CHAPTER 6
Defining Classes II and Arrays

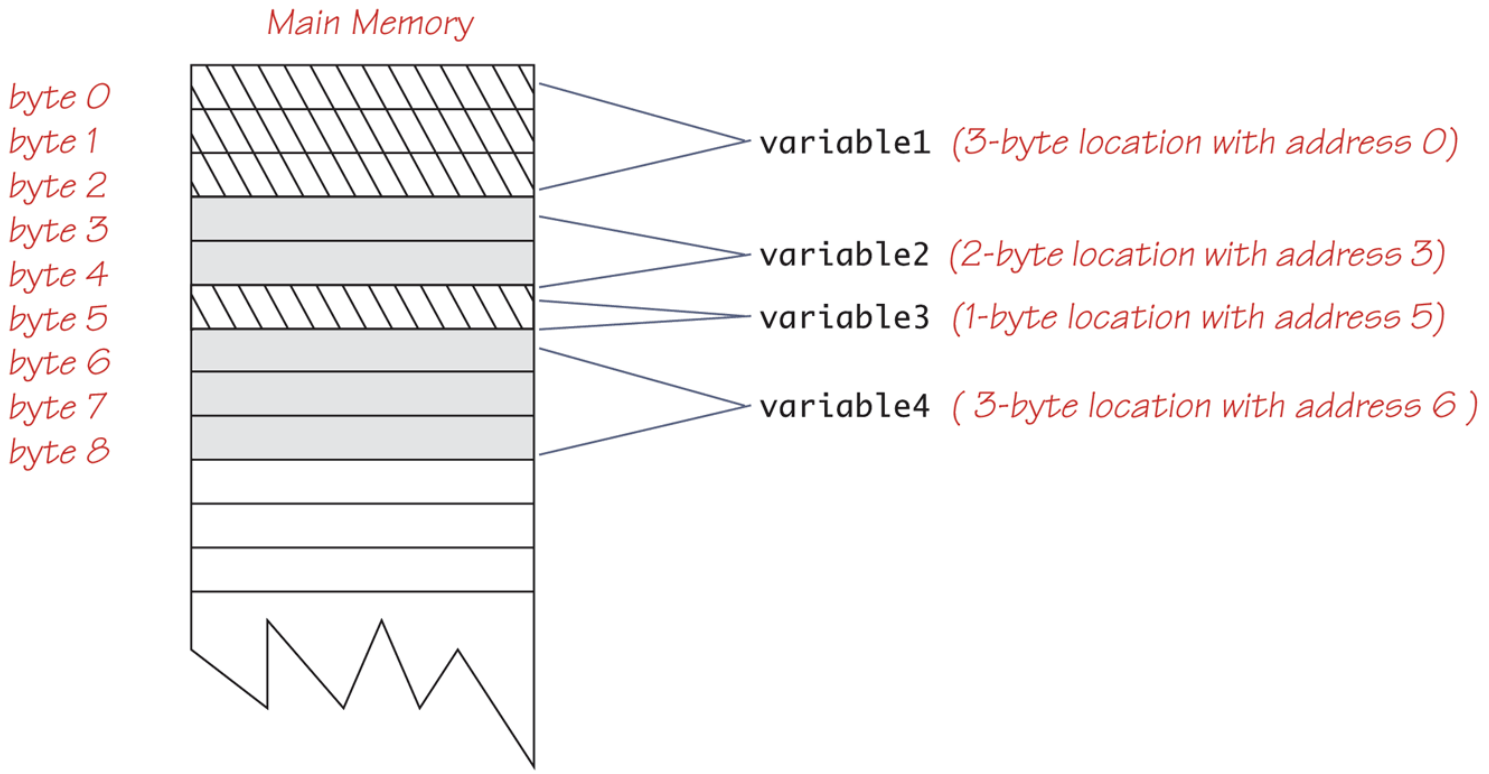
Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of Computer Science and Information Engineering
National Cheng Kung University

Variables and Memory

- Values of most data types require more than one byte of storage
  - Several **adjacent bytes** are then used to hold the data item
  - The entire **chunk of memory** that holds the data is called its ***memory location***
  - The address of the **first byte** of this memory location is used as the **address** for the data item
- A computer's main memory can be thought of as a long list of memory locations of *varying sizes*

Variables in Memory

Display 5.10 Variables in Memory



References

- Every variable is implemented as a location in computer memory
- When the variable is a **primitive type, the value of the variable is stored in the memory location** assigned to the variable
  - Each primitive type always require the same amount of memory to store its values

References

- When the variable is **a class type**, **only the memory address (or *reference*)** where its object is located is stored in the memory location assigned to the variable
  - The object named by the variable is stored in some other location in memory
  - Like primitives, the value of a class variable is a fixed size
  - Unlike primitives, the value of a class variable is a memory address or reference
  - The object, whose address is stored in the variable, can be of any size

Class Type Variables Store a Reference (Part 1 of 2)

