Classes and Objects Miscellany: I/O, Statics, Wrappers & Packages

CMSC 202H (Honors Section)

John Park

Basic Input/Output

Printing to the Screen

In addition to `System.out.print()` (and `println()`):

- Formatted output
  - `System.out.printf("Printing integer %d%n",5);`
  - `System.out.printf("%d %c %d", 1, 'a', 2);`
- Place holders can be added to represent variables to be output in the format string.
  - %d, %c, %f, %s – What does each stand for?
  - Every place holder that appears inside the output string must have a matching value separated by a comma.
- Add proceeding white space characters and precision to variables printed.
  - `System.out.printf("2 points of precision %10.2d", 89.999);`
  - "Two points of precision ____90.00" ← no newline character
- Other special formatting
  - %n – platform independent newline character
  - \t – horizontal tab

Reading From the Console

- Java’s ***Scanner object*** reads in input that the user enters on the command line.

```
Scanner input = new Scanner(System.in);
```

- System.in is a reference to the ***standard input buffer.***
- We can read values from the Scanner object using the dot notation to invoke a number of functions.
  - nextInt() — returns the next integer from the buffer
  - nextFloat() — returns the next float from the buffer
  - nextLine() — returns the entire line as a String

Scanner Notes

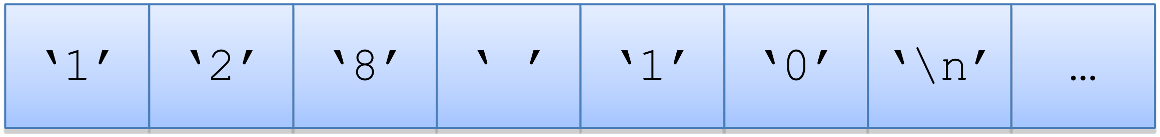
- In order to use the Scanner class, you’ll need to add the following line to the top of your code…

```
import java.util.Scanner;
```

- You should ***never*** declare more than one Scanner object on a given input stream.
- The Scanner object will wait for a user to type, and read all text entered up until the user presses the “enter” key (including the newline character).

Reading from the Console

```
System.out.print("Enter 2 numbers to sum: ");
Scanner input = new Scanner(System.in);
int n1 = input.nextInt();
int n2 = input.nextInt();
System.out.printf("%d + %d = %d", n1, n2, n1 + n2);
```



‘1’	‘2’	‘8’	‘ ’	‘1’	‘0’	‘\n’	…

- Let’s assume the user has entered “128 10” .
- The first call to nextInt() reads the characters “128” leaving “ 10\n” in the input buffer.
- The second call to nextInt() reads the “10” and leaves the “\n” in the buffer.

Reading via UNIX Redirection

```java
int sum = 0;
Scanner input = new Scanner(System.in);
while(input.hasNextInt()) {
        sum += input.nextInt();
}
System.out.println("Sum: " + sum);
```

```
% cat numbers
1 2 3
4
5 6 7
8
% java Sum < numbers
Sum: 36
%
```

- The Scanner class also has a bunch of hasNextX() methods to detect if there’s another data item of the given type in the stream.
- For example, this is useful if we were reading an unknown quantity of integers from a file that is redirected into our program (as above).

Packages, Compilation and Execution

Packages

- Java allows you to partition your classes into sets and subsets, called *packages.*
- You place your class into a package with the directive:

```
package myPackage;
```

- If the “package” directive is missing, the class is placed into the *unnamed package*
- A Java package is similar to a “namespace”: it implicitly prepends a prefix of your choice to all classes you define.

Packages

- You can refer to all objects via its fully-qualified name, e.g.:

```
myPackage.MyClass foo = new myPackage.MyClass();
```

- Within a class definition, class references without explicit package name prefixes refer to other classes in your package
  - This is modified by importing other packages
- In addition to its use for namespaces, packages affect the function of some *visibility modifiers* (later)

Importing Packages

- Import single *class* by using:

```
import java.util.Random;
```

- Or, import many classes, with wildcard:

```
import java.util.*;
```

  - Cannot “`import java.*.*;`”
  - Importing is not recursive (e.g. java.* != java.util.*)
  - Importing singly is preferred (why?)
- java.lang.* is already implicitly imported
- However, all other java.*… must be explicitly imported

Package Naming Conventions

- Initially, beginners use the *unnamed package*
- For simple, standalone applications, use simple one-token package names, e.g.: “proj1” (note lowercase)
- For packages to be deployed outside the organization, use inverse-domain-address-like notation, e.g.:

  edu.umbc.csee.cmsc202.utilityPackage

Packages: Example

```
package proj3;
import java.util.Random;
public class MyClass {
     // Stuff inside this class definition
     public static int someMethod() {
            Random rand = new Random();
            …
     }
}
---------------------------------------------------------

// No “package” directive, so in unnamed package
// No “import” directive, so all class names must be full
public class MyOtherClass {
     // Stuff inside this class definition
     public static int someMethod() {
            proj3.MyClass myClassInst = new proj3.MyClass();
            java.util.Random rand = new java.util.Random();
            …
     }
}
```

Java Program Review

```java
package demos;

public class SimpleProgram {
  public static void main (String[] args){
    System.out.println("Hello World");
  }
}
```

```java
package demos;

public class OtherProgram {
  public static void main (String[] args){
    System.out.println("Hello World 2");
  }
}
```

- Java source code can be compiled under any operating system.
  - javac -d . SimpleProgram.java
  - javac -d . OtherProgram.java
- Java will create a directory named *demos* containing
  - SimpleProgram.class
  - OtherProgram.class
- We can execute SimpleProgram with the following.
  - java demos.SimpleProgram
- We can execute OtherProgram with the following.
  - Java demos.OtherProgram
- We can execute any class’ main in a similar manner.
  - java <package name>.<Class name>

Command Line Arguments

```java
package demos;

public class ArgsDemo {
        public static void main (String[] args){
                for(int i = 0; i < args.length; i++){
                        System.out.println(args[i]);
                }
        }
}
```

- Anything that follows the name of the main class to be executed will be read as a ***command line argument***.
- All text entered will be stored in the String array specified in main (typically ***args*** by convention).
  - java demos.ArgsDemo Hi
  - Results in “Hi” stored at args[0]
- Individual arguments can be separated by spaces like so
  - java demos.ArgsDemo foo 123 bar
  - Results in “foo” stored at args[0], “123” at args[1] and “bar” at args[2]

What Does “Static” Mean?

