Passing the Value of a Parameter to a Method

James Brucker

Method Parameters

Pass data to methods using parameters.

```
public void deposit( long amount ) {
```

access   return value   parameter(s)

Actual argument type must be compatible with parameter:

```
BankAccount acct = new BankAccount();
long a = 1000000L;
int    b = 1000;
acct.deposit( a );        // OK ... (long) a matches parameter type
acct.deposit( b );        // OK ...but why?
acct.deposit( 50.25 );    // ERROR ... incompatible type
```

Method Parameters, again

- Both the number and type of argument must match the method *signature*.

```
// overloaded method:
int max(int m, int n) {   . . .   }
float max(float x, float y) {   . . .   }
float max(float x, float y, float z) { . . . }
```

- Which "max" method will be called?

```
int   r = max( 20, 45 );
float q = max( 20, 33.F);  // mixed arguments
float z = max(1, 2, 3.5F); // mixed arguments
int   p = (int) max( 2 , -9.3F );
```

Parameters and Arguments

- Method Parameter means the variable in a method signature.
- Argument means the value that is given for a parameter when you call a method. Also called "parameter value".
- Example:

```
public void increment( int a ) { // a is parameter
     a = a + 1;
}
public static void main(String [] args) {
     int x = 10;
     increment( x );                // x is argument
```

Arguments are Passed by Value

- In Java, arguments are always passed by value.
- The method parameter gets a copy of the value of the caller's argument
- The method cannot change the caller's own argument.

```
public void increment( int a ) { // add 1 to a
     a = a + 1;
}
public static void main(String[] args) {
     int x = 10;
     increment( x );
     System.out.println( "x = " + x );
```

```
x = 10
```

Pass by Value Example

- What will this code print?

```
public void swap( int a, int b ) { // swap args
    int temp = a;
    a = b;
    b = temp;
}
public static void main(String[] args) {
    int a = 10;  int b = 20;
    swap( a, b );
    System.out.println( "a = " + a );  //what?
    System.out.println( "b = " + b );  //what?
```

Passing objects as arguments

- A method can not change the *value* of the caller's arguments. Setting "date = new value" doesn't work.

```
public void changeDate( LocalDate date ) {
    date = LocalDate.of(2018,1,1); //new year
}

public static void main(String [] args) {
    // Christmas is 25 December.
    LocalDate xmas = LocalDate.of(2017,12,25);
    changeDate( xmas );
    System.out.printf( "Date is %tF", xmas );
}
```

```
Date is 2017-12-25
```

Passing objects as arguments (2)

- A method can change the object that the parameter *refers to*. (Use Date, because LocalDate is *immutable*.)

```
public void changeDate( Date date ) {
    date.setMonth( 1 );  // set to New Year
    date.setDate( 1 );
}
public static void main(String[] args) {
    Date xmas =
        new Date(100, Calendar.DECEMBER, 25);
    changeDate( xmas );
    System.out.printf( "Date is %tF", xmas );
}
```

```
Date is 2000-01-01
```

Passing array as argument

- The same rule applies to arrays:

```
public void swap( int[] a ) {// swap a[0],a[1]
     int tmp = a[0];
     a[0] = a[1];
     a[1] = tmp;
}
public static void main(String[] args) {
     int[] a = new int[] { 10, 20 };
     swap( a );
     System.out.println("a[0] = " + a[0] );
     System.out.println("a[1] = " + a[1] );
```

a is a reference to an *array object*.

