

Lesson 4

Classes

Absolute Java

Walter Savitch

Class

- A **class** is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors.
- **Java class** objects exhibit the properties and behaviors **defined** by its **class**.
- A **class** can contain fields and methods to describe the behavior of an object.
- For example Automobile, chair, Book are some examples of a class .

Object

- **Object** refers to a particular instance of a class where the **object** can be a combination of variables, functions, and data structures.
- For example for automobile class, Car can be an instance/object.
- An Object has the similar relationship to a class that a variable has to a data type.
- An object is said to be an instance of a class, in the same way 'YOU' and 'ME' are the instances of a class 'HUMAN'.
- Class may contain data as well as methods.
- An object has both data and actions. **The actions are called methods.**
- Each object can have different data, but all objects of a class have the same types of data and all objects in a class have the same methods.

Class

Syntax for defining a class in Java :

```
public class Class_Name
{
    Instance_Variable_Declaration_1
    Instance_Variable_Declaration_2
    ...
    Instance_Variable_Declaration_Last

    Method_Definition_1
    Method_Definition_2
    ...
    Method_Definition_Last
}
```

- **public** simply means that there are no restrictions on how these instance variables are used.
- Example Code: Example

Class

```
1 public class DateFirstTry
2 {
3     public String month;
4     public int day;
5     public int year; //a four digit number.
6
7     public void writeOutput()
8     {
9         System.out.println(month + " " + day + ", " + year);
10    }
```

*This class definition goes in a file named
DateFirstTry.java.*

*Later in this chapter, we will
see that these three public
modifiers should be replaced
with private.*

Class

```
1 public class DateFirstTryDemo
2 {
3     public static void main(String[] args)
4     {
5         DateFirstTry date1, date2;
6         date1 = new DateFirstTry();
7         date2 = new DateFirstTry();
8         date1.month = "December";
9         date1.day = 31;
10        date1.year = 2012;
11        System.out.println("date1:");
12        date1.writeOutput();
13
14        date2.month = "July";
15        date2.day = 4;
16        date2.year = 1776;
17        System.out.println("date2:");
18        date2.writeOutput();
19    }
20 }
```

*This class definition (program) goes in a file named
DateFirstTryDemo.java.*

Class

- **new operator** : The new operator is used to create an object of a class and associate the object with a variable that names it.
- **Example:**
`date1 = new DateFirstTry();`
- **Note:** This code should be saved as DateFirstTry.java and compiled to create DateFirstTry.class.

Methods

- A method is a named unit of a group of statements that can be invoked from other parts of the program. Methods have:
- **<return type>**: It is the data type of the value that is returned to the calling statement after the method is executed. It can be any of the data types, available in Java.
- **<method name>**: It is the name of the method, which is given with accordance to the naming conventions used for naming an identifier. A method is called with the help of it's name.
- **method definition**: The method definition contains the method header and the method body i.e. the set of statements which are executed when the method is called.

Methods

- Syntax for method definition:

public/private <return type> <method name> (<data type> <name of parameter>, ...) *//method header*

{

statement 1;

statement 2;

.

.

// method Body

.

return (value/variable/expression); *//return statement*

}

Methods

- Example:

```
public int getYear( )  
{  
    return year;  
}
```

- Java does not have any stand-alone methods that are not in any class.
- The ***return statement*** is used to return a value to the call statement and it ends the method invocation.
- It is always the last statement in the method definition (if the method has a return type).
- The data type of the value returned should match with the return type of the method.

Example of Class

- **Bicycle.java** **//Class with more methods**

```
1 public class Bicycle {  
2  
3     // the Bicycle class has  
4     // three instance variables  
5     public int cadence;           // public is a modifier  
6     public int gear;  
7     public int speed;  
8  
9     // the Bicycle class has  
10    // one constructor  
11    public Bicycle(int startCadence, int startSpeed, int startGear) {  
12        gear = startGear;  
13        cadence = startCadence;  
14        speed = startSpeed;  
15    }
```

Example of Class

- **Bicycle.java** //Class with more methods

```
16
17 // the Bicycle class has
18 // six methods
19 public void setCadence(int newValue) {
20     cadence = newValue;
21 }
22
23 public void setGear(int newValue) {
24     gear = newValue;
25 }
26
27 public void setSpeed(int newValue) {
28     speed = newValue;
29 }
30
31 public void applyBrake(int decrement) {
32     speed -= decrement;
33 }
```