The Problem of Words

"When I use a word," Humpty Dumpty said in rather a scornful tone, "it means just what I choose it to mean -- neither more nor less."
"The question is," said Alice, "whether you can make words mean so many different things."
"The question is," said Humpty Dumpty, "which is to be master - - that's all."
Lewis Carroll, *Through the Looking Glass*

- So, what do `static` (and `final`) mean in Java?
  - `public static final float PI = 3.14159;`
  - `public static int timesCreated;`
  - `public static void main(String[] args) {…}`
- …and *why* do they mean ***that***?!

History of `static`

- In C, originally needed a way to let a variable keep its value unchanged across calls, i.e., keep it “static”
- Extended scope to repurpose `static` keyword for file-scope global variables
- Java repurposed the word multiple times again, in an OOP context
- Humpty Dumpty would have loved `static`

What Does “static” Mean in Java?

- Instance variables, constants, and methods may all be labeled as `static`.
- In this context, static means that there is one copy of the variable, constant, or method that belongs to the class as a whole, and not to a particular instance.
- It is not necessary to instantiate an object to access a static variable, constant or method.

Static Variables

- A ***static variable*** belongs to the class as a whole, not just to one object.
- There is only one copy of a static variable per class.
- All objects of the class can read and change this static variable.
- A static variable is declared with the addition of the modifier **`static`**.

```
    static int myStaticVariable;
```

- Static variables can be declared and initialized at the same time.

```
    static int myStaticVariable = 0;
```

Static Constants

- A ***static constant*** is used to symbolically represent a constant value.
- In some languages (e.g., C) constants are simply implemented as macros, used to replace text.
- In Java, constants derive from regular variables, by "finalizing" them
  - The declaration for a static defined constant must include the modifier `final`, which indicates that its value cannot be changed.

```
public static final int BIRTH_YEAR = 1954;
```

(The modifier `final` is also overloaded, and means other things in other contexts, as we shall see later.)

- Static constants belong to the class as a whole, not to each object, so there is only one copy of a static constant

- When referring to such a defined constant outside its class, use the name of its class in place of a calling object.

```
int year = MyClass.BIRTH_YEAR;
```

Static Methods

So far,

- class methods required a calling object in order to be invoked.

```
Date birthday = new Date(1, 23, 1982);
String s = birthday.toString( );
```

- These are sometimes known as ***non-static methods***.

Static methods:

- still belong to a class, but need no calling object, and
- often provide some sort of utility function.

monthString Method

Recall the Date class private helper method monthString.

- Translates an integer month to a string
- Note that the monthString method
  - Does not call any other methods of the Date class, and
  - Does not use any instance variables (month, day, year) from the Date class.

- This method can be made available to users of the Date class without requiring them to create a Date object.

```
public static String monthString( int monthNumber ) {
     switch ( monthNumber )   {
              case 1:   return "January";
              case 2:   return "February";
              case 3:   return "March";
              case 4:   return "April";
              case 5:   return "May";
              case 6:   return "June";
              case 7:   return "July";
              case 8:   return "August";
               case 9:   return "September";
              case 10: return "October";
              case 11: return "November";
              case 12: return "December";
              default: return “????”;
     }
}
```

It is now a public static method.

monthString Demo

- Code outside of the Date class can now use the monthString method without creating a Date object.
- Prefix the method name with the name of the class instead of an object.

Date is a class name, not an object name.

monthString is the name of a static method

```
class MonthStringDemo
{
   public static void main( String [ ] args )
   {
         String month = Date.monthString( 6 );
         System.out.println( month );
   }
}
```

Rules for Static Methods

- Static methods have no calling/host object (they have no `this`).
- Therefore, static methods cannot:
  - Refer to any instance variables of the class
  - Invoke any method that has an implicit or explicit `this` for a calling object
- Static methods may invoke other static methods or refer to static variables and constants.
- A class definition may contain both static methods and non-static methods.

Static F^{o} to C^{o} Convert Example

```
public class FtoC
{
   public static double convert( double degreesF )
        { return 5.0 / 9.0 * (degreesF – 32 ); }
}
```

```
public class F2CDemo
{
   public static void main( String[ ] args )
   {
        double degreesF = 100;

        // Since convert is static, no object is needed
        // The class name is used when convert is called

        double degreesC = FtoC.convert( degreesF );
        System.out.println( degreesC );
   }
}
```

main is a Static Method

Note that the method header for main( ) is

```
public static void main(String [] args)
```

Being static has two effects:

- main can be executed without an object.
- “Helper” methods called by main must also be static.

Any Class Can Have a main( )

- Every class can have a public static method name main( ).
- Java will execute main in whichever class is specified on the command line.

```
java <className>
```

