Basics of the Object Oriented Programming II

Associate Professor Viorica Rozina Chifu
viorica.chifu@cs.utcluj.ro

Comments and embedded documentation

- There are two types of comments in Java
  - First type
    - Is the traditional C style comment
    - Begin with a ***/**** and continue, possibly across many lines, until a ****/***
    - Example: /* This is a comment * that continues * across lines*/
  - Second type
    - Comes from C++
    - It is the single-line comment
    - Starts at a ***//*** and continues until the end of the line
    - Example: // This is a one-line comment

Comment documentation

- Javadoc
  - Is a documentation generator for Java source code
  - It was created by Sun Microsystems (now Oracle Corporation) and is part of the Java Development Kit (JDK)
  - Javadoc reads special comments in Java code and generates a set of HTML pages that document the code
  - The tool to extract the comments and generate the documentation is called *javadoc*, and it is included in the JDK installation.
- Javadoc comments can be placed above any class, method, or field declaration in the Java source code, and they are used to provide documentation for that code.
- Javadoc comments typically have two sections:
  - a description of what is being documented, and
  - a set of standalone block tags marked with the "@" symbol that provide additional information or metadata about the code.

Overview of the basics of **javadoc**

- Javadoc commands or tags occur only within /*...*/ comments, which are also known as block comments in Java.
- These comments must start with /** and end with */.
- **Standalone doc tags**
  - Start with a ‘@’ and are placed at the beginning of a comment line

Overview of the basics of **javadoc**

- **@see**
  - **Form**: **@see** *classname,* where classname is the name of the class that the documentation should link to
  - **Used Where:** Any item being commented
  - **Used For:** If we want to refer a class documentation from other class in order to provide more clarity; in this case we use **see** tag to provide a link to that class
    - » Allow to refer to the documentation in other classes
  - Javadoc will generate HTML with the **@see** tags hyperlinked to the other documentation

```java
/**
 * This class represents a rectangle with a given width and height.
 *
 * @see Shape
 */
public class Rectangle extends Shape {
    // implementation details
}
```

Overview of the basics of **javadoc**

- {**@docRoot**}
  - **Form**: {**@docRoot**}
  - **Used Where:** Interface and Class comments
  - **Used For:** Represents the relative path to the generated document's root directory from any generated page; This tag can be used in interface and class comments to provide links to other parts of the documentation

```
/**
 * This class represents a point in two-dimensional space.
 *
 * For more information, see the documentation at {@docRoot}/com/example/Point.html
 */
public class Point {
    // implementation details
}
```

Overview of the basics of *javadoc* - Some example tags

- **@version**
  - **Form:** **@version** *version-information*
  - **Used Where:** Interface and Class comments
  - **Used For:** Describes the current version number of the source code. Is often just a version number, but can also include other information like the date or release name.

```java
/**
 * This class represents a person with a name and an age.
 *
 * @version 1.0
 */
public class Person {
    // implementation details
}
```

In this example, the @version tag is used to indicate that the Person class is part of version 1.0 of the code

Overview of the basics of *javadoc* - Some example tags

- **@author**
  - **Form: @author** *author-information*
    - » *author-information:*
      - Your *name*
      - *Email address* or any other appropriate information
    - » You can have multiple **author** tags for a list of authors, but they must be placed consecutively
  - **Used Where:** Interface and Class comments
  - **Used For:** Indicate the author or authors of the source code being documented.

```java
/**
 * This class represents a person with a name and an age.
 *
 * @author John Smith
 * @version 1.0
 */
public class Person {
    // implementation details
}
```

In this example, the @author tag is used to indicate that the Person class was authored by John Smith. This helps developers understand who created the code and can be especially helpful in collaborative projects where multiple developers may be contributing to the codebase.

