- Chapter 4

- Methods, arrays and references

Method Definitions

- Methods belong to a class
  - Defined inside the class
- *Heading*
  - Return type (e.g. `int, float, void`)
  - Name (e.g. `nextInt, println`)
  - Parameters (e.g. `println(…)`)
  - More…
- *Body*
  - enclosed in braces `{}`.
  - Declarations and/or statements.

The Method `main`

- A program
  - a class that has a method named `main`.
  - Execution begins in the `main` method
- So far
  - no attributes (instance variables)
  - no methods other than method `main`.

```
------------ Person.java ------ defining Person ---------------------
public class Person
{
  private String _name;
  private String _iceCream;

  public void setName(String newName)
  {
    this._name = newName;   // this. is optional
  }

  public void setIceCream(String newIceCream)  { … }

  public void print()
  {
     System.out.println(this._name + " likes " + this._IceCream); // this. optional
  }
}

------------ PersonTest.java ----- using Person ------------------
public class PersonTest
{
  public static void main(String[] args)
  {
    Person joe = new Person();
    joe.setName("Joseph");
    joe.setIceCream("Rocky Road");

    Person mary = new Person();
    mary.setName("Mary");
    mary.setIceCream("Chocolate Fudge");
    mary.print();
  }
}
```

Example

- `class SpeciesFirstTry`

```
import java.util.*;

public class SpeciesFirstTry
{
    public String name;
    public int population;
    public double growthRate;

    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is the species' name?");
        name = keyboard.nextLine();

        System.out.println("What is the population of the species?");
        population = keyboard.nextInt();
        while (population < 0)
        {
            System.out.println("Population cannot be negative.");
            System.out.println("Reenter population:");
            population = keyboard.nextInt();
        }

        System.out.println(
                    "Enter growth rate (percent increase per year):");
        growthRate = keyboard.nextDouble();
    }

    public void writeOutput()
    {
        System.out.println("Name = " + name);
        System.out.println("Population = " + population);
        System.out.println("Growth rate = " + growthRate + "%");
    }

    public int populationIn10()
    {
        double populationAmount = population;
        int count = 10;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = (populationAmount +
                                (growthRate/100) * populationAmount);
            count   ;
        }
        if (populationAmount > 0)
            return (int)populationAmount;
        else
            return 0;
    }
}
```

We will give a better version of this class later in this chapter.

Later in this chapter you will see that the modifier `public` should be replaced with `private`.

`(int)` is a type cast, as discussed in Chapter 2.

Display 4.3
A Class Definition

Example, contd.

- `class SpeciesFirstTryDemo`

```java
public class SpeciesFirstTryDemo
{
    public static void main(String[] args)
    {
        SpeciesFirstTry speciesOfTheMonth = new SpeciesFirstTry();
        int futurePopulation;

        System.out.println("Enter data on the Species of the Month:");
        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();

        futurePopulation = speciesOfTheMonth.populationIn10();
        System.out.println("In ten years the population will be "
                                        + futurePopulation);

        speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In ten years the population will be "
                        + speciesOfTheMonth.populationIn10());
    }
}
```

Sample Screen Dialog

```
Enter data on the Species of the Month:
What is the species' name?
Ferengie fur ball
What is the population of the species?
1000
Enter growth rate (percent increase per year):
-20.5
Name = Ferengie fur ball
Population = 1000
Growth rate = -20.5%
In ten years the population will be 100
The new Species of the Month:
Name = Klingon ox
Population = 10
Growth rate = 15.0%
In ten years the population will be 40
```

Display 4.4
Using Classes and Methods

  - Each object of type `SpeciesFirstTry` has its own values for the three attributes

Two Types of Methods

1. Return a value
   - `next = keyboard.nextInt();`
   - `keyboard` is the *calling object.*
2. Don't return a value, called `void` *method*
   - `System.out.println("Enter data:");`
   - `System.out` is the calling object

void Method Definitions

- example

```
public void writeOuput()  //heading
{  //body
   System.out.println("Name: " + name);
   System.out.println("Age: " + age);
}
```

Using `return` in a `void` Method

- form

```
return;
```

- usage
  - end the invocation of the method prematurely for dealing with some problem
- caution
  - better ways to deal with a potential problem (“exceptions”) [later...]

Defining Methods That Return a Value

- example

```
public float density(float area) // heading
{   // body
    return population / area;
}
```

Defining Methods That Return a Value, cont.