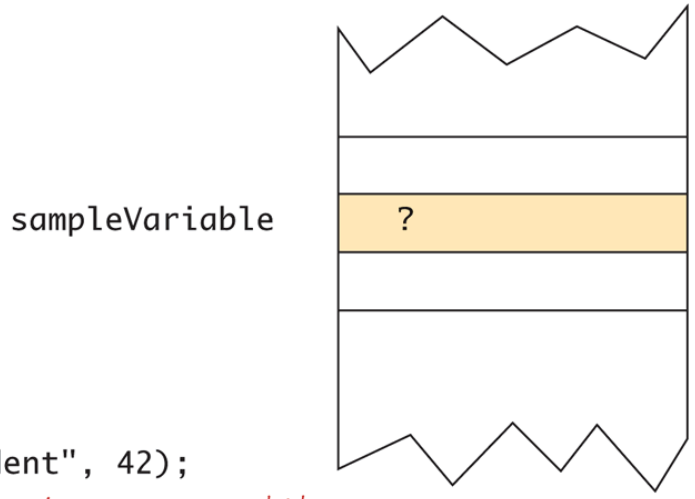
Display 5.12 Class Type Variables Store a Reference

```
public class ToyClass
{
    private String name;
    private int number;
```

The complete definition of the class `ToyClass` *is given in Display 5.11.*

```
ToyClass sampleVariable;
```

Creates the variable `sampleVariable` *in memory but assigns it no value.*



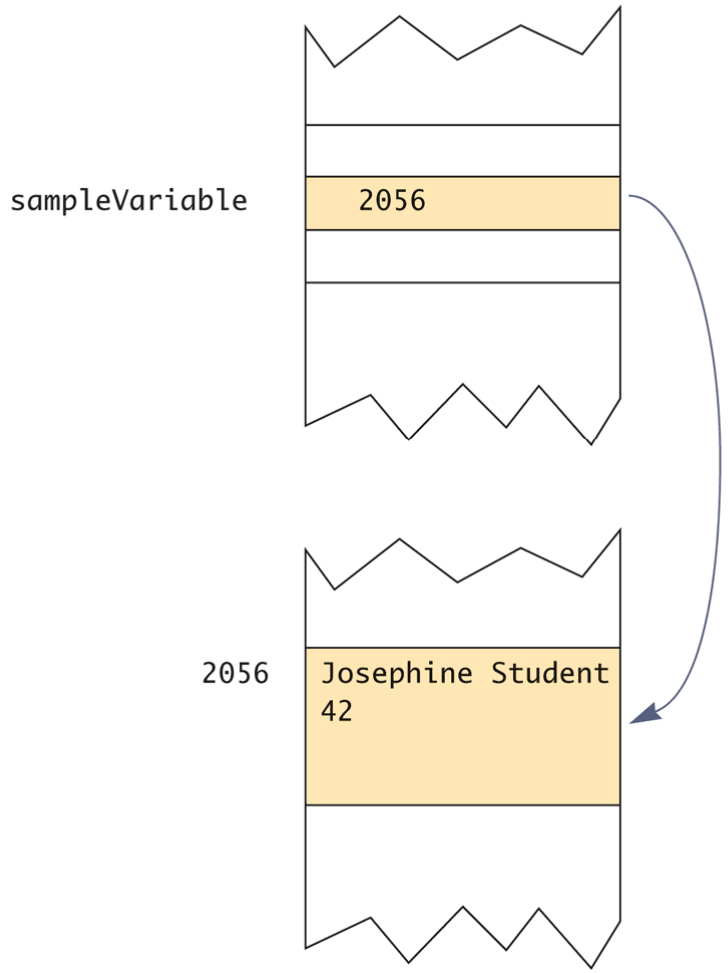

```
sampleVariable =
new ToyClass("Josephine Student", 42);
```

Creates an object, places the object someplace in memory, and then places the address of the object in the variable `sampleVariable`*. We do not know what the address of the object is, but let's assume it is* `2056`*. The exact number does not matter.*

(continued)

Class Type Variables Store a Reference (Part 2 of 2)

Display 5.12 Class Type Variables Store a Reference



For emphasis, we made the arrow point to the memory location referenced.

References

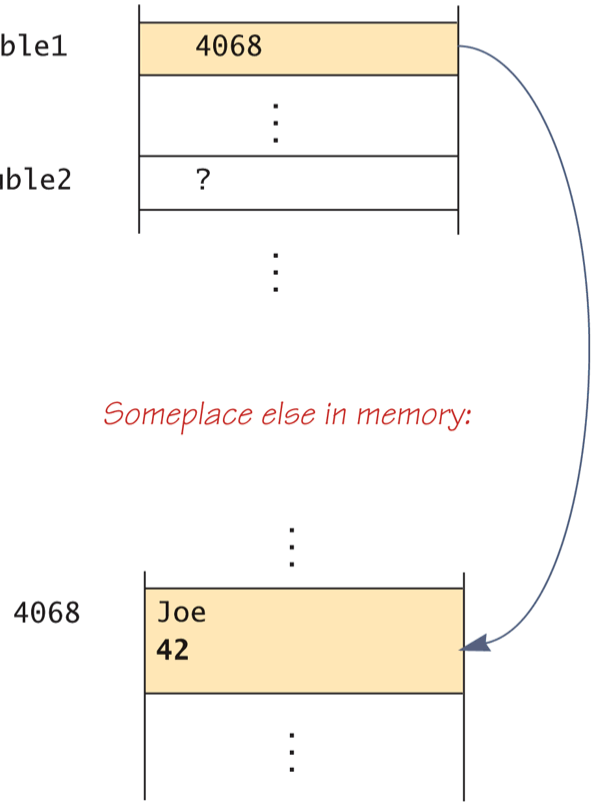
- **Two reference variables can contain the same reference**, and therefore name the same object
  - The assignment operator sets the reference (memory address) of one class type variable equal to that of another
  - Any change to the object named by one of theses variables will produce a change to the object named by the other variable, since they are the same object

```
variable2 = variable1;
```

