1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 14: Basic memory concepts

References and learning objects

- Today's sessions are largely based on ***Java: How to Program***, Chapter 7, and ***Java in a Nutshell***
- After today's session, you should be able to:
    - Explain the basic memory concepts, such as difference between destructive and nondestructive process, as well as pass-by-value and pass-by-reference
    - Implement algorithms using recursion
    - Handle exceptions with `try…catch` structure in your Java code
    - Define and use your own custom exceptions

Memory concepts

- Every variable in Java has a *name*, a *type*, a *size* (in bytes), and a *value*
    - Name differentiates variables from each other
    - Size depends on type
    - Value can change during the execution of the program (unless the variable is a final variable)
- When a variable is initialised, JVM allocates space for the variable in the computer's working memory
    - When new value is assigned to the variable, the old value is lost (destructive process)
    - When the value of a variable is read, it can be used but it does not change (nondestructive process)

Memory concepts example

```java
public class MemoryExample {
  public static void main(String[] args){
    int x = 5;
    int y = 10;
    int z = x + y;
    System.out.println("Result: " + z);
  }
}
```

```
$ java MemoryExample
```

Program starts in the beginning of method **`main`**

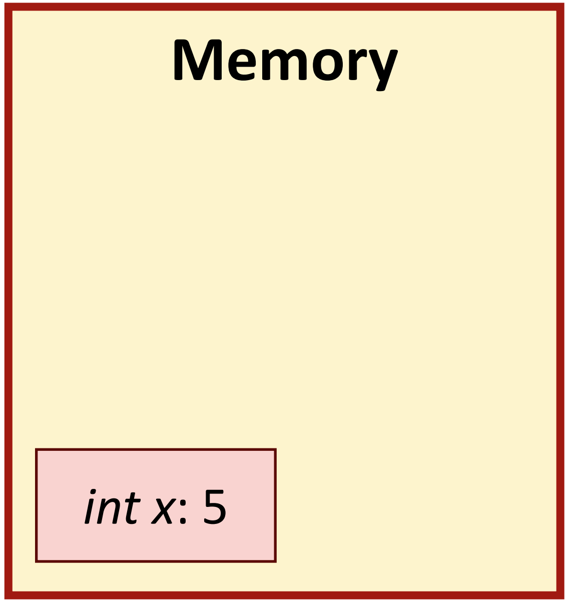
Memory

Memory concepts: initialise x

```java
public class MemoryExample {
  public static void main(String[] args){
    int x = 5;
    int y = 10;
    int z = x + y;
    System.out.println("Result: " + z);
  }
}
```

```
$ java MemoryExample
```

Space is allocated and value is assigned to variable **x**

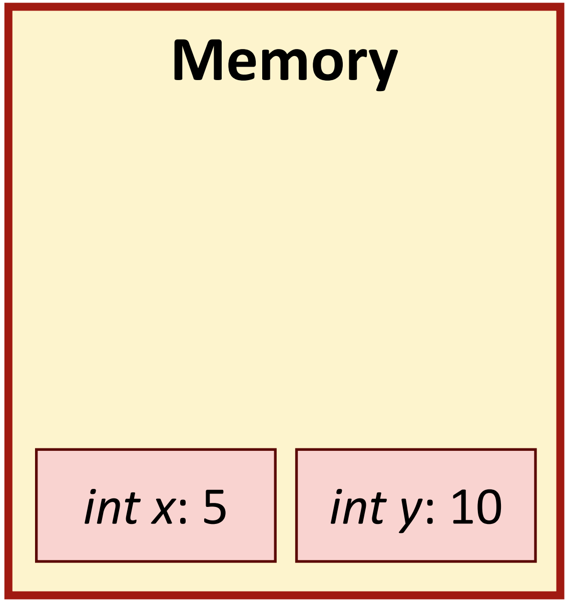


Memory concepts: initialise y

```java
public class MemoryExample {
  public static void main(String[] args){
    int x = 5;
    int y = 10;
    int z = x + y;
    System.out.println("Result: " + z);
  }
}
```

```
$ java MemoryExample
```

