

Lesson 5

Classes

Absolute Java

Walter Savitch

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

```
37 public static double toCelsius(double degreesF)
38 {
39
40     return 5*(degreesF - 32)/9;
41 }
42 public static void main(String[] args)
43 {
44     double degreesF, degreesC;
45
46     Scanner keyboard = new Scanner(System.in);
47     System.out.println("Enter degrees Fahrenheit:");
48     degreesF = keyboard.nextDouble();
49
50     degreesC = toCelsius(degreesF);
51
52     Temperature temperatureObject = new Temperature(degreesC);
53     System.out.println("Equivalent Celsius temperature is"
54         + temperatureObject.toString());
55 }
56 }
```

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

Because main is a static method, toString must have a specified calling object such as temperatureObject.

Static Methods

//Sample Output

```
Enter degrees Fahrenheit:
```

```
212
```

```
Equivalent Celsius temperature is 100.0 C
```

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?

Static Variables

- A class can have static variables as well as static methods.
- A static variable is a variable that belongs to the class as a whole and not just to one object.
- Each object has its own copies of the instance variables. However, with a static variable, there is only one copy of the variable, and all the objects can use this one variable.
- Thus, a static variable can be used by objects to communicate between the objects.
- One object can change the static variable, and another object can read that change.

Static Variables

- To make a variable static, you declare it like an instance variable but add the modifier static as follows:

- **Syntax**

```
private static Type Variable_Name;  
private static Type Variable_Name = Initial_Value;  
public static final Type Variable_Name = Constant_Value;
```

- **Examples**

```
private static String lastUser;  
private static int turn = 0;  
public static final double PI = 3.14159;  
private static int turn;
```

Static Variables

- Static variables can also be initialized.

Example:

```
private static int turn = 0;
```

- **If you do not initialize a static variable, it is automatically initialized to a default value:** Static variables of type boolean are automatically initialized to false. Static variables of other primitive types are automatically initialized to the zero of their type.

Static Variables

- **Defined constants** are a special kind of static variable:

public static final double PI = 3.14159;

- The modifier `final` in this example means that the static variable `PI` cannot be changed.
- Such defined constants are normally `public` and can be used outside the class.
- Good programming style dictates that static variables should normally be marked `private` unless they are marked `final`, that is, unless they are defined constants. The reason is the same as the reason for making instance variables `private`.

The Math Class

- The class Math provides a number of standard mathematical methods. The class Math is provided automatically and requires no import statement. **All of the methods in Math class are static, which means that you normally use the class name Math in place of a calling object.**
- The Math class is in the java.lang package, so it requires no import statement.

The Math Class

- **pow()**: Returns base to the power exponent.

public static double pow(double base, double exponent)

- **Example**

Math.pow(2.0,3.0) returns 8.0.

The Math Class

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

public static double abs(double argument)

public static float abs(float argument)

public static long abs(long argument)

public static int abs(int argument)

- **Example**

Math.abs(-6) and Math.abs(6) both return 6.

Math.abs(-5.5) and Math.abs(5.5) both return 5.5.

The Math Class

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

```
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)
```

- **Example :**

Math.min(3, 2) returns 2.

The Math Class

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

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)

- **Example :**

Math.max(3, 2) returns 3.

The Math Class

- **round()**: Rounds its argument.

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

- **Example**

Math.round(3.2) returns 3;

Math.round(3.6) returns 4.

The Math Class

- **ceil():** Returns the smallest whole number greater than or equal to the argument.

public static double ceil(double argument)

- **Example**

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

- **floor():** Returns the largest whole number less than or equal to the argument.

public static double floor(double argument)

- **Example**

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

The Math Class

- **ceil():** Returns the smallest whole number greater than or equal to the argument.

public static double ceil(double argument)

- **Example**

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

- **floor():** Returns the largest whole number less than or equal to the argument.

public static double floor(double argument)

- **Example**

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

The Math Class

- **sqrt()**: Returns the square root of its argument.

public static double sqrt(double argument)

- **Example**

Math.sqrt(4) returns 2.0.

- **random()**: Returns a random number greater than or equal to 0.0 and less than 1.0.

public static double random()

- **Example**

Math.random() returns 0.5505562535943004 (sample number; returns a pseudorandom number that is less than 1 and greater than or equal to 0).

