# Object-Oriented Programming Fundamentals

## Lecture/Workshop (Week 10)

### Arrays of Objects

*Note: After the workshop, you can make a directory in your latcs account for this worksheet and cd into it. Some files in this worksheet can be copied from the library area into your current directory using:*

```
cp  /home/1st/csilib/cse1oof/ws10/*  .
```

---

We have seen that creating arrays in Java is a 2 step process.

**First,** we need to declare the **object reference** (variable) that will contain the starting memory address of the array, when we instantiate (construct) the array, using the **`new`** operator. We indicate that this object reference is going to contain the address of an array by using the **`[ ]`**

For example, an array of ints

```
int [ ] intArray;
```

this is **not** an array declaration, just the 4 byte object reference that can contain the starting memory address of an array of **`ints`**, when we call the **`new`** operator.

Second, we allocate the memory for the array using the **`new`** operator.

For example, using the object reference declaration above

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

Arrays are contiguous, meaning that the memory allocated must be in one continuous block. As well, each element of an array must be of the same type. The type comes from our object reference declaration. In the example above it is **`int`**. So each element of the example array, **`intArray`**, can contain an **`int`**.

If we have an array of primitive data types, then each element of the array is initialized to the default value for that primitive data type.

see **`ArrayExample10_1.java`** in the **`ws10`** library area for the default values of arrays of primitive data types and an example of method overloading.

Arrays of objects are actually **arrays of object references**.

To declare an array of object (references) we do the same things as we do for arrays of primitive data types.

Put the type first. It is just that in this case the type is a class, not a primitive data type.

Refering to the **Person** class handout that accompanies this Lecture / Workshop

```
Person [ ] people;
```

As before, **people** is NOT an array of **Person** class objects, it is not even an array of any kind, it is an object reference that can contain the starting memory address of an array of **Person** class object references.

As with all arrays, we have to use the **new** operator to allocate the memory for the array.

```
people = new Person[ 10 ];
```

We still do not have any **Person** objects. What we have is an array of object references to **Person** objects, 4 byte variables that can contain the starting memory address of a **Person** object.

The contents of each element of the array is **null**, indicating no address. This is true of any array of object references.

Even though there is a **getName( )** method in the **Person** class, if we were to write this line of code.

```
System.out.println( people[ 1 ].getName( ) );
```

what java would see would be:

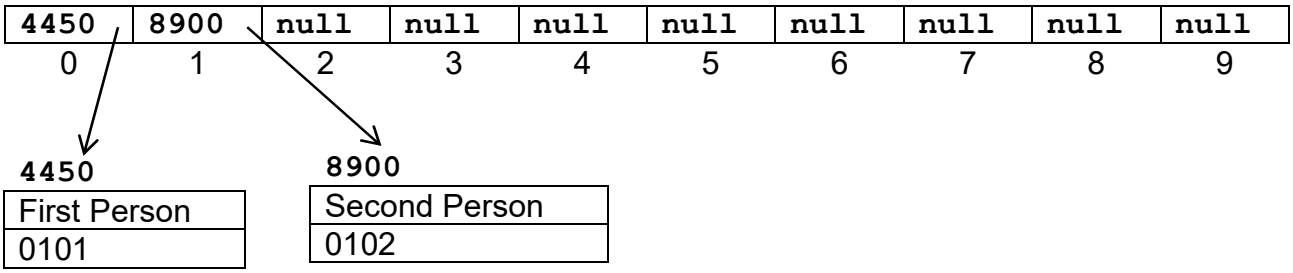
```
System.out.println( null.getName( ) );
```

because the contents of index 1 of the **people** array is **null**. This would cause a **nullPointerException** (crash the program) when we tried to run the program.

So we need to instantiate **Person** objects first, storing the starting memory addresses that **new** returns in the **people** array.

```
people[ 0 ] = new Person( "First Person", "0101" );
people[ 1 ] = new Person("Second Person", "0102" );
```

Now our array (and the program) would look like this:




Now if we try to run the line of code

```
System.out.println( people[ 1 ].getName( ) );
  // look in people[ 1 ] - memory address and go to that memory address if it is not null
```

what java would see would be:

```
System.out.println( (object at memory address 8900).getName( ) );
```

Since there is a **`Person`** object at this memory address, and **`Person`** objects have a **`getName( )`** method, now our program returns a copy of the name.

Using the **`Person`** class and the following skeleton of a driver program, complete the **Tasks** that add methods to the driver program **`PersonTester.java`**

```
import java.util.Scanner;

public class PersonTester
{
    private Scanner kb;
    private Person [ ] people;
    private final int SIZE = 15;
    private int current;

    public static void main( String [ ] args )
    {
         PersonTester pt = new PersonTester( );
    }

}
```

**Task 1**

Write the java code for the **`PersonTester( )`** constructor. This constructor should instantiate the array and do any other initialization required

## Task 2

Write the code for a method named **`addPerson`**. This method must check that there is space in the array. If there is space in the array then the user is prompted (asked) to enter the details of a **`Person`**, which is then added to the array

```
public void addPerson( )
{









}
```

## Task 3

Write the code for a method named **`display`**. This method displays the contents of the **`people`** array. This method should display an appropriate message if there are no **`Person`** objects in the array.

```
public void display( )
{




}
```

## Task 4

Write a method **`search`**, which takes as a parameter an existing **`Person`** object. This method returns the index in the **`people`** array of a **`Person`** object that matches the **`Person`** object that is passed in as a parameter. The objects match if the **`names`** and the **`phone numbers`** are the same. This method returns -1 if the **`Person`** is not found in the array.

```
public int search( Person p )
{




}
```

## Task 5

Write a method **`removePerson`** that takes as a parameter the **`index`** of the **`Person`** to remove from the array. This method returns the **`Person`** that is removed from the array.

Note that this method must check that the **`index`** passed in as a parameter is valid. If the **`index`** is not valid then the method displays an appropriate message to the screen (what is the value of the return?)

If the **`index`** is valid then the **`Person`** at that **`index`** is returned, but before returning the **`Person`**, the array must be compacted. That is, the existing **`Person`** objects (if any) are moved down one **`index`** and the last **`Person`** object is set to **`null`**. Remember that the value of **`current`** needs to be changed as there is one less **`Person`** object in the array.

Also note, when returning the **`Person`** object, consider privacy leaks, make sure your method does not have a privacy leak.

```
public Person removePerson( int index )
{




}
```