JAVA TIP NO.

github.com/Teabeans/Java_Learn/

UWB_TADEMO_000.pdf

- PASSING ARGUMENTS BY VALUE OR REFERENCE-

(OR BY REFERENCE VALUE)

Prepared For:
The University of Washington Bothell
CSS TA Admissions Committee
2019.08

By:
Tim Lum
twhlum@gmail.com
2019.07.16

0 - Prerequisites:

Primitive Data Types:

Any form of data that is provided by the programming language as a foundational building block. Support for these data types are provided by the language. In Java, there are (8) primitives:

- Byte
- Short
- Int
- Long
- Float
- Double
- Boolean
- Char

Note:
Java primitives are also called 'Value Types' in that their bit states directly store their values.
In Java, it is not possible to create additional value data types.

Composite Data Types:

Any data object which aggregates or combines primitive data types. These include, among many others:

- Arrays
- Class Objects
- Strings

Variable Declaration and Assignment:

In Java, instructing the computer to set aside named memory of a particular type for later reference (declaration), then instructing the computer to give that memory a particular value or state (assignment).

Note:
Assignment' and 'Initialization' are often used synonymously. To be precise 'initialization' refers to an act of assignment that occurs at the same time as declaration.

Method Definitions:

Sometimes also called a 'function', methods are a reusable set of procedures similar to a recipe or instruction manual. They are comprised of a **signature** (sometimes called a 'header' defining the name, inputs, outputs, and some other details of the method) and a **body** (where the actual instructions reside).

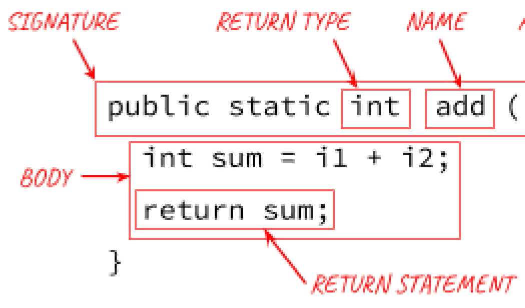
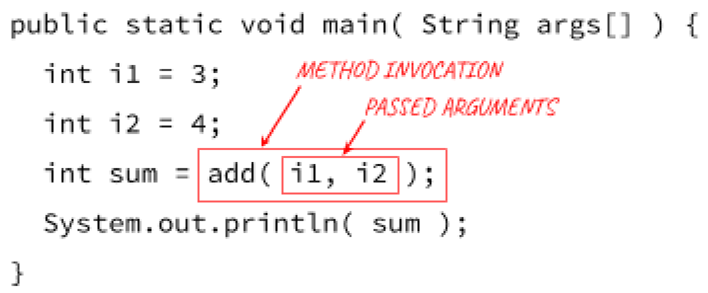

```
public static int add ( int i1, int i2 ) {
    int sum = i1 + i2;
    return sum;
}
```

Fig 0.0 - A method definition for a simple addition operation.

For the purposes of this TIP, we are concerned primarily with the arguments and return statement.

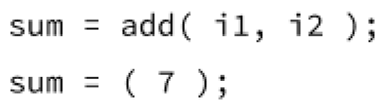
Argument Passing:
When a method is invoked (also sometimes referred to as 'calling' or 'using' a method), appropriate arguments must be passed in to the method based upon the method's signature.



```
public static void main( String args[] ) {
  int i1 = 3;
  int i2 = 4;
  int sum = add( i1, i2 );
  System.out.println( sum );
}
```

Fig 0.1 - Arguments being passed to a method during invocation

Argument Returns:
Where a method definition defines a return value, this is the value that the method invocation resolves to upon completion.



```
sum = add( i1, i2 );
sum = ( 7 );
```

Fig 0.2 - Because 'add(i1, i2)' returns the value 7, these two lines of code have the same effect.

Note:
If a method's return value is not properly assigned at the time of invocation, it will be lost.