Assignment Operator with Class Type Variables (Part 1 of 3)

Display 5.13 Assignment Operator with Class Type Variables

```
ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;
```



We do not know what memory address (reference) is stored in the variable `variable1`. *Let's say it is* `4068`. *The exact number does not matter.*

Note that you can think of

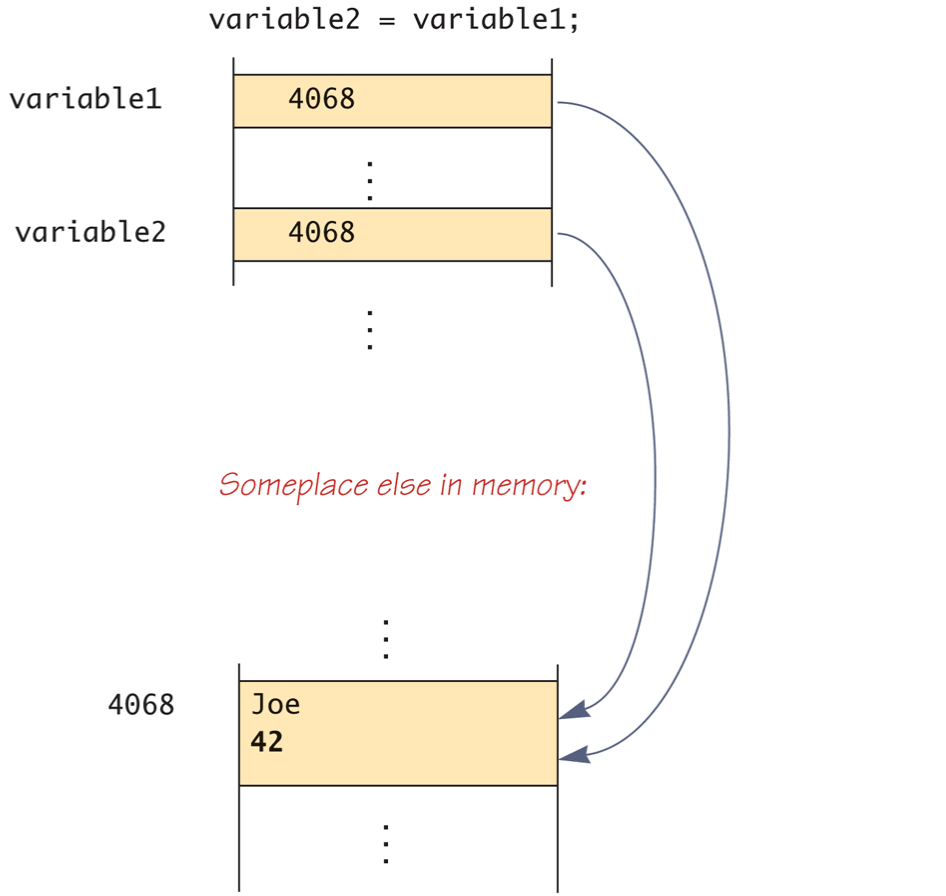
```
new ToyClass("Joe", 42)
```

as returning a reference.

(continued)

Assignment Operator with Class Type Variables (Part 2 of 3)

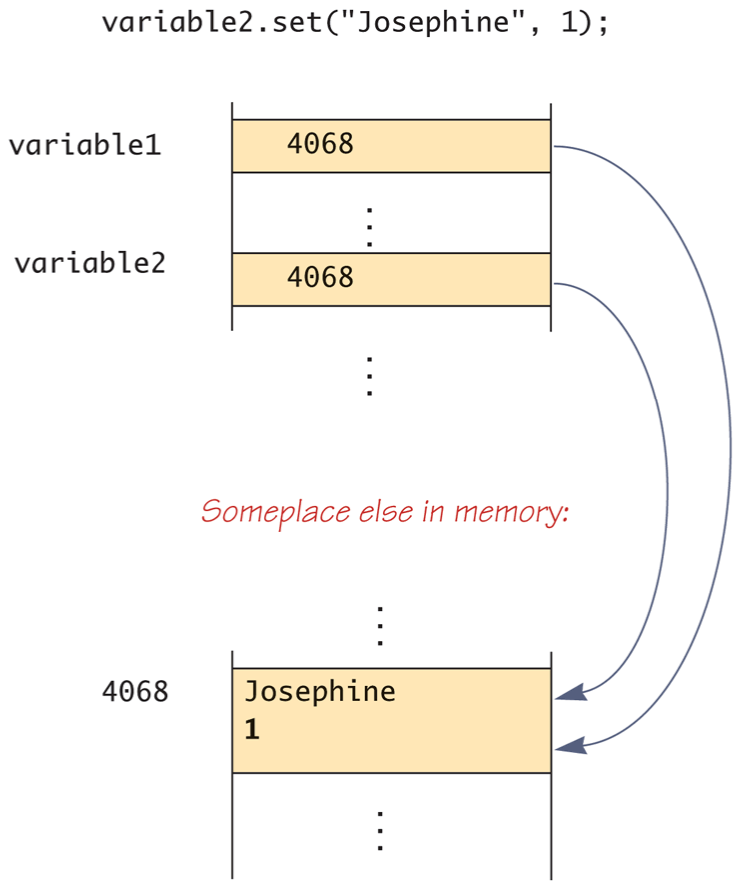
Display 5.13 Assignment Operator with Class Type Variables