Space is allocated and value is assigned to variable **y**

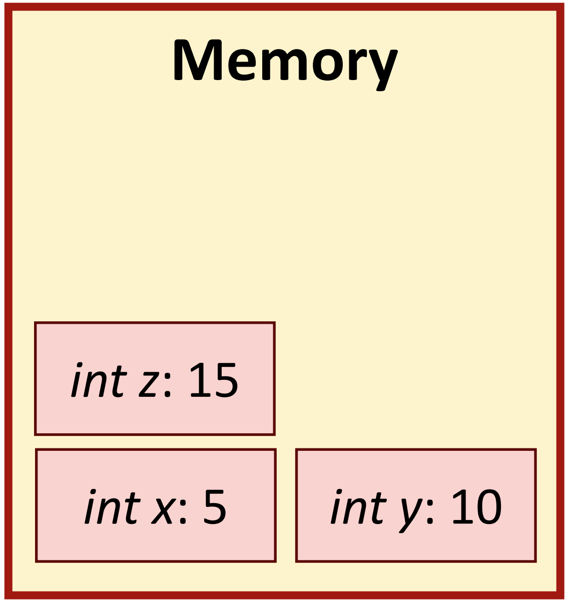


Memory concepts: initialise z

```java
public class MemoryExample {
  public static void main(String[] args){
    int x = 5;
    int y = 10;
    int z = x + y;
    System.out.println("Result: " + z);
  }
}
```

```
$ java MemoryExample
```

Space is allocated and value is assigned to variable **z**, using variables **x** and **y**. This is a *nondestructive* memory process, since the values of **x** and **y** are not lost.

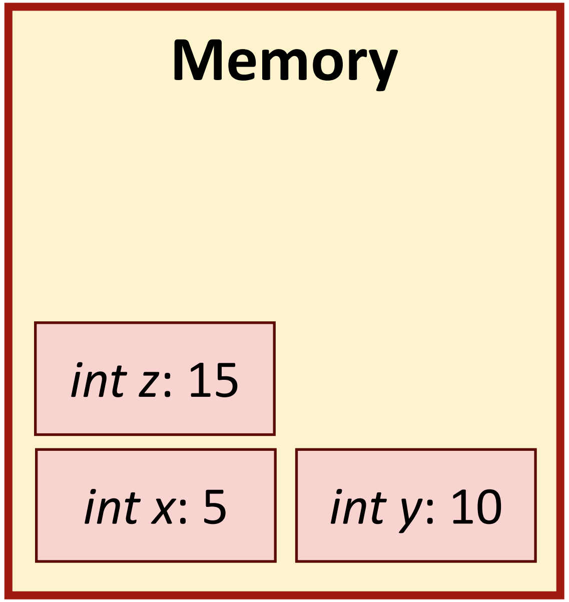


Memory concepts: use variable z

```java
public class MemoryExample {
  public static void main(String[] args){
    int x = 5;
    int y = 10;
    int z = x + y;
    System.out.println("Result: " + z);
  }
}
```

```
$ java MemoryExample
Result: 15
```

Finally, value of **z**, is used as console output. This is also a *nondestructive* memory process, since the value of **z** is not lost.



Memory concepts: execution ends

```java
public class MemoryExample {
  public static void main(String[] args){
    int x = 5;
    int y = 10;
    int z = x + y;
    System.out.println("Result: " + z);
  }
}
```

```
$ java MemoryExample
Result: 15
$
```

Execution of the program ends, and the working memory is cleared.

Memory

1495 UNIVERSITY OF ABERDEEN

Memory concept example 2

```
public class MemoryExample2 {
  public static void main(String[] args){
    int x = 5;
    x = x + 10;
    System.out.println("Result: " + x);
  }
}
```

```
$ java MemoryExample2
```

Also in the second example, program starts in the beginning of method **`main`**

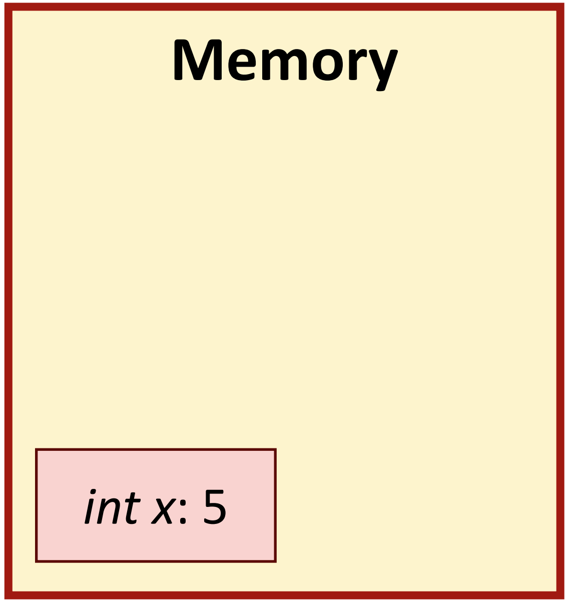
Memory

Memory concepts 2: initialise x

```java
public class MemoryExample2 {
  public static void main(String[] args){
    int x = 5;
    x = x + 10;
    System.out.println("Result: " + x);
  }
}
```

```
$ java MemoryExample2
```

