Chapter 5
Defining Classes II

(Part 1)

Prof. Choonhwa Lee

Dept. of Computer Science and Engineering
Hanyang University

Static Methods

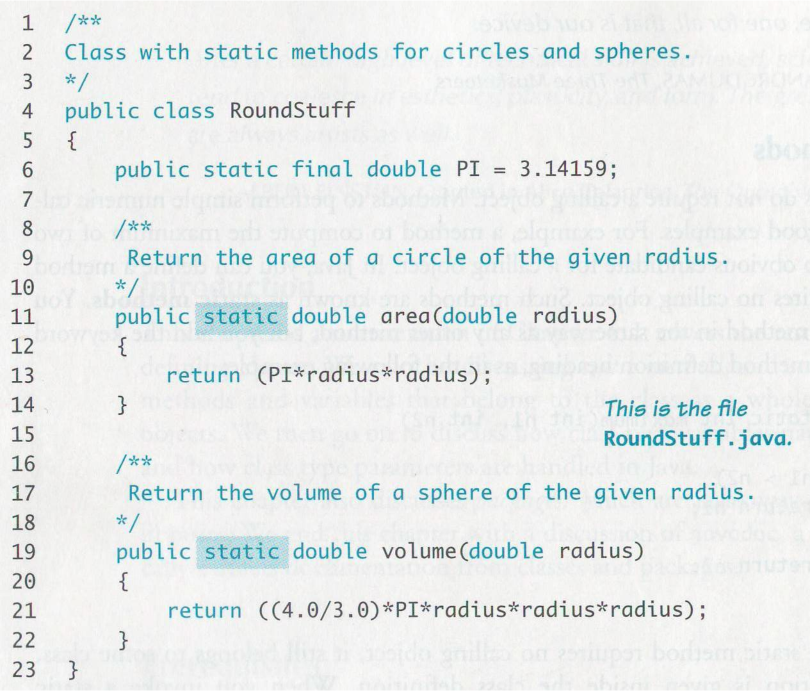
- A *static method* is one that can be used without a calling object
- A static method still belongs to a class, and its definition is given inside the class definition
- When a static method is defined, the keyword **`static`** is placed in the method header

```
public static returnedType myMethod(parameters)
{ . . . }
```

- Static methods are invoked using the class name in place of a calling object

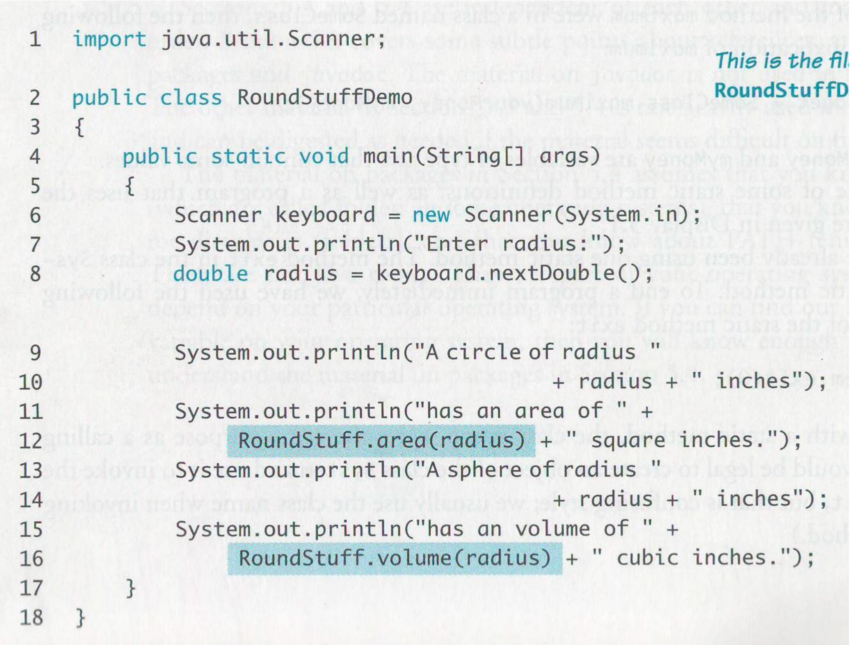
```
returnedValue = MyClass.myMethod(arguments);
```

Display 5.1 Static Methods



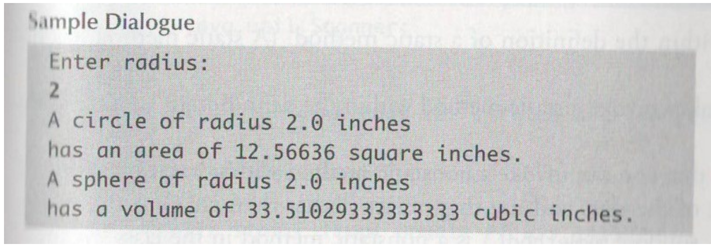
```
/**
Class with static methods for circles and spheres.
*/
public class RoundStuff
{
    public static final double PI = 3.14159;

    /**
     Return the area of a circle of the given radius.
    */
    public static double area(double radius)
    {
        return (PI*radius*radius);
    }

    /**
     Return the volume of a sphere of the given radius.
    */
    public static double volume(double radius)
    {
        return ((4.0/3.0)*PI*radius*radius*radius);
    }
}
```

This is the file **RoundStuff.java.**



```
import java.util.Scanner;

public class RoundStuffDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter radius:");
        double radius = keyboard.nextDouble();

        System.out.println("A circle of radius "
                                + radius + " inches");
        System.out.println("has an area of " +
             RoundStuff.area(radius) + " square inches.");
        System.out.println("A sphere of radius "
                                + radius + " inches");
        System.out.println("has an volume of " +
             RoundStuff.volume(radius) + " cubic inches.");
    }
}
```