- must contain *return statement*

  `return` *`Expression`*`;`

  - *`Expression`* must have a type that matches the return type
- multiple `return` statements are allowed
  - a single `return` statement is better (one exit point for the method).

Naming Methods

- Use a verb to name methods
  - Actions
  - getBalance, deposit, changeAddress
- Start a method name with a lowercase letter.

`public` Method Definitions

- syntax for a `void` method

```
public void methodName(parameters)
{
    <statement(s)>
}
```

`public` Method Definitions

- syntax for methods that return a value

```
public returnType methodName(parameters)
{
    <statement(s), including a
     return statement>
}
```

Local Variables

- Declared within a method
  - "local to" (confined to) the method definition
  - can't be accessed outside the method
- *Not attributes (instance variables)*

Local Variables, cont.

- `class BankAccount`
- `class LocalVariablesDemoProgram`

```java
/**
 This class is used in the program LocalVariablesDemoProgram.
*/
public class BankAccount
{
    public double amount;
    public double rate;

    public void showNewBalance()
    {
        double newAmount = amount + (rate/100.0)*amount;
        System.out.println("With interest added the new amount is $"
                                    + newAmount);
    }
}

/**
 A toy program to illustrate how local variables behave.
*/
public class LocalVariablesDemoProgram
{
    public static void main(String[] args)
    {
        BankAccount myAccount = new BankAccount();
        myAccount.amount = 100.00;
        myAccount.rate = 5;

        double newAmount = 800.00;
        myAccount.showNewBalance();
        System.out.println("I wish my new amount were $" + newAmount);
    }
}
```

This class definition goes in a file named `BankAccount.java`

Two different variables named `newAmount`

This program goes in a file named `LocalVariableDemoProgram.java`

This does not change the value of the variable `newAmount` *in* `main`

Screen Output

```
With interest added the new amount is $105.0
I wish my new amount were $800.0
```

Display 4.5
Local Variables

Passing Values to a Method: Parameters

- Input values for methods (within the program, not from user)
  - *passed* values or *parameters*
- More flexibility for methods
- *formal* parameters
  - Part of method definition
  - After the method name, within parentheses
    - *type*
    - *name*
- *arguments*, or *actual* parameters
  - Calling a method with an object within the parentheses
    - matching data type
    - in the same order

Formal vs Actual Parameters

```
public static void main(String[] args)
{
  print("George Bush");
}

public static void print(String name)
{
  System.out.print("The name is: " + name);
}
```

Scope of Formal Parameters

- Start: begin of the method
- End: end of the method

```
public float square(float num)
{   // begin of num's scope
    …
}   // end of num's scope
```

Parameter Passing Example

```
//Definition of method to double an integer
public int doubleValue(int numberIn)
{
   return 2 * numberIn;
}
```

```
//Invocation of the method... somewhere in main...
...
int next = keyboard.nextInt();
System.out.println(someObj.doubleValue(next));
```

- What is the formal parameter in the method definition?
  - `numberIn`
- What is the argument (actual parameter) in the method invocation?
  - `next`

Pass-By-Value: *Primitive* Data Types as Parameters

- When the method is called
  - *value* of each argument is ***copied*** (assigned) to its corresponding formal parameter
- Formal parameters
  - initialized to the values passed
  - local to their method
- Variables used as arguments cannot be changed by the method
  - the method only gets a copy of the variable's value

Example for Pass-by-Value

```
public static void main(String[] args)
{
  int x = 10, num = 20;
  int sq = MyClass.square(x);
  System.out.println(x);
  System.out.println(num);
}

public static int square(int num)
{
   num = num * num;
   return num;
}
```

Arguments to Methods

- An argument in a method invocation can be
  - a literal such as `2` or `'A'`
  - a variable
  - an expression that yields a value

    [technically a literal or variable is also an “expression”]

Example for Pass-by-Value

```
public static void main(String[] args)
{
  int x = 10, area;
  area = MyClass.square(x);
  area = MyClass.square(5);
  area = MyClass.square(x + 5 % 2);
}

public static int square(int num)
{
   return num * num;
}
```

Multiple Arguments to Methods

```
anObject.doStuff(42, 100, 9.99, 'Z');
public void doStuff(int n1, int n2, double d1, char c1);
```

- arguments and formal parameters are matched by position
- Corresponding types need to match