Space is allocated and value is assigned to variable **x**



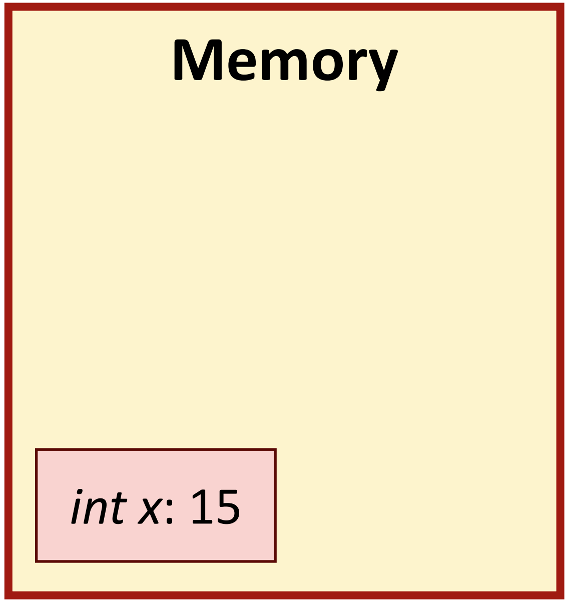
UNIVERSITY OF ABERDEEN

Memory concepts 2: reassign value to x

```java
public class MemoryExample2 {
  public static void main(String[] args){
    int x = 5;
    x = x + 10;
    System.out.println("Result: " + x);
  }
}
```

```
$ java MemoryExample2
```

New value is assigned to **x** and the old value is lost; this is a *destructive* memory process



Memory concepts 2: use value of x

```java
public class MemoryExample2 {
  public static void main(String[] args){
    int x = 5;
    x = x + 10;
    System.out.println("Result: " + x);
  }
}
```

```
$ java MemoryExample2
Result: 15
```

Value of **x** is used but not changed

Memory

int x: 15

Memory concepts 2: execution ends

```java
public class MemoryExample2 {
  public static void main(String[] args){
    int x = 5;
    x = x + 10;
    System.out.println("Result: " + x);
  }
}
```

```
$ java MemoryExample2
Result: 15
```

Execution ends and memory is cleared

Memory

Reference type variables

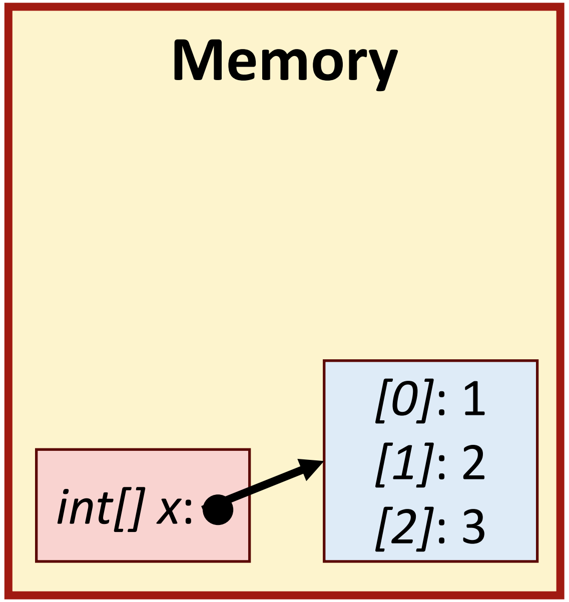
- Variables that are not primitive type variables are *reference type variables*: the variable contains a reference (pointer) to the memory location with the actual data
    - Arrays and references to objects are reference type variables
- Be careful to note the difference between modifying the variable (pointer) and modifying the data it refers to!