This is the fil **RoundStuffD**



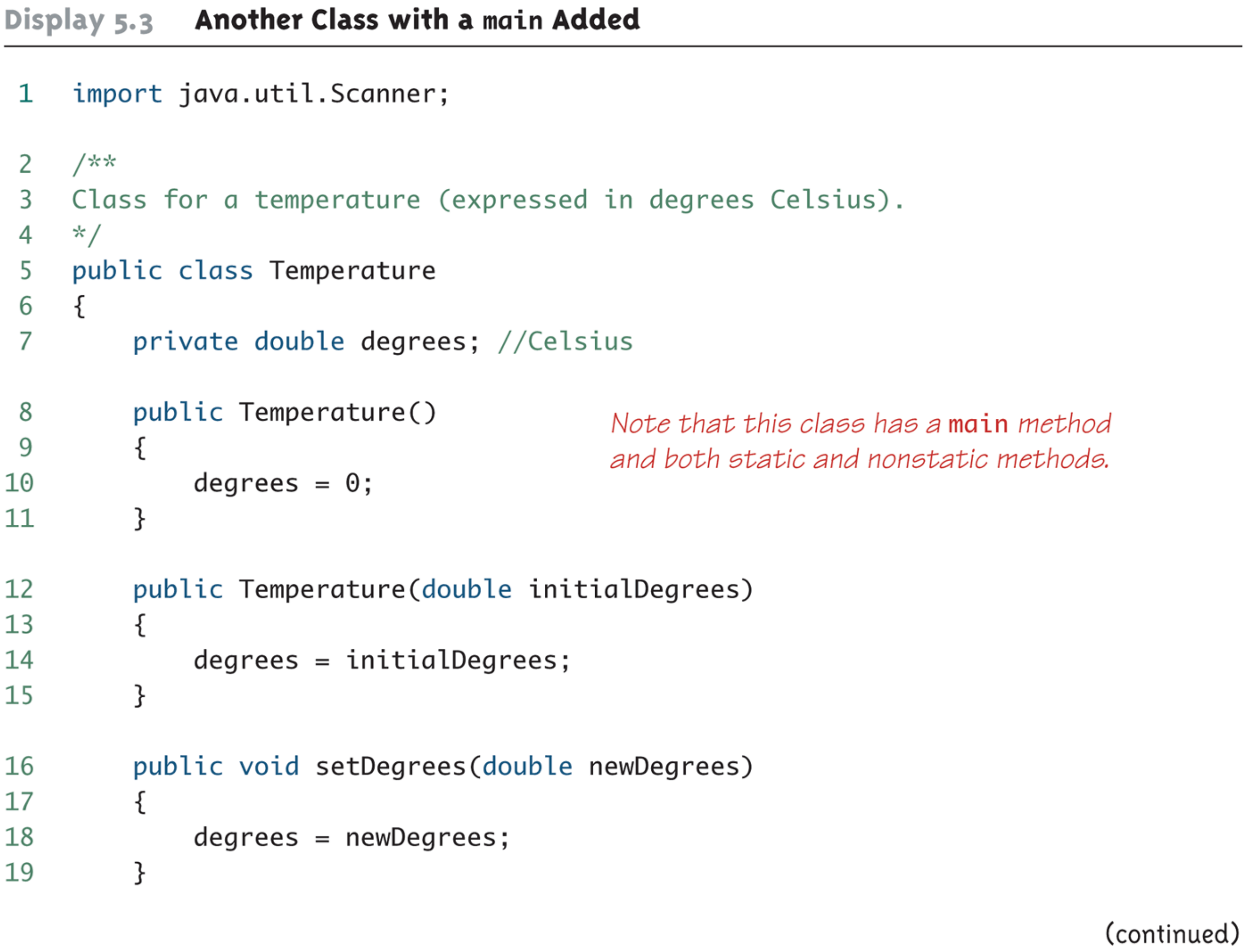
Sample Dialogue

```
Enter radius:
2
A circle of radius 2.0 inches
has an area of 12.56636 square inches.
A sphere of radius 2.0 inches
has a volume of 33.51029333333333 cubic inches.
```

Pitfall: Invoking a Nonstatic Method Within a Static Method

- A static method cannot refer to an instance variable of the class, and it cannot invoke a nonstatic method of the class
  - A static method has no **`this`**, so it cannot use an instance variable or method that has an implicit or explicit **`this`** for a calling object
  - A static method can invoke another static method, however

Another Class with a **`main`** Added (Part 1 of 4)



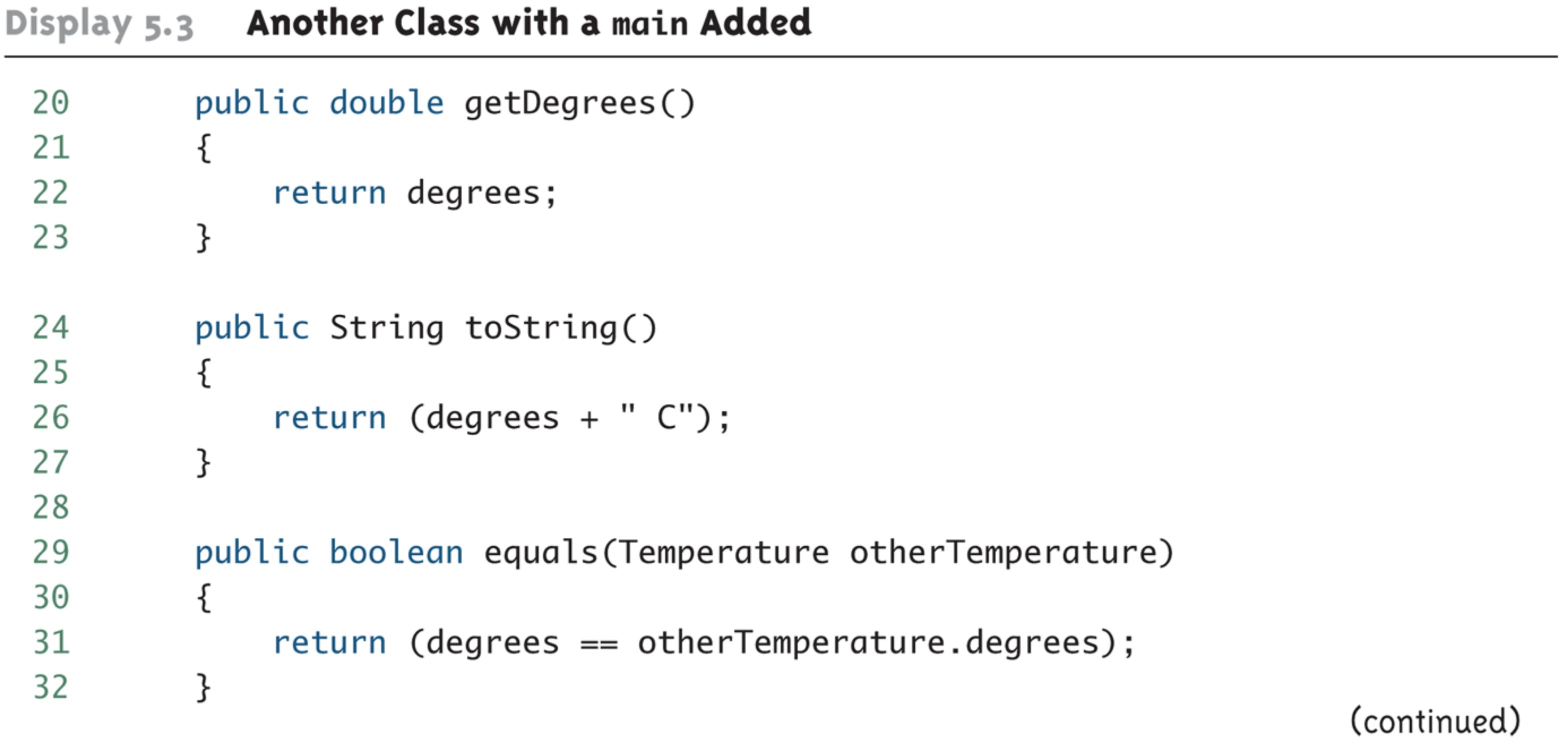
Display 5.3 **Another Class with a `main` Added**

```
import java.util.Scanner;

/**
Class for a temperature (expressed in degrees Celsius).
*/
public class Temperature
{
    private double degrees; //Celsius

    public Temperature()
    {
        degrees = 0;
    }

    public Temperature(double initialDegrees)
    {
        degrees = initialDegrees;
    }

    public void setDegrees(double newDegrees)
    {
        degrees = newDegrees;
    }
```

Note that this class has a `main` *method and both static and nonstatic methods.*

(continued)

Another Class with a **`main`** Added (Part 2 of 4)