The Math Class

- **Math.PI** : Constant PI is defined in the class Math, it must have the class name Math and a dot before them.
- **Example:**

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

The Math Class

Check your understanding:

- What values are returned by each of the following?
- `Math.round(3.2)`
- `Math.round(3.6)`
- `Math.floor(3.2)`
- `Math.floor(3.6)`
- `Math.ceil(3.2)`
- `Math.ceil(3.6)`

Wrapper Classes

Java treats the primitive types, such as `int` and `double`, differently from the class types, such as the class `String` and the programmer-defined classes.

At times, you may find yourself in a situation where you want to use a value of a primitive type but you want or need the value to be an object of a class type.

Wrapper classes provide a class type corresponding to each of the primitive types so that you can have an object of a class type that behaves somewhat like its corresponding value of a primitive type.

Wrapper Classes

- **Boxing:** The process of going from a value of a primitive type to an object of its wrapper class is sometimes called boxing.
- To convert a value of a primitive type to an “equivalent” value of a class type, you create an object of the corresponding wrapper class using the primitive type value as an argument to the wrapper class constructor. The wrapper class for the primitive type `int` is the predefined class `Integer`.
- **Example:** If you want to convert an `int` value, such as 42, to an object of type `Integer`, you can do so as follows:

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

The variable `integerObject` now names an object of the class `Integer` that corresponds to the `int` value 42. The object `integerObject` does, in fact, have the `int` value 42 stored in an instance variable of the object `integerObject`.

Wrapper Classes

- More examples of boxing:

`Integer numberOfSamuri = new Integer(47);`

`Double price = new Double(499.99);`

`Character grade = new Character('A');`

Wrapper Classes

- **Unboxing:** The process of going from an object of a wrapper class to the corresponding value of a primitive type is called unboxing,
- To go from an object of type Integer to the corresponding int value, you can do the following:

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

- The method intValue() recovers the corresponding int value from an object of type Integer.

Wrapper Classes

- The wrapper classes for the primitive types byte, short, long, float, double, and char are Byte, Short, Long, Float, Double, and Character, respectively.
- The methods for converting from the wrapper class object to the corresponding primitive type are intValue for the class Integer, byteValue for the class Byte, shortValue for the class Short, longValue for the class Long, floatValue for the class Float, doubleValue for the class Double, and charValue for the class Character.

Wrapper Classes

- Starting with version 5.0, Java will automatically do this boxing, so the previous three assignments can be written in the following equivalent, but simpler, forms:

Integer numberOfSamuri = 47;

Double price = 499.99;

Character grade = 'A';

- This is an automatic type cast.

Wrapper Classes

- Unboxing is also done automatically in Java (starting in version 5.0).
The following are examples of automatic unboxing:

```
Integer numberOfSamuri = new Integer(47);
```

```
int n = numberOfSamuri;
```

```
Double price = new Double(499.99);
```

```
double d = price;
```

```
Character grade = new Character('A');
```

```
char c = grade;
```

Wrapper Classes

- Java automatically applies the appropriate accessor method (intValue, doubleValue, or charValue in these cases) to obtain the value of the primitive type that is assigned to the variable.
- So the previous examples of automatic unboxing are equivalent to the following code, which is what you had to write in older versions of Java that did not do automatic unboxing:

```
Integer numberOfSamuri = new Integer(47);  
int n = numberOfSamuri.intValue();  
Double price = new Double(499.99);  
double d = price.doubleValue();  
Character grade = new Character('A');  
char c = grade.charValue();
```

Wrapper Classes

- Code can often involve a combination of automatic boxing and unboxing. For example, consider the following code, which uses both automatic boxing and automatic unboxing:

```
Double price = 19.90;  
price = price + 5.12;
```

- This code is equivalent to the following, which is what you had to write in older versions of Java that did not do automatic boxing and unboxing:

```
Double price = new Double(19.90);  
price = new Double(price.doubleValue() + 5.12);
```

Wrapper Classes

Check your understanding

1. Which of the following are legal?

```
Integer n = new Integer(42);  
int m = 42;  
n = m;  
m = n;
```

2. In the following, is the value of the variable price after the assignment statement an object of the class Double or a value of the primitive type double?

```
Double price = 1.99;
```

3. In the following, is the value of the variable count after the assignment statement an object of the class Integer or a value of the primitive type int?

```
int count = new Integer(12);
```