Reference type variable example (x[])

```java
public class RefVarExample {
  public static void main(String[] args){
    int[] x = {1,2,3};
    int[] y = x;
    int[] z = {0,0,0};
    System.out.println("Array x: " + x.hashCode());
    System.out.println("Array y: " + y.hashCode());
    System.out.println("Array z: " + z.hashCode());
  }
}
```

```
$ java RefVarExample
```

Space is allocated for the reference variable **x** as well as the area containing the values of **x[0]**, **x[1]**, and **x[2]**

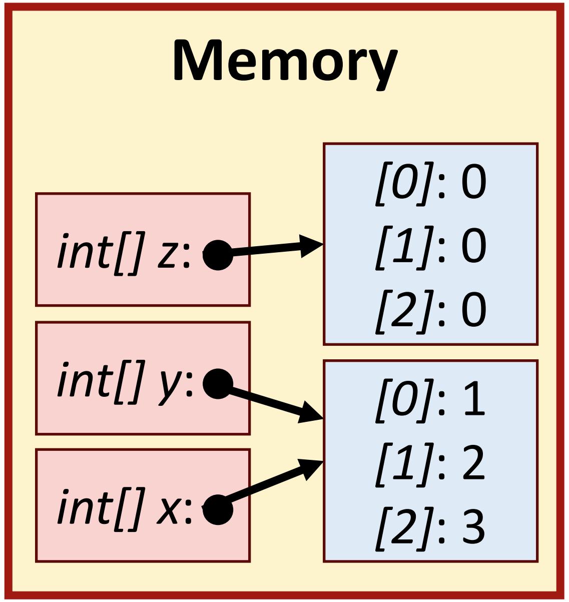


Reference type variable example (z[])

```java
public class RefVarExample {
  public static void main(String[] args){
    int[] x = {1,2,3};
    int[] y = x;
    int[] z = {0,0,0};
    System.out.println("Array x: " + x.hashCode());
    System.out.println("Array y: " + y.hashCode());
    System.out.println("Array z: " + z.hashCode());
  }
}
```

```
$ java RefVarExample
```

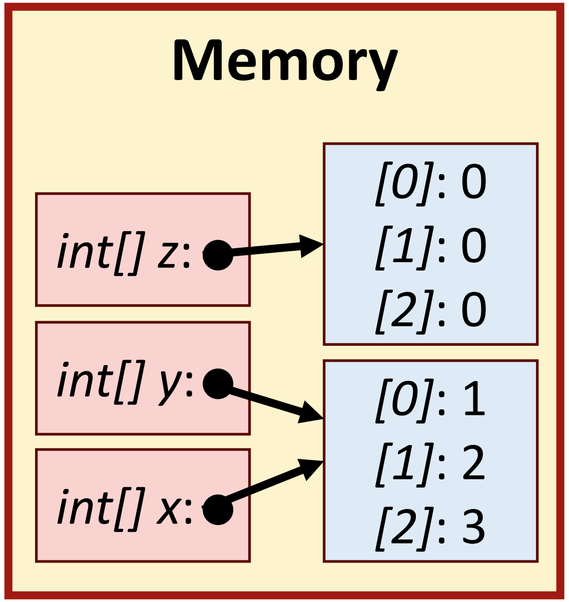
Space is allocated for the reference variable **`z`** and the values of **`z[0]`**, **`z[1]`**, and **`z[2]`**



Reference type variable example (hashes)

```java
public class RefVarExample {
  public static void main(String[] args){
    int[] x = {1,2,3};
    int[] y = x;
    int[] z = {0,0,0};
    System.out.println("Array x: " + x.hashCode());
    System.out.println("Array y: " + y.hashCode());
    System.out.println("Array z: " + z.hashCode());
  }
}
```

```
$ java RefVarExample
Array x: 1510467688
```

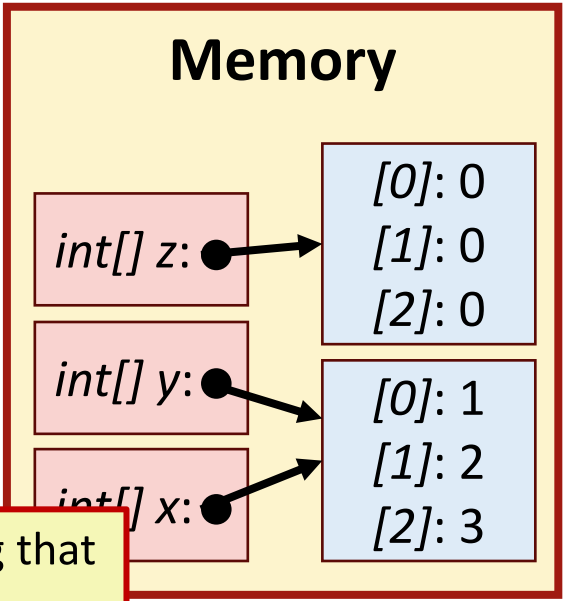


We can print the hash codes, i.e., unique representations for memory locations, for each reference variable