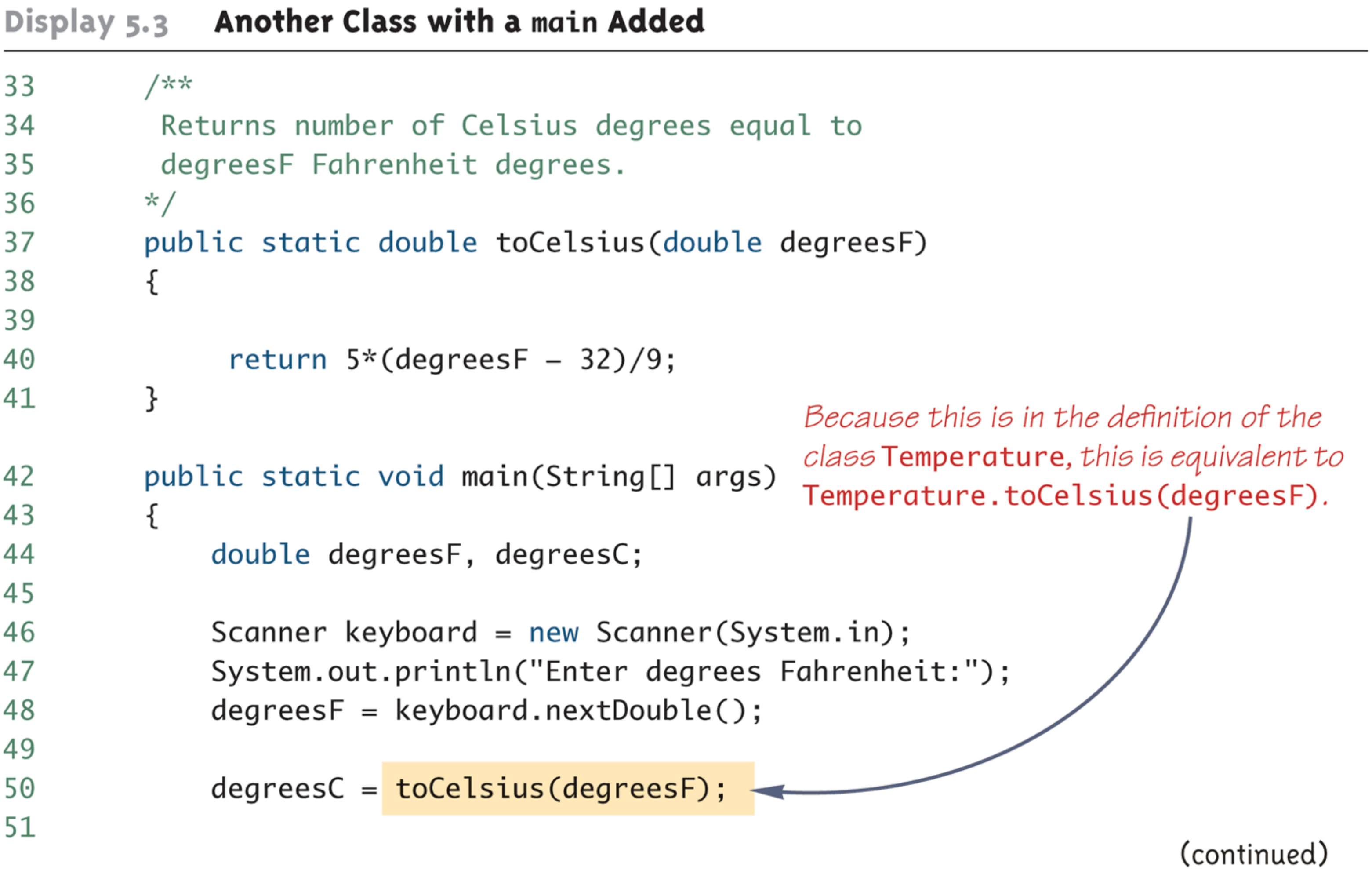


Display 5.3 **Another Class with a `main` Added**

```
    public double getDegrees()
    {
        return degrees;
    }

    public String toString()
    {
        return (degrees + " C");
    }

    public boolean equals(Temperature otherTemperature)
    {
        return (degrees == otherTemperature.degrees);
    }
```

(continued)

Another Class with a `main` Added (Part 3 of 4)



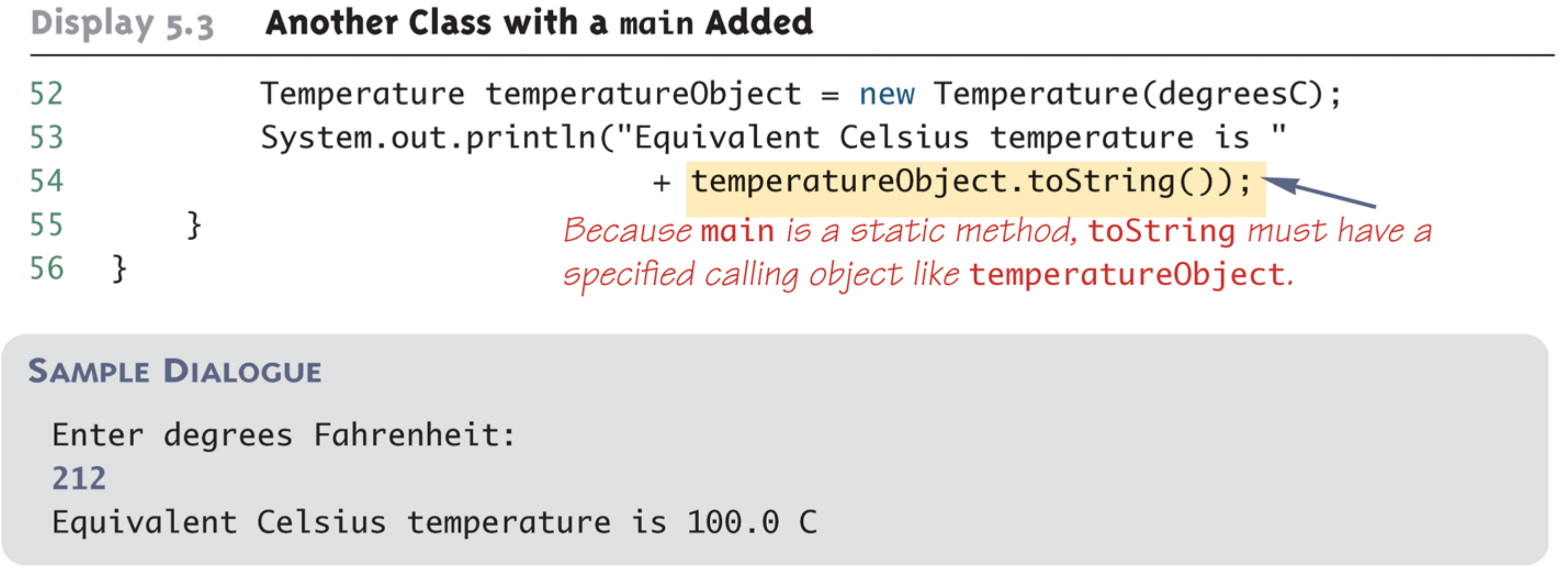
Display 5.3 **Another Class with a main Added**

```
    /**
     Returns number of Celsius degrees equal to
     degreesF Fahrenheit degrees.
    */
    public static double toCelsius(double degreesF)
    {

         return 5*(degreesF – 32)/9;
    }

    public static void main(String[] args)
    {
        double degreesF, degreesC;

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter degrees Fahrenheit:");
        degreesF = keyboard.nextDouble();

        degreesC = toCelsius(degreesF);

```

Because this is in the definition of the class `Temperature`*, this is equivalent to* `Temperature.toCelsius(degreesF)`.