(continued)

Assignment Operator with Class Type Variables (Part 3 of 3)

Display 5.13 Assignment Operator with Class Type Variables



Lab

```
public class Cat {

    int age = 1;

    public static void main(String[] args)
    {
        Cat cat1 = new Cat();
        Cat cat2 = cat1;

        cat1.age = 2;
        System.out.println(cat2.age);

     }

}
```

Class Parameters

- **Primitive type parameters** in Java are ***call-by-value*** parameters
  - A parameter is a *local variable* that is set equal to the value of its argument
  - Therefore, any change to the value of the parameter cannot change the value of its argument
- **Class type parameters** appear to behave differently from primitive type parameters
  - They appear to behave in a way similar to parameters in languages that have the ***call-by-reference*** parameter passing mechanism

Differences Between Primitive and Class-Type Parameters

- A method **cannot change the value of a variable of a primitive type** that is an argument to the method

- In contrast, a method **can change the values of the instance variables of a class type** that is an argument to the method

Lab (Call by Value)

```
public class PrimitiveParameterDemo {

    public static void main(String[] args)
    {
        int speed = 50;
        System.out.println("argument value:" + speed);

        changer(speed);

        System.out.println("argument value:" + speed);
     }

    public static void changer(int speed)
    {
    speed = 100;
    System.out.println("parameter value:" + speed);
    }

}
```

Lab (Call by Reference)

```
public class ToyClass
{
    private String name;
    private int number;

    public ToyClass(String initialName, int initialNumber)
    {
        name = initialName;
        number = initialNumber;
    }

    public String toString( )
    {
        return (name + " " + number);
    }


    public void set(String newName, int newNumber)
    {
        name = newName;
        number = newNumber;
    }
}
```

Lab (Call by Reference)

```
public class ClassParameterDemo
{
    public static void main(String[] args)
    {
        ToyClass anObject = new ToyClass("Robot Dog", 10);
        System.out.println(anObject);

        changer(anObject);
        System.out.println(anObject);
     }

    public static void changer(ToyClass aParameter)
    {
    aParameter.set("Robot Cat",20);
    }

}
```

The Constant `null`

- `null` is a special constant that may be assigned to a **variable of any class type**

```
YourClass yourObject = null;
```

- **It is used to indicate that the variable has no "real value"**
  - It is often used in constructors to initialize class type instance variables when there is no obvious object to use
- `null` is not an object: It is, rather, a kind of "**placeholder**" for a reference that does not name any memory location
  - Because it is like a memory address, use `==` or `!=` (instead of `equals`) to test if a class variable contains null

```
if (yourObject == null)  . . .
```

Pitfall: Null Pointer Exception

- A method cannot be invoked using a variable that is initialized to `null`
  - The calling object that must invoke a method does not exist
- Any attempt to do this will result in a "Null Pointer Exception" error message
  - For example, if the class variable has not been initialized at all (and is not assigned to `null`), the results will be the same

Lab

```
public class NullTest {

  public static void main(String[] args) {

    NullTest nt = new NullTest();
    nt.showMessage();

    NullTest nt2 = null;
    nt2.showMessage();
  }

  public void showMessage(){
    System.out.println("Hi!");
  }
}
```

The new Operator and Anonymous Objects

- The **new** operator invokes a constructor which initializes an object, and returns a reference to the location in memory of the object created
  - This reference can be assigned to a variable of the object's class type
- **Sometimes the object created is used as an argument to a method, and never used again**
  - In this case, the object need not be assigned to a variable, i.e., given a name
- **An object whose reference is not assigned to a variable is called an `anonymous object`**

Lab

```
public class AnonymousObjectTest {

  public static void main(String[] args) {
    AnonymousObjectTest obj = new AnonymousObjectTest();
    obj.showMessage(new String("I am an anonymous object"));
  }

  public void showMessage(String message){
    System.out.println(message);
  }

}
```

Packages and Import Statements

- Java uses *packages* to form libraries of classes
- A package is a group of classes that have been placed in a directory or folder, and that can be used in any program that includes an *import statement* that names the package
  - The import statement must be located at the beginning of the program file: Only blank lines, comments, and package statements may precede it
  - The program can be in a different directory from the package

Import `Statements`

- We have already used import statements to include some predefined packages in Java, such as `Scanner` from the `java.util` package

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

- It is possible to make all the classes in a package available instead of just one class:

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

  - Note that there is no additional overhead for importing the entire package

The package Statement

- To make a package, group all the classes together into a single directory (folder), and add the following package statement to the beginning of each class file:

```
package package_name;
```