Example of Class

- Bicycle.java **//Class with more methods**

```
34
35
36  public void speedUp(int increment) { //increment is the formal parameter
37      speed += increment;
38  }
39
40  public void Display()
41  {
42      System.out.print(" Speed=" + speed);
43      System.out.print(" Cadence=" + cadence);
44      System.out.print(" Gear=" + gear);
45      System.out.println();
46  }
47
48 }
```

Example of Class

- **TriBicycle.java** **//Demo Class to test Bicycle.java**

```
1 import java. util. Scanner;
2
3 public class TryBicycle
4 {
5     public static void main(String args[])
6     {
7         Bicycle B=new Bicycle(34,80,4); //Constructor Called
8         B.Display(); // Displays the values set using Constructor
9         B.setCadence(36);
10        B.setGear(4);
11        B.setSpeed(95);
12        B.Display(); //Displays the changed values
13        B.applyBrake(20); //literal 20 is the actual parameter
14        B.Display(); //Displays the changed speed
15        B.speedUp(15);
16        B.Display(); //Displays the changed speed
17
18    }
19 }
```

Example of Class

- **Sample Output from above code:**

```
Speed=80    Cadence=34    Gear=4
Speed=95    Cadence=36    Gear=4
Speed=75    Cadence=36    Gear=4
Speed=90    Cadence=36    Gear=4
```

- You can compile a Java class before you have a program in which to use it.
- The compiled bytecode for the class will be stored in a file of the same name, but ending in .class rather than .java.
- So compiling the file Bicycle.java will create a file called Bicycle.class.
- Later, you can compile a program file that uses the class Bicycle, and you will not need to recompile the class definition for Bicycle.

Methods

- **Actual Parameters:** Values/ Variables which are used while making a call to the method are called actual parameters. In the above example , integer literal **15** is an actual parameter.
- **Formal Parameters:** The parameters mentioned in the method header are called the formal parameters. In the above class definition example , parameter **increment** is the formal parameter.

Methods

- Corresponding arguments must match the type of their corresponding formal parameter, although in some simple cases, an automatic type cast might be performed by Java.
- For example, if you plug in an argument of type `int` for a parameter of type `double`, Java automatically type casts the `int` value to a value of type `double`.
- The following list shows the type casts that Java automatically performs for you. An argument in a method invocation that is of any of these types is automatically type cast to any of the types that appear to its right, if that is needed to match a formal parameter.

`byte -> short -> int -> long -> float -> double`

- **Note:** An argument of type `char` is also converted to a matching number type, if the formal parameter is of type `int` or any type to the right of `int` in the above list of types.

Methods

- **Local Variables:** Local variables are those variables which are declared within a method or a compound statement and these variables can only be used within that method/scope .
- They cannot be accessed from outside the method or a scope of it's declaration.
- This means that we can have variables with the same names in different methods/scope.
- **Local variables are local to the method/scope in which they are declared.**
- One method's local variables have no meaning within another method.
- **Moreover, if two methods each have a local variable with the same name, they are considered two different variables.**

Methods

- In a method invocation, there must be exactly the same number of arguments in parentheses as there are formal parameters in the method definition heading.
- The first argument in the method invocation is plugged in for the first parameter in the method definition heading, the second argument in the method invocation is plugged in for the second parameter in the heading of the method definition, and so forth.

Methods

Parameters are passed by call-by-value mechanism:

- When the method is invoked, the parameter is initialized to the value of the corresponding argument in the method invocation.
- This mechanism is known as the call-by-value parameter mechanism.
- The argument in a method invocation can be a literal constant, such as 2 or 'A'; a variable; or any expression that yields a value of the appropriate type.
- This is the only kind of parameter that Java has for parameters of a primitive type.

Use of This

- Within a method definition, you can use the keyword **this** as a name for the calling object.
- If an instance variable or another method in the class is used without any calling object, then this is understood to be the calling object.
- The instance variables are understood to have <the calling object>. in front of them.
- Sometimes it is handy, and on rare occasions even necessary, to have an explicit name for the calling object.
- Inside a Java method definition, you can use the keyword **this** as a name for the calling object.