Overview of the basics of *javadoc* - Some example tags

- **@since**
  - **Form:** @**since** version
  - **Used Where:** Interface and Class comments to provide information about the history of the code being documented
  - **Used For:** Is used to indicate the version of the source code in which the item being documented was introduced. It is usually just a version number but may also contain a specific date
  - By indicating the version in which an item was introduced, developers can help other developers understand the evolution of the code and how it has changed over time.

```java
/**
 * This class represents a person with a name and an age.
 *
 * @since 1.0
 */
public class Person {
    // implementation details
}
```

Overview of the basics of *javadoc* - Some example tags

- **@deprecated**
  - **Form: @deprecated**
  - **Used Where:** Interface, class and method comments
  - **Used For:** Is used to indicate that an item (class, method, field, etc.) is deprecated (i.e., a member of the deprecated API) and should no longer be used.
  - Deprecated items are included in the API for backwards compatibility but should not be used in new code. Instead, developers should use the recommended alternative, which is usually indicated in the Javadoc comment using the @deprecated tag.

```java
/**
 * This method calculates the square of a number.
 *
 * @param x the number to square
 * @return the square of x
 * @deprecated This method is deprecated and should not be used. Use Math.pow(x, 2)
 */
@Deprecated
public static int square(int x) {
    return x * x;
}
```

Overview of the basics of *javadoc* - Some example tags

- **@ param**
  - **Form: @param** *parameter-name description*
    - » *parameter-name* - is the identifier of the parameter in the method signature
    - » *description* - is a description of the parameter. The description can include multiple lines, and it is considered finished when a new Javadoc tag is encountered.
  - **Used Where:** methods comments
  - **Used For:** Used for method documentation
  - You can have any number of **param** tags presumably one for each parameter

```java
/**
 * Adds two numbers and returns the result.
 *
 * @param num1 the first number to be added
 * @param num2 the second number to be added
 * @return the sum of num1 and num2
 */
public int add(int num1, int num2) {
    return num1 + num2;
}
```

In this example, the @param tag is used twice to document the parameters num1 and num2 of the add method. By including a description of each parameter, developers reading the Javadoc can better understand how to use the method and what each parameter does.

Overview of the basics of *javadoc* - Some example tags

- **@return**
  - **Form:** @**return** description
    - » *description* is a description of the value that the method returns
  - **Used Where:** Methods comments
  - **Used For:** Is used to document the return value of a method.

```java
/**
 * Returns the sum of two numbers.
 *
 * @param num1 the first number to be added
 * @param num2 the second number to be added
 * @return the sum of num1 and num2
 */
public int add(int num1, int num2) {
    return num1 + num2;
}
```

- In this example, the @return tag is used to document the return value of the add method, which returns the sum of num1 and num2.
- By including a description of the return value, developers reading the Javadoc can better understand what the method does and what value it returns.
- It's worth noting that the @return tag is not used for methods that return void or constructors, as they do not return a value.

Overview of the basics of *javadoc* - Some example tags

- **@throws**
  - **Form:** **@throws** exception description
    - » *exception* the name of the exception class that the method might throw
    - » *description* is a description of why the exception might be thrown
  - **Used Where:** Methods comments
  - **Used For:** To document exceptions that a method might throw
  - By including this information in the method comments, it helps other developers understand the potential errors that may occur and how to handle them appropriately.

```java
/**
 * Reads the contents of a file and returns it as a string.
 *
 * @param filePath the path of the file to read
 * @return the contents of the file
 * @throws FileNotFoundException if the file does not exist
 */
public String readFile(String filePath) throws FileNotFoundException {
    File file = new File(filePath);
    Scanner scanner = new Scanner(file);
    String content = scanner.useDelimiter("\\Z").next();
    scanner.close();
    return content;
}
```

In this example, we use the "@throws" tag to document that the method might throw a "FileNotFoundException" if the specified file does not exist. This information can help other developers understand the potential errors that may occur when calling this method and how to handle them appropriately.