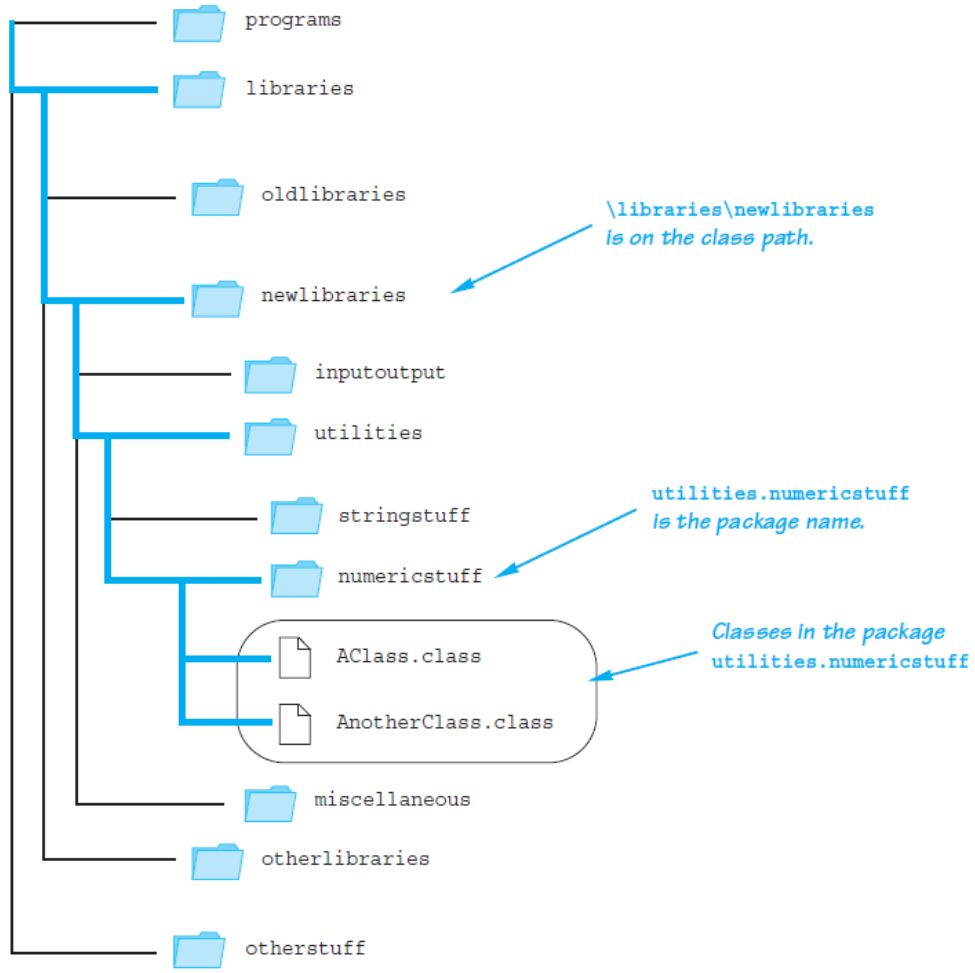
  - Only the `.class` files must be in the directory or folder, the `.java` files are optional
  - Only blank lines and comments may precede the package statement
  - If there are both import and package statements, the package statement must precede any import statements

The Package `java.lang`

- The package **`java.lang`** contains the classes that are fundamental to Java programming
  - **It is imported automatically, so no import statement is needed**
  - Classes made available by **`java.lang`** include **`Math`**, **`String`**, and the wrapper classes

A Package Name

Display 5.22 A Package Name



Pitfall: Subdirectories Are Not Automatically Imported

- When a package is stored in a subdirectory of the directory containing another package, importing the enclosing package does not import the subdirectory package
- The import statement:

```
import utilities.numericstuff.*;
```

  imports the `utilities.numericstuff` package only
- The import statements:

```
import utilities.numericstuff.*;
import utilities.numericstuff.statistical.*;
```

  import both the `utilities.numericstuff` and `utilities.numericstuff.statistical` packages

The Default Package

- All the classes in the current directory belong to an unnamed package called the *default package*
- As long as the current directory (**.**) is part of the **CLASSPATH** variable, all the classes in the default package are automatically available to a program

Introduction to `javadoc`

- Unlike a language such as C++, Java places both the interface and the implementation of a class in the same file
- However, Java has a program called **`javadoc`** that automatically extracts the interface from a class definition and produces documentation
  - This information is presented in HTML format, and can be viewed with a Web browser
  - If a class is correctly commented, a programmer need only refer to this *API (Application Programming Interface)* documentation in order to use the class
  - **`javadoc`** can obtain documentation for anything from a single class to an entire package

Commenting Classes for `javadoc`

- The `javadoc` program extracts class headings, the headings for some comments, and headings for all public methods, instance variables, and static variables
  - In the normal default mode, no method bodies or private items are extracted
- To extract a comment, the following must be true:
  1. The comment must *immediately precede* a public class or method definition, or some other public item
  2. The comment must be a block comment, and the opening `/*` must contain an extra `*` (`/** . . . */`)
  - Note: Extra options would have to be set in order to extract line comments (`//`) and private items

Commenting Classes for javadoc

- In addition to any general information, the comment preceding a public method definition should include descriptions of parameters, any value returned, and any exceptions that might be thrown
  - This type of information is preceded by the @ symbol and is called an @ *tag*
  - @ tags come after any general comment, and each one is on a line by itself

```
/**
 General Comments about the method . . .
 @param aParameter Description of aParameter
 @return What is returned
 . . .
*/
```

@ Tags

- @ tags should be placed in the order found below
- If there are multiple parameters, each should have its own **`@param`** on a separate line, and each should be listed according to its left-to-right order on the parameter list
- If there are multiple authors, each should have its own **`@author`** on a separate line

```
@param Parameter_Name Parameter_Description
@return Description_Of_Value_Returned
@throws Exception_Type Explanation
@deprecated
@see Package_Name.Class_Name
@author Author
@version Version_Information
```