Wrapper Classes

- The wrapper classes contain a number of useful constants and static methods. So, wrapper classes have two distinct personalities:
 - One is their ability to produce class objects corresponding to values of primitive types.
 - The other is as a repository of useful constants and methods.
- **Largest and smallest value of primitive types:** The associated wrapper class can be used to find the value of the largest and smallest values of any of the primitive number types.
- For example, the largest and smallest values of type int are **Integer.MAX_VALUE** and **Integer.MIN_VALUE**
- The largest and smallest values of type double are **Double.MAX_VALUE** and **Double.MIN_VALUE**

Wrapper Classes

- The wrapper classes contain a number of useful constants and static methods. So, wrapper classes have two distinct personalities:
 - One is their ability to produce class objects corresponding to values of primitive types.
 - The other is as a repository of useful constants and methods.
- **Largest and smallest value of primitive types:** The associated wrapper class can be used to find the value of the largest and smallest values of any of the primitive number types.
- For example, the largest and smallest values of type int are **Integer.MAX_VALUE** and **Integer.MIN_VALUE**
- The largest and smallest values of type double are **Double.MAX_VALUE** and **Double.MIN_VALUE**

Wrapper Classes

- Wrapper classes have static methods that can be used to convert back and forth between string representations of numbers and the corresponding number of type int, double, long, or float.
- For example, the static method **parseDouble** of the wrapper class Double converts a string to a value of type double.

Double.parseDouble("199.98") //returns the double value 199.98

Double.parseDouble(theString.trim())

//trims the leading or trailing blanks, and then converts theString to
//double

Wrapper Classes

- The method `trim` is a method in the class `String` that trims off leading and trailing whitespace, such as blanks.
- Similarly, the static methods `Integer.parseInt`, `Long.parseLong`, and `Float.parseFloat` convert from string representations to numbers of the corresponding primitive types `int`, `long`, and `float`, respectively.
- Each of the numeric wrapper classes also has a static method called `toString` that converts in the other direction, from a numeric value to a string representation of the numeric value.
- Example

`Double.toString(123.99)` //returns the string value "123.99"

Wrapper Classes

- **Character:** It is a wrapper class for the primitive type char, contains a number of static methods that are useful for string processing. The class Character is in the java.lang package, so it requires no import statement.
- Some of the methods are:
- **public static char toUpperCase(char argument)**
- **public static char toLowerCase(char argument)**
- **public static boolean isUpperCase(char argument)**
- **public static boolean isLowerCase(char argument)**
- **public static boolean isWhitespace(char argument)**
- **public static boolean isLetter(char argument)**
- **public static boolean isDigit(char argument)**
- **public static boolean isLetterOrDigit(char argument)**

Wrapper Classes

Boolean: There is also a wrapper class Boolean corresponding to the primitive type boolean.

It has names for two constants of type Boolean: Boolean.TRUE and Boolean.FALSE, which are the Boolean objects corresponding to the values true and false of the primitive type boolean.

Wrapper Classes

Example Code

```
1 import java.util.Scanner;
2 /**
3  Illustrate the use of a static method from the class Character.
4  */
5
6 public class StringProcessor
7 {
8     public static void main (String[] args)
9     {
10         System.out.println("Enter a one line sentence:");
11         Scanner keyboard = new Scanner(System.in);
12         String sentence = keyboard.nextLine();
13     }
```

Wrapper Classes

```
14     sentence = sentence.toLowerCase();
15     char firstCharacter = sentence.charAt(0);
16     sentence = Character.toUpperCase(firstCharacter)
17         + sentence.substring(1);
18
19     System.out.println("The revised sentence is:");
20     System.out.println(sentence);
21 }
22 }
```


Wrapper Classes

Sample Output

Enter a one line sentence:

is you is OR is you ain't my BABY?

The revised sentence is:

Is you is or is you ain't my baby?

References and Class Parameters

- Variables of a class type and variables of a primitive type behave quite differently in Java.
- Variables of a primitive type name their values in a straightforward way. For example, if `n` is an `int` variable, then `n` can contain a value of type `int`, such as 42.
- If `v` is a variable of a class type, then `v` does not directly contain an object of its class.
- Instead, `v` names an object by containing the memory address of where the object is located in memory.

References and Class Parameters

- For a variable of a primitive type, the value of the variable is stored in the memory location assigned to the variable.
- However, a variable of a class type stores only the memory address of where an object is located. The object named by the variable is stored in some other location in memory, and the variable contains only the memory address of where the object is stored. This memory address is called a [reference](#) .