Reference type variable example (hashes)

```java
public class RefVarExample {
  public static void main(String[] args){
    int[] x = {1,2,3};
    int[] y = x;
    int[] z = {0,0,0};
    System.out.println("Array x: " + x.hashCode());
    System.out.println("Array y: " + y.hashCode());
    System.out.println("Array z: " + z.hashCode());
  }
}
```

```
$ java RefVarExample
Array x: 1510467688
Array y: 1510467688
```

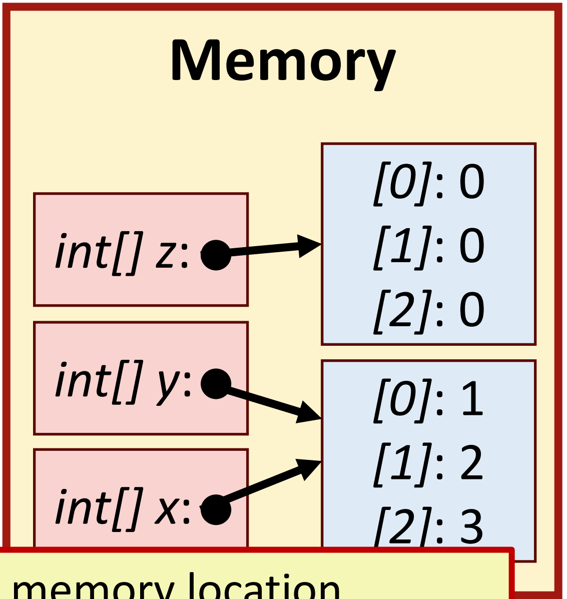


Hash codes for **x** and **y** are the same, indicating that they indeed refer to the same memory location

Reference type variable example (hashes)

```java
public class RefVarExample {
  public static void main(String[] args){
    int[] x = {1,2,3};
    int[] y = x;
    int[] z = {0,0,0};
    System.out.println("Array x: " + x.hashCode());
    System.out.println("Array y: " + y.hashCode());
    System.out.println("Array z: " + z.hashCode());
  }
}
```

```
$ java RefVarExample
Array x: 1510467688
Array y: 1510467688
Array z: 868693306
```



Hash code for **z** is different, indicating different memory location

Reference type variable example (end)

```java
public class RefVarExample {
  public static void main(String[] args){
    int[] x = {1,2,3};
    int[] y = x;
    int[] z = {0,0,0};
    System.out.println("Array x: " + x.hashCode());
    System.out.println("Array y: " + y.hashCode());
    System.out.println("Array z: " + z.hashCode());
  }
}
```

```
$ java RefVarExample
Array x: 1510467688
Array y: 1510467688
Array z: 868693306
$
```

Execution ends and memory is cleared.

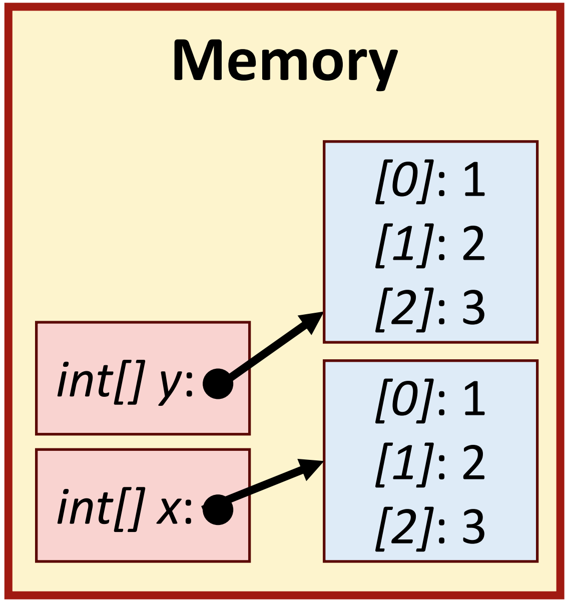
Memory

Deep copy method clone()

```java
public class RefVarExample {
  public static void main(String[] args){
    int[] x = {1,2,3};
    int[] y = x.clone();
    int[] z = {0,0,0};
    System.out.println("Array x: " + x.hashCode());
    System.out.println("Array y: " + y.hashCode());
    System.out.println("Array z: " + z.hashCode());
  }
}
```

```
$ java RefVarExample
```