1 - Problem Statement (with Primitives):

When passing primitive arguments to a method, changes made inside the method appear to be lost. Values appear to revert to their prior state. For instance, given a function called '`increment`' that appears to add (1) to a number:

```
public static void increment ( int i) {
   i = i + 1;
}
```

And an associated invocation as follows:

```
public static void main ( String arg[] ) {
   int number = 0;
   increment( number );
   System.out.println( 'number == ' + number );
}
```

We might expect the state of '`number`' to be equal to (1). This is not the case, as the compiler reveals the state of 'number' to remain as (0) after the method has performed its work.

```
number == 0
```

2 - Correction for Primitives:

When passing primitives, in order for the changes to 'stick', two corrections must be made. First, the function definition must return the new value. This may be accomplished by first **adding the appropriate return type to the method signature** and **specifying a return statement**.

```
public static int increment ( int i) {
   i = i + 1;
   return i;
}
```

Second, **the returned value must be assigned** where the method was invoked.

```
public static void main ( String arg[] ) {
   int number = 0;
   number = increment( number ); ⇐ THIS INVOCATION RESOLVES TO THE RETURN VALUE
   System.out.println( 'number == ' + number );
}
```

The console now reports the expected value.

```
number == 1
```

3 - 'Problem' Statement with Composites:

It should be noted, however, that composite data types don't express the same behavior. Given a function which increments all values in an array:

```
public static void incrementArray( int[] arr ) {
  for( int i = 0 ; i < arr.length ; i++ ) {
    arr[i] = arr[i] + 1;
  }
}
```

And an associated invocation as follows:

```
public static void main( String []args ) {
  int[] array = { 2, 4, 8, 16, 32, 64, 128, 256 };
  printArray( array ); ⇐ METHOD DEFINITION IN THE APPENDIX
  incrementArray( array );
  printArray( array );
}
```

The console reports the (perhaps naively) expected values:

```
[ 2, 4, 8, 16, 32, 64, 128, 256 ]
[ 3, 5, 9, 17, 33, 65, 129, 257 ]
```

No further correction is required and modifications made inside the '`incrementArray`' method do not require a return and associated assignment in order to persist.

4 - Explanation

So what's going on? Is Java just inconsistent in how it handles primitives versus composites? Not quite! A hint as to why may be found by simply printing the array object to console at various points during the program's execution. In the following case, that's before and after incrementing the contents:

```
public static void main( String []args ) {
  int[] array = { 2, 4, 8, 16, 32, 64, 128, 256 };
  System.out.println( array );
  incrementArray( array );
  System.out.println( array );
}
```

The console appears to report… gibberish (and your results will vary):

```
[I@6d06d69c ⇐ BEFORE CALLING INCREMENT()
[I@6d06d69c ⇐ AFTER CALLING INCREMENT()
```

The output is the result of the default toString() method of the array object, but why that particular value and how is it possible that it remains the same despite the array changing?

What we're actually looking at is a representation of the object's memory address, expressed in hexadecimal notation. This is a type of **reference**.

Note:
Hexadecimal is a human-readable means of representing bits, most often 8-bit bytes. A hexadecimal digit ranges from 0-9 and A-F, and tend to come in pairs. With 16 states per digit, a hexadecimal pair can represent 256 discrete states (16 x 16). This is exactly the same number of states that can be represented by 8 bits (or 2 ^ 8, which is 256).

Usually, the author of a hexadecimal value should assign a prefix of '`0x`' to denote that the characters to follow are formatted as hex, since some ambiguity can exist (e.g. '`10`' and '`0x10`' are very different) if context is lacking.

https://en.wikipedia.org/wiki/Hexadecimal

You'll come back to references in the future. For now, you can imagine that a reference is like an object's business card or mailing address, which the program uses to find and interact with the object in question.

Note:
Java composite types are also sometimes called 'Reference Types', reflecting the fact that they are memory references which inform the computer of where to look for the object.

Which brings us back to the question at hand: what is happening when we pass a composite data object to a method? Java is actually passing a reference (by value) during the method invocation, which the method is able to use to alter the object at its place of residence. When that object is inspected at any point in the future, such changes will persist.

To summarize, the value of the reference does not change. What the reference points to does.

Note:
As a final detail, there is a slight distinction between passing the reference itself and passing the value of the reference. Using the analogy of a business card, these situations may be understood as follows:

Passing the reference - The card is given to the method
Passing the reference by value - The method makes a copy of the card

Java passes the reference by value; it does not pass the reference directly

5 - Challenge

With this in mind, consider the following method 'replaceArray':

```
public static int[] replaceArray( int[] oldArray ) {
  int[] newArray = new int[oldArray.length]; ⇐ MAKE AN EMPTY ARRAY OF SAME SIZE
  for( int i = 0 ; i < oldArray.length ; i++ ) {
    newArray[i] = oldArray[i]; ⇐ COPY ALL VALUES ACROSS
  }
  oldArray = newArray;
  return oldArray;
}
```

Using method definitions seen previously (refer to the Appendix if you need a refresher), fill in the missing elements:

```
public static void main( String []args ) {
  int[] oldArray = { 2, 4, 8, 16, 32, 64, 128, 256 };

  int[] newArray = ____________( oldArray );
  incrementArray( newArray );

  printArray( ________ );
  printArray( ________ );
}
```