References and Class Parameters

- **Reasoning:** Variables of a primitive type and variables of a class type are different for a reason. A value of a primitive type, such as the type `int`, always requires the same amount of memory to store one value. There is a maximum value of type `int`, so values of type `int` have a limit on their size. However, an object of a class type, such as an object of the class `String`, might be of any size. The memory location for a variable of type `String` is of a fixed size, so it cannot store an arbitrarily long string. It can, however, store the address of any string because there is a limit to the size of an address.
- Because variables of a class type contain a reference (memory address), two variables may contain the same reference, and in such a situation, both variables name the same object. Any change to the object named by one of these variables will produce a change to the object named by the other variable, because they are the same object.

References and Class Parameters

```
1  public class ToyClass
2  {
3      private String name;
4      private int number;

5      public ToyClass(String initialName, int initialNumber)
6      {
7          name = initialName;
8          number = initialNumber;
9      }

10     public ToyClass()
11     {
12         name = "No name yet.";
13         number = 0;
14     }
```

References and Class Parameters

```
15      public void set(String newName, int newNumber)
16      {
17          name = newName;
18          number = newNumber;
19      }
```

References and Class Parameters

```
20     public String toString()
21     {
22         return (name + " " + number);
23     }
24     public static void changer(ToyClass aParameter)
25     {
26         aParameter.name = "Hot Shot";
27         aParameter.number = 42;
28     }

29     public boolean equals(ToyClass otherObject)
30     {
31         return ((name.equals(otherObject.name))
32                 && (number == otherObject.number) );
33     }
34 }
```

References and Class Parameters

Test Code

```
ToyClass variable1 = new ToyClass("Joe", 42);  
ToyClass variable2;  
variable2 = variable1; //Now both variables name the same  
                        //object.  
variable2.set("Josephine", 1);  
System.out.println(variable1); //Invokes variable1's toString  
                                //method
```

Output

Josephine 1

The object named by **variable1** has been changed without ever using the name **variable1**.

References and Class Parameters

Class Parameters

- When a method has a parameter of a class type, the value plugged in is a reference (memory address). So, the parameter becomes another name for the argument, and any change made to the object named by the parameter is made to the object named by the argument, because they are the same object. **Thus, a method can change the instance variables of an object given as an argument.**

References and Class Parameters

```
1 public class ClassParameterDemo
2 {
3     public static void main(String[] args)
4     {
5         ToyClass anObject = new ToyClass("Mr. Cellophane", 0);
6         System.out.println(anObject);
7         System.out.println(
8             "Now we call changer with anObject as argument.");
9         ToyClass.changer(anObject);
10        System.out.println(anObject);
11    }
12 }
```

ToyClass is defined in Display 5.11.

*Notice that the method **changer** changed the instance variables in the object **anObject**.*

Sample Dialogue

```
Mr. Cellophane 0
Now we call changer with anObject as argument.
Hot Shot 42
```

References and Class Parameters

- **== with objects:** The operator == with objects does not check that the objects have the same values for their instance variables.
- It merely checks for equality of memory addresses, so two objects in two different locations in memory would test as being “not equal” when compared using ==, even if their instance variables contain equivalent data.

References and Class Parameters

```
ToyClass2 variable1 = new ToyClass2("Chiana", 3),  
            variable2 = new ToyClass2("Chiana", 3);  
if (variable1 == variable2)  
    System.out.println("Equal using ==");  
else  
    System.out.println("Not equal using ==");
```

Output

```
Not equal using ==
```

- Even though these two variables name objects that are intuitively equal, they are stored in two different locations in the computer's memory. This is why we usually use an equals method to compare objects of a class type. **The variables variable1 and variable2 would be considered "equal" if compared using the equals method as defined for the class ToyClass2.**

References and Class Parameters

- **The Constant null:** The constant null is a special constant that may be assigned to a variable of any class type. It is used to indicate that the variable has no “real value.”
- **Example:**
`YourClass yourObject = null;`
- It is also common to use null in constructors to initialize instance variables of a class type when there is no obvious object to use.
- Null is not an object. It is like a reference (memory address) that does not refer to any object (does not name any memory location).
- To test whether a class variable contains null, use == or !=:
- **Example:**
`if (yourObject == null)
 System.out.println("No real object here.");`

References and Class Parameters

- **Null Pointer Exception:** You cannot invoke a method using a variable that is initialized to null. If you try, you will get an error message that says “Null Pointer Exception.”