If you want to make a full copy of the referred data and not just another reference, you can use method **`clone()`**



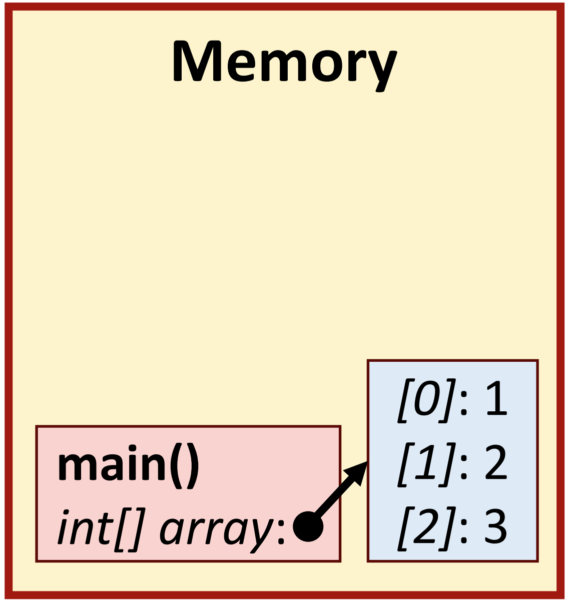
Pass-by-value vs. pass-by-reference

- In many programming languages (e.g., C++), there are two ways to pass arguments in method calls: *pass-by-value* (or *call-by-value*) and *pass-by-reference* (*call-by-reference*)
  - **Pass by value:** a copy of the argument's value is passed to the called method. The called method works exclusively with the copy. Changes to the copy do not affect the original variable's value.
  - **Pass by reference:** the called method can access the argument's value in the caller directly and modify that data, if necessary. This can improve performance by eliminating the need to copy large amounts of data.
- In Java, you cannot choose: *all arguments are passed by value*

Pass-by-value example (1)

```java
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int element) {
    element *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

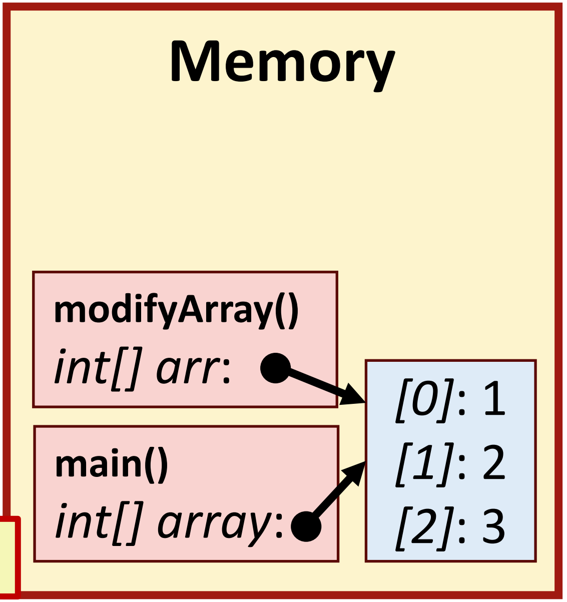
Start program, assign values to array



Pass-by-value example (2)

```
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int element) {
    element *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

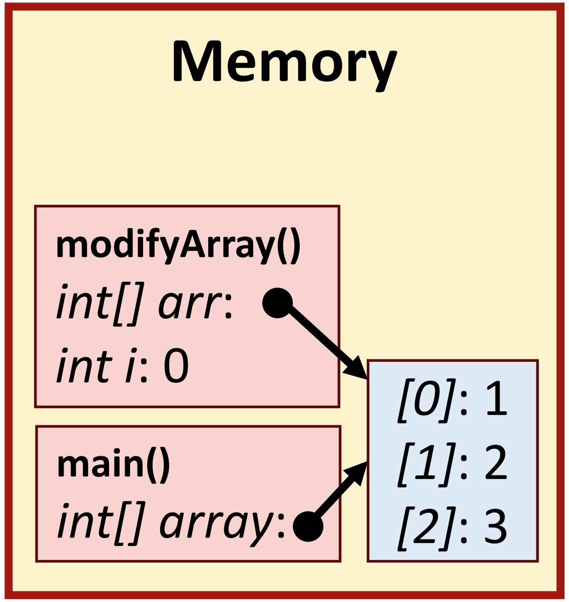
Method `main()` calls method `modifyArray()`