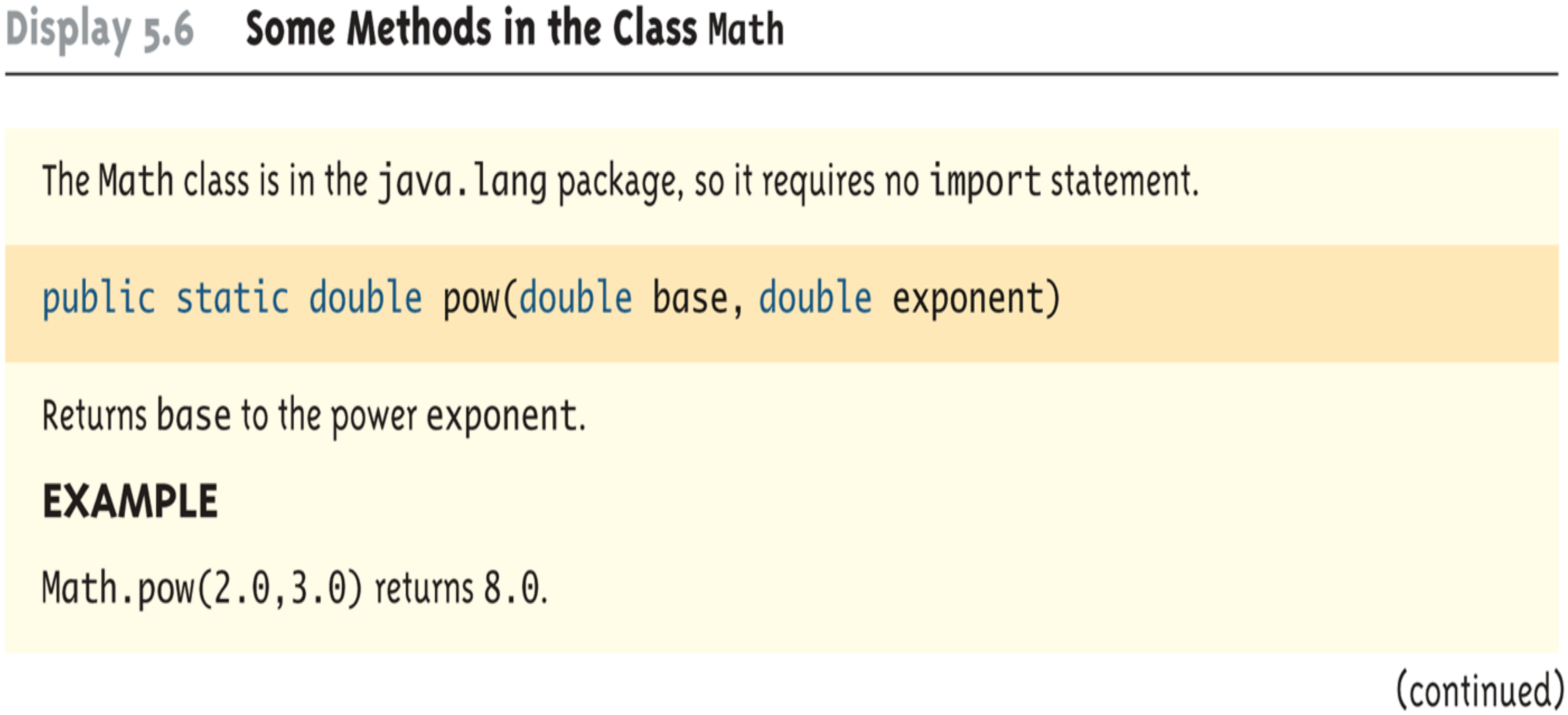
- A convenient way to write test code for your class.

The `Math` Class

- The `Math` class provides a number of standard mathematical methods.
  - Found in the `java.lang` package, so it does not require an `import` statement
  - All of its methods and data are static.
    - They are invoked with the class name `Math` instead of a calling object.
  - The `Math` class has two predefined constants, `E` (*e*, the base of the natural logarithm system) and `PI` (π, 3.1415 . . .).

```
area = Math.PI * radius * radius;
```

Some Methods in the Class **`Math`**
(Part 1 of 5)



Display 5.6 **Some Methods in the Class** Math

The Math class is in the `java.lang` package, so it requires no `import` statement.

```
public static double pow(double base, double exponent)
```

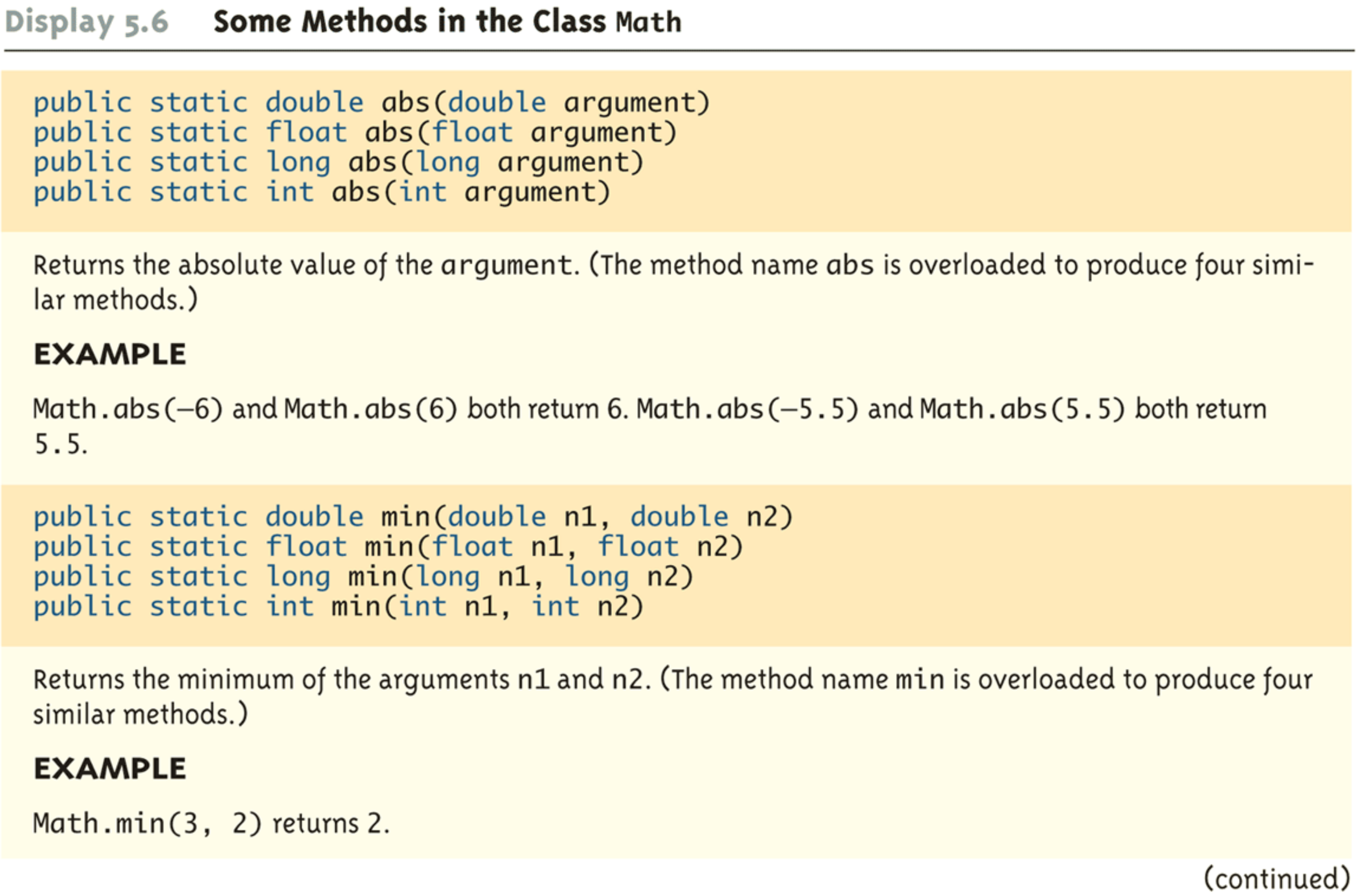
Returns base to the power exponent.