Documentation example

```
/** This class represents a bank account.
A bank account has a balance, which can be increased or
    decreased by depositing or withdrawing money.
It also has an owner, which is identified by their name and account
    number.
This class provides methods to get and set the balance, deposit and
    withdraw money, and get the owner's name and account number.
 */
public class BankAccount {
      private double balance;
   private String ownerName;
   private int accountNumber;

   /**Constructs a new bank account with the specified owner name
    and account number. The initial balance is set to 0.
     @param ownerName the name of the account owner
     @param accountNumber the account number
    */
   public BankAccount(String ownerName, int accountNumber) {
      this.ownerName = ownerName;
      this.accountNumber = accountNumber;
      this.balance = 0.0;
   }
   /**Gets the balance of this bank account.
     @return the current balance
    */
   public double getBalance() {
      return balance;
   }

   /*** Sets the balance of this bank account.
     @param balance the new balance
    ***/
   public void setBalance(double balance) {
      this.balance = balance;
   }

   /** Deposits the specified amount of money into this bank account.
    @param amount the amount to deposit
    ***/
   public void deposit(double amount) {
      balance += amount;
   }

   /*** Withdraws the specified amount of money from this bank account.
      @param amount the amount to withdraw
      @throws InsufficientFundsException if the account balance is not enough to
withdraw the specified amount
    */
   public void withdraw(double amount) throws InsufficientFundsException {
      if (balance < amount) {
         throw new InsufficientFundsException("Account balance is not enough to
withdraw " + amount);
      }
      balance -= amount;
   }
```

Using Java operators

- Operator
  - Takes one or more arguments and produces a new value
  - Almost all operators work only with primitives
    - The exceptions are ‘=’, ‘==’ and ‘!=’, which work with all objects
  - In addition, the String class supports ‘+’
- Precedence Operator
  - Defines how an expression evaluates when several operators are present
  - Java has specific rules that determine the order of evaluation
    - Multiplication and division are performed before addition and subtraction
  - Parentheses can be used to make the order of evaluation explicit

```
a = x + (y - 2)/(2 + z);
```

Using Java operators

- Assignment
  - Is performed with the operator =
  - Means “take the value of the right-hand side and copy it into the left-hand side”
    - »You can assign a constant value to a variable:

      a = 4;

    - »You cannot assign anything to constant value

      (You can’t say 4 = a;)
  - When you assign an object to another object a reference from one place to another is copied
    - »If you say *c* = *d* for objects, you end up with both *c* and *d* pointing to the object that, originally, only *d* pointed to

Using Java operators

- Mathematical operators
  - Addition (+)
  - Subtraction (-)
  - Division (/)
  - Multiplication (*)
  - Modulus (%, which produces the remainder from integer division)
  - Java also uses a shorthand notation to perform an operation and an assignment at the same time
    - This is denoted by an operator followed by an equal sign
    - E.g., to add 4 to the variable x and assign the result to x, use: x += 4 ⇔ x=x+4;

Using Java operators

- Auto increment and decrement
  - Decrement operator
    - Is represented as: “**- -**”
    - Means decrease by one unit
  - Increment operator
    - Is represented as: ++
    - Means increase by one unit
  - Increment and decrement operators
    - Not only modify the variable, but also produce the value of the variable as a result

Using Java operators

- Auto increment and decrement
  - The *prefix* and *postfix* versions of auto increment and decrement
    - »*Pre-increment*
      - Means the “**++**” operator appears before the variable or expression
    - »*Post-increment*
      - Means the “**++**” operator appears after the variable or expression
    - »*Pre-decrement*
      - Means the “ **- -** “ operator appears before the variable or expression
    - »*Post-decrement*
      - Means the “ **--** “ operator appears after the variable or expression
- For post-increment and post-decrement (i.e., *a*++ or *a*--), the value is produced, then the operation is performed

Using Java operators - Auto increment and decrement example

```
public class AutoInc{
  public static void main(String[] args) {
        int i = 1;
        System.out.println("i : " + i);
        System.out.println("++i : " + ++i); // Pre-increment
        System.out.println("i++ : " + i++); // Post-increment
        System.out.println("i : " + i);
        System.out.println("--i : " + --i); // Pre-decrement
        System.out.println("i-- : " + i--); // Post-decrement
        System.out.println("i : " + i);
   }
}
```

```
>> i : 1
    ++i : 2
     i++ : 2
     i : 3
     --i : 2
     i-- : 2
     i : 1
```

