# Arrays 2/4

# Outline

- Arrays and References
- Arrays and Objects
- Arrays Parameters
- Example

# - Arrays and References

- Like class types, a variable of an array type holds a *reference*
  - Arrays are objects
  - A variable of an array type holds the address of where the array object is stored in memory
  - Array types are (usually) considered to be class types

# - Arrays are Objects ...

- An array can be viewed as a collection of indexed variables
- An array can also be viewed as a single item whose value is a collection of values of a base type
    - An array variable names the array as a single item

```
double[] a;
```

    - A **new** expression creates an array object and stores the object in memory

```
new double[10]
```

    - An assignment statement places a reference to the memory address of an array object in the array variable

```
a = new double[10];
```

# ... - Arrays Are Objects

- The previous steps can be combined into one statement

```
double[] a = new double[10];
```

- Note that the **new** expression that creates an array invokes a constructor that uses a nonstandard syntax
- Not also that as a result of the assignment statement above, **a** contains a single value:  a memory address or *reference*
- Since an array is a reference type, the behavior of arrays with respect to assignment (**=**), equality testing (**==**), and parameter passing are the same as that described for classes

# 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

# - 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;
    . . .
  }
. . .
}
```

# … - Array Parameters

- Arrays of double may be defined as follows:

```
double[] a = new double[10];
double[] b = new double[30];
```

- Given the arrays above, the method `doubleElements` from class `SampleClass` can be invoked as follows:

```
SampleClass.doubleElements(a);
SampleClass.doubleElements(b);
```

  - Note that no square brackets are used when an entire array is given as an argument
  - Note also that a method that specifies an array for a parameter can take an array of any length as an argument

# Pitfall: Use of = and == with Arrays

- Because an array variable contains the memory address of the array it names, the assignment operator (**=**) only copies this memory address
  - It does not copy the values of each indexed variable
  - Using the assignment operator will make two array variables be different names for the same array

```
b = a;
```

  - The memory address in **a** is now the same as the memory address in **b**: They reference the same array

# Pitfall: Use of = and == with Arrays

- A **for** loop is usually used to make two different arrays have the same values in each indexed position:

```
int i;
for (i = 0;
     (i < a.length)  && (i < b.length); i++)
  b[i] = a[i];
```

  - Note that the above code will not make **b** an exact copy of **a**, unless **a** and **b** have the same length

# Pitfall: Use of = and == with Arrays

- For the same reason, the equality operator **(==)** only tests two arrays to see if they are stored in the same location in the computer's memory
  - It does not test two arrays to see if they contain the same values

    ```
    (a == b)
    ```

  - The result of the above **`boolean`** expression will be **`true`** if **`a`** and **`b`** share the same memory address (and, therefore, reference the same array), and **`false`** otherwise

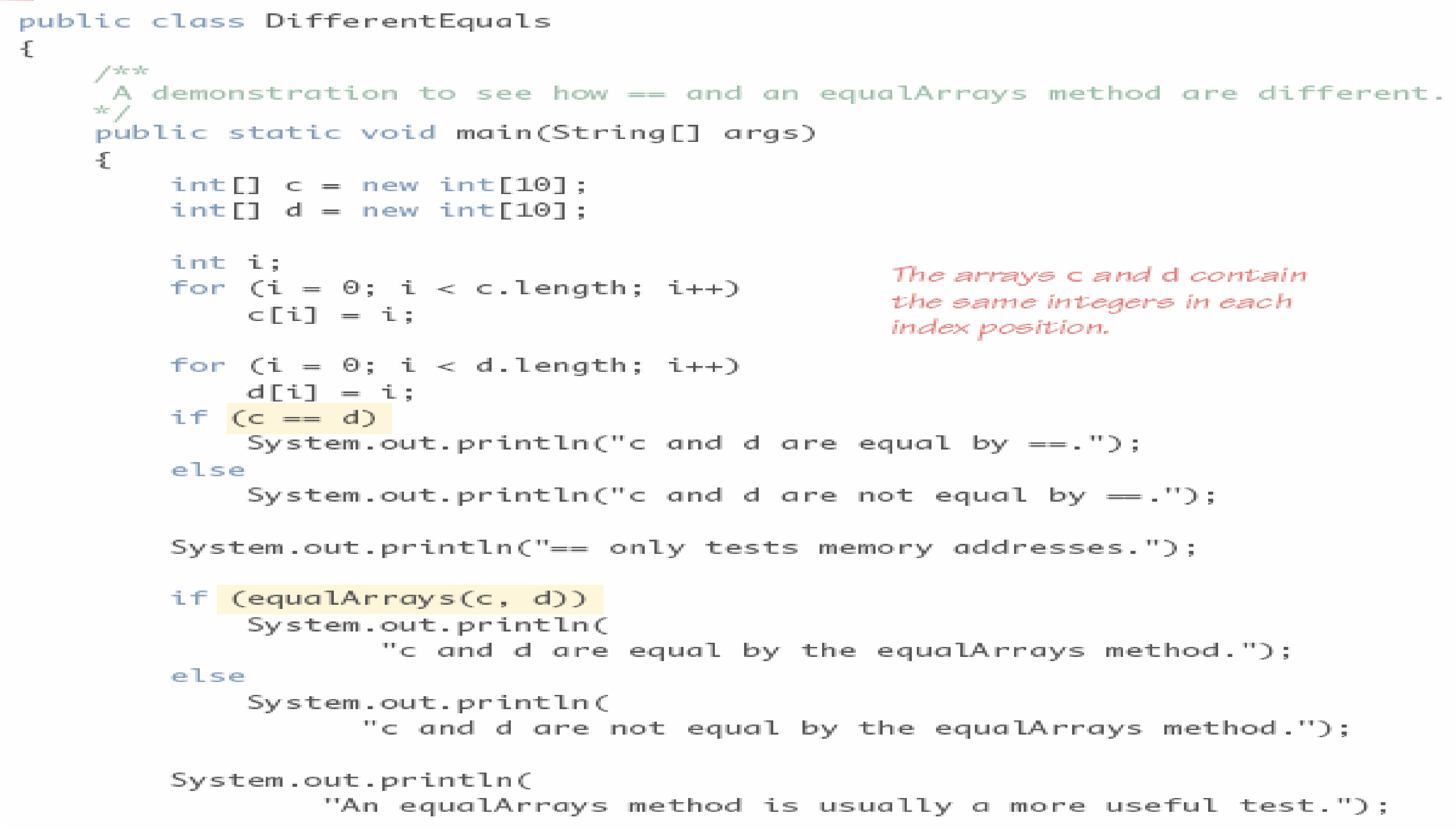
# Pitfall: Use of = and == with Arrays

- In the same way that an **`equals`** method can be defined for a class, an ***`equalsArray`*** method can be defined for a type of array
  - This is how two arrays must be tested to see if they contain the same elements
  - The following method tests two integer arrays to see if they contain the same integer values

# Pitfall:  Use of = and == with Arrays