EXAMPLE

`Math.pow(2.0,3.0)` returns `8.0`.

(continued)

Some Methods in the Class **Math** (Part 2 of 5)



Display 5.6 Some Methods in the Class Math

```
public static double abs(double argument)
public static float abs(float argument)
public static long abs(long argument)
public static int abs(int argument)
```

Returns the absolute value of the argument. (The method name abs is overloaded to produce four similar methods.)

EXAMPLE

Math.abs(−6) and Math.abs(6) both return 6. Math.abs(−5.5) and Math.abs(5.5) both return 5.5.

```
public static double min(double n1, double n2)
public static float min(float n1, float n2)
public static long min(long n1, long n2)
public static int min(int n1, int n2)
```

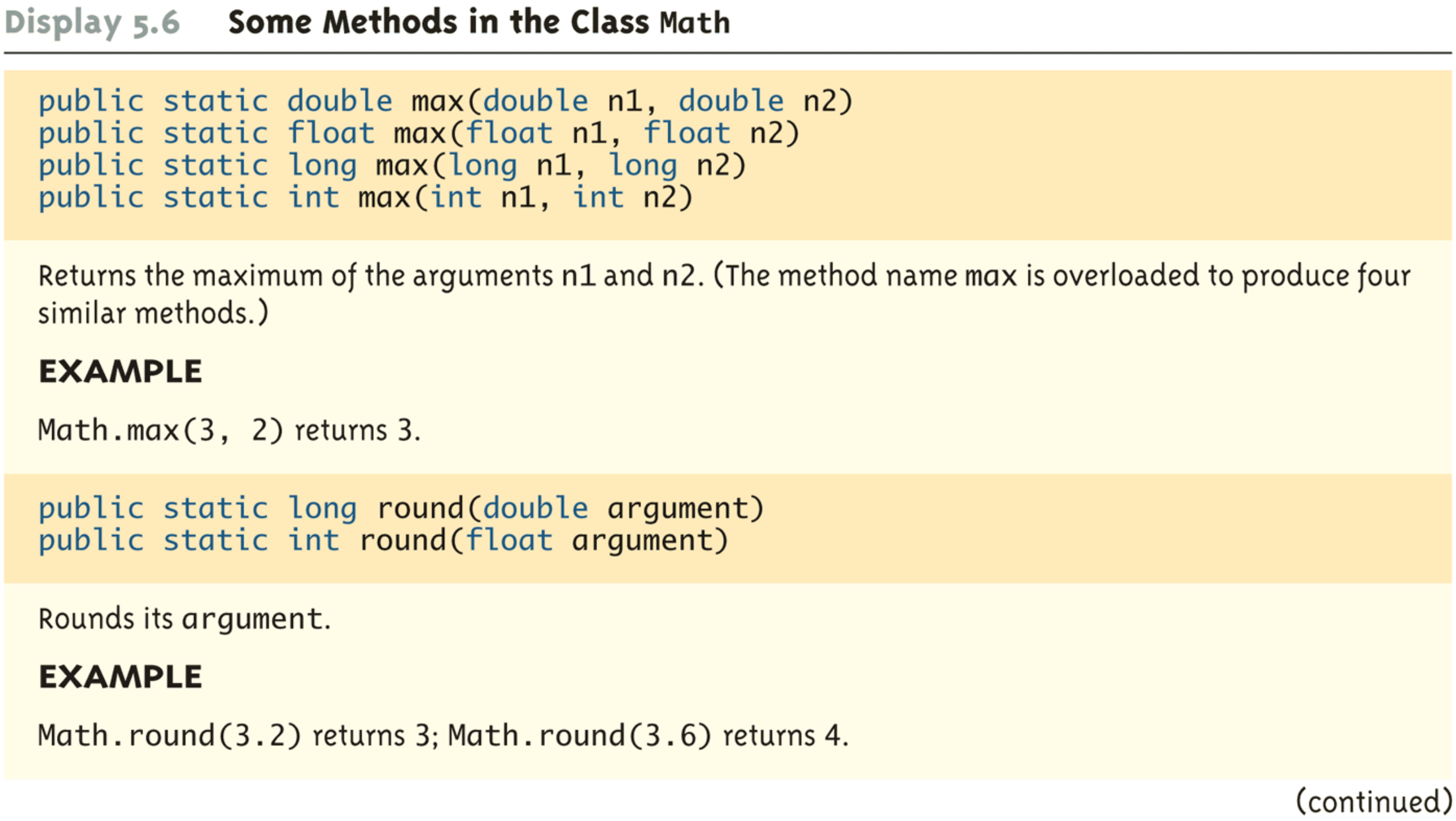
Returns the minimum of the arguments n1 and n2. (The method name min is overloaded to produce four similar methods.)

EXAMPLE

Math.min(3, 2) returns 2.

