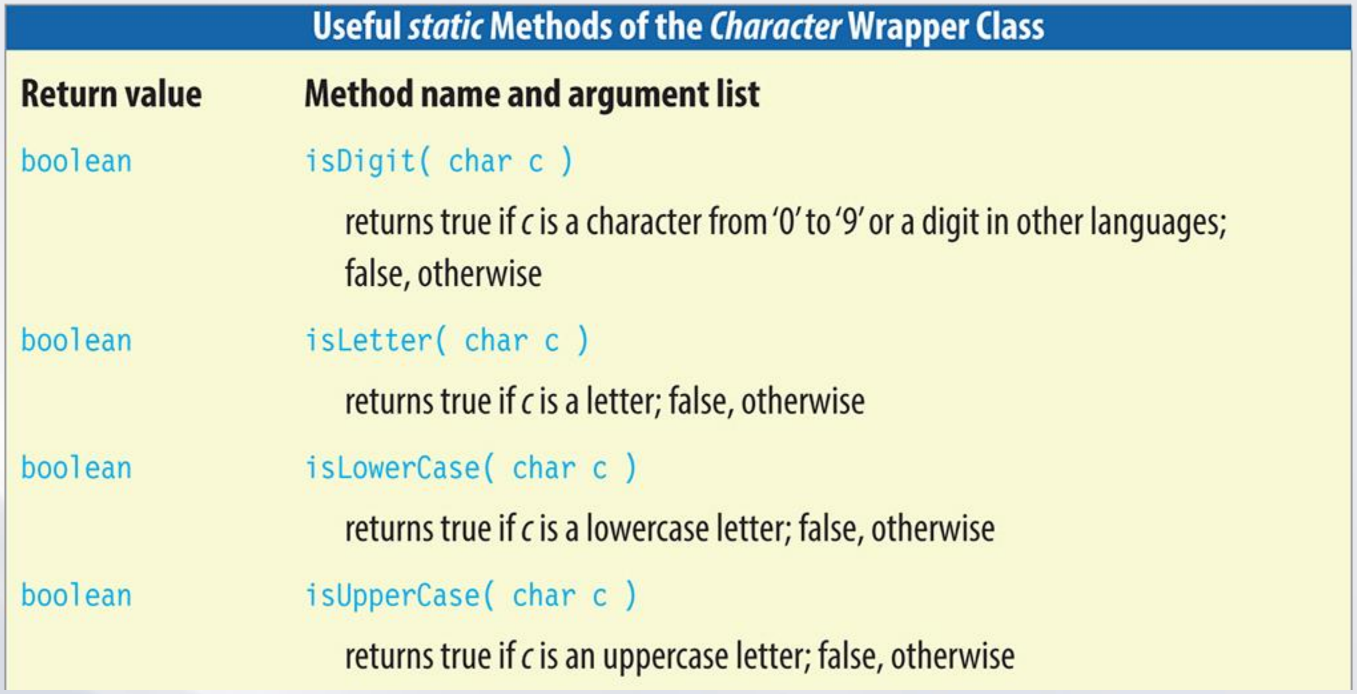
Useful Methods of the *Double* Wrapper Class

Return value	Method name and argument list
`double`	`parseDouble( String s )`
	static method that converts the *String s* to a *double* and returns that value
`Double`	`valueOf( String s )`
	static method that converts the *String s* to a *Double* object and returns that object

See Example 3.19 DemoWrapper.java

Methods of the *Character* Class (1 of 2)

The *Character* class includes methods to test if a single *char* is a digit, a letter, upper case, ..

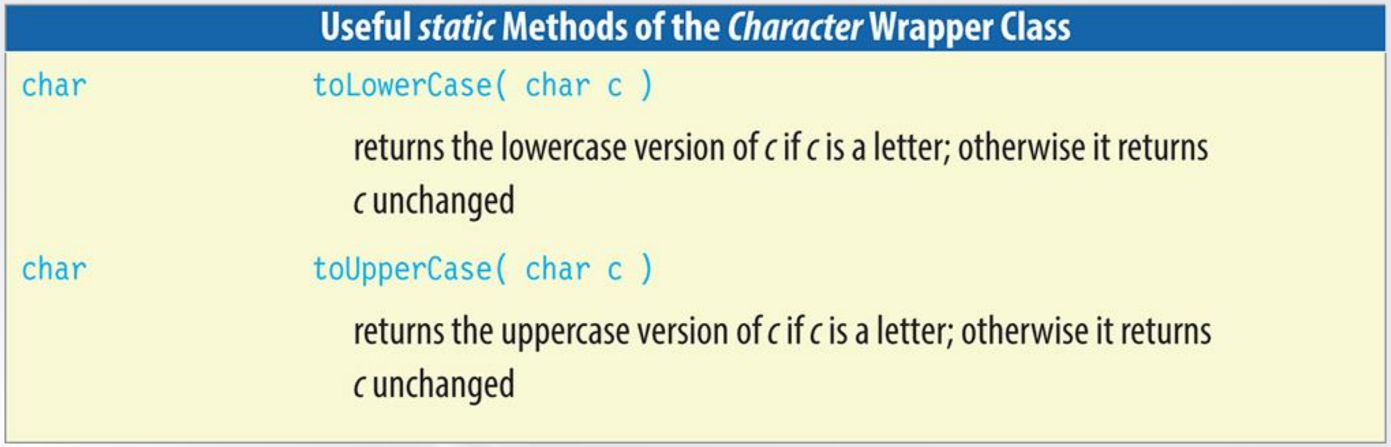


Useful *static* Methods of the *Character* Wrapper Class

Return value	Method name and argument list
`boolean`	`isDigit( char c )`
	returns true if *c* is a character from '0' to '9' or a digit in other languages; false, otherwise
`boolean`	`isLetter( char c )`
	returns true if *c* is a letter; false, otherwise
`boolean`	`isLowerCase( char c )`
	returns true if *c* is a lowercase letter; false, otherwise
`boolean`	`isUpperCase( char c )`
	returns true if *c* is an uppercase letter; false, otherwise

Methods of the *Character* Class (2 of 2)

Other *Character* class methods can convert letters to uppercase or lowercase.



Useful *static* Methods of the *Character* Wrapper Class

`char`	`toLowerCase( char c )`
	returns the lowercase version of *c* if *c* is a letter; otherwise it returns *c* unchanged
`char`	`toUpperCase( char c )`
	returns the uppercase version of *c* if *c* is a letter; otherwise it returns *c* unchanged

See Example 3.20 CharacterMethods.java