Using Java operators

- Relational operators
  - Generate a **Boolean** result
  - Evaluate the relationship between the values of the operands
    - A relational expression produces **true** if the relationship is true, and **false** if the relationship is untrue
  - Example of relational operators
    - less than (<)
    - greater than (>)
    - less than or equal to (<=)
    - greater than or equal to (>=)
    - equivalent (==)
    - not equivalent (!=)

Using Java operators

- Logical operators
  - Produces a **Boolean** value of **true** or **false** based on the logical relationship of its arguments
  - Example of logical operator
    - AND (&&)
    - OR (||)
    - NOT (!)

Using Java operators

- Ternary **if-else** operator
  - Has three operands
  - Produces a value, unlike the ordinary **if-else** statement

boolean-exp?value$_0$:value$_1$

  - »If **boolean-exp** is **true**, *value*$_0$ is evaluated and its result becomes the value produced by the operator
  - »If **boolean-exp** is **false,** *value*$_1$ is evaluated and its result becomes the value produced by the operator

```
static int ternary(int i)
 { return i < 10 ? i * 100 : i * 10;}
```

Using Java operators

- String operator “**+**” is used to concatenate string

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

  - Java compiler will convert *x*, *y*, and *z* into their **String** representations instead of adding them together first

Using Java operators

- **Cast** is used to make a type conversion explicit, or to force it when it wouldn't normally happen
- Java will automatically change one type of data into another when appropriate

```
void casts() {
    int i = 200;
    long l = (long)i;
    long l2 = (long)200;
}
```

  - If you assign an integer value to a floating-point variable, the compiler will automatically convert the **int** to a **float**

Execution control in Java

- Java uses all of C's execution control statements
  - **if-else, while, do-while, for, switch**, **return**
- **If-else**
  - Produce a **Boolean** result
  - **Else** is optional, so you can use **if** in two forms:

```
if(Boolean-expression)
    statement
```

  - or

```
if(Boolean-expression)
    statement
else
    statement
```

- **return**
  - Specifies the value return by a method

Execution control in Java - Example of **if-else** statement

```
public class IfElse {
   static int test(int testval, int target) {
         int result = 0;
         if(testval > target)
               result = +1;
          else if(testval < target)
                result = -1;
                else
                  result = 0;
         return result;
                 }
    public static void main(String[] args) {
         System.out.println(test(10, 5));
         System.out.println(test(5, 10));
         System.out.println(test(5, 5));
       }
 }
```

Execution control in Java - **Iteration**

- Iteration statements
  - **while**, **do-while** and **for** control looping
  - In programming, iteration statements are used to execute a block of code repeatedly until a certain condition is met
  - Example for **while**

```
while(Boolean-expression)
    statement
```

  - The while loop executes a block of code repeatedly as long as the controlling Boolean-expression evaluates to true

Execution control in Java - Example for **while**

```
public class WhileTest
{
    public static void main(String[] args)
     {
        int i = 0;
       while (i < 10) {
                    System.out.println(i);
                    i++;
       }
     }
    }
```

Execution control in Java - Example for **while**

- Infinite loop

```
int item = 0;
while (item < 5000)
 {
    item = item * 5;
  }
```

  – Because **item** is initialized to 0, item will never be larger than 5000 (0 = 0 * 5), so the loop will never terminate

Execution control in Java

- **do-while**
  - The do-while loop is similar to the while loop, but the block of code is executed at least once, and then repeatedly as long as the controlling Boolean-expression evaluates to true.

do

statement

while(*Boolean-expression*);

```
public class WhileTest
    {
        public static void main(String[] args)
         {
            int i = 0;
            do {
                System.out.println(i);
                  i++;
              } while (i < 10);
         }
        }
```

Execution control in Java

- **for**
  - The form of the **for loop** is:

for(*initialization*; *Boolean-expression*; *step*)
statement

  - » The for loop is a more structured way to repeat a block of code, and it is often used when the number of iterations is known in advance.
  - » It consists of three parts: an initialization statement, a controlling Boolean-expression, and an update statement.
  - » *Boolean-expression* is tested before each iteration, and as soon as it evaluates to **true**, execution will continue
  - » At the end of each loop, the *step* executes
  - » Any of the expressions, *initialization, Boolean-expression* or *step* can be empty