Method with a Parameter

- `class SpeciesSecondTry`

```
import java.util.*;

public class SpeciesSecondTry
{
    public String name;
    public int population;
    public double growthRate;

    public void readInput()
    {
        <The definition of the method readInput is the same as in Display 4.3.>
    }

    public void writeOutput()
    {
        <The definition of the method writeOutput is the same as in Display 4.3.>
    }

    /**
     Returns the projected population of the calling object
     after the specified number of years.
    */
    public int projectedPopulation(int years)
    {
        double populationAmount = population;
        int count = years;
        while ((count > 0) && (populationAmount > 0))
        {
            populationAmount = (populationAmount +
                                (growthRate/100) * populationAmount);
            count   ;
        }
        if (populationAmount > 0)
            return (int)populationAmount;
        else
            return 0;
    }
}
```

Later in the chapter, you will see that the modifier `public` *should be replaced with* `private`.

We will give an even better version of the class later in the chapter.

Display 4.6
A Method with a Parameter

Using a Method with a Parameter

- `class SpeciesSecondTryDemo`

```
/**
 Demonstrates the use of a parameter
 with the method projectedPopulation.
*/
public class SpeciesSecondTryDemo
{
    public static void main(String[] args)
    {
        SpeciesSecondTry speciesOfTheMonth = new SpeciesSecondTry();
        int futurePopulation;

        System.out.println("Enter data on the Species of the Month:");
        speciesOfTheMonth.readInput();
        speciesOfTheMonth.writeOutput();

        futurePopulation = speciesOfTheMonth.projectedPopulation(10);
        System.out.println("In ten years the population will be " +
                                                futurePopulation);

        speciesOfTheMonth.name = "Klingon ox";
        speciesOfTheMonth.population = 10;
        speciesOfTheMonth.growthRate = 15;
        System.out.println("The new Species of the Month:");
        speciesOfTheMonth.writeOutput();
        System.out.println("In ten years the population will be " +
                        speciesOfTheMonth.projectedPopulation(10));
    }
}
```

Sample Screen Dialog

The dialog is exactly the same as in Display 4.4.

Display 4.7
Using a Method with a Parameter

Access specifiers for Methods

- public
- private
- protected

Method Modifiers

- static
- abstract
- final
- native
- synchronized
- volatile

Class Constructors

- Special method used to initialize member variables of the class
- Same name as the Class name and does not have a return type
- Called when an object is created
- Types:
  - Explicit constructors
  - Implicit constructors

ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Arrays in JAVA

Declaring an Array Variable

- Do not have to create an array while declaring array variable
  - *<type>* [] variable_name;
  - *int* [] prime;
  - *int* prime[];
- Both syntaxes are equivalent
- No memory allocation at this point

Defining an Array

- Define an array as follows:
  - variable_name=new <*type*>[N];
  - primes=new int[10];
- Declaring and defining in the same statement:
  - *int*[] primes=new *int*[10];
- In JAVA, int is of 4 bytes, total space=4*10=40 bytes

- Declaring and defining in the same statement: (Khai báo mảng)

Kiểu dữ liệu tên_mảng = new Kiểu dữ liệu [];

- *Ví dụ: int*[] primes=new *int*[10];

  double [] mangA = new double [100];

Graphical Representation

prime

Index

0	1	2	3	4	5	6	7	8	9
2	1	11	-9	2	1	11	90	101	2

value

What happens if …

- We define
  - int[] prime=new long[20];

```
MorePrimes.java:5: incompatible types
found: long[]
required: int[]
int[] primes = new long[20];
                ^
```

- The right hand side defines an array, and thus the array variable should refer to the same type of array

What happens if …

- We define
  - int prime[100];

```
MorePrimes.java:5: ']' expected
long primes[20];
                ^
```

- The C++ style is not permitted in JAVA syntax

What happens if …

- Valid code:

```
int k=7;
long[] primes = new long[k];
```

- Invalid Code:

```
int k;
long[] primes =new long[k];
```

Compilation Output:

```
MorePrimes.java:6: variable k might not have been initialized
long[] primes = new long[k];
                          ^
```

Array Size through Input

```
....
BufferedReader stdin = new BufferedReader (new InputStreamReader(System.in));
String inData;
int    num;
System.out.println("Enter a Size for Array:");
inData = stdin.readLine();
num    = Integer.parseInt( inData ); // convert inData to int
long[] primes = new long[num];
System.out.println("Array Length="+primes.length);
....
```

