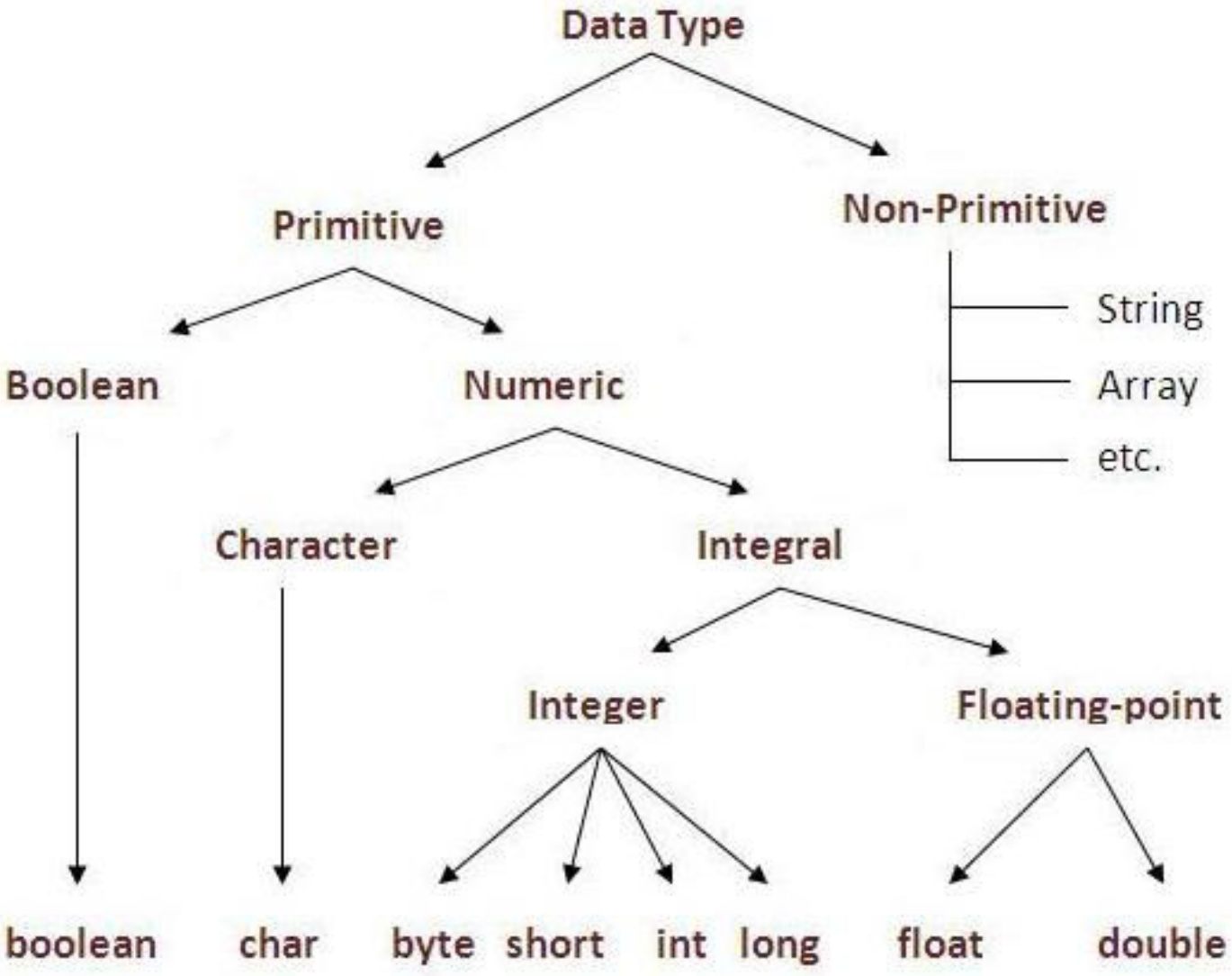
# Programming Language II
# CSE-215

Prof. Dr. Mohammad Abu Yousuf

yousuf@juniv.edu

# Data Types, Variables, and Arrays

- In java, there are two types of data types
  - primitive data types
  - non-primitive data types



- Java defines eight *primitive types of data: byte, short, int, long, char, float, double,* and boolean. The primitive types are also commonly referred to as *simple types*. These can be put in four groups:
  - Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers.
  - Floating-point numbers This group includes float and double, which represent numbers with fractional precision.
  - Characters This group includes char, which represents symbols in a character set, like letters and numbers.
  - Boolean This group includes boolean, which is a special type for representing true/false values.