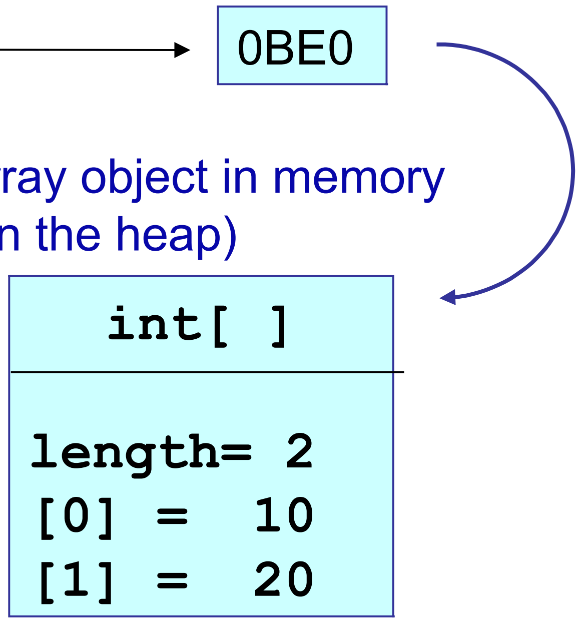
```
a[0] = 20

a[1] = 10
```

Array parameters explained

Value of "p" is memory is the address of an array

```
int [] p
        = new int[]{ 10, 20};
```



Array object in memory (on the heap)

An array is an object.

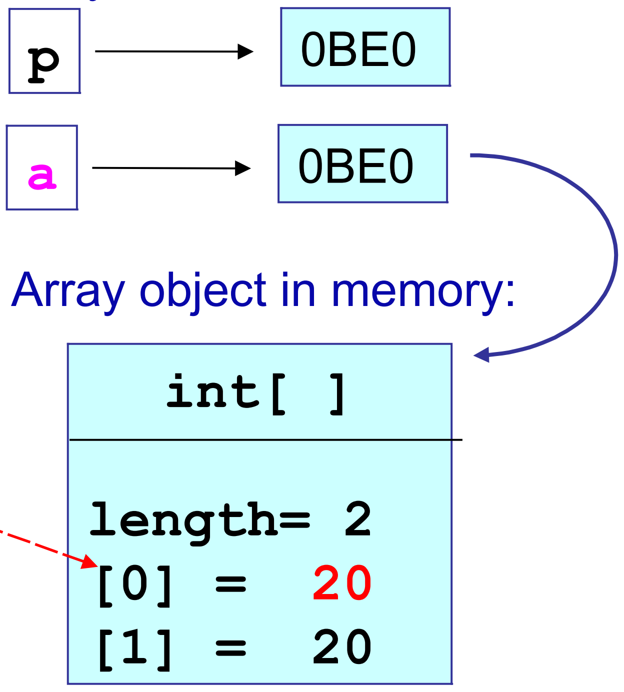
The array variable p refers to the memory area where the array object is stored.

Array parameters, continued

```
int[] p = new int[]{10, 20};
swap( p );
```

```
void swap(int[] a) {
   int tmp = a[0];
   a[0] = a[1];
```

a is a *copy* of p, but both refer to the same array address



The swap parameter a gets a copy of p's value... the address of the array.

swap( ) can use the address (a) to change elements of the array.

Quiz: What does this do?

```
public void swap2( int[] a ) {
     int[] b = new int[2]; // copy and swap
     b[0] = a[1];
     b[1] = a[0];
     a = b;
}
public static void main(String[] args) {
     int[] a = new int[] { 10, 20 };
     swap2( a );
     System.out.println("a[0] = " + a[0] );
     System.out.println("a[1] = " + a[1] );
```

```
a[0] = ___

a[1] = ___
```

Quiz: What does this do?

```java
public void makeNewYear( Date date ) {
    int year = date.getYear();
    // change to January 1
    date = new Date(year, Calendar.JANUARY, 1);
}
public static void main(String[] args) {
    Date xmas =
          new Date(100,Calendar.DECEMBER,25);
    makeNewYear( xmas );
    System.out.printf( "Xmas is %tF", xmas );
```

```
Xmas is 2000-___-___
```

Quiz: Does this work?

```
/** Copy array a into array b.  Really? */
public void arraycopy( int[] a, int[] b ) {
    if (b.length < a.length) /*throw exception*/;
    for(int k=0; k<a.length; k++) b[k] = a[k];
}
public static void main(String[] args) {
    int[] x = new int[] { 10, 20 };
    int[] y = new int[] {  0,  0 };
    arraycopy( x, y);
    System.out.printf("y = [%d, %d]", y[0], y[1]);
```

```
y = [ ___, ___ ]
```

Summary

- In Java, arguments are always passed by value.
- The method parameter gets a copy of the value of the caller's argument
- The method cannot change the value of the caller's argument.
- A method can change the object that a parameter *refers to*.