Pass-by-value example (3)

```java
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int element) {
    element *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

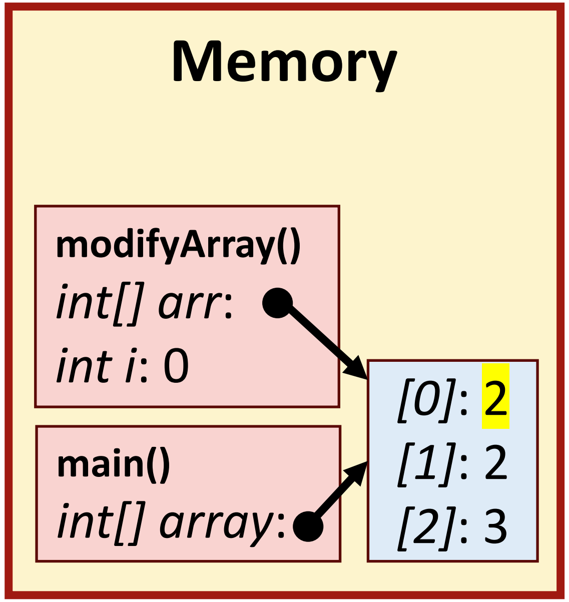
Start a loop to modify elements of arr[ ]



Pass-by-value example (4)

```
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int element) {
    element *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

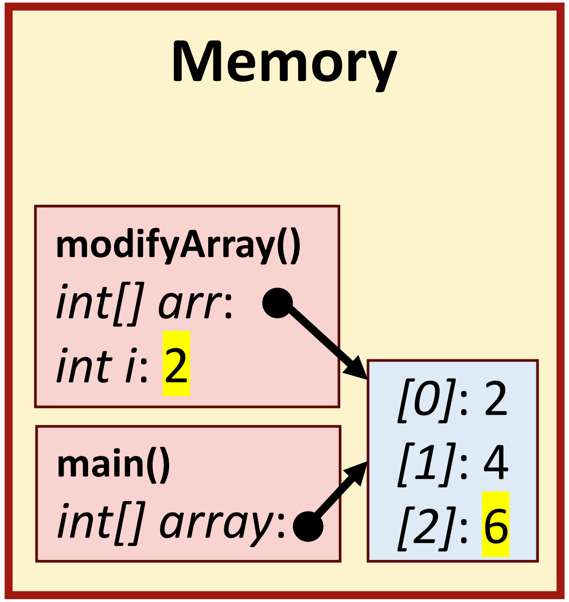
Modify `arr[0]`



Pass-by-value example (6)

```java
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int element) {
    element *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

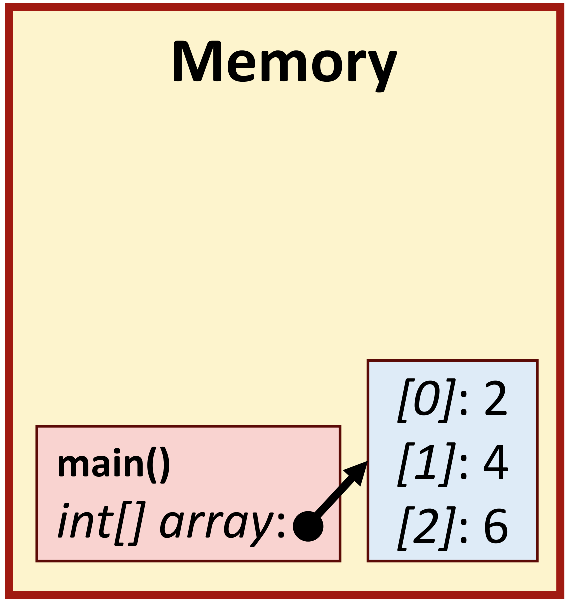
Modify arr[2]



Pass-by-value example (7)

```java
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int element) {
    element *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