SAMPLE RUN:

```
Enter a Size for Array:
4
Array Length=4
```

Default Initialization

- When array is created, array elements are initialized
  - Numeric values (int, double, etc.) to 0
  - Boolean values to false
  - Char values to ‘\u0000’ (unicode for blank character)
  - Class types to null

Accessing Array Elements

- Index of an array is defined as
  - Positive int, byte or short values
  - Expression that results into these types
- Any other types used for index will give error
  - long, double, etc.
  - Incase Expression results in long, then type cast to int
- Indexing starts from 0 and ends at N-1

```
primes[2]=0;
int k = primes[2];
…
```

Validating Indexes

- JAVA checks whether the index values are valid at runtime
  - If index is negative or greater than the size of the array then an IndexOutOfBoundException will be thrown
  - Program will normally be terminated unless handled in the try {} catch {}

What happens if …

```
long[] primes = new long[20];
primes[25]=33;
….
```

Runtime Error:

Exception in thread “main” java.lang.ArrayIndexOutOfBoundsException: 25

at MorePrimes.main(MorePrimes.java:6)

Reusing Array Variables

- Array variable is separate from array itself
  - Like a variable can refer to different values at different points in the program
  - Use array variables to access different arrays

```
int[] primes=new int[10];
……
primes=new int[50];
```

- Previous array will be discarded
- Cannot alter the type of array

Initializing Arrays

- Initialize and specify size of array while declaring an array variable

```
int[] primes={2,3,5,7,11,13,17}; //7 elements
```

- You can initialize array with an existing array

```
int[] even={2,4,6,8,10};
int[] value=even;
```

  - One array but two array variables!
  - Both array variables refer to the same array
  - Array can be accessed through either variable name

Graphical Representation

even

value

0	1	2	3	4
2	4	6	8	10

Demonstration

```
long[] primes = new long[20];
primes[0] = 2;
primes[1] = 3;
long[] primes2=primes;
System.out.println(primes2[0]);
primes2[0]=5;
System.out.println(primes[0]);
```

Output

2

5

Array Length

- Refer to array length using *length*
  - A data member of array object
  - array_variable_name.length
  - for(int k=0; k<primes.length;k++)

    ….
- Sample Code:

```
long[] primes = new long[20];
System.out.println(primes.length);
```

- Output: 20

Change in Array Length

- If number of elements in the array are changed, JAVA will automatically change the length attribute!

Sample Program

```
class MinAlgorithm
{
   public static void main ( String[] args )
   {
         int[] array = { -20, 19, 1, 5, -1, 27, 19, 5 } ;
         int min=array[0]; // initialize the current minimum
         for ( int index=0; index < array.length; index++ )
            if ( array[ index ] < min )
                        min = array[ index ] ;
         System.out.println("The minimum of this array is: " + min );
   }
}
```

*Program taken from: http://chortle.ccsu.edu/CS151/Notes/chap47/ch47_10.html

Arrays of Arrays

- Two-Dimensional arrays
  - float[][] temperature=new float[10][365];
  - 10 arrays each having 365 elements
  - First index: specifies array (row)
  - Second Index: specifies element in that array (column)
  - In JAVA float is 4 bytes, total Size=4*10*365=14,600 bytes

Graphical Representation

Sample[0]

0	1	2	3	4	5	6	7	8	9

Sample[1]

0	1	2	3	4	5	6	7	8	9

Sample[2]

0	1	2	3	4	5	6	7	8	9

Initializing Array of Arrays

```
int[][] array2D = { {99, 42, 74, 83, 100}, {90, 91, 72, 88, 95}, {88,
  61, 74, 89, 96}, {61, 89, 82, 98, 93}, {93, 73, 75, 78, 99}, {50,
  65, 92, 87, 94}, {43, 98, 78, 56, 99} };
//5 arrays with 5 elements each
```

Arrays of Arrays of Varying Length

- All arrays do not have to be of the same length

```
float[][] samples;
samples=new float[6][];//defines # of arrays
samples[2]=new float[6];
samples[5]=new float[101];
```

- Not required to define all arrays

Initializing Varying Size Arrays

```
int[][] uneven = { { 1, 9, 4 }, { 0, 2}, { 0, 1, 2, 3, 4 } };
//Three arrays
//First array has 3 elements
//Second array has 2 elements
//Third array has 5 elements
```