Use of This

Example:

```
public void writeOutput()  
{  
    System.out.println(<the calling object>.month + " "  
        + <the calling object>.day + ", " + <the calling object>.year);  
}
```

So, the following is a valid Java method definition that is equivalent to the one above:

```
public void writeOutput()  
{  
    System.out.println(this.month + " " + this.day + ", " + this.year);  
}
```

Methods equals() and toString()

- Java expects certain methods to be in all, or almost all, classes.
- This is because some of the standard Java libraries have software that assumes such methods are defined.
- Two of these methods are **equals** and **toString**.
- **equals**: The method **equals** is a boolean valued method to compare two objects of the class to see if they satisfy the intuitive notion of “being equal.” So, the heading should be equals
public boolean equals(Class_Name Parameter_Name)

Methods equals() and toString()

- **Example:**

public boolean equals(Person P2)

- When you use the method equals to compare two objects of the class Person, one object is the calling object and the other object is the argument, like so:

P1.equals(P2)

or equivalently,

P2.equals(P1)

- Because the method equals returns a value of type boolean, you can use an invocation of equals as the Boolean expression in an if-else statement. Similarly, you can also use it anywhere else that a Boolean expression is allowed.

Methods `equals()` and `toString()`

- **toString:** The method **toString** is a method to display the values of all instance variables of an object.
- The method **toString** should be defined so that it returns a String value that represents the data in the object.
- One nice thing about the method **toString** is that it makes it easy to output an object to the screen.
- If a class has a **toString** method, you can use an object of the class as an argument to the methods `System.out.println` and `System.out.print`.

Methods equals() and toString()

- **Example:**

```
System.out.println(date.toString());
```

- **toString** is also called automatically when the object is connected to some other string with a +, as in

```
System.out.println(date1 + " equals " + date2);
```

Methods equals() and toString()

Sample Code

// Using the Methods equals and toString

```
1  import java.util.Scanner;

2  public class DateFourthTry
3  {
4      private String month;
5      private int day;
6      private int year; //a four digit number.

7      public String toString()
8      {
9          return (month + " " + day + ", " + year);
10     }
```

Methods equals() and toString()

```
11  public void writeOutput()  
12  {  
13      System.out.println(month + " " + day + ", " + year);  
14  }  
  
15  public boolean equals(DateFourthTry otherDate)  
16  {  
17      return ( month.equals(otherDate.month) )  
18              && (day == otherDate.day) && (year == otherDate.year) );  
19  }
```

This is the method equals in the class DateFourthTry.

This is the method equals in the class String.

Methods equals() and toString()

```
20     public boolean precedes(DateFourthTry otherDate)
21     {
22         return ( (year < otherDate.year) ||
23                 (year == otherDate.year && getMonth() <
24                  otherDate.getMonth()) ||
25                 (year == otherDate.year && month.equals(otherDate.month)
26                  && day < otherDate.day) );
26     }
```

Methods equals() and toString()

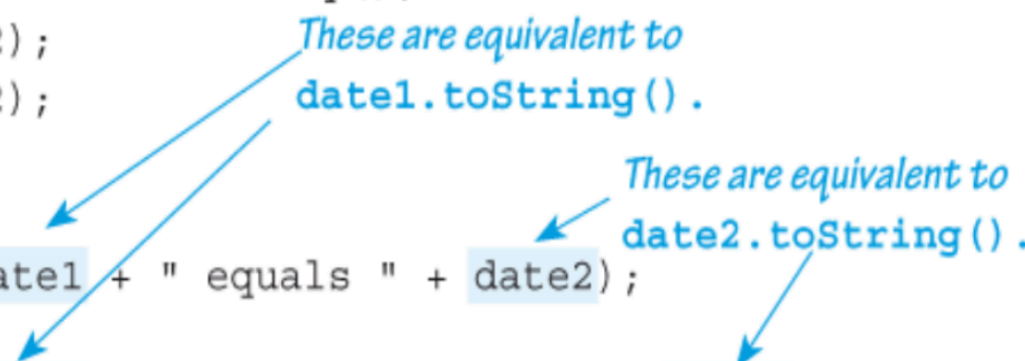
//Demo Class to check the working of equals and toString methods

```
1 public class EqualsAndToStringDemo
2 {
3     public static void main(String[] args)
4     {
5         DateFourthTry date1 = new DateFourthTry(),
6             date2 = new DateFourthTry();
7         date1.setDate(6, 17, 1882);
8         date2.setDate(6, 17, 1882);
9
10        if (date1.equals(date2))
11            System.out.println(date1 + " equals " + date2);
12        else
13            System.out.println(date1 + " does not equal " + date2);
14
15        date1.setDate(7, 28, 1750);

```

These are equivalent to date1.toString().