- You can define multiple variables within a **for** statement, but they must be of the same type:

for(**int** *i* = 0, *j* = 1; *i* < 10 && *j* != 11; *i*++, *j*++)

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Execution control in Java

- **break** and **continue**
  - Control the flow of the loop by using **break** and **continue**
    - »**break** quits the loop without executing the rest of the statements in the loop
    - »**continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration

Execution control in Java - Example of break and continue statements

```
public class BreakAndContinue
 {
   public static void main(String[] args)
    {
      for(int i = 0; i < 100; i++)
       {
           if(i == 74) break; // Out of for loop
           if(i % 9 != 0) continue; // Next iteration
          System.out.println(i);
        }
    }
 }
```

Execution control in Java

- **Continue** with a Label
  - Skips the new iteration of an outer loop marked with the given label

```
public class Example {
 public static void main (String args[]) {
  ciclu1: for (int i=0; i<6; i++)
  {
    System.out.println("Control 1: i =" + i);
    for (int j = 3; j > 0; j--)
    {
      System.out.println("Control 2: i =" + i + ", j=" + j);
       if(i >1 && i<5) continue ciclu1;
       System.out.println("Control 3: i ="+i + ", j="+j);
    }
   //Sfarsitul ciclului interior
  System.out.println("Control 4: i="+i);
}
// Sfarsitul ciclului exterior
}}
```

```
>>output
Control 1: i =0
Control 2: i =0, j=3
Control 3: i =0, j=3
Control 2: i =0, j=2
Control 3: i =0, j=2
Control 2: i =0, j=1
Control 3: i =0, j=1
Control 1: i =1
Control 2: i =1, j=3
Control 3: i =1, j=3
Control 2: i =1, j=2
Control 3: i =1, j=2
Control 2: i =1, j=1
Control 3: i =1, j=1
Control 1: i =2
Control 2: i =2, j=3
Control 1: i =3
Control 2: i =3, j=3
Control 1: i =4
Control 2: i =4, j=3
Control 1: i =5
Control 2: i =5, j=3
Control 3: i =5, j=3
Control 2: i =5, j=2
Control 3: i =5, j=2
Control 2: i =5, j=1
Control 3: i =5, j=1
```

Execution control in Java

- **Switch**
  - Selects from among pieces of code based on the value of an integral expression

```
switch(integral-selector) {
  case integral-value1: statement; break;
  case integral-value2: statement; break;
  case integral-value3: statement; break; // ...
  default: statement;
}
```

  - » **Integral-selector** is an expression that produces an integral value
  - » **Switch** compares the result of **integral-selector** to each integral-value
    - If it finds a match, the corresponding **statement** *(*simple or compound*)* executes
    - If no match occurs, the **default statement** executes

Execution control in Java

- Switch example

```
public class SwitchDemo
{
  public static void main(String[] args) {
     int month = 8;
     String monthString;
     switch (month) {
       case 1: monthString = "January"; break;
       case 2: monthString = "February"; break;
        ……
       case 8: monthString = “August"; break;
       …….
       default: monthString = "Invalid month";
                                          break;
    }
   System.out.println(monthString);
} }
```

Array in Java

- Stores a group of data items all of the same type
- Is an object that must be instantiated separately
  - Once instantiated, the array object contains a block of memory locations for the individual elements
    - If the individual elements are not explicitly initialized, they will be set to zero
- Can be created with a size that is determined dynamically (but once created, the size is fixed)
- Declaring the array variable does not create an array object instance
  - It creates the reference variable

Arrays in Java - Declaring an array

- An array is **declared** by specifying the type of data to be stored, followed by square brackets

  dataType [] *variableName*;
  int [] *nums*;