(continued)

Some Methods in the Class **Math** (Part 3 of 5)



Display 5.6 **Some Methods in the Class Math**

```
public static double max(double n1, double n2)
public static float max(float n1, float n2)
public static long max(long n1, long n2)
public static int max(int n1, int n2)
```

Returns the maximum of the arguments n1 and n2. (The method name max is overloaded to produce four similar methods.)

EXAMPLE

Math.max(3, 2) returns 3.

```
public static long round(double argument)
public static int round(float argument)
```

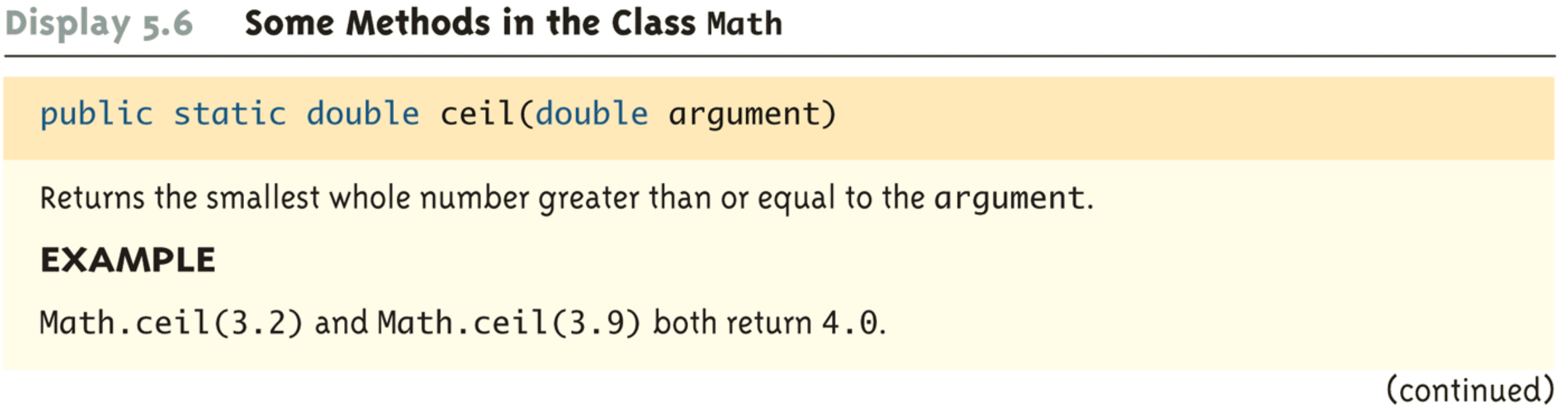
Rounds its argument.

EXAMPLE

Math.round(3.2) returns 3; Math.round(3.6) returns 4.

(continued)

Some Methods in the Class **`Math`**
(Part 4 of 5)



Display 5.6 **Some Methods in the Class `Math`**

```
public static double ceil(double argument)
```

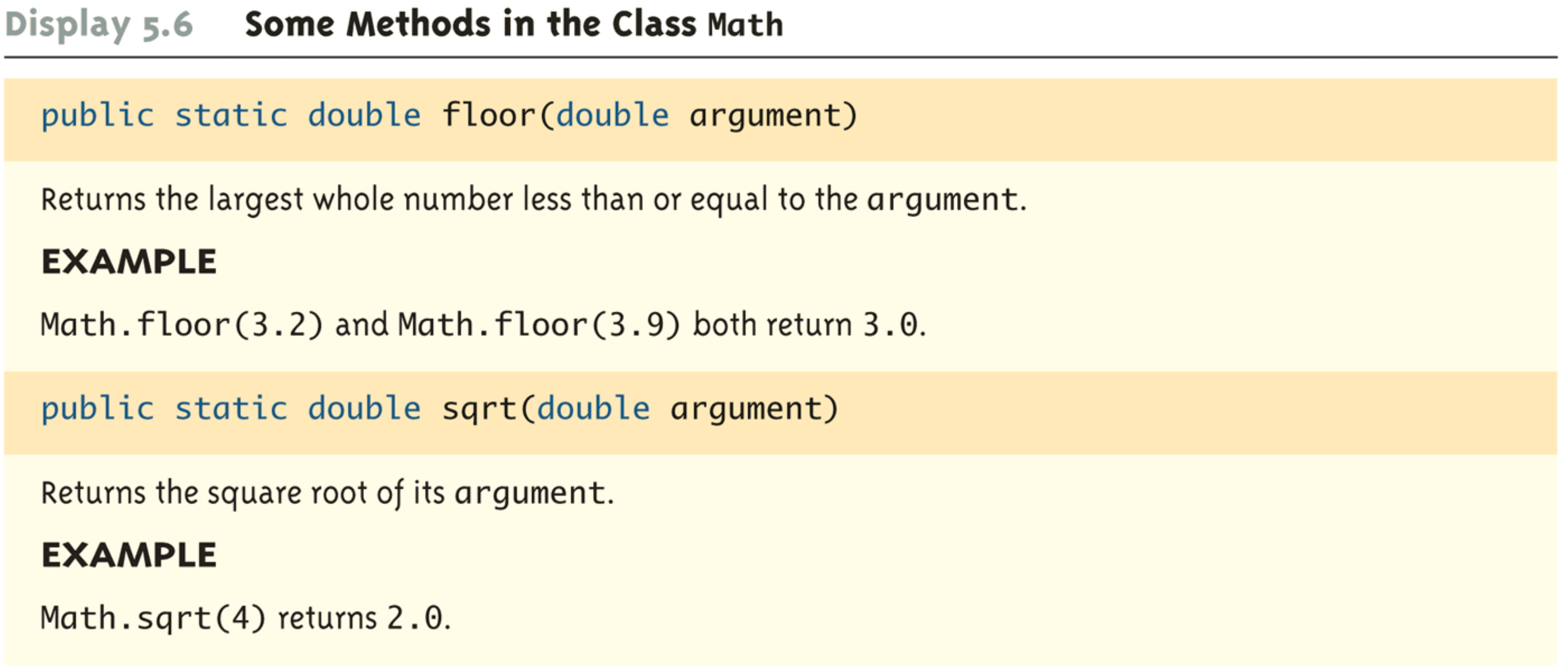
Returns the smallest whole number greater than or equal to the `argument`.