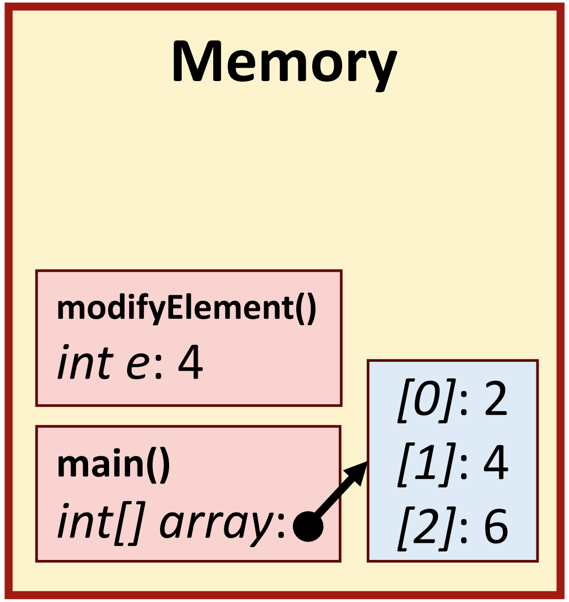
Return to `main()`, `array` has been modified.
Call `modifyElement()` next.



Pass-by-value example (8)

```java
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int e) {
    e *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

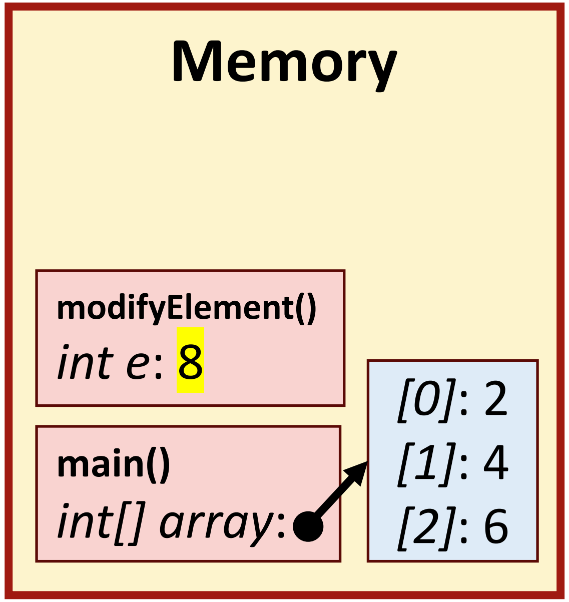
Note that the passed variable e contains a *copy* of element array[3]!



Pass-by-value example (9)

```java
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int e) {
    e *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

Value of e is modified, but the change is not visible in main()

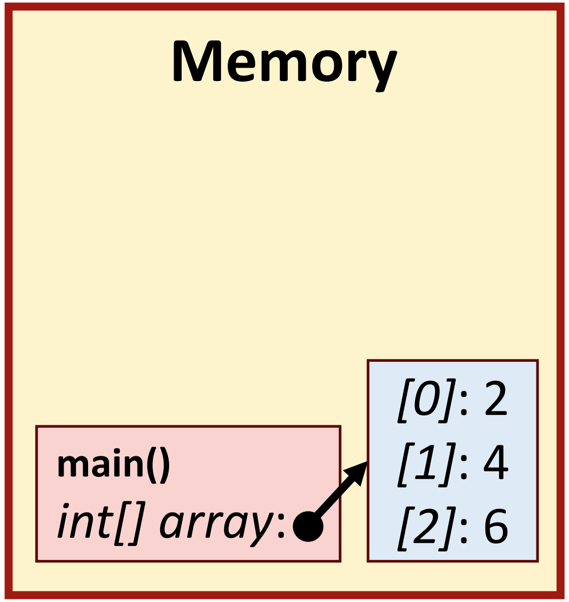


UNIVERSITY OF ABERDEEN

Pass-by-value example (10)

```
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int e) {
    e *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

Return to main()



Pass-by-value example (11)

```
public class RefVarExample {
  public static void modifyArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] *= 2;
    }
  }
  public static void modifyElement(int e) {
    e *= 2;
  }
  public static void main(String[] args) {
    int[] array = {1, 2, 3, 4, 5};
    modifyArray(array);
    modifyElement(array[3]);
  }
}
```

Execution ends, memory is cleared

Memory

Questions, comments?