  – The brackets can be putted after or before the variable name

  int *nums* []; // not recommended, but legal

Arrays in Java - Instantiating Arrays

- An array is instantiated by using
  - **new** operator, the data type, and the array size in square brackets
- Example
  - Constructing an array object with 10 integer elements, all initialized to 0, and stores into *nums*

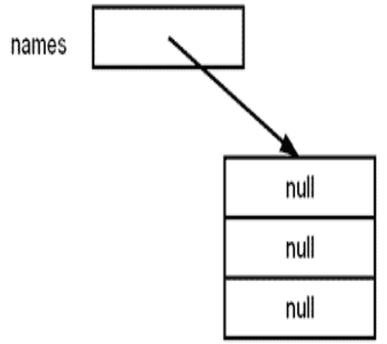
```
int [] nums;
 nums = new int [10];
```

Arrays in Java - Declaring and Instantiating Arrays

int[] *moreNums*;

int *size* = 7;

moreNums = **new int**[*size*];



- You can declare and instantiate all at once

  String[] *names* = **new String**[3];

  – The elements of the array, **String** references, are initialized to null

- The length of an array is obtained by using the property **length**

  – For the considered example

  *names.***length** = 3

Arrays in Java - Initializing Arrays

- An array can be initialized when it is created:

String[] *names* = {"Joe", "Jane", "Herkimer"};
String[] *names* = **new String**[] {"Joe", "Jane", "Herkimer"};

– This automatically creates an array of length 3, because there were 3 items supplied
– e.g., array of length of 6

int[] *nums* = **new int** [] { 2, 4, 6, 8, 10, 12 };

Arrays in Java - Initializing Arrays

```
String[] names;
names = new String[] {"Joe", "Jane", "Herkimer"};
```

- For arrays of other types of objects:

```
Book[] titles;
 titles = new Book[]{new Book(5011, "Fishing
                    Explained"), new Book(1234,
                    "Help is on the Way")};
```

```
class Book {
  int id;
  String name;
  Book(int i, String n){
     id=i;
     name=n;
   }
  public static void main(String args[]){
  Book[] titles;
   titles = new Book[]{new Book(5011, "Fishing
                              Explained"), new Book(1234,
                              "Help is on the Way")};
  }
```

Arrays in Java - Working With Arrays

- Array elements can be accessed
  - By their array index (the element number)

```
String[] names = new String[3];
        names[0] = "Sam";
        names[1] = "Sue";
        names[2] = "Mary";
```

» For the considered example, the valid elements are 0, 1, and 2

» In a **for** loop:

```
for (int i = 0; i < 3; i++)
    System.out.println(names[i]);
```

» Or using the length property:

```
for (int i = 0; i < names.length; i++) System.out.println(names[i]);
```

Arrays in Java - Working With Arrays

- Java compiler does no checking to ensure that the bounds of the array is exceed
- But the JVM does check at runtime
  - If the bounds of the array are exceeded, an exception will occur

Arrays in Java - 2D Arrays

- Array can have 2, 3, or more dimensions
- When declaring a variable for such an array, use a pair of square brackets for each dimension
- Example of 2D arrays:

```
char[][] board;
board = new char[3][3];
board[1][1] = ‘X’;
board[0][0] = ‘O’;
board[0][1] = ‘X
```

- In this example, we are creating a 2D array with 3 rows and 3 columns, and then setting some elements of array.
- It's important to note that for 2D arrays, the elements are indexed using [row][column] notation. This means that the first index refers to the row number, and the second index refers to the column number.

Arrays in Java

```
import java.util.*;
public class Arrays1
 {
    public static void main(String[] args){
           Random r = new Random();
            int[] nums = new int[10];
            for (int i = 0; i < nums.length; i++)
               { nums[i] = r.nextInt(100); }
            System.out.println("Element 7 is: " + nums[7]);
            String[] names = new String[3];
            names[0] = "Joe";
            names[1] = "Jane";
            names[2] = "Herkimer";
            for (int i = 0; i < names.length; i++)   {System.out.println(names[i]); }
            //this line should throw an exception
            System.out.println(names[6]);  }
    }
```