Lab

```
public class AnonymousObjectTest {

public static void main(String[] args) {
  AnonymousObjectTest obj = new AnonymousObjectTest();
 obj.showMessage(new String("I am an anonymous object"));
}

/**
This method is to show a given message
@param The message to be shown
 */
public void showMessage(String message){
  System.out.println(message);
}
}
```

Lab (Run Javadoc in Eclipse)

- Menubar->Project->Generate Javadoc->Select types (classes) -> Finish

Creating and Accessing Arrays

- An array that behaves like this collection of variables, all of type **`double`**, can be created using one statement as follows:

```
double[] score = new double[5];
```

- Or using two statements:

```
double[] score;
score = new double[5];
```

  - The first statement declares the variable **`score`** to be of the array type **`double[]`**
  - The second statement creates an array with five numbered variables of type **`double`** and makes the variable **`score`** a name for the array

Creating and Accessing Arrays

- The individual variables that together make up the array are called ***indexed variables***
  - They can also be called *subscripted variables* or *elements* of the array
  - The number in square brackets is called an *index* or *subscript*
  - In Java, *indices must be numbered starting with* **`0`**, *and nothing else*

```
score[0], score[1], score[2], score[3], score[4]
```

Creating and Accessing Arrays

- The number of indexed variables in an array is called the ***length* or *size* of the array**
- When an array is created, the length of the array is given in square brackets after the array type
- The indexed variables are then numbered starting with **0**, and ending with the integer that is *one less than the length of the array*

```
double[] score = new double[5];

score[0], score[1], score[2], score[3], score[4]
```

Declaring and Creating an Array

- An array is declared and created in almost the same way that objects are declared and created:

```
BaseType[] ArrayName = new BaseType[size];
```

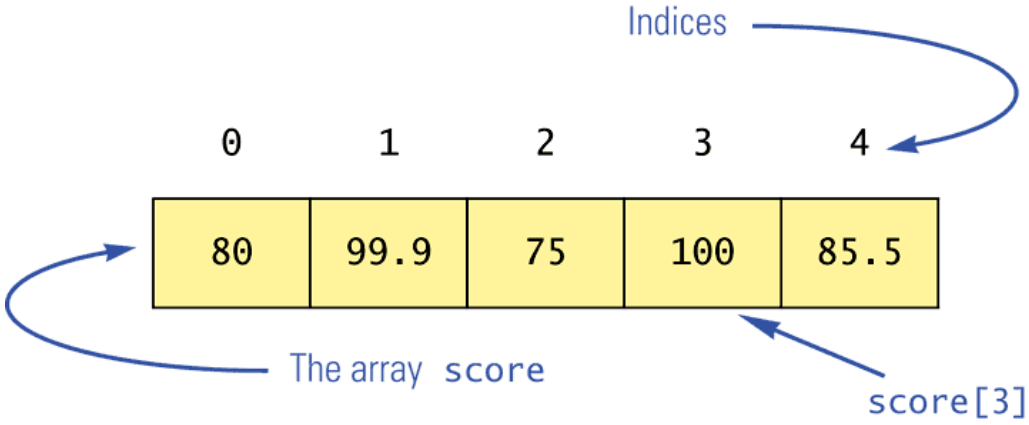
  - The ***size*** may be given as an expression that evaluates to a nonnegative integer, for example, an **int** variable

```
char[] line = new char[80];
double[] reading = new double[count];
Person[] specimen = new Person[100];
```

Using the `score` Array in a Program

- The `for` loop is ideally suited for performing array manipulations:

```
for (index = 0; index < 5; index++)
  System.out.println(score[index]);
```



Three Ways to Use Square Brackets [] with an Array Name

- Square brackets can be used to create a type name:

```
double[] score;
```

- Square brackets can be used with an integer value as part of the special syntax Java uses to create a new array:

```
score = new double[5];
```

- Square brackets can be used to name an indexed variable of an array:

```
max = score[0];
```

The `length` Instance Variable

- An array is considered to be an object
- Since other objects can have instance variables, so can arrays
- Every array has exactly one instance variable named ***`length`***
  - When an array is created, the instance variable **`length`** is automatically set equal to its size
  - The value of **`length`** cannot be changed (other than by creating an entirely new array with **`new`**)

```
double[] score = new double[5];
```

  - Given **`score`** above, **`score.length`** has a value of 5

Initializing Arrays

- An array can be initialized when it is declared
  - Values for the indexed variables are enclosed in braces, and separated by commas
  - The array size is automatically set to the number of values in the braces

```
int[] age = {2, 12, 1};
```

  - Given **`age`** above, **`age.length`** has a value of 3

Initializing Arrays

- Another way of initializing an array is by using a **`for`** loop

```
double[] reading = new double[100];
for (int i = 0; i < reading.length; i++)
  reading[i] = 42.0;
```

- If the elements of an array are not initialized explicitly, they will automatically be initialized to the default value for their base type

Lab

```
public class ArrayTest {
  public static void main(String[] args) {
  double[] reading = new double[100];
  for (int i = 0; i < reading.length; i++){
    reading[i] = 42.0;
  }

  System.out.println(reading[38]);


  int[] age = {12, 24, 36};
    System.out.println(age.length);
    System.out.println(age[2]);
  }
}
```