- **Example:**

```
ToyClass2 aVariable = null;
```

```
String representation = aVariable.toString();
```

```
//Generates Null Pointer Exception
```

- **Note:** Any time you get a “Null Pointer Exception,” look for an uninitialized class variable.

References and Class Parameters

- **The new Operator and Anonymous Objects:** There are times when you create an object using new and use the object as an argument to a method, but then never again use the object. In such cases, you need not give the object a variable name. You can instead use the expression with the new operator and the constructor directly as the argument.

- **Example:**

```
if (variable1.equals(new ToyClass("JOE", 42)))  
    System.out.println("Equal");  
else  
    System.out.println("Not equal");
```

- This is equivalent to the following:

```
ToyClass temp = new ToyClass("JOE", 42);  
if (variable1.equals(temp))  
    System.out.println("Equal");  
else  
    System.out.println("Not equal");
```

References and Class Parameters

- **Anonymous Objects:** An expression with a new operator and a constructor creates a new object and returns a reference to the object. If this reference is not assigned to a variable, but instead the expression with new and the constructor is used as an argument to some method, then the object produced is called an anonymous object.
- **Example**

```
if (variable1.equals(new ToyClass("JOE", 42)))  
    System.out.println("Equal");  
else  
    System.out.println("Not equal");
```


References and Class Parameters

- **null Can Be an Argument to a Method**
- If a method has a parameter of a class type, then **null** can be used as the corresponding argument when the method is invoked.

```
private static boolean consistent(Date birthDate, Date deathDate)
{
    if (birthDate == null)
        return false;
    else if (deathDate == null)
        return true;
    else
        return (birthDate.precedes(deathDate)
                || birthDate.equals(deathDate));
}
```

References and Class Parameters

- **Note:** The calling object cannot be null.
- **Example:** In the following statement
`object1.equals(object2)`
- The calling object object1 must be a true object of a class;
- a calling object cannot be null.
- However, the argument object2 can be either a true object or null. If the argument is null, then equals should return false, because a true object cannot reasonably be considered to be equal to null.

Copy Constructor

- It is a constructor with a single argument of the same type as the class. The copy constructor should create an object that is a separate, independent object but with the instance variables set so that it is an exact copy of the argument object.

Copy Constructor


```
1 public class Date
2 {
3     private String month;
4     private int day;
5     private int year; //A four digit number.

6     public Date(String monthString, int day, int year)
7     {
8         setDate (monthString, day, year);
9     }

10    public Date(Date aDate)
11    {
12        if (aDate == null) //Not a real date.
13        {
14            System.out.println("Fatal Error.");
15            System.exit(0);
16        }

17        month = aDate.month;
18        day = aDate.day;
19        year = aDate.year;
20    }
```

*This is not a complete definition of the class Date.
The complete definition of the class Date is in Display 4.11,
but this has the details that are important to what we are
discussing in this chapter.*

Copy constructor 

Copy Constructor

```
21  public void setDate(String monthString, int day, int year)
22  {
23      if (dateOK(monthString, day, year))
24      {
25          this.month = monthString;
26          this.day = day;
27          this.year = year;
28      }
29      else
30      {
31          System.out.println("Fatal Error");
32          System.exit(0);
33      }
34  }
35  public void setYear(int year)
36  {
37      if ( (year < 1000) || (year > 9999) )
38      {
```

The method dateOK checks that the date is a legitimate date, such as not having more than 31 days.

Copy Constructor

After this code is executed, both `date1` and `date2` represent the date January 1, 2015, but they are two different objects. So, if we change one of these objects, it will not change the other.

```
Date date1 = new Date("January", 1, 2015);  
Date date2 = new Date(date1);
```

Copy Constructor

```
1  /**
2   Class for a person with a name and dates for birth and death.
3   Class invariant: A Person always has a date of birth, and if the Person
4   has a date of death, then the date of death is equal to or later than the
5   date of birth.
6  */
7  public class Person
8  {
9      private String name;
10     private Date born;
11     private Date died; //null indicates still alive.

12     public Person(String initialName, Date birthDate, Date deathDate)
13     {
14         if (consistent(birthDate, deathDate))
15         {
16             name = initialName;
17             born = new Date(birthDate);
18             if (deathDate == null)
19                 died = null;
20             else
21                 died = new Date(deathDate);
22         }
23         else
24         {
25             System.out.println("Inconsistent dates.Aborting.");
26             System.exit(0);
27         }
28     }
}
```

The class `Date` was defined in Display 4.11 and many of the details are repeated in Display 5.20.