```
public static boolean equalsArray(int[] a, int[] b)  {
  if (a.length != b.length)  return false;
  else {
    int i = 0;
    while (i < a.length)  {
      if (a[i] != b[i])
        return false;
      i++;
    }
  }
  return true;
}
```

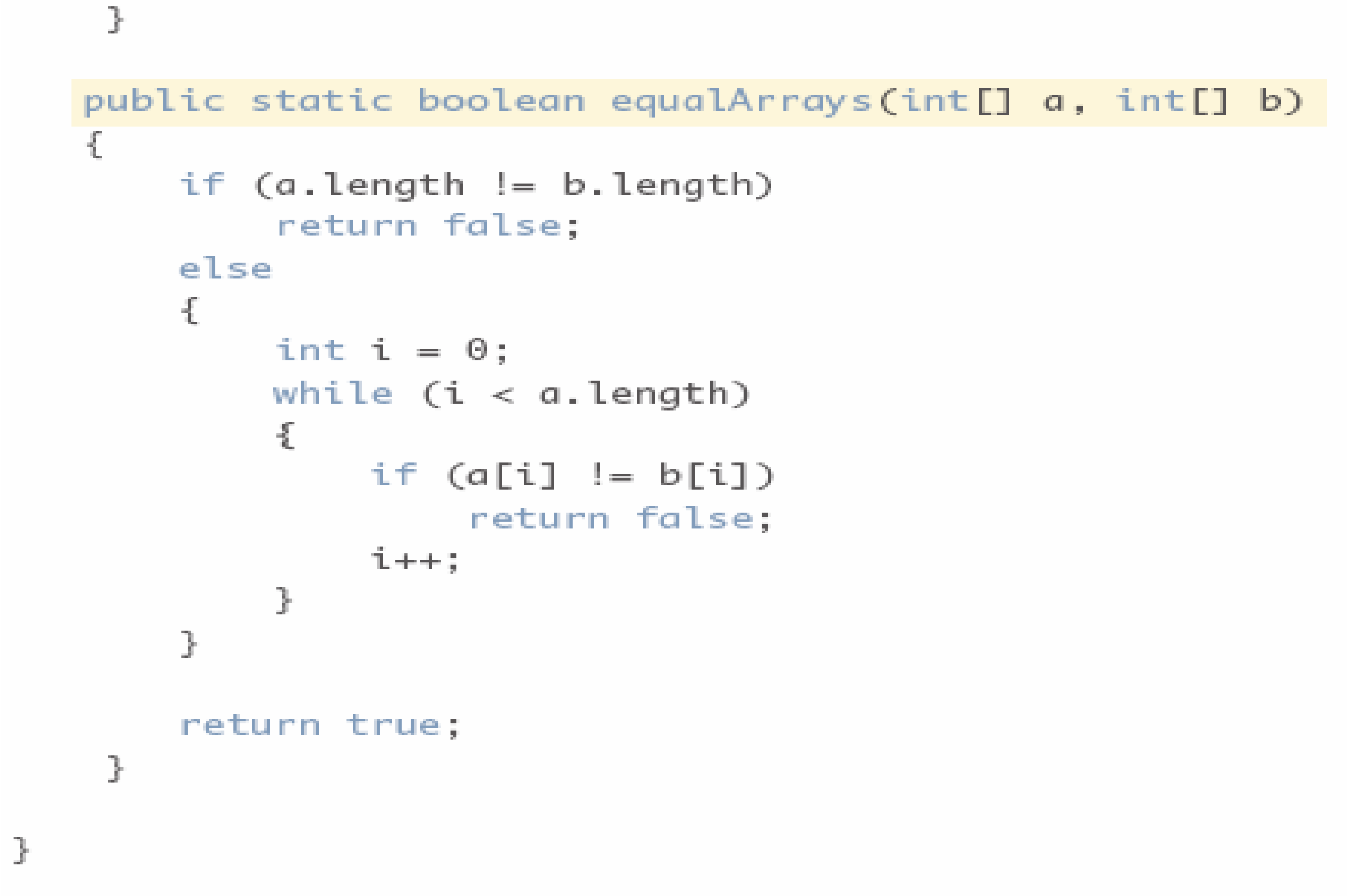
# - Example ...



```
public class DifferentEquals
{
    /**
     A demonstration to see how == and an equalArrays method are different.
    */
    public static void main(String[] args)
    {
        int[] c = new int[10];
        int[] d = new int[10];

        int i;
        for (i = 0; i < c.length; i++)
            c[i] = i;

        for (i = 0; i < d.length; i++)
            d[i] = i;
        if (c == d)
            System.out.println("c and d are equal by ==.");
        else
            System.out.println("c and d are not equal by ==.");

        System.out.println("== only tests memory addresses.");

        if (equalArrays(c, d))
            System.out.println(
                       "c and d are equal by the equalArrays method.");
        else
            System.out.println(
                     "c and d are not equal by the equalArrays method.");

        System.out.println(
                "An equalArrays method is usually a more useful test.");
```

*The arrays c and d contain the same integers in each index position.*

# ... - Example



```
    }

   public static boolean equalArrays(int[] a, int[] b)
   {
       if (a.length != b.length)
           return false;
       else
       {
           int i = 0;
           while (i < a.length)
           {
               if (a[i] != b[i])
                   return false;
               i++;
           }
       }

       return true;
    }

}
```

THE END