# CSC 1051 Algorithms & Data Structures I

Methods & Arrays

## MORE WITH METHODS

## Method Overloading

Method overloading is a technique in which two or more methods in a class can have the same name

The compiler must be able to figure out which method is being called by analyzing the parameter list

A method signature is the name of the method along with the types of its parameters

All method signatures in a class must be unique

That means the names can be the same as long as the number, types, and order of the parameters are distinct

## Method Overloading

For example:

```
public void setCoordinates(int x, int y)
{
    // whatever
}

public void setCoordinates(Point p)
{
    // whatever
}
```

The signatures for these methods are

```
setCoordinates(int, int)
setCoordinates(Point)
```

## Method Overloading

Suppose the following invocation is made:

```
target.setCoordinates(13, 58);
```

The compiler would map this invocation to the version of the method that takes two int parameters

Method overloading is appropriate in situations where the same operation is being applied to different data

Constructors are often overloaded

## Method Overloading

Another example:

```
public static int add(int i, int j)
{
    return i + j;
}

public static double add(double i, double j)
{
    return i + j;
}

public static double add(double i, double j, double k)
{
    return i + j + k;
}
```

## Method Overloading

The signatures of those methods are

```
add(int, int)
add(double, double)
add(double, double, double)
```

The compiler looks for the most specific match it can find

If two or more methods are equally appropriate, the compiler will issue an error (ambiguous invocation)

The return type is NOT part of the method signature

Overloaded methods cannot differ only by their return type

## Method Overloading

The print and println methods are overloaded several times

Each version takes a single type as a parameter

```
println(int)
println(double)
println(String)
```

etc.

So the following two calls invoke different methods

```
System.out.println("Hello there!");
System.out.println(total);
```

# Collections Overview

A collection is an object that manages a group of other objects.

The collection classes defined in the Java API are some of the most useful and versatile.

Each type of collection manages objects in a particular way:

- List – list of elements you an add, remove or replace
- Queue – list you can only add on one end & remove on other
- Stack – list where you add and remove only on one end
- Map – collection that uses keys to lookup values
- Set – unordered collection of unique elements

## Collections Framework

Collections in Java are organized by a design architecture known as the Collections Framework.

The framework includes a set of Java interfaces that define the operations on a collection, as well as one or more classes that implement the interfaces.

Java Collections are defined in the `java.util` package.

## Collections vs. Data Structures

These two terms sometimes get used interchangeably. For our purposes, they are defined as:

collection - manages a group of objects in a particular way

data structure - the programming technique used to implement a collection

The collection is the concept and the data structure is how it gets done.

## Collections are Generic

All Java collection classes are **generic**.

A generic class is a class that specifies the type of data the class manages using a placeholder.

For example, the ArrayList class is named **`ArrayList<E>`**, where the E is a placeholder for the type of element to be stored. It is used like this:

```
ArrayList<String> nameList = new ArrayList<String>();
```

## The for-each loop

Traversing a collection is made easy by the for-each statement, which is a variation of the for loop:

```
for (Member mem : memberList)
{
    System.out.println(mem.getName());
    System.out.println(mem.getMembershipNumber());
}
```

On each iteration of the loop, the variable `mem` is assigned the next `Member` object from the list, starting with the first one. The loop repeats until the last object in the list is used.

## WORKING WITH ARRAYS

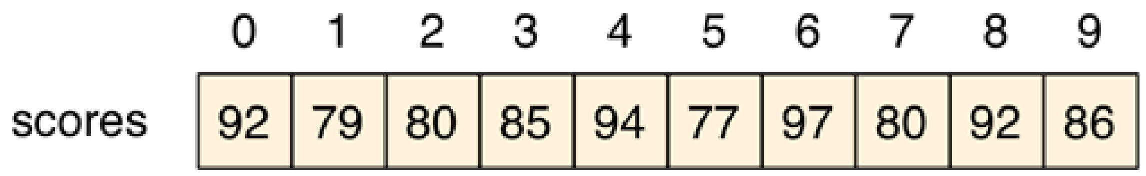
Rephactor

# Arrays

An array is an object that holds a set of values

Each value can be accessed by a numeric index

An array of length N is indexed from 0 to N-1



The scores array can hold 10 integers, indexed from 0 to 9

The name of the array is an object reference variable

## Arrays

Square brackets (the index operator) are used to refer to a specific element in the array

```
int num = scores[3];
```

An exception is thrown if you attempt to access an array outside of the range 0 to N-1

This is called a bounds error

Each element of **scores** can be treated as an individual integer

```
scores[7] = 83;
```