We will discuss `Date` and the significance of these constructor invocations later in this chapter in the subsection entitled "Copy Constructors."

Copy Constructor

```
29 public Person(Person original)
30 {
31     if (original == null )
32     {
33         System.out.println("Fatal error.");
34         System.exit(0);
35     }
36
37     name = original.name;
38     born = new Date(original.born);
39
40     if (original.died == null)
41         died = null;
42     else
43         died = new Date(original.died);
44 }
```

Copy constructor



Copy Constructor

```
1 public class PersonDemo
2 {
3     public static void main(String[]args)
4     {
5         Person bach =
6             new Person("Johann Sebastian Bach",
7                 new Date("March", 21, 1685), new Date("July", 28, 1750));
8         Person stravinsky =
9             new Person("Igor Stravinsky",
10                 new Date("June", 17, 1882), new Date("April", 6, 1971));
11         Person adams =
12             new Person("John Adams",
13                 new Date("February", 15, 1947), null );
14         System.out.println("A Short List of Composers:");
15         System.out.println(bach);
16         System.out.println(stravinsky);
17         System.out.println(adams);
18         Person bachTwin = new Person(bach); //Copy Constructor
```

Copy Constructor

```
19         System.out.println("Comparing bach and bachTwin:");
20         if (bachTwin == bach)
21             System.out.println("Same reference for both.");
22         else
23             System.out.println("Distinct copies.");
24         if (bachTwin.equals(bach))
25             System.out.println("Same data.");
26         else
27             System.out.println("Not same data.");
28     }
29 }
```

Copy Constructor

Sample Output

A Short List of Composers:

Johann Sebastian Bach, March 21, 1685–July 28, 1750

Igor Stravinsky, June 17, 1882–April 6, 1971

John Adams, February 15, 1947–

Comparing bach and bachTwin:

Distinct copies.

Same data.

Copy Constructor

Example of wrong practice

Now let's consider the copy constructor for the class `Person`

```
public Person(Person original)
{
    if (original == null)
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = new Date(original.born);
    if (original.died == null)
        died = null;
    else
        died = new Date(original.died);
}
```

Copy Constructor

We want the object created to be an independent copy of **original**. That would not happen if we had used the following instead:

```
public Person(Person original) //Unsafe
{
    if (original == null )
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = original.born; //Not good.
    died = original.died; //Not good.
}
```

Copy Constructor

If we had used the unsafe version of the copy constructor, the “Not good.” code simply copies references from `original.born` and `original.died` to the corresponding arguments of the object being created by the constructor. So, the object created is not an independent copy of the original object.

Example

```
Person original =  
    new Person("Natalie Dressed",  
        new Date("April", 1, 1984), null);  
Person copy = new Person(original);  
copy.setBirthYear(1800);  
System.out.println(original);
```

Output

```
Natalie Dressed, April 1, 1800-
```

Copy Constructor

- When we changed the birth year in the object copy, we also changed the birth year in the object original. This is because we are using our unsafe version of the copy constructor. Both `original.born` and `copy.born` contain the same reference to the same `Date` object.
- Safer version of the copy constructor that sets the born instance variables:
`born = new Date(original.born);`

which is equivalent to

`this.born = new Date(original.born);`

- This version, which we did use, makes the instance variable `this.born` an independent `Date` object that represents the same date as `original.born`. So if you change a date in the `Person` object created by the copy constructor, you will not change that date in the original `Person` object.

Copy Constructor

- **Note that if a class, such as Person, has instance variables of a class type, such as the instance variables born and died, then to define a correct copy constructor for the class Person, you must already have copy constructors for the class Date of the instance variables. The easiest way to ensure this for all your classes is to always include a copy constructor in every class you define.**

Copy Constructor

Leaking Accessor methods:

```
public Date getBirthDate()  
{  
    return new Date(born);  
}
```

Do not make the mistake of defining the accessor method as follows:

```
public Date getBirthDate() //Unsafe  
{  
    return born; //Not good  
}
```