(continued)

Another Class with a **`main`** Added (Part 4 of 4)



Display 5.3 **Another Class with a `main` Added**

```
        Temperature temperatureObject = new Temperature(degreesC);
        System.out.println("Equivalent Celsius temperature is "
                               + temperatureObject.toString());
    }
}
```

Because `main` *is a static method,* `toString` *must have a specified calling object like* `temperatureObject`*.*

Sample Dialogue

```
Enter degrees Fahrenheit:
212
Equivalent Celsius temperature is 100.0 C
```

Static Variables

- A *static variable* is a variable that belongs to the class as a whole, and not just to one object
  - There is only one copy of a static variable per class, unlike instance variables where each object has its own copy
- All objects of the class can read and change a static variable
- Although a static method cannot access an instance variable, a static method can access a static variable
- A static variable is declared like an instance variable, with the addition of the modifier **`static`**

```
    private static int myStaticVariable;
```

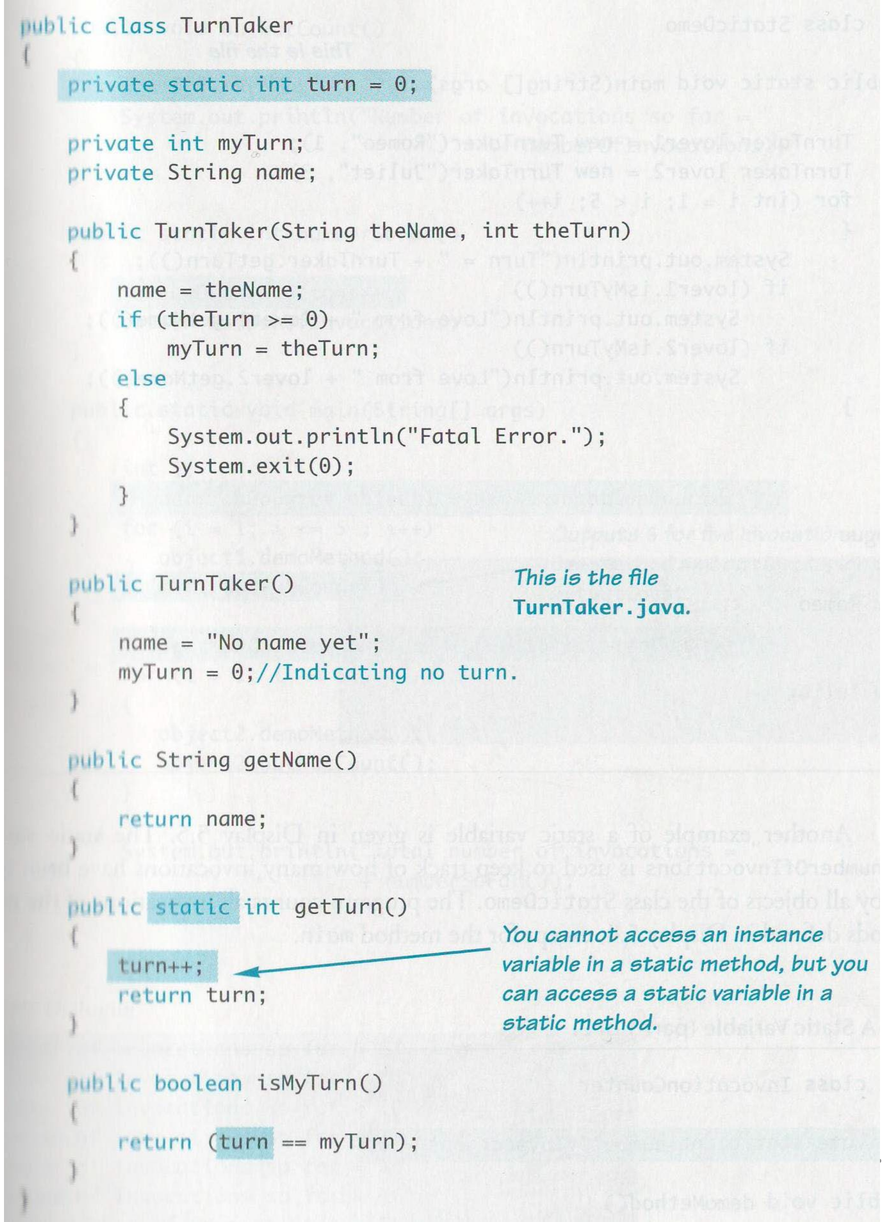
Static Variables

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

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

- If not explicitly initialized, a static variable will be automatically initialized to a default value
  - `boolean` static variables are initialized to `false`
  - Other primitive types static variables are initialized to the zero of their type
  - Class type static variables are initialized to `null`
- It is always preferable to explicitly initialize static variables rather than rely on the default initialization

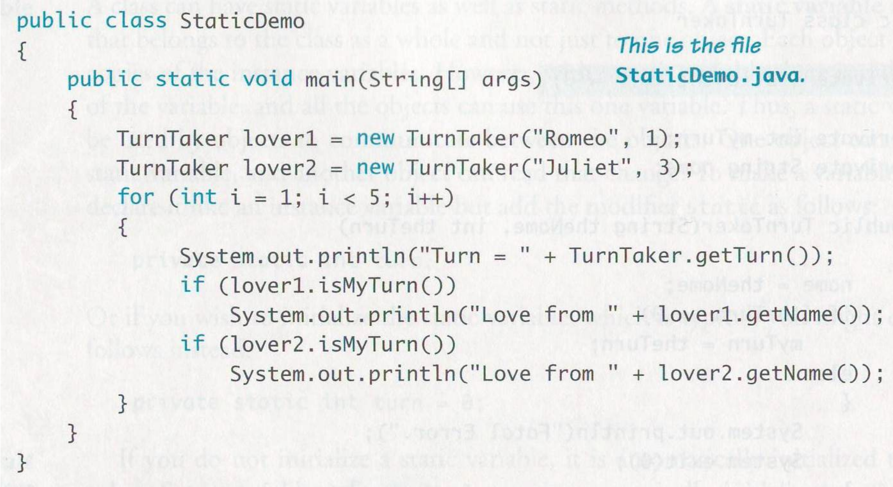
Display 5.4 A Static Variable



```
public class TurnTaker
{
    private static int turn = 0;

    private int myTurn;
    private String name;

    public TurnTaker(String theName, int theTurn)
    {
        name = theName;
        if (theTurn >= 0)
            myTurn = theTurn;
        else
        {
            System.out.println("Fatal Error.");
            System.exit(0);
        }
    }

    public TurnTaker()
    {
        name = "No name yet";
        myTurn = 0;//Indicating no turn.
    }

    public String getName()
    {
        return name;
    }

    public static int getTurn()
    {
        turn++;
        return turn;
    }

    public boolean isMyTurn()
    {
        return (turn == myTurn);
    }
}
```

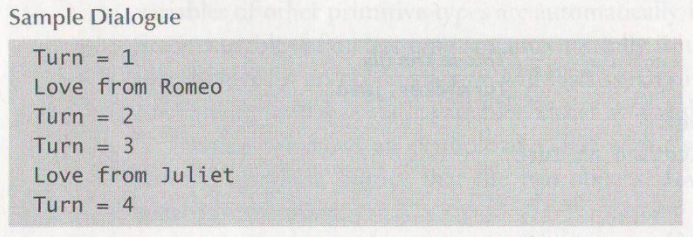
This is the file **TurnTaker.java.**

You cannot access an instance variable in a static method, but you can access a static variable in a static method.



```
public class StaticDemo
{
    public static void main(String[] args)
    {
        TurnTaker lover1 = new TurnTaker("Romeo", 1);
        TurnTaker lover2 = new TurnTaker("Juliet", 3);
        for (int i = 1; i < 5; i++)
        {
            System.out.println("Turn = " + TurnTaker.getTurn());
            if (lover1.isMyTurn())
                System.out.println("Love from " + lover1.getName());
            if (lover2.isMyTurn())
                System.out.println("Love from " + lover2.getName());
        }
    }
}
```

This is the file **StaticDemo.java.**



Sample Dialogue

```
Turn = 1
Love from Romeo
Turn = 2
Turn = 3
Love from Juliet
Turn = 4
```