EXAMPLE

`Math.ceil(3.2)` and `Math.ceil(3.9)` both return `4.0`.

(continued)

Some Methods in the Class **`Math`** (Part 5 of 5)



Display 5.6 **Some Methods in the Class `Math`**

```
public static double floor(double argument)
```

Returns the largest whole number less than or equal to the `argument`.

EXAMPLE

`Math.floor(3.2)` and `Math.floor(3.9)` both return `3.0`.

```
public static double sqrt(double argument)
```

Returns the square root of its `argument`.

EXAMPLE

`Math.sqrt(4)` returns `2.0`.

Static Review

- Given the skeleton class definition below

```
public class C
{
  public int a = 0;
  public static int b = 1;

  public void f( ) { …}
  public static void g( ) {…}
}
```

- Can body of f( ) refer to a?
- Can body of f( ) refer to b?
- Can body of g( ) refer to a?
- Can body of g( ) refer to b?
- Can f( ) call g( )?
- Can g( ) call f( )?
  - For each, explain why or why not.

Wrapper Classes

Wrapper Classes

- ***Wrapper classes***
  - Provide a class type corresponding to each of the primitive types
  - Makes it possible to have class types that behave somewhat like primitive types
  - The wrapper classes for the primitive types:

`byte`, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, and **`char`**

are (in order)

`Byte`, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`**

  - Wrapper classes also contain useful
    - predefined constants
    - static methods

Constants and Static Methods in Wrapper Classes

- Wrapper classes include constants that provide the largest and smallest values for any of the primitive number types.
  - `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Double.MAX_VALUE`, `Double.MIN_VALUE`, etc.
- The `Boolean` class has names for two constants of type `Boolean`.
  - `Boolean.TRUE` corresponds to `true`
  - `Boolean.FALSE` corresponds to `false`

  of the primitive type `boolean`.

Constants and Static Methods in Wrapper Classes

- Some static methods convert a correctly formed string representation of a number to the number of a given type.
  - The methods `Integer.parseInt()`, `Long.parseLong()`, `Float.parseFloat()`, and `Double.parseDouble()`

    do this for the primitive types (in order) `int`, `long`, `float`, and `double`.
- Static methods convert from a numeric value to a string representation of the value.
  - For example, the expression

    ```
    Double.toString(123.99);
    ```

    returns the string value `"123.99"`
- The `Character` class contains a number of static methods that are useful for string processing.

Wrappers and Command Line Arguments

- Command line arguments are passed to main via its parameter conventionally named args.

```
public static void main (String[ ] args)
```

- For example, if we execute our program as

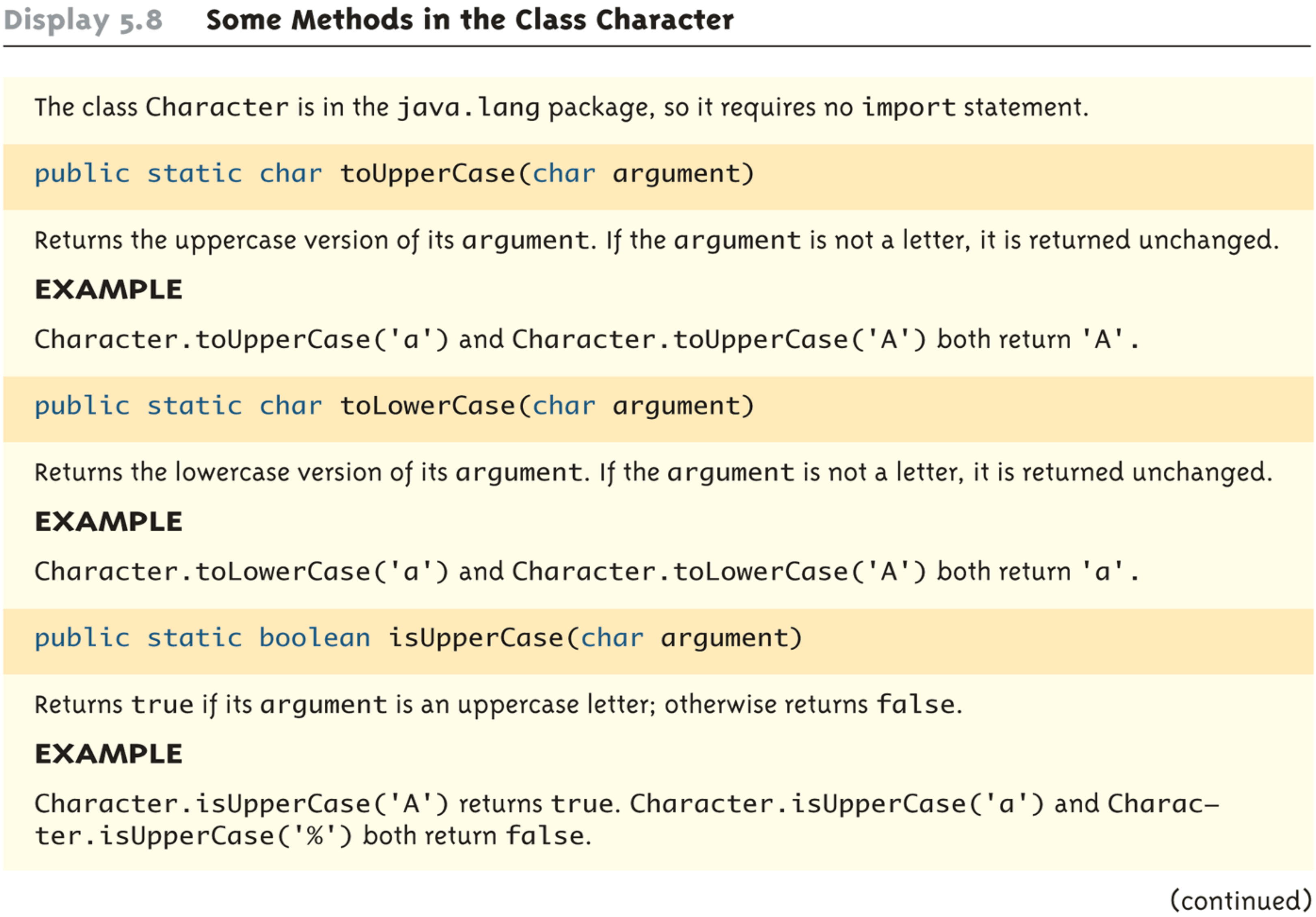
```
java proj1.Project1 Bob 42
```

then args[0] = "Bob" and args[1] = "42".