Pitfall: Arrays with a Class Base Type

- The base type of an array can be a class type

```
Date[] holidayList = new Date[20];
```

- The above example creates 20 indexed variables of type `Date`
  - **It does not create 20 objects of the class `Date`**
  - Each of these indexed variables are automatically initialized to `null`
  - Any attempt to reference any them at this point would result in a "null pointer exception" error message

Pitfall: Arrays with a Class Base Type

- Like any other object, each of the indexed variables requires a separate invocation of a constructor using **new** (singly, or perhaps using a **for** loop) to create an object to reference

```
holidayList[0] = new Date();
                 . . .
holidayList[19] = new Date();
```

OR

```
for (int i = 0; i < holidayList.length; i++)
  holidayList[i] = new Date();
```

- Each of the indexed variables can now be referenced since each holds the memory address of a **Date** object

Lab

```
public class ArrayTest {
  public static void main(String[] args) {

    String[] names = new String[3];
    System.out.println(names[0]);

    names[0] = "Apple";
    System.out.println(names[0]);

  }
}
```

Array Parameters

- Both array indexed variables and entire arrays can be used as arguments to methods
  - An indexed variable can be an argument to a method in exactly the same way that any variable of the array base type can be an argument

Array Parameters

```
double n = 0.0;
double[] a = new double[10];//all elements
                    //are initialized to 0.0
int i = 3;
```

- Given **myMethod** which takes one argument of type **double**, then all of the following are legal:

```
myMethod(n);//n evaluates to 0.0
myMethod(a[3]);//a[3] evaluates to 0.0
myMethod(a[i]);//i evaluates to 3,
                    //a[3] evaluates to 0.0
```

Array Parameters

- An argument to a method may be an entire array
- Array arguments behave like objects of a class
  - Therefore, a method can change the values stored in the indexed variables of an array argument
- A method with an array parameter must specify the base type of the array only

```
BaseType[]
```

  - It does not specify the length of the array

Array Parameters

- The following method, **`doubleElements`**, specifies an array of **`double`** as its single argument:

```
public class SampleClass
{
  public static void doubleElements(double[] a)
  {
    int i;
    for (i = 0; i < a.length; i++)
      a[i] = a[i]*2;
    . . .
  }
. . .
}
```

Lab

```
public class ArrayTest {
  public static void main(String[] args) {

    String[] names = new String[3];
    System.out.println(names[0]);
    names[0] = "Apple";
    System.out.println(names[0]);

    showMessage(names);
  }

  public static void showMessage(String[] message){
    System.out.println(message[0]);
  }
}
```

Lab

```
public class ArrayTest2 {

  public static void main(String[] args) {

    int[] scores = new int[100];

    for(int i=0;i<scores.length;i++){
      scores[i] = i;
    }

    System.out.println(scores[98]);
  }

}
```

Sorting an Array

- A sort method takes in an array parameter **a**, and rearranges the elements in **a**, so that after the method call is finished, the elements of **a** are sorted in ascending order
- A *selection sort* accomplishes this by using the following algorithm:

```
for (int index = 0; index < count; index++)
  Place the indexth smallest element in
  a[index]
```

Multidimensional Arrays

- It is sometimes useful to have an array with more than one index
- Multidimensional arrays are declared and created in basically the same way as one-dimensional arrays
  - You simply use as many square brackets as there are indices
  - Each index must be enclosed in its own brackets

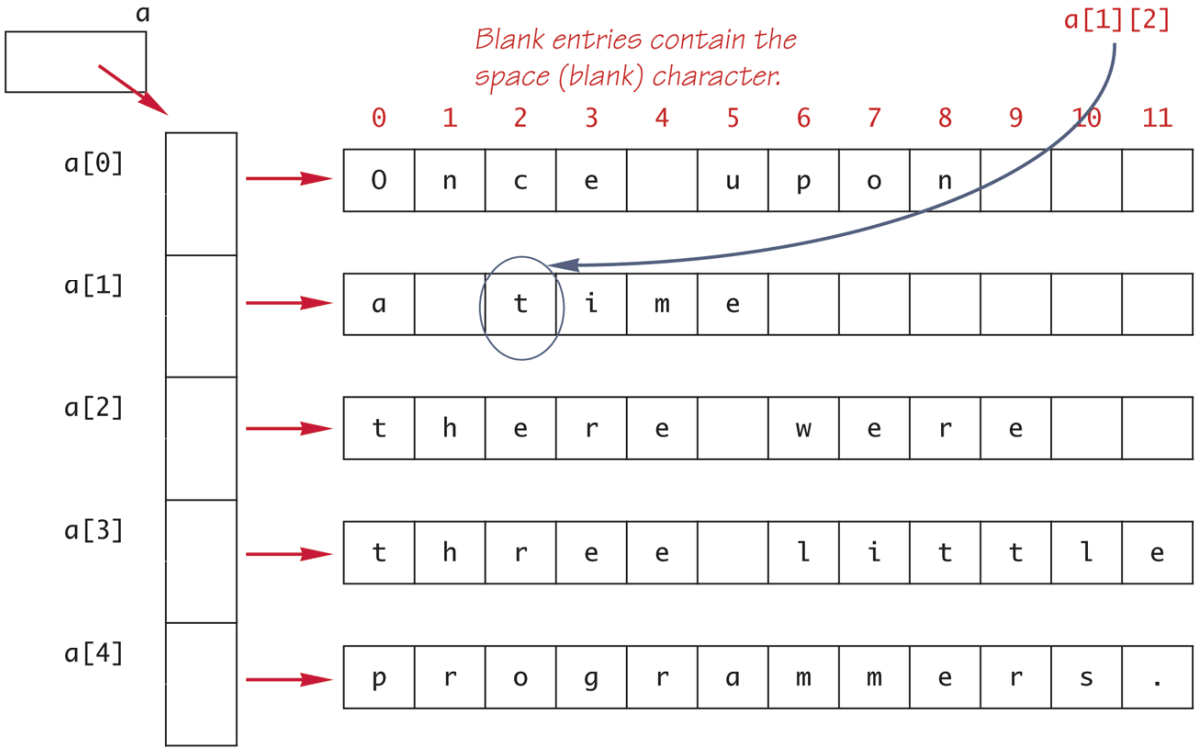
```
double[][]table = new double[100][10];
int[][][] figure = new int[10][20][30];
Person[][] = new Person[10][100];
```