Static Variables

- A static variable should always be defined private, unless it is also a defined constant
  - The value of a static defined constant cannot be altered, therefore it is safe to make it `public`
  - In addition to `static`, 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;
```

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

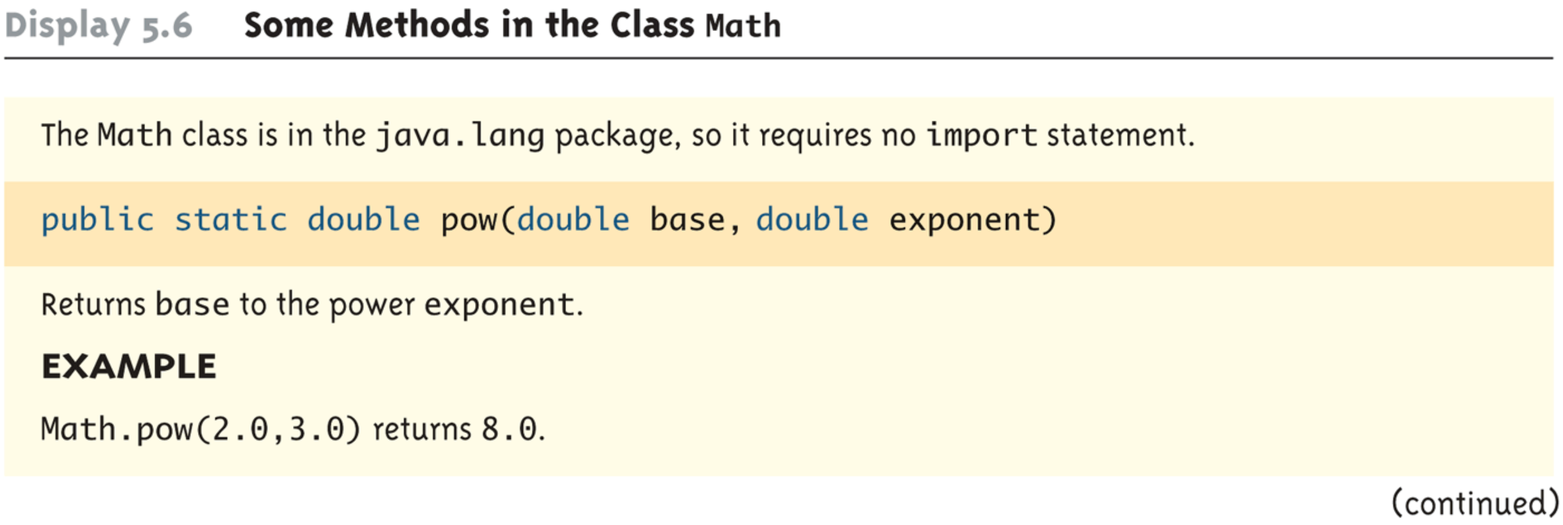
```
int year = MyClass.BIRTH_YEAR;
```

The **`Math`** Class

- The **`Math`** class provides a number of standard mathematical methods
  - It is found in the **`java.lang`** package, so it does not require an **`import`** statement
  - All of its methods and data are static, therefore 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)
```

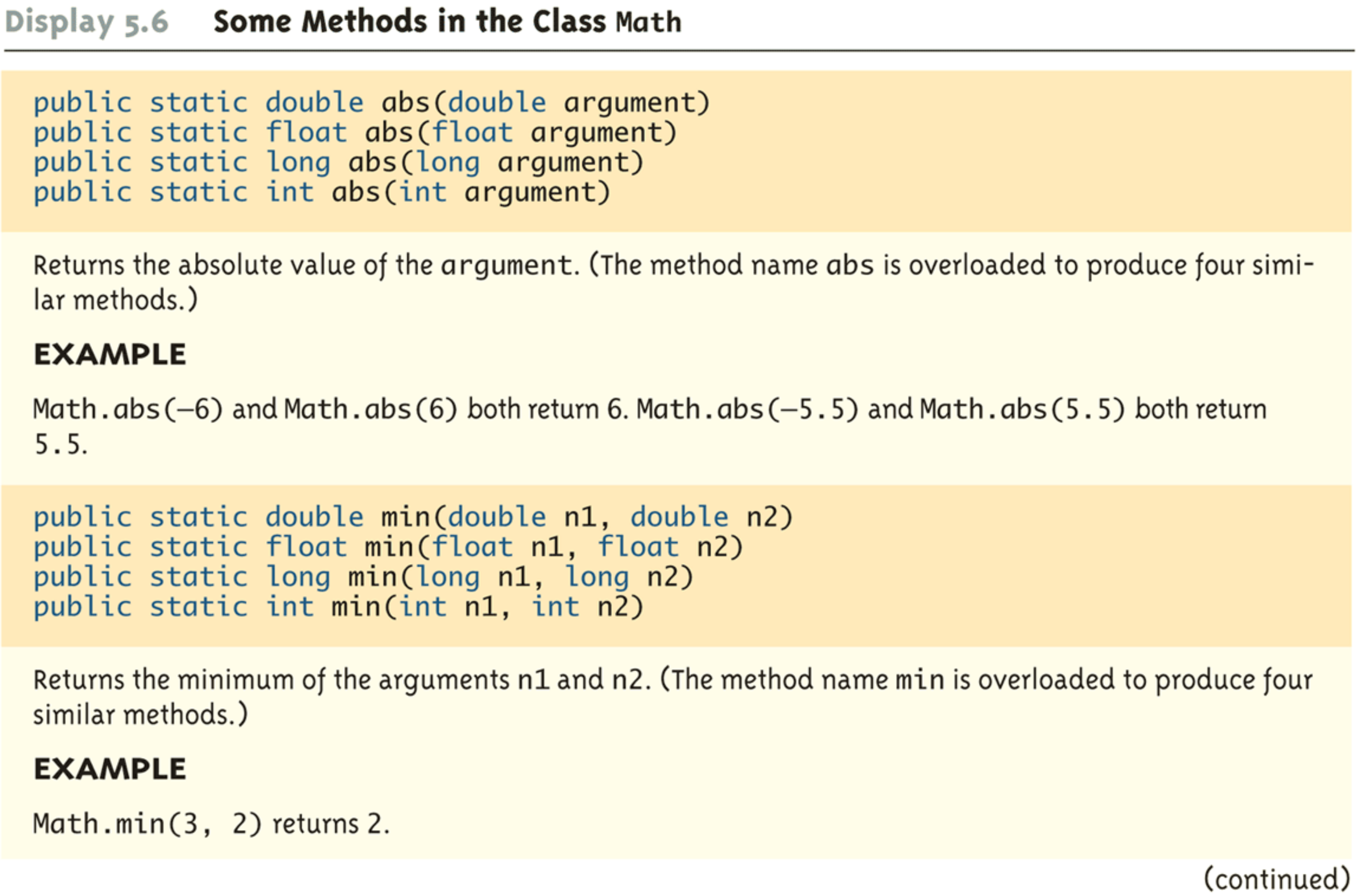
Returns `base` to the power `exponent`.