These are equivalent to date2.toString().



Methods equals() and toString()

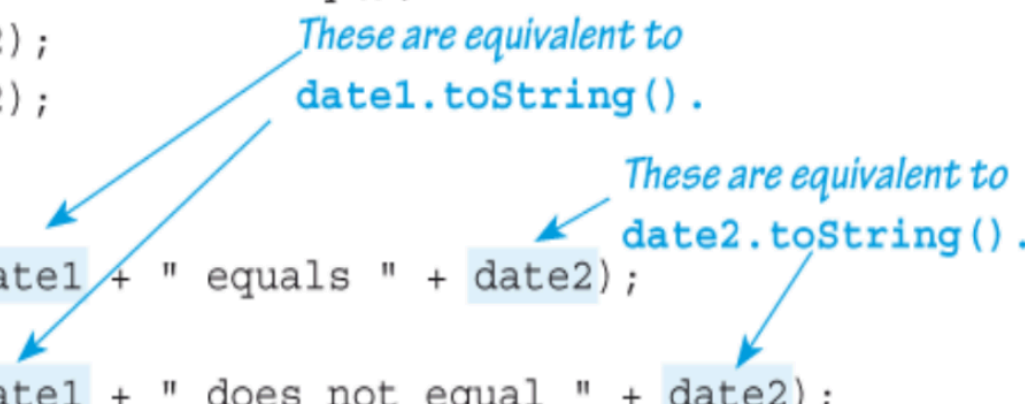
//Demo Class to check the working of equals and toString methods

```
1 public class EqualsAndToStringDemo
2 {
3     public static void main(String[] args)
4     {
5         DateFourthTry date1 = new DateFourthTry(),
6             date2 = new DateFourthTry();
7         date1.setDate(6, 17, 1882);
8         date2.setDate(6, 17, 1882);
9
10        if (date1.equals(date2))
11            System.out.println(date1 + " equals " + date2);
12        else
13            System.out.println(date1 + " does not equal " + date2);
14
15        date1.setDate(7, 28, 1750);

```

These are equivalent to date1.toString().

These are equivalent to date2.toString().



Methods equals() and toString()

```
14         if (date1.precedes(date2))
15             System.out.println(date1 + " comes before " + date2);
16         else
17             System.out.println(date2 + " comes before or is equal to "
18                                 + date1);
19     }
20 }
```

- **Sample Output**

June 17, 1882 equals June 17, 1882

July 28, 1750 comes before June 17, 1882

Constructor

- A constructor is a method in the class which is used to initialize the instance variables in the class.
- A class contains constructors that are invoked to create objects from the class blueprint.
- Constructor declarations look like method declarations—except that they use the name of the class and have no return type. It is a special method of class with the following unique features:
 - It has same name as the name of the class they belong to.
 - It has no return type.
 - It is defined as public

Constructor

- A constructor is called when you create a new object, such as with the operator new. An attempt to call a constructor in any other way, such as the following, is illegal:

birthday.Date("January", 27, 1756); *//Illegal!*

- Moreover, constructor can be overloaded.