- We can use the static method **Integer.parseInt( )** to change the argument "42" to an integer variable via

```
int age = Integer.parseInt( args[ 1 ] );
```

Methods in the Class `Character` (1 of 3)



Display 5.8 Some Methods in the Class Character

The class `Character` is in the `java.lang` package, so it requires no `import` statement.

`public static char toUpperCase(char argument)`

Returns the uppercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE

`Character.toUpperCase('a')` and `Character.toUpperCase('A')` both return `'A'`.

`public static char toLowerCase(char argument)`

Returns the lowercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE

`Character.toLowerCase('a')` and `Character.toLowerCase('A')` both return `'a'`.

`public static boolean isUpperCase(char argument)`

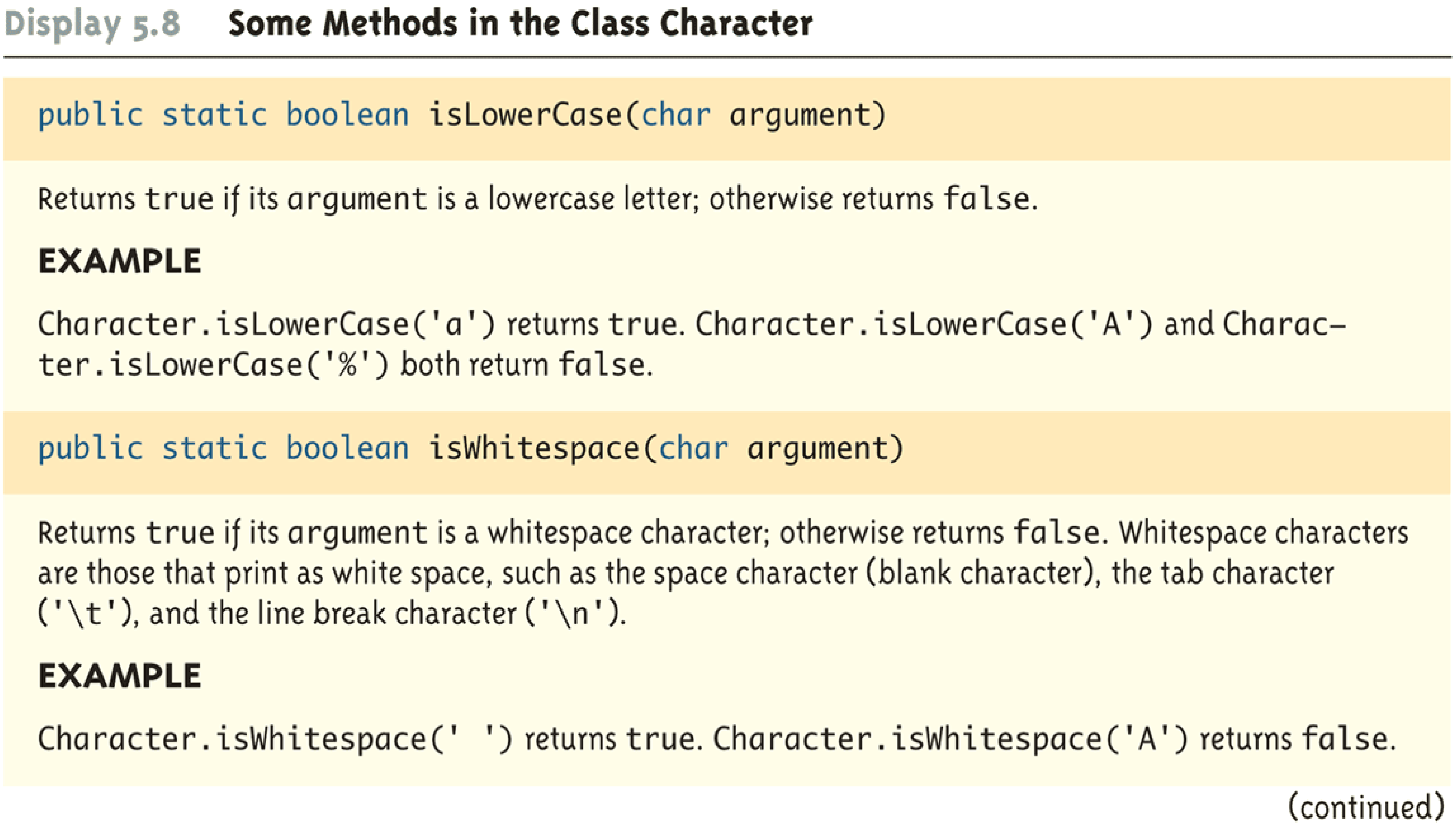
Returns `true` if its `argument` is an uppercase letter; otherwise returns `false`.

EXAMPLE

`Character.isUpperCase('A')` returns `true`. `Character.isUpperCase('a')` and `Character.isUpperCase('%')` both return `false`.

(continued)

Methods in the Class **Character** (2 of 3)



Display 5.8 **Some Methods in the Class Character**

`public static boolean isLowerCase(char argument)`

Returns `true` if its argument is a lowercase letter; otherwise returns `false`.

EXAMPLE

`Character.isLowerCase('a')` returns `true`. `Character.isLowerCase('A')` and `Character.isLowerCase('%')` both return `false`.