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)
```

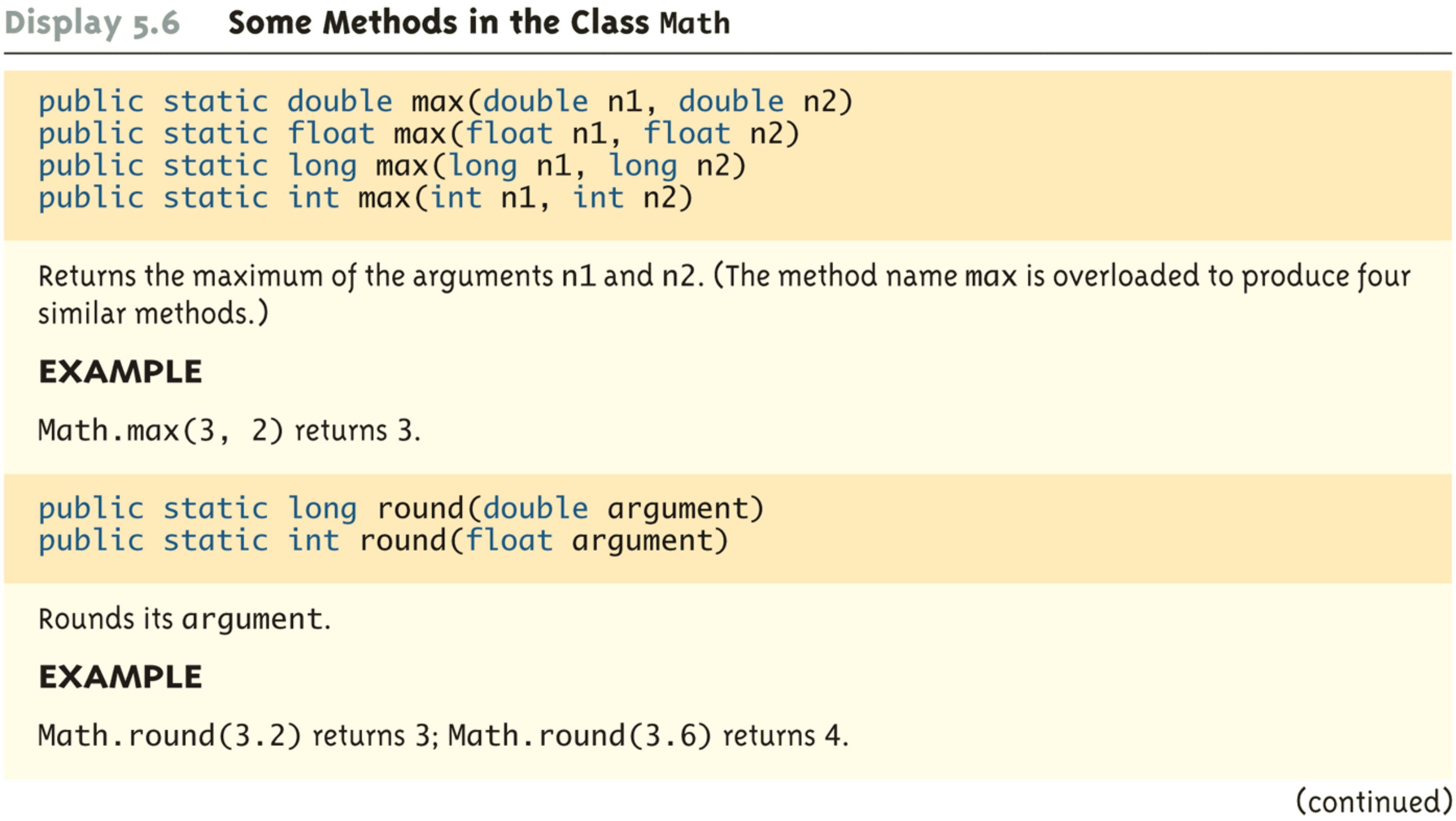
Returns the minimum of the arguments `n1` and `n2`. (The method name `min` is overloaded to produce four similar methods.)

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)
```

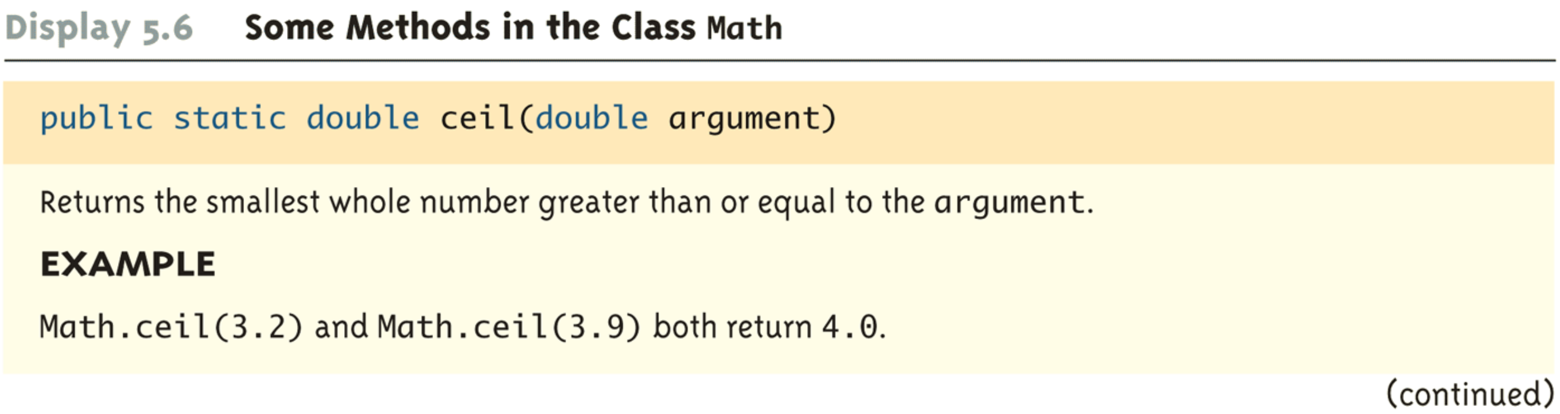
Rounds its `argument`.

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)
```

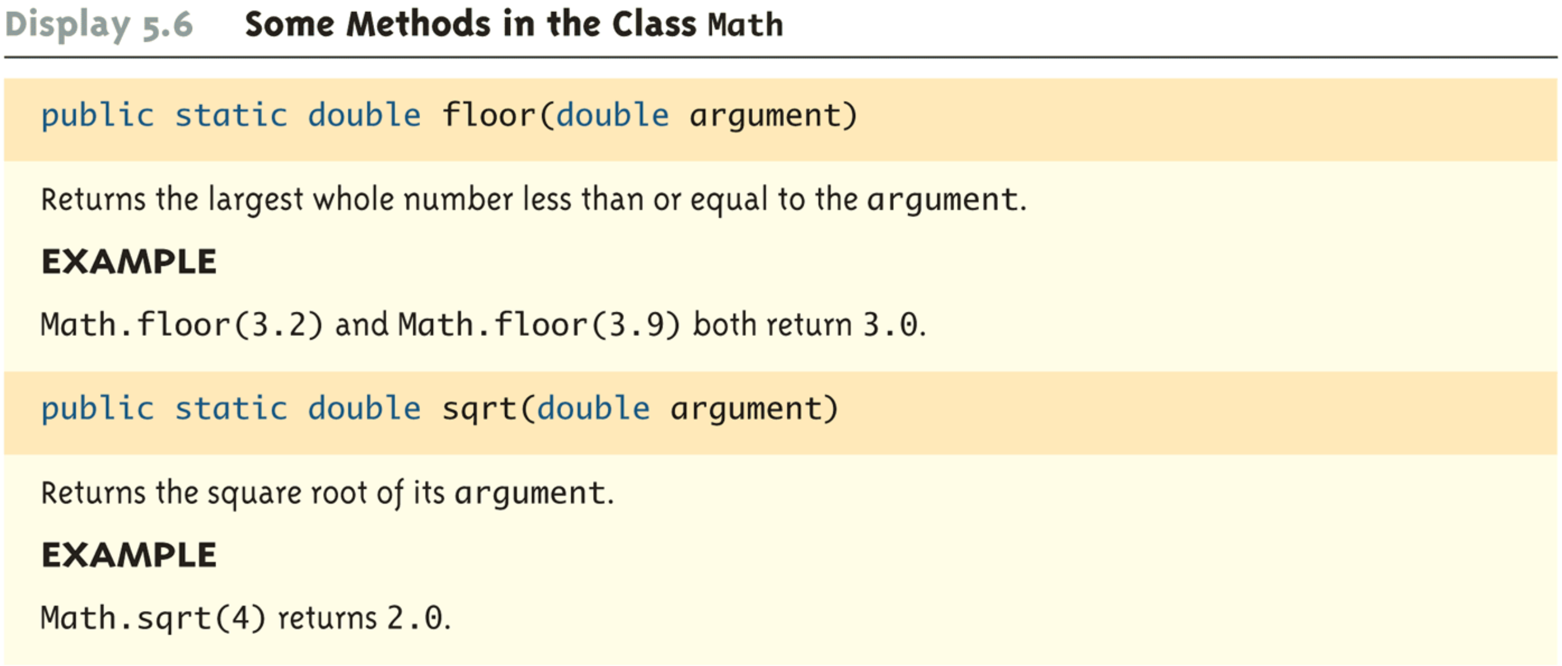
Returns the smallest whole number greater than or equal to the `argument`.

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`.

Random Numbers

- The **Math** class also provides a facility to generate pseudo-random numbers