Array of Arrays Length

```
long[][] primes = new long[20][];
primes[2] = new long[30];
System.out.println(primes.length); //Number of arrays
System.out.println(primes[2].length);//Number of elements in the second array
```

OUTPUT:

20

30

Sample Program

```
class unevenExample3
{
    public static void main( String[] arg )
    { // declare and construct a 2D array
            int[][] uneven = { { 1, 9, 4 }, { 0, 2}, { 0, 1, 2, 3, 4 } };
            // print out the array
            for ( int row=0; row < uneven.length; row++ ) //changes row
            {
                    System.out.print("Row " + row + ": ");
                    for ( int col=0; col < uneven[row].length; col++ ) //changes column
                            System.out.print( uneven[row][col] + " ");
            System.out.println();
            }
    }
}
```

Output

Row 0: 1 9 4

Row 1: 0 2

Row 2: 0 1 2 3 4

Triangular Array of Arrays

- Triangular Array

```
for(int k=0; k<samples.length;k++)
    samples[k]=new float[k+1];
```

Multidimensional Arrays

- A farmer has 10 farms of beans each in 5 countries, and each farm has 30 fields!
- Three-dimensional array

```
long[][][] beans=new long[5][10][30];
//beans[country][farm][fields]
```

Varying length in Multidimensional Arrays

- Same features apply to multi-dimensional arrays as those of 2 dimensional arrays

```
long beans=new long[3][][];//3 countries
beans[0]=new long[4][];//First country has 4 farms
beans[0][4]=new long[10];
//Each farm in first country has 10 fields
```

ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Introduction to reference in java

What's a reference

- In Java, reference is a typed named memory space which holds address of an Object of that type.
  - Reference in Java is similar with pointer in C/C++.
- Java is safer, C/C++ is more powerful.

Reference vs. Pointer

- 1) You cannot put your hands on an Object without through its reference in Java. In C/C++, you can.
- 2) In C/C++, you can do arithmetic on pointers, but you cannot do arithmetic on references in Java.
- 3) In C/C++, you can **dereference** a pointer, you cannot dereference a reference in Java.
- 4)In Java, all object are put on the heap only. In C/C++, you can put an Object/struct onto the stack.
- 5) In C/C++, you can cast pointer to an totally different type without compiler errors. In Java, you cannot, Java is more strong typed language.
- 6) In C/C++, pointer can point to primitive types too. In Java, reference cannot reference a primitive type, unless the wrapper class is used.
- 7) In C/C++, pointer can point to another pointer, in Java, reference cannot reference another reference.

example

- Java:

```
Class A{…}
A a;
a=new A();
```

reference

- C/C++:

```
struct a{…}
struct a*  p = malloc(sizeof(a));
p=&a;
```

pointer

Addition

A swap method?

- Does the following swap method work? Why or why not?

```
public static void main(String[] args) {
    int a = 7;
    int b = 35;

    // swap a with b?
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Value semantics

- **value semantics**: Behavior where values are copied when assigned, passed as parameters, or returned.
  - All primitive types in Java use value semantics.
  - When one variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;
int y = x;        // x = 5, y = 5
y = 17;           // x = 5, y = 17
x = 8;            // x = 8, y = 17
```

Reference semantics (objects)

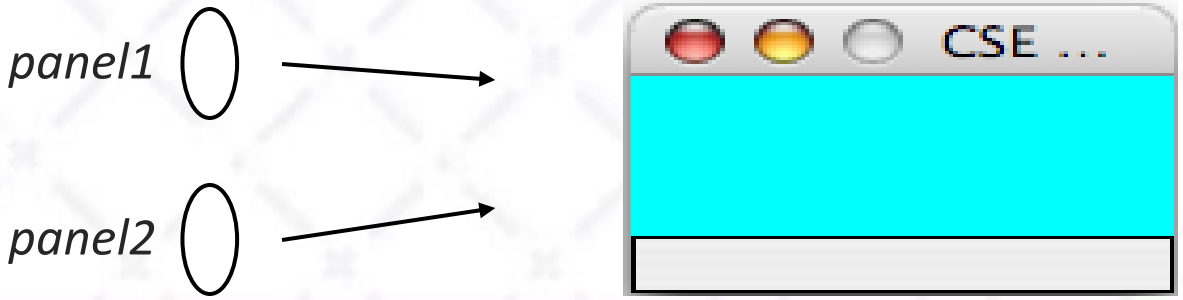
- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable *will* affect others.

```
int[] a1 = {4, 15, 8};
int[] a2 = a1;          // refer to same array as a1
a2[0] = 7;
System.out.println(Arrays.toString(a1)); // [7, 15, 8]
```