# Arrays

The object reference variable is declared without specifying the size of the array

```
int[] scores;
```

The variable scores can refer to any array of integers

The array size is specified when the array object is created

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

As with other objects, those two steps may be combined

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

## Arrays

In Java it is valid to associate the brackets with the array name in a declaration

```
int myList[];
```
✗

However, this is not a good idea

It is far more readable to associate the brackets with the element type

```
int[] myList;
```
✓

Together, they define the type of the variable (an array of int)

# Arrays

The array's element type is the type of values it stores

An array can be declared to hold any primitive or object type

Only values consistent with the element type can be stored in it

```
int[] widths = new int[500];
double[] myArray = new double[20];
boolean[] flags = new boolean[80];
String[] names = new String[150];
```

When an array is created it is filled with the default value for the element type

# Arrays

The size of an array is stored in a constant called length

Once an array has been created, its size cannot change

The for and for-each loops are often used when processing arrays

```
int[] list = new int[15];

for (int i = 0; i < list.length; i++)
    list[i] = i * 10;

for (int value : list)
    System.out.print(value + "  ");
```

```
0   10   20   30   40   50   60   70   80   90   100   110   120   130   140
```

# Arrays

Modifying particular elements of that array:

```
list[3] = 999;
list[0] = list[5];
list[12] = list[13] + list[14];

for (int value : list)
    System.out.print(value + "  ");
```

```
50   10   20   999   40   50   60   70   80   90   100   110   270   130   140
```

Rephactor

## Arrays

An array can also be created using an initialization list, which both creates the array and fills it with values

```
int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
    31, 37, 41, 43, 47, 53, 59, 61, 67, 71};
```

If an initialization list is used, the new operator is not

The array size is determined by the number of elements in the list

## The for-each Statement

Both arrays and collections allow you to manage multiple values in a single object

It's often necessary to traverse the elements – access them one at a time

A for-each statement traverses the elements of an array or collection without using an index variable

It's often more convenient than a standard for loop

## The for-each Statement

```java
int[] list = {44, 88, 11, 77, 55, 66, 33, 99, 22};

for (int number : list)
    System.out.println(number);
```

```
44
88
11
77
55
66
33
99
22
```

That loop would be read "for each number in list…"

## The for-each Statement

Suppose library is a collection (such as an ArrayList) of Book objects

```
for (Book book : library)
{
    System.out.println(book.getTitle());
    System.out.println(book.getAuthor());
    System.out.println();
}
```

The condition of a for-each loop is implied—the loop continues as long as there are elements to process

## The for-each Statement

A for-each loop is often convenient, but it has limitations

You can't, for instance, use it set the values in an array

```
for (int element : list)
    element = 0;
```

❌

All this does is overwrite the value of the variable element

The for-each loop is also not helpful if you want to traverse the elements backwards, or visit every other one, for instance

The standard for loop is better in those situations

## Arrays of Objects

An array can have elements that are primitive values or objects. An array of objects uses the class name as the element type.

Creating an array of objects looks like this:

```
String[] words = new String[5];
```

This array is declared to hold 5 strings, but it doesn't have any values yet.

## Filling an Array

To fill the array of objects, an object is assigned to each element of the array, like this:

```
words[0] = "hullaballoo";
words[1] = "sozzled";
words[2] = "lollygag";
words[3] = "flabbergast";
words[4] = "skirl";
```

Since array indexes are numbered starting with 0, the last element is 1 less than the length of the array.

# Iterating an Array

To iterate through the array of objects, the for-each loop makes it simple:

```
for (String word : words)
    System.out.println(word.toUpperCase())
```

Each element in turn is assigned to word, converted to uppercase, and then printed.

```
HULLABALLOO
SOZZLED
LOLLYGAG
FLABBERGAST
SKIRL
```

# Another Initialization

An initialization list is an alternate way to create a fill an array of objects.

```
String words[] = {"hullaballoo", "sozzled",
    "lollygag", "flabbergast", "skirl" };
```

This allows the array to be created and initialized in a single step.

## Arrays of Person Objects

Here's another example where an array of objects contains Person objects rather than strings:

```
Person[] athletes = new Person[4];

athletes[0] = new Person("Michael", "Jordon");
athletes[1] = new Person("Babe", "Ruth");
athletes[2] = new Person("Martina", "Navratilova");
athletes[3] = new Person("Wayne", "Gretzky");

for (Person athlete : athletes)
    System.out.println(athlete.getLastName() + ", " +
                       athlete.getFirstName());
```

The array is created, initialized, and the elements are printed.