`public static boolean isWhitespace(char argument)`

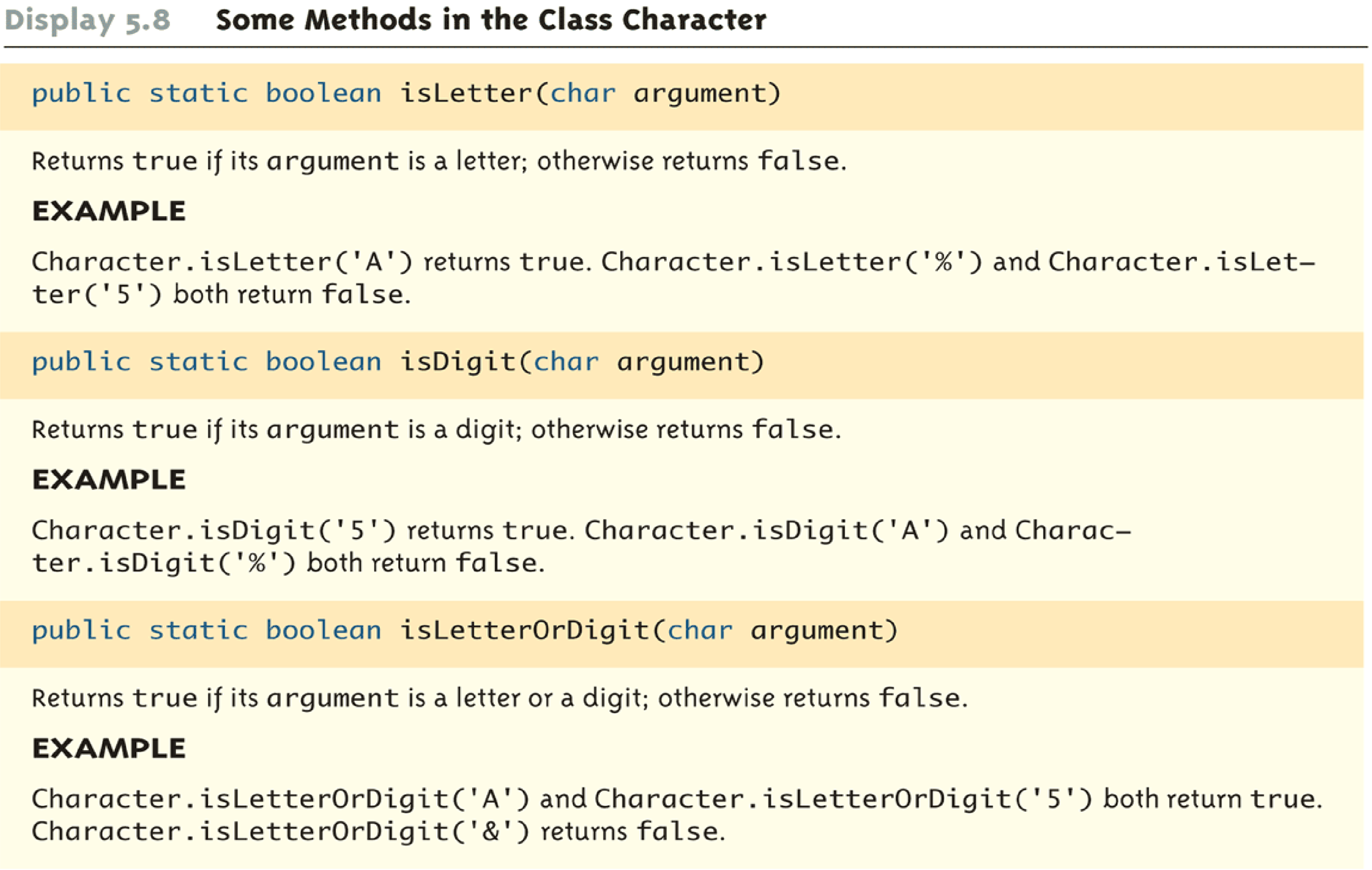
Returns `true` if its argument is a whitespace character; otherwise returns `false`. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character (`'\t'`), and the line break character (`'\n'`).

EXAMPLE

`Character.isWhitespace(' ')` returns `true`. `Character.isWhitespace('A')` returns `false`.

(continued)

Methods in the Class **`Character`** (3 of 3)



Display 5.8 Some Methods in the Class Character

```
public static boolean isLetter(char argument)
```

Returns `true` if its argument is a letter; otherwise returns `false`.

EXAMPLE

`Character.isLetter('A')` returns `true`. `Character.isLetter('%')` and `Character.isLetter('5')` both return `false`.

```
public static boolean isDigit(char argument)
```

Returns `true` if its argument is a digit; otherwise returns `false`.

EXAMPLE

`Character.isDigit('5')` returns `true`. `Character.isDigit('A')` and `Character.isDigit('%')` both return `false`.

```
public static boolean isLetterOrDigit(char argument)
```

Returns `true` if its argument is a letter or a digit; otherwise returns `false`.

EXAMPLE

`Character.isLetterOrDigit('A')` and `Character.isLetterOrDigit('5')` both return `true`. `Character.isLetterOrDigit('&')` returns `false`.

Boxing

- ***Boxing***: The process of converting from a value of a primitive type to an object of its wrapper class.
  - Create an object of the corresponding wrapper class using the primitive value as an argument
  - The new object will contain an instance variable that stores a copy of the primitive value.

```
Integer integerObject = new Integer(42);
```

  - Unlike most other classes, a wrapper class does not have a no-argument constructor.
  - The value inside a Wrapper class is ***immutable***.

Unboxing

- ***Unboxing***: The process of converting from an object of a wrapper class to the corresponding value of a primitive type.
  - The methods for converting an object from the wrapper classes

    `Byte`, `Short`, `Integer`, `Long`, `Float`,
    `Double`, and `Character`

    to their corresponding primitive type are (in order)

    `byteValue`, `shortValue`, `intValue`,
    `longValue`, `floatValue`, `doubleValue`,
    and `charValue`.

  - None of these methods take an argument.

```
int i = integerObject.intValue();
```

Automatic Boxing and Unboxing

Starting with version 5.0, Java can automatically do boxing and unboxing for you.

- Boxing:

```
Integer integerObject = 42;
```

rather than:

```
Integer integerObject = new Integer(42);
```

- Unboxing:

```
int i = integerObject;
```

rather than:

```
int i = integerObject.intValue();
```