References and objects

- Arrays and objects use reference semantics. Why?
  - *efficiency.* Copying large objects slows down a program.
  - *sharing.* It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;    // same window
panel2.setBackground(Color.CYAN);
```

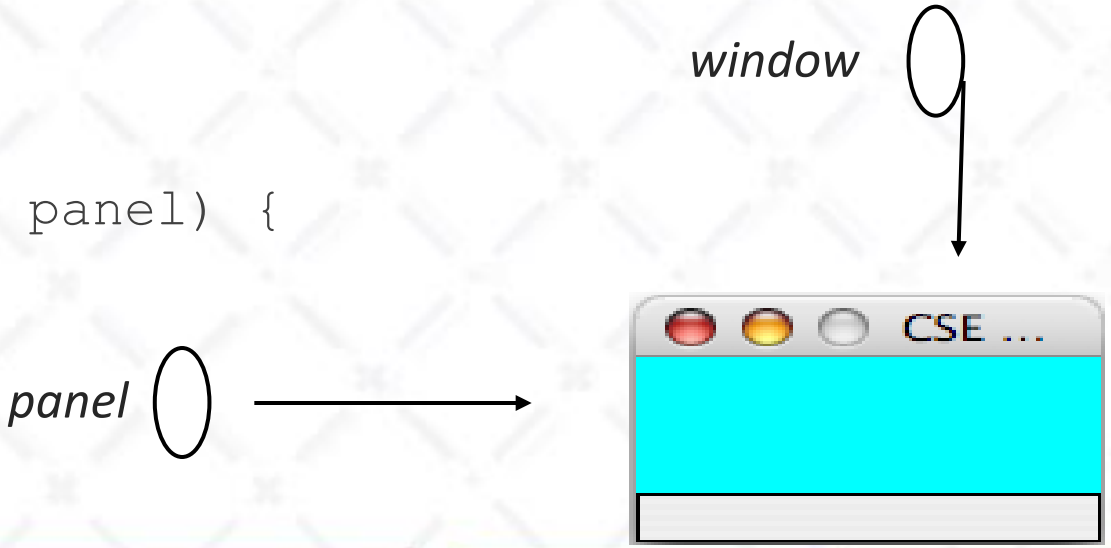


Objects as parameters

- When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.
  - If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {
    DrawingPanel window = new DrawingPanel(80, 50);
    window.setBackground(Color.YELLOW);
    example(window);
}

public static void example(DrawingPanel panel) {
    panel.setBackground(Color.CYAN);
    ...
}
```



Arrays as parameters

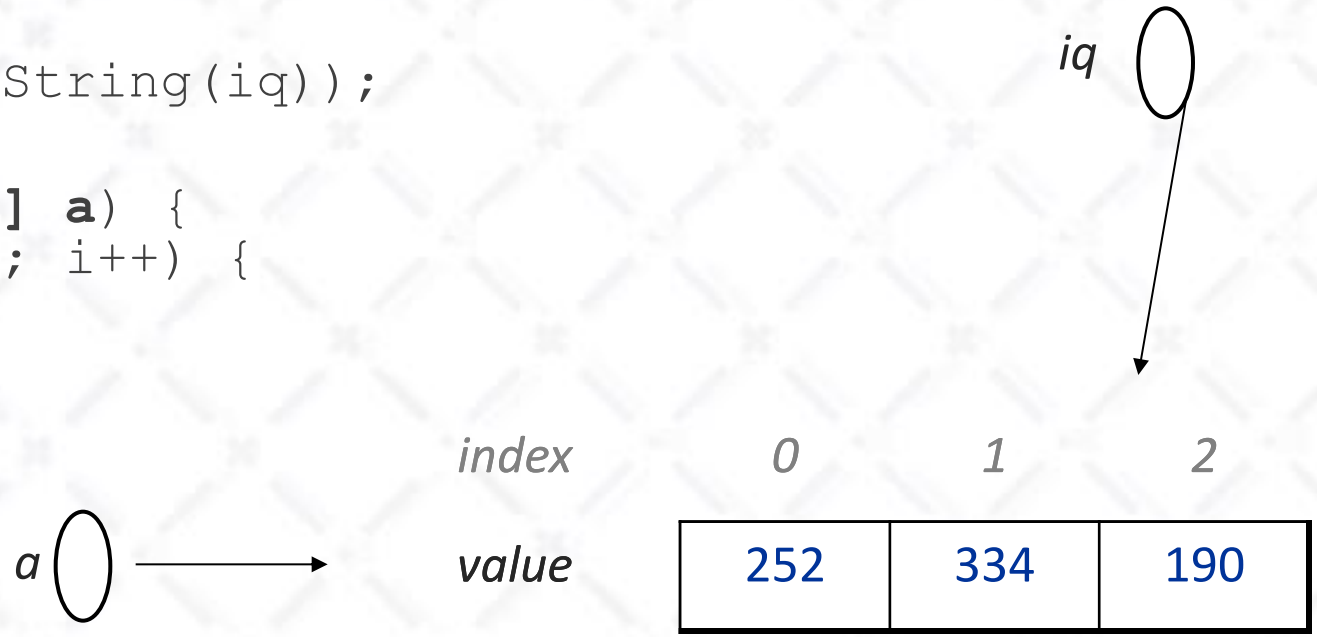
- Arrays are also passed as parameters by reference.
  - Changes made in the method are also seen by the caller.

```
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}

public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```

- Output:

```
[252, 334, 190]
```

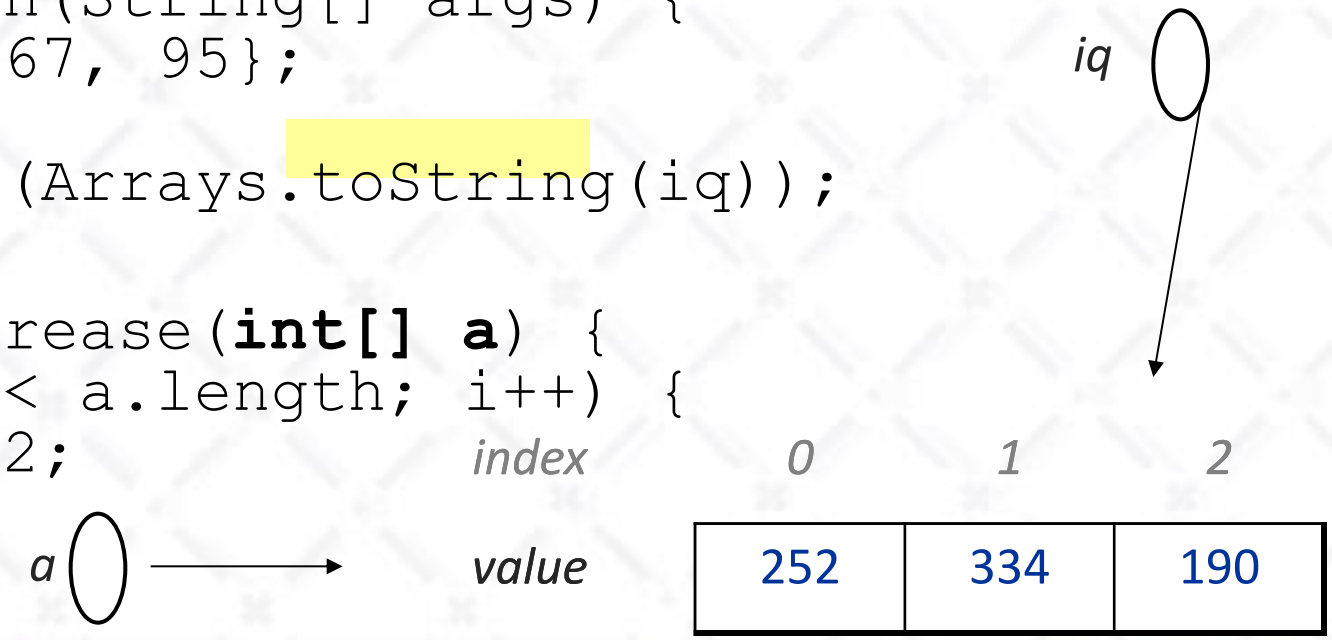


Arrays pass by reference

- Arrays are also passed as parameters by reference.
  - Changes made in the method are also seen by the caller.

```
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}

public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```



- Output:

```
[252, 334, 190]
```