Such that the program generates the output:

```
[ 2, 4, 8, 16, 32, 64, 128, 256 ]
[ 3, 5, 9, 17, 33, 65, 129, 257 ]
```

Question:

In the method 'replaceArray', the variable oldArray (passed in as the initial argument) is replaced before being returned.

```
public static int[] replaceArray( int[] oldArray ) {
  int[] newArray = new int[oldArray.length];
  for( int i = 0 ; i < oldArray.length ; i++ ) {
    newArray[i] = oldArray[i];
  }
  oldArray = newArray;
  return oldArray;
}
```

1) What type of data was overwritten by this line of code?
2) Should you expect this change to persist outside of the method?
3) How might you shorten this method while maintaining identical program behavior?

6 - Rationale

So why might Java make this distinction? Why engage the additional complexity of using and passing references to objects, rather than just copying the objects into and out of methods in a similar fashion to how primitives are handled?

As with many reasons in computer science, the likely answer is time and space performance.

Note:
Computer or processor time and memory or storage space are two of the most important and common metrics you'll encounter in computer science. Except in very specific applications, the better solution will take less time (process faster) and/or use less space to complete.

Because a reference uses little space (in most cases, a single 'word' of either 4 or 8 bytes), Java is able to read, copy, and handle references very easily. The object being referenced, however, can be mind-bogglingly large, with a maximally sized int array weighing in among the gigabytes. If Java were to attempt to copy an object that large into a method (then copy it back out at the return), the memory requirements could overwhelm even high performance computers (to say nothing of the time differences involved).

Question:

How large an array can your Java Virtual Machine (JVM) handle before throwing an error?
How many bytes does your JVM use to represent an int?
How many megabytes (MB) does your JVM need to hold a maximum sized int array?

This principle holds true for almost all but the simplest composite classes, including those that may be defined by the user (that's you!). Java has no way of knowing in advance how large and cumbersome your classes will be, but by adopting the strategy of consistently using references, it never really has to care.

Primitives, however, can never be found in a large size, hence there is no performance advantage to passing them by reference; a direct copy of their value works at about the same speed and if the 4-byte memory requirement is breaking the program, there are probably other issues at play.

Note:
Immutable objects such as Strings are still passed by reference value. Their assignment operators, however, behave differently than mutable objects. This may have the effect of making them appear to be inconsistent compared to mutable composite types, but be assured that this is not the case.

9 - Appendix

```
// A simple method to print the contents of an integer array
public static void printArray( int[] arr ) {
  System.out.print("[ ");
  for( int i = 0 ; i < arr.length-1 ; i++ ) {
    System.out.print( arr[i] + ", " );
  }
  System.out.print( arr[arr.length - 1] );
  System.out.println(" ]");
}
```

Note:
A language-provided functioning for resolving an array to a String may be invoked with the following:

`java.util.Arrays.toString( array )`

While it may be more rigid in its formatting, it gets the job done and can be used on arrays of varying data types.

```
// A method to increment all values in an int array withOUT a return
public static void incrementArray( int[] arr ) {
  for( int i = 0 ; i < arr.length ; i++ ) {
    arr[i] = arr[i] + 1;
  }
}
```

```
// A method to increment all values in an int array with a return
public static int[] incrementArray( int[] arr ) {
  for( int i = 0 ; i < arr.length ; i++ ) {
    arr[i] = arr[i] + 1;
  }
  return arr;
}
```

```
// A method to generate a deep copy of an array and return the new reference address
public static int[] replaceArray( int[] oldArray ) {
  int[] newArray = new int[oldArray.length];
  for( int i = 0 ; i < oldArray.length ; i++ ) {
    newArray[i] = oldArray[i];
  }
  return newArray;
}
```

EXERCISE 00

Using the following driver, modify the method 'passTest' and 'main' to confirm whether the following datatypes (4 minimum) are passed by value or by reference value:

```
public static void passTest( int i ) {
  i++;
}

public static void main( String []args ) {
  int i = 0;
  System.out.println( "Before: " + i );

  passTest( i );

  System.out.println( "After : " + i );
}
```

PRIMITIVES	byte	
	short	
	int	
	long	
	float	
	double	
	boolean	
	char	
COMPOSITES	Array[]	
	String	
	Queue	
	Stack	
	Deque	
	LinkedList	
	Point	
	HashMap	
	Graph	
	TreeMap	