For example, Bicycle Class has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear)  
{  
    gear = startGear;  
    cadence = startCadence;  
    speed = startSpeed;  
}
```

- You Can Invoke Another Method in a Constructor.

Constructor

- **No-Argument Constructor:** A constructor that takes no arguments is called a no-argument constructor or no-arg constructor.
- If you define a class and include absolutely no constructors of any kind, then a no-argument constructor is automatically created.
- A no-argument constructor is also known as a **default constructor**.
- **If your class definition contains absolutely no constructor definitions, then Java will automatically create a no-argument constructor.**
- **If your class definition contains one or more constructor definitions, then Java does not automatically generate any constructor;** in this case, what you define is what you get.
- Most of the classes you define should include a definition of a no-argument constructor.

Creating objects of class

- Objects of a class can be created after class definition using the following syntax:

<class name> <object name> = new Constructor(arg 1, arg 2,...);

Example:

- To create a new Bicycle object called myBike, a constructor is called by the new operator:

Bicycle myBike = new Bicycle(30, 0, 8);

This statements has three parts :

- **Declaration:** Associate a variable name with an object type.
- **Instantiation:** The new keyword is a Java operator that creates the object.
- **Initialization:** The new operator is followed by a call to a constructor, which initializes the new object.

Creating objects of class

- The **new operator** creates space in memory for the object and initializes the data members or fields.
- **For example:**
- **new Bicycle(30, 0, 8)** creates space in memory for the object and initializes its fields.
- **A class can have multiple constructors (Constructor Overloading) but they should have a unique argument list.**
- **Depending upon how many arguments are passed at object creation, respective constructor is called.**

Creating objects of class

- Although Bicycle only has one constructor, it could have others, including a no-argument constructor as follows:

```
public Bicycle() {  
    gear = 1;  
    cadence = 10;  
    speed = 0;  
}
```

Bicycle yourBike = new Bicycle();

- This statement invokes the no-argument constructor to create a new Bicycle object called yourBike.

Creating objects of class

- Both constructors could have been declared in Bicycle because they have different argument lists.
- **As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types.**
- You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.
- **You don't have to provide any constructors for your class, but you must be careful when doing this.**
- **The compiler automatically provides a no-argument, default constructor for any class without constructors.**
- **This default constructor will call the no-argument constructor of the superclass.**

Creating objects of class

Referencing an Object's Fields within a class:

- Object fields are accessed by their name.
- Fields may be accessed with their names within its own class.
- For example, we can add a statement within the Bicycle class that prints the gear and speed:

```
System.out.println("Speed and Gear are: " + speed + ", " + gear);
```

- In this case, speed and gear are simple names.

Creating objects of class

Usage of Dot Operator outside the class:

- Fields and methods of a class can be accessed outside the class using an object reference followed by the dot (.) operator, followed by a simple field/method name using the following syntax:

objectReference.fieldName

For example:

```
System.out.println("Width of rectOne: " + rectOne.width);  
System.out.println("Height of rectOne: " + rectOne.height);
```

Creating objects of class

- To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference.
- Recall that the new operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

```
int height = new Rectangle().height;
```

- This statement creates a new Rectangle object and immediately gets its height.
- In essence, the statement calculates the default height of a Rectangle.
- **Note that after this statement has been executed, the program no longer has a reference to the created Rectangle, because the program never stored the reference anywhere.**
- The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

Creating objects of class

Calling an Object's Methods

- You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

objectReference.methodName(argumentList);

or:

objectReference.methodName();

Creating objects of class

- For Example:

```
System.out.println("Area of rectOne: " + rectOne.getArea());
```

```
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

- In this case, the object that `getArea()` is invoked on is the rectangle returned by the constructor.

Access Modifiers

- Within a class definition, each instance variable declaration and each method definition, as well as the class itself, can be preceded by either **public** or **private**.
- These access modifiers specify where a class, instance variable, or method can be used.
- **Private:** The modifier **private** means that the instance variable/method cannot be accessed by name outside of the class definition. However, it can be used within the definitions of methods in its class.
- **Public:** The modifier **public** means that there are no restrictions on where the instance variable/method can be used.
- **Normally, all instance variables are private and most methods are public.**

Access Modifiers

/ Class that represents a rectangle. */**

```
public class Rectangle  
{ private int width;  
  private int height;  
  private int area;  
  
  public void setDimensions(int newWidth, int newHeight)  
  { width = newWidth;  
    height = newHeight;  
    area = width * height;  
  }  
  
  public int getArea()  
  { return area;  
  }  
}
```

Access Modifiers

- **Note: Public instance variables can lead to the corruption of an object's data.**
- **Private instance variables enable the class to restrict how they are accessed or changed.**

Example for Object creation and usage:

```
Rectangle box = new Rectangle( );  
box.setDimensions(10, 5);  
System.out.println("The area of a rectangle is " + box.getArea());
```

The output from these statements would be:

The area of a rectangle is 50

Accessor Methods

- Accessor methods allow you to look at the data in a private instance variable. It is a public method that returns data from a private instance variable and is called an accessor method, a get method, or a getter. They allow you to obtain the data. The names of accessor methods typically begin with get.