Arrays in Java - Copying Arrays

- Use **System.arraycopy** to copy an array into another

public static void *arraycopy*(**Object** *src*, **int** *srcPos*, **Object** *dest*, **int** *destPos*, **int** *length*)

Arrays in Java

```
public class CopyArray
 {
    public static void main(String[] args)
     {
       int nums[] = { 1, 2, 3, 4, 5 };
       int biggerNums[] = new int[10];
       System.arraycopy(nums, 0, biggerNums,  0, nums.length);
       System.arraycopy(nums, 0, biggerNums, nums.length, nums.length);
       for (int i = 0; i < biggerNums.length; i++)
                System.out.println(biggerNums[i]);
     }
  }
```

Classes in Java

```
public class Bicycle {

 public int speed;

 public Bicycle( int stSpeed)
    { speed = stSpeed;}

 public void setSpeed(int newVal)
      {speed = newVal;}

public void speedUp(int incr)
      {speed += inc;}
}
```

// Class declaration

//Class fields (attributes)

// Class constructors

// Class methods

Class in Java - Declaring Classes

```
class MyClass
  {
   //field, constructor, and method declarations
  }
```

- The class body contains all the necessary code for the life cycle of the objects created from the class:
  - **Constructors** for initializing new objects
  - **Declarations** for the **fields** that provide the **state** of the class and its objects
  - **Methods** to implement the **behavior** of the class and its objects
- This class declaration is a minimal one
  - It contains only those components of a class declaration that are required

Class in Java - Declaring Classes

- Generally, a class declarations can include the following components:
  - Modifiers such as **public, private**, and a number of others that you will encounter later
  - The name of the class's parent, if any, preceded by the keyword **extends**
    - A class can only **extend** one parent
  - A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword **implements**
    - A class can **implement** more than one interface
  - The class body, surrounded by braces{}

Class in Java - Declaring Member Variables

- Kinds of variables:
  - Member variables in a class (*fields*)
  - Variables in a method or block of code (*local variables*)
  - Variables in method declarations (*parameters*)
- *Bicycle* class uses the following lines of code to define its field:

```
Class Bicycle {
   public int speed;
    …….
}
```

Class in Java - Declaring Member Variables

- Field declarations are composed of three components, in order:
  - Zero or more modifiers (e.g., **public**, **private, protected)**
  - The field's **type**
  - The field's **name**

Class in Java - Declaring Member Variables

- Access Modifiers
  - Control the access of other classes to the member field of class
    - **public** – the field is accessible from all classes
    - **private** – the field is accessible only within its own class
  - In the spirit of encapsulation, it is common to make fields **private**
    - Private fields can only be directly accessed from the *Bicycle* class
    - Access to private field values can be done indirectly by adding public methods

```
Class Bicycle {
   private int speed;
    int getSpeed(){return speed;}
    void setSpeed(int newValue) {speed = newValue;}
}
```

Class in Java - Declaring Member Variables

- Types
  - All variables must have a type
    - Primitive types such as **int**, **float**, **boolean**, etc.
    - Reference types, such as **Strings**, **arrays**, or **Objects**
- Variable Names
  - All variables follow the same Java naming rules and conventions

Class in Java - Defining Methods

- Example of a method declaration:

double ***calculateGlobalPrice*****(int** quantity, **double** *pricePerUnit*)
{ //do the calculation here}

  - The only required elements of a method declaration are:
    - The method's **return type**
    - The methods' **name**
    - A **pair of parentheses**, ()
    - A **body** between braces {}

Class in Java- Defining Methods

- More generally, method declarations have six components, in order
  - Modifiers (i.e., **public**, **private**, etc.)
  - The return type
    - data type of the value returned by the method
    - **void** if the method does not return a value
  - The method name
    - Typically, a method has a unique name within its class
    - But a method might have the same name as other methods due to **method overloading**
  - The parameter list in parenthesis
    - A comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, ()
    - If there are no parameters, you must use empty parentheses
  - An exception list
    - To be discussed later
  - The method body, enclosed between braces
    - The method's code, including the declaration of local variables