```
public static double random()
```

  - A pseudo-random number appears random but is really generated by a deterministic function
    - There is also a more flexible class named **Random**

- Sample use: `double num = Math.random();`
- Returns a pseudo-random number greater than or equal to 0.0 and less than 1.0

Wrapper Classes

- *Wrapper classes* provide a class type corresponding to each of the primitive types
  - This makes it possible to have class types that behave somewhat like primitive types
  - The wrapper classes for the primitive types **`byte`**, **`short`**, **`long`**, **`float`**, **`double`**, and **`char`** are (in order) **`Byte`**, **`Short`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`**
- Wrapper classes also contain a number of useful predefined constants and static methods

Wrapper Classes

- *Wrapper classes* provide a class type corresponding to each of the primitive types
  - This makes it possible to have class types that behave somewhat like primitive types
  - The wrapper classes for the primitive types **`byte`**, **`short`**, **`long`**, **`float`**, **`double`**, and **`char`** are (in order) **`Byte`**, **`Short`**, **`Long`**, **`Float`**, **`Double`**, and **`Character`**
- Wr
  pre

Basic types	Derived types
Primitive types	**Class types**
Buit-in types	User-defined types

int → Integer

double → Double

Wrapper Classes

- *Boxing*: the process of going from a value of a primitive type to an object of its wrapper class
  - To convert a primitive value to an "equivalent" class type value, 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
  - Unlike most other classes, a wrapper class does not have a no-argument constructor

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

Wrapper Classes

- *Unboxing*: the process of going 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
- Instead of creating a wrapper class object using the **`new`** operation (as shown before), it can be done as an automatic type cast:

```
Integer integerObject = 42;
```

- Instead of having to invoke the appropriate method (such as **`intValue`**, **`doubleValue`**, **`charValue`**, etc.) in order to convert from an object of a wrapper class to a value of its associated primitive type, the primitive value can be recovered automatically