Example:

```
public int getwidth()  
{    return width;  
}
```


Mutator/Set Methods

- They allow you to change the data stored in private instance variables. A mutator method changes data in an object. It is a public method that changes the data stored in one or more private instance variables and is called a mutator method, a set method, or a setter. The names of mutator methods typically begin with set.

- **Example:**

```
public void setDimensions(int newWidth, int newHeight)  
{ width = newWidth;  
  height = newHeight;  
  area = width * height;  
}
```

Preconditions and Postconditions

- One good way to write a method comment is to break it down into two kinds of information, called the precondition and the postcondition.
- **The precondition states what is assumed to be true when the method is called.** The method should not be used and cannot be expected to perform correctly unless the precondition holds.
- **The postcondition describes the effect of the method call; that is, the postcondition tells what will be true after the method is executed in a situation in which the precondition holds.** For a method that returns a value, the postcondition describes the value returned by the method.

Preconditions and Postconditions

- Example

/**

Precondition: All instance variables of the calling object have values.

Postcondition: The data in the calling object has been written to the screen.

***/**

public void writeOutput()

- You do not need to know the definition of the method writeOutput() to use this method, if precondition and postcondition are specified.

Static Methods

- A **static method** is one that can be used without a calling object.
- With a static method, you normally use the class name in place of a calling object.
- When you define a static method, you place the keyword `static` in the heading of the definition.
- **Example:**

```
public static int maximum(int n1, int n2)
{
    if (n1 > n2)
        return n1;
    else
        return n2;
}
```

Static Methods

- Although a static method requires no calling object, it still belongs to some class, and its definition is given inside the class definition.
- When you invoke a static method, you normally use the class name in place of a calling object.
- So if the above definition of the method `maximum` were in a class named **SomeClass**, then the following is a sample invocation of **maximum()**:

```
int budget = SomeClass.maximum(yourMoney, myMoney);
```

where **yourMoney** and **myMoney** are variables of type `int` that contain some values.

Static Methods

- The method `exit` in the class `System` is also a static method. To end a program immediately, we have used the following invocation of the static method `exit`:
- **`System.exit(0);`**
- **With a static method, the class name serves the same purpose as a calling object.**
- It would be legal to create an object of the class `System` and use it to invoke the method `exit`, but that is confusing style; we usually use the class name when invoking a static method.

Static Methods

//Example Code

```
1  /**
2   Class with static methods for circles and spheres.
3   */
4   public class RoundStuff
5   {
6       public static final double PI = 3.14159;
7
8       /**
9        Return the area of a circle of the given radius.
```

Static Methods

```
10  */
11  public static double area(double radius)
12  {
13      return (PI*radius*radius) ;
14  }                                     This is the file
                                         RoundStuff.java.
15
16  /**
17   Return the volume of a sphere of the given radius.
18   */
19  public static double volume(double radius)
20  {
21      return ((4.0/3.0)*PI*radius*radius*radius) ;
22  }
23 }
```


Static Methods

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

This is the file

```
2 public class RoundStuffDemo
```

RoundStuffDemo.java.

```
3 {
```

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

```
5     {
```

```
6         Scanner keyboard = new Scanner(System.in);
```

```
7         System.out.println("Enter radius:");
```

```
8         double radius = keyboard.nextDouble();
```

```
9         System.out.println("A circle of radius"
```

```
10                                + radius + "inches");
```

```
11         System.out.println("has an area of " +
```

```
12             RoundStuff.area(radius) + " square inches.");
```

```
13         System.out.println("A sphere of radius"
```

```
14                                + radius + "inches");
```

```
15         System.out.println("has an volume of " +
```

```
16             RoundStuff.volume(radius) + "cubic inches.");
```

```
17     }
```

```
18 }
```

Static Methods

Sample Output

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.

Static Methods

- **Within the definition of a static method, you cannot do anything that refers to a calling object, such as accessing an instance variable.**
- This is because a static method can be invoked without using any calling object and so can be invoked when there are no instance variables.
- Remember instance variables belong to the calling object.
- In a static method, you cannot use the `this` parameter, either explicitly or implicitly.
- For example, the name of an instance variable by itself has an implicit **this** and a dot before it.
- **So you cannot use an instance variable in the definition of a static method.**
- In other words, since it does not need a calling object, a static method cannot refer to an instance variable of the class, nor can it invoke a non-static method of the class (unless it creates a new object of the class and uses that object as the calling object).