Variable Length Parameters

- A method can have a variable number of parameters.
- We can write **one** `sum` method to do this:

```
sum = MyMath.sum( x1 ); // = x1
sum = MyMath.sum( x1, x2 ); // = x1+x2
sum = MyMath.sum( x1, x2, x3 ); // = x1+x2+x3
sum = MyMath.sum( x1, x2, x3, x4 );
sum = MyMath.sum( x1, x2, x3, x4, x5 );
```

Variable Length Parameter Syntax

- Use "`. . .  name`" to declare a variable length param.
- Use `name[k]` as array <u>inside</u> the method.

```
public static double sum( double ... x )
{
    double sum = x[0];
    for (int k=1; k<x.length; k++)
        sum = sum + x[k];
    return sum;
}
```

Be Careful!

- The *actual* number of parameters may be zero.
- Be careful of zero-length array.

```
double sum = MyMath.sum( ); // stupid but legal
```

```
public static double sum( double ... x )
{
    double sum = x[0];
    ... ArrayIndexOutOfBoundsException
```

Required Parameter

- If your method requires at least one parameter, add a fixed parameter:

```
public static double sum( double first,
                          double ... x )
{
     double sum = first;
     for (int k=0; k<x.length; k++) sum += x[k];
```

For sum( ), we could just return 0.0 if no parameters.
But what about `max(double ... x)`?
`max( )` makes no sense.

Rules for Variable Length Parameter

- May have only 1 variable length parameter per method.
- Must be the *last parameter* of the method.

```
double power( double ... x, int ... y ) // ERROR

void addMany( List list, String ... item) // OK

void addMany(String ... item, List list) // ERROR
```

How printf( ) works

How can `printf(    )` print any number of items?

```
System.out.printf("hello\n");
System.out.printf("%s", s1);
System.out.printf("(x,y)=(%f,%f)", x, y);
System.out.printf("%s %f %s", s1, x, s3);
```

Format string

variable number of Objects

Parameter passing in other Languages

Some programming literacy.

These slides are optional.

Parameter Passing

Pass by Value ("call by value") means a function gets a copy of the caller's arguments. Changes to the copy to not effect the caller.

Pass by Reference ("call by reference") means that the function parameters refer to the same storage as the caller's arguments. Any changes *will* affect the caller.

Java *always* uses "**pass by value**".

- a method cannot change values of the caller's arguments
- a method <u>can</u> change the object that a parameter refers to (this will change the caller's data)

Parameter Passing in C#

- C# has both call by value and call by reference.
- Use "**`ref`**" to indicate call-by-reference parameters

```
/* this is call by value (can't change args) */
static void swap(int a, int b) {
  int temp = a;  a = b;   b = temp;
}
```

call by value

```
/* this is call by reference (can change args)*/
static void swap(ref int a, ref int b) {
  int temp = a;  a = b;  b = temp;
}
```

call by reference

How does C Pass Parameters?

- C always passes parameters by value (same as Java).
- To enable a function to change values of caller's arguments, you must use a pointer ("**`int *a`**" in C).

```
/* this doesn't work (pass by value) */
void swap(int a, int b) {
  int temp = a;   a = b;     b = temp;
}
```

```
/* this works: use pointers */
void swap(int *a, int *b) {
  int temp = *a;   *a = *b;   *b = temp;
}
```

Parameter Passing in C

- An array name is a pointer (reference) to an array. So, even using "call by value" a function can change the caller's array elements!

```
/* double the first element of the array */
void double(int a[ ]) {
  a[0] = 2*a[0];  // change the storage a points to
}
```

```
int main( ) {
  int p[1];
  p[0] = 100;
  double( p );
  printf("%d\n", p[0]);  // prints "200"
}
```

Parameter Passing in C++

- C++ has both "call by value" and "call by reference"
- Use "&" to indicate reference parameters

```
/* this does not change the caller's a or b */
void swap(int a, int b) {
  int temp = a;   a = b;    b = temp;
}
```

pass by value

```
/* this does change the caller's a and b values */
void swap(int &a, int &b) {
  int temp = a;   a = b;   b = temp;
}
```

pass by reference

You can write "int& a" or "int &a" or "int & a".