```
int i = integerObject;
```

Constants and Static Methods in Wrapper Classes

- Wrapper classes include useful constants that provide the largest and smallest values for any of the primitive number types
  - For example, `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` and `Boolean.FALSE` are the Boolean objects that correspond to the values `true` and `false` of the primitive type `boolean`

Constants and Static Methods in Wrapper Classes

- Wrapper classes have static methods that 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`**
- Wrapper classes also have static methods that convert from a numeric value to a string representation of the value
  - For example, the expression

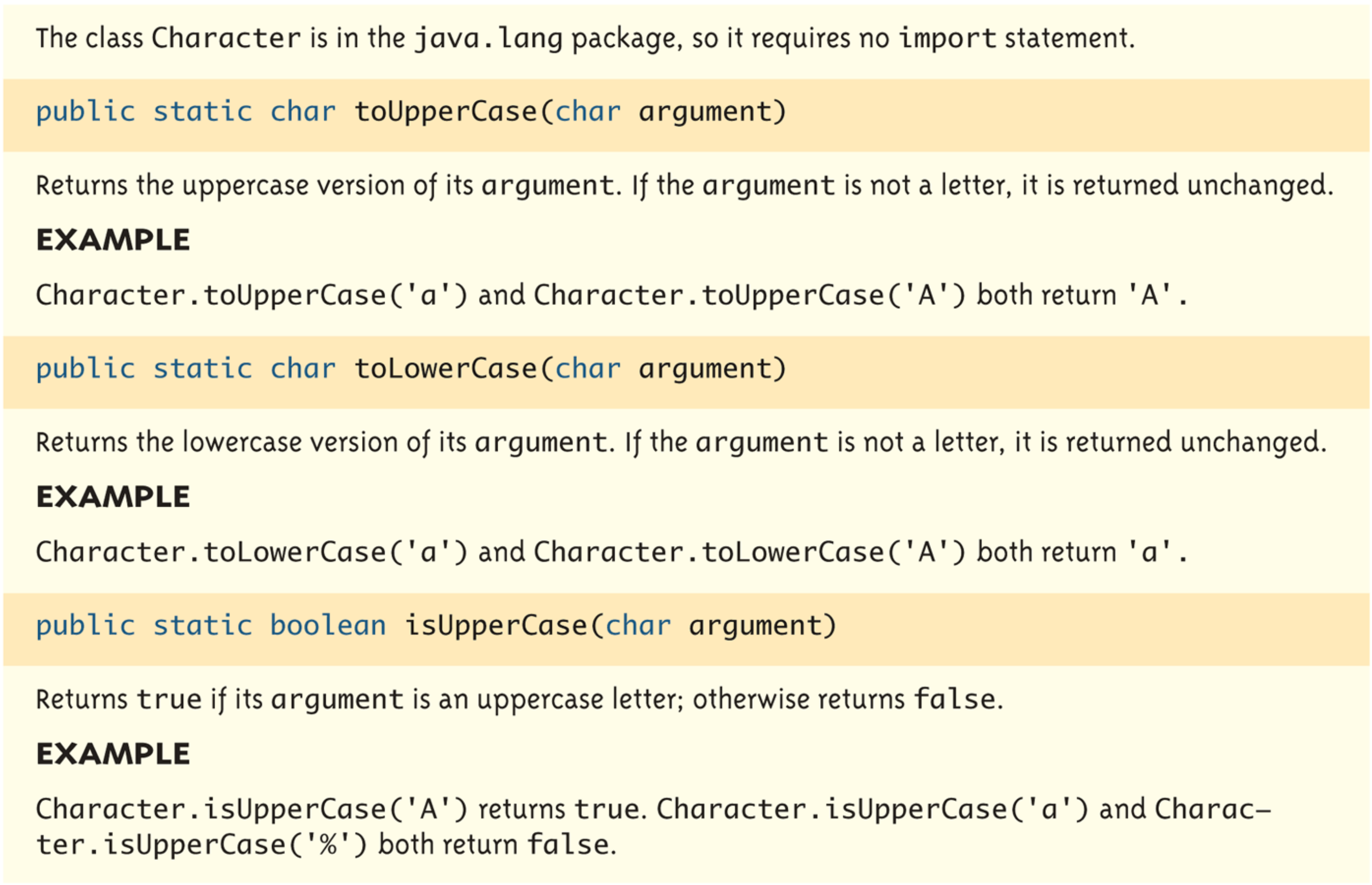
```
Double.toString(123.99);
```

returns the string value **`"123.99"`**

- The **`Character`** class contains a number of static methods that are useful for string processing

Some Methods in the Class **`Character`** (Part 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)`

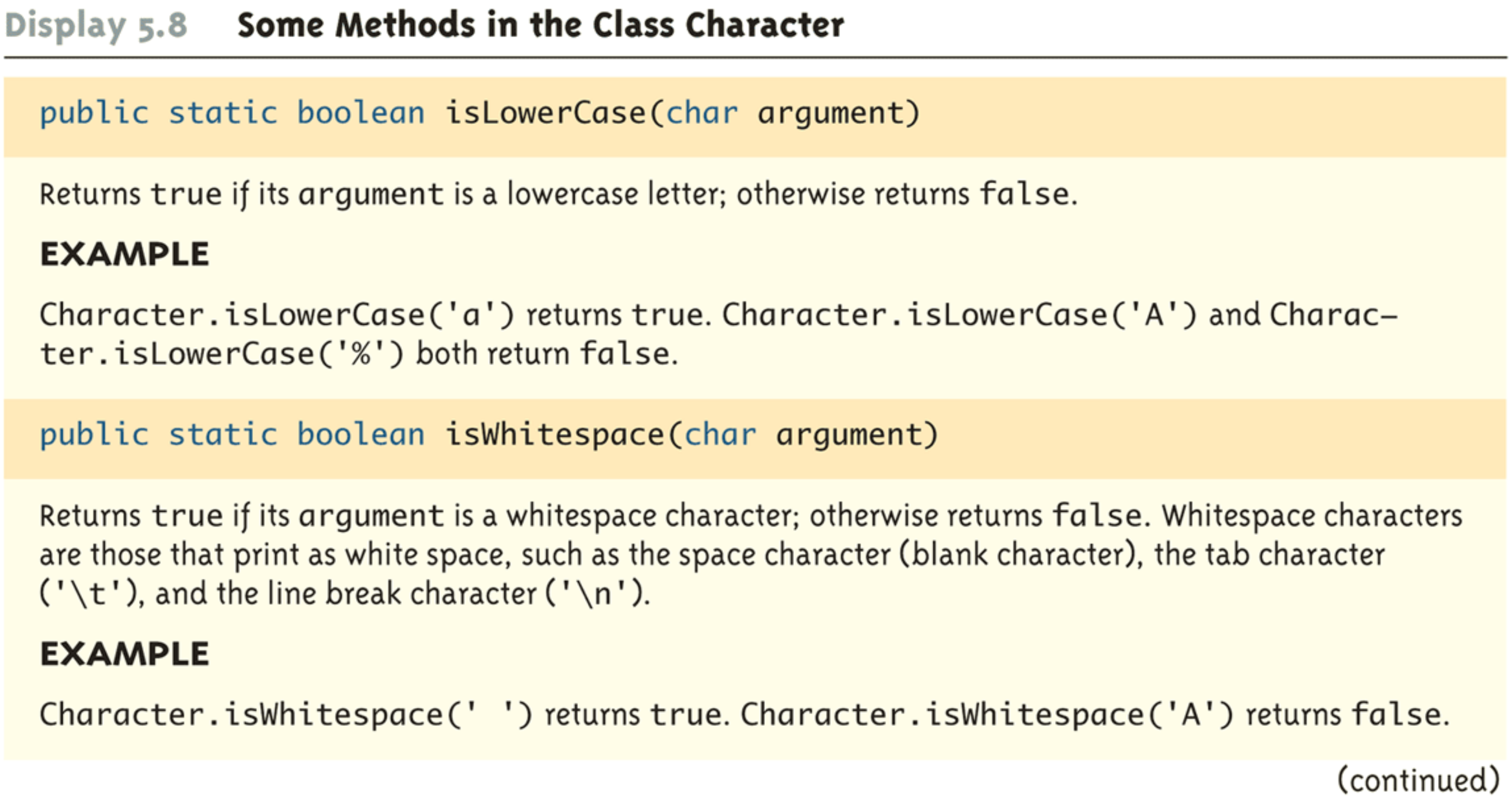
Returns `true` if its `argument` is an uppercase letter; otherwise returns `false`.

EXAMPLE

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

(continued)

Some Methods in the Class **Character** (Part 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)`

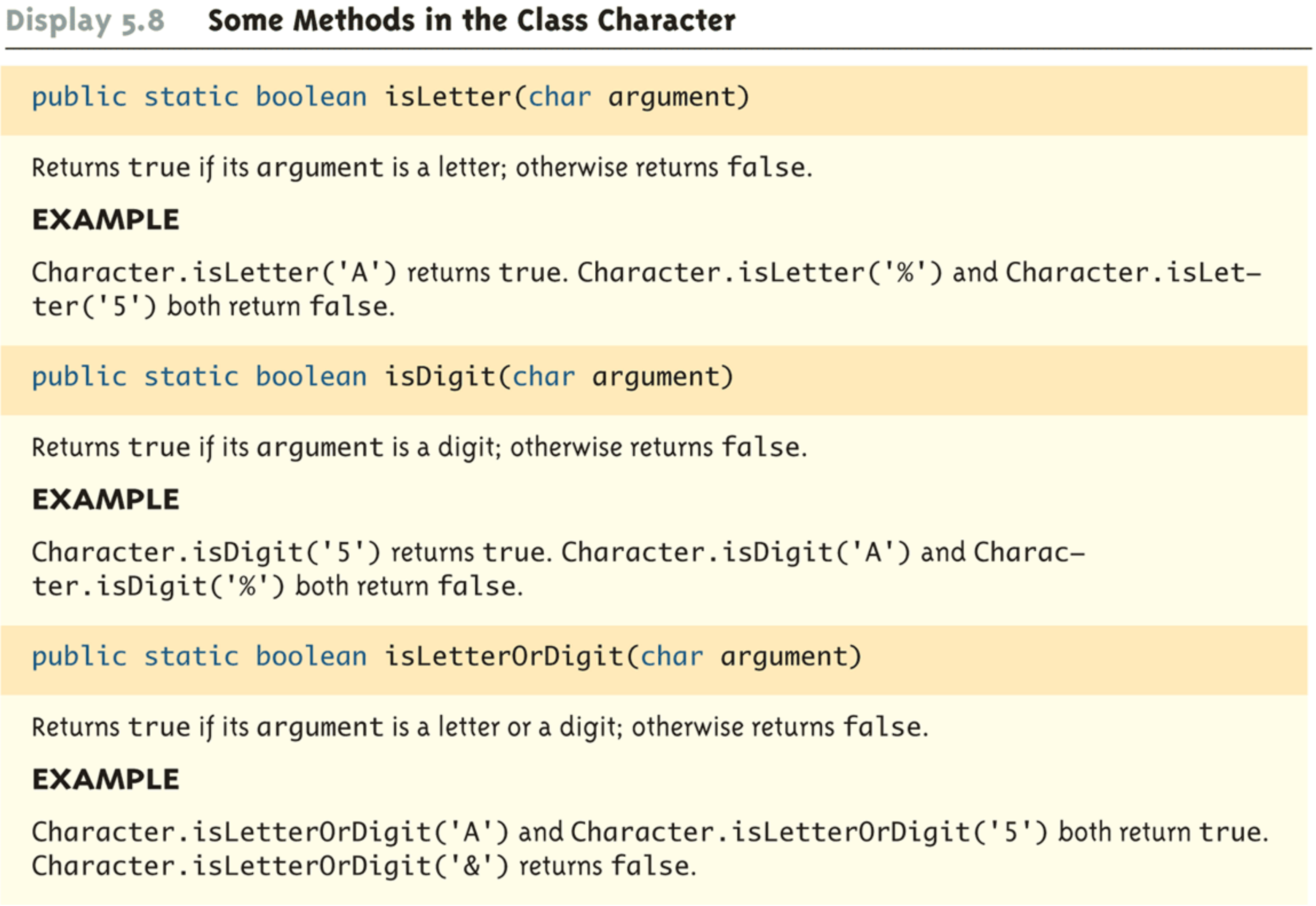
Returns true if its argument is a whitespace character; otherwise returns false. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character ('\t'), and the line break character ('\n').

EXAMPLE

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

(continued)

Some Methods in the Class **`Character`** (Part 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`.