Two-Dimensional Array as an Array of Arrays (Part 1 of 2)

Display 6.17 Two-Dimensional Array as an Array of Arrays

```
char[][] a = new char[5][12];
```

Code that fills the array is not shown.



(continued)

Two-Dimensional Array as an Array of Arrays (Part 2 of 2)

Display 6.17 Two-Dimensional Array as an Array of Arrays

We will see that these can and should be replaced with expressions involving the `length` *instance variable.*

```
int row, column;
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 12; column++)
       System.out.print(a[row][column]);
    System.out.println();
}
```

Produces the following output:

```
Once upon
a time
there were
three little
programmers.
```

Using the `length` Instance Variable

```
char[][] page = new char[30][100];
```

- The instance variable **`length`** does not give the total number of indexed variables in a two-dimensional array
  - Because a two-dimensional array is actually an array of arrays, the instance variable **`length`** gives the number of first indices (or "rows") in the array
    - **`page.length`** is equal to 30
  - For the same reason, the number of second indices (or "columns") for a given "row" is given by referencing **`length`** for that *"row" variable*
    - **`page[0].length`** is equal to 100

Lab

```
public class ArrayTest3 {

  public static void main(String[] args){

    int[][] seat = new int[100][10];

    for(int i=0;i<seat.length;i++){
      for(int j=0;j<seat[i].length;j++){
        seat[i][j] = i*j;
      }
    }

    System.out.println(seat[5][3]);
  }
}
```

ArrayList

- An **ArrayList** is a dynamic data structure, meaning items can be added and removed from the list.

- To set up an ArrayList, you first have to import the package from the **java.util library**:
  - **import java.util.ArrayList;**
- You can then create a new ArrayList object:
  - **ArrayList listTest = new ArrayList( );**
- Once you have a new ArrayList objects, you can add elements to it with the add method:
  - **listTest.add("first item");**

Lab

```
import java.util.ArrayList;

public class ArrayListTest {

  public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<String>();
        names.add("Apple");
        names.add("Orange");
        names.add("pear");

        System.out.println(names.get(1));

  }

}
```

Lab

```
import java.util.ArrayList;

public class ArrayListDemo {

  public static void main(String[] args) {

   ArrayList<String> names = new ArrayList<String>();
   names.add("A");
   names.add("B");
   names.add("C");
   names.remove(1);
   System.out.println(names.get(1));
  }

}
```

Lab

```
import java.util.ArrayList;

public class ArrayListDemo2 {

  public static void main(String[] args) {

    ArrayList<String> names = new ArrayList<String>();
    names.add("A");
    names.add("B");
    names.add("C");

    for(int i=0;i<3;i++){
      names.remove(i);
    }
    System.out.println(names.size());
  }

}
```

Lab

```
public class Sum
{
    public static void main( String[] args )
    {
        int[] a;
        a = new int[3];

        for ( int i = 0; i < a.length; i++ ){
            a[i] = i + 2;
        }

        int result = 0;
        for ( int i = 0; i < a.length; i++ ){
            result += a[i];
        }

        System.out.println( "Result is:" + result );
    }
}
```

Lab

```
public class SumTest
{
   public static void main( String[] args )
   {
      int[] a = { 99, 22, 11, 3, 11, 55, 44, 88, 2, -3 };

      int result = 0;

      for ( int i = 0; i < a.length; i++ )
      {
         if ( a[ i ] > 30 )
            result += a[ i ];
      }

      System.out.printf( "Result is: %d\n", result );
   }
}
```

Lab

- Which statement below initializes array items to contain 3 rows and 2 columns?

a. int[][] items = { { 2, 4 }, { 6, 8 }, { 10, 12 } };.

b. int[][] items = { { 2, 6, 10 }, { 4, 8, 12 } };.

c. int[][] items = { 2, 4 }, { 6, 8 }, { 10, 12 };.

d. int[][] items = { 2, 6, 10 }, { 4, 8, 12 };.

Reference

- "Absolute Java". Walter Savitch and Kenrick Mock. Addison-Wesley; 5 edition. 2012
- "Java How to Program". Paul Deitel and Harvey Deitel. Prentice Hall; 9 edition. 2011.
- "A Programmers Guide To Java SCJP Certification: A Comprehensive Primer 3rd Edition". Khalid Mughal, Rolf Rasmussen. Addison-Wesley Professional. 2008