Static Methods

- **Invoking a Non-static Method Within a Static Method is illegal.**
- If **myMethod()** is a nonstatic (that is, ordinary) method in a class, then within the definition of any method of this class, an invocation of the form

myMethod();

means

this.myMethod();

- and so it is illegal within the definition of a static method. (A static method has no this.)
- However, it is legal to invoke a static method within the definition of another static method.

Static Methods

- There is one way that you can invoke a non-static method within a static method: if you create an object of the class and use that object (rather than this) as the calling object.
- The following is legal in a static method or any method definition:

```
MyClass anObject = new MyClass(); //The method is called with object  
anObject.myMethod();
```

- We see the same syntax utilized in method main which is a static method.

Static Methods

- **You Can Put a main in Any Class**
- So far, whenever we have used a class in the main part of a program, that main method was by itself in a different class definition within another file.
- However, sometimes it makes sense to have a main method within a regular class definition.
- The class can then be used for two purposes:
 - It can be used to create objects in other classes,
 - It can be run as a program.

Static Methods

//Example code of Class Definition with a main Added

```
1  import java.util.Scanner;
2  /**
3   Class with static methods for circles and spheres.
4   */
5  public class RoundStuff2
6  {
7      public static final double PI = 3.14159;
```

Static Methods

```
8      /**
9      Return the area of a circle of the given radius.
10     */
11     public static double area(double radius)
12     {
13         return (PI*radius*radius);
14     }
15
16     /**
17     Return the volume of a sphere of the given radius.
18     */
19     public static double volume(double radius)
20     {
21         return ((4.0/3.0)*PI*radius*radius*radius);
22     }
```


Static Methods

```
23     public static void main(String[] args)
24     {
25         Scanner keyboard = new Scanner(System.in);
26         System.out.println("Enter radius:");
27         double radius = keyboard.nextDouble();
28
29         System.out.println("A circle of radius "
30                             + radius + "inches");
31         System.out.println("has an area of " +
```

Static Methods

```
32         RoundStuff2.area(radius) + " square inches.");
33     System.out.println("A sphere of radius "
34                         + radius + "inches");
35     System.out.println("has an volume of " +
36                         RoundStuff2.volume(radius) + " cubic inches.");
37 }
38 }
```

Static Methods

//Temperature class

- In the following example, in addition to the static method **main**, the class has another static method named **toCelsius**. Note that the static method **toCelsius** can be invoked without the class name or a calling object because it is in another static method (namely **main**) in the same class. However, the nonstatic method **toString** requires an explicit calling object (**temperatureObject**).
- Java requires that a program's main method be static.
- **Thus, within a main method, you cannot invoke a nonstatic method of the same class (such as toString) unless you create an object of the class and use it as a calling object for the nonstatic method.**

Static Methods

// Example code of class with static and non static method

```
1  import java.util.Scanner;

2  /**
3   Class for a temperature (expressed in degrees Celsius).
4   */
5  public class Temperature
6  {
7      private double degrees; //Celsius

8      public Temperature()
9      {
10         degrees = 0;
11     }
```

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

Static Methods

```
12     public Temperature (double initialDegrees)
13     {
14         degrees = initialDegrees;
15     }

16     public void setDegrees (double newDegrees)
17     {
18         degrees = newDegrees;
19     }

20     public double getDegrees()
21     {
22         return degrees;
23     }
```

Static Methods

```
24     public String toString()
25     {
26         return (degrees + "C");
27     }
28
29     public boolean equals(Temperature otherTemperature)
30     {
31         return (degrees == otherTemperature.degrees);
32     }
33     /**
34      Returns number of Celsius degrees equal to
35      degreesF Fahrenheit degrees.
36      */
```

Static Methods

Check your understanding

1. Is the following legal for the class RoundStuff in example..

```
RoundStuff roundObject = new RoundStuff();  
System.out.println("A circle of radius 5.5 has area"  
                    + roundObject.area(5.5);
```

2. Can a class contain both static and nonstatic (that is, regular) methods?
3. Can you invoke a non-static method within a static method?
4. Can you invoke a static method within a nonstatic method?
5. Can you reference an instance variable within a static method? Why or why not?