# Integers

- Java defines four integer types: **byte, short, int, and long. All of these are signed,** positive and negative values.

| Name | Width | Range |
|---|---|---|
| long | 64 | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| int | 32 | –2,147,483,648 to 2,147,483,647 |
| short | 16 | –32,768 to 32,767 |
| byte | 8 | –128 to 127 |

# Integers

**byte:**

- The smallest integer type is **byte. This is a signed 8-bit type that has a range from –128** to 127.

- Byte variables are declared by use of the **byte keyword. For example, the following** declares two **byte variables called b and c:**

```
byte  b, c;
```

# Integers

**short:**

- **short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the** least-used Java type. Here are some examples of **short variable declarations:**

```
short  s;
short  t;
```

# Integers

**long:**

- **long is a signed 64-bit type and is useful for those occasions where an int type is not** large enough to hold the desired value.

- For example, the following is a program that computes the number of miles that light will travel in a specified number of days:

```
// Compute distance light travels using long variables.
class Light {
  public static void main(String args[]) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // approximate speed of light in miles per second
    lightspeed = 186000;

    days = 1000; // specify number of days here

    seconds = days * 24 * 60 * 60; // convert to seconds

    distance = lightspeed * seconds; // compute distance

    System.out.print("In " + days);
    System.out.print(" days light will travel about ");
    System.out.println(distance + " miles.");
  }
}
```

In 1000 days light will travel about 16070400000000 miles.

# Characters

- **char in Java is not the same as char in C or C++. In C/C++, char is 8 bits** wide. This is *not the case in Java. Instead, Java uses Unicode to represent characters.*

- Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.
- For this purpose, it requires 16 bits. Thus, in Java **char is a 16-bit type. The range of a char** is 0 to 65,536. There are no negative **chars.**

# Characters

- Here is a program that demonstrates **char variables:**

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88; // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

```
ch1 and ch2: X Y
```

# Characters

- Notice that **ch1 is assigned the value 88, which is the ASCII (and Unicode) value that** corresponds to the letter *X. As mentioned, the ASCII character set occupies the first 127* values in the Unicode character set.

```
// char variables behave like integers.
class CharDemo2 {

  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " + ch1);
  }
}
```

ch1 contains X
ch1 is now Y

# Booleans

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

# Booleans

- Output of previous program:

```
b is false
b is true
This is executed.
10 > 9 is true
```

# Arrays

- **Arrays:**
- The general form of a one dimensional array declaration is:

  dataType[] arr; (or)

  dataType []arr; (or)

  dataType arr[];

  Example: int month_days[];
- **new is a special** operator that allocates memory.
- The general form of **new** as it applies to one**-dimensional** arrays appears as follows:

  *array-var = new type [size];*

  Example: month_days = new int[12];

# Arrays

- It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```
int  month_days[] = new int[12];
```

- This is the way that you will normally see it done in professionally written Java programs.
- Let's review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new, and assign it to the array variable. Thus, in Java all arrays are** dynamically allocated.

# Arrays

```
class Testarray{
public static void main(String args[]){

int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;

//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}
}
```

# Passing Array to method in java

```
class Testarray2{
static void min(int arr[]){
    int min=arr[0];
    for(int i=1;i<arr.length;i++)
    if(min>arr[i])
    min=arr[i];
    System.out.println(min);
}

public static void main(String args[]){
     int a[]={33,3,4,5};
     min(a);//passing array to method
}
}
```

Output: 3

# Multidimensional Arrays

- **Multidimensional Arrays:**
- In Java, *multidimensional arrays are actually arrays of arrays.*
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, the following declares a two-dimensional array variable called **twoD:**

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

  - **Example:**

```
int  twoD[][] = new  int[4][5];
```

# Multidimensional Arrays

```
class Testarray3{
public static void main(String args[]){

//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}

}}
```

Output:1 2 3
2 4 5
4 4 5

# Multidimensional Arrays

- Example:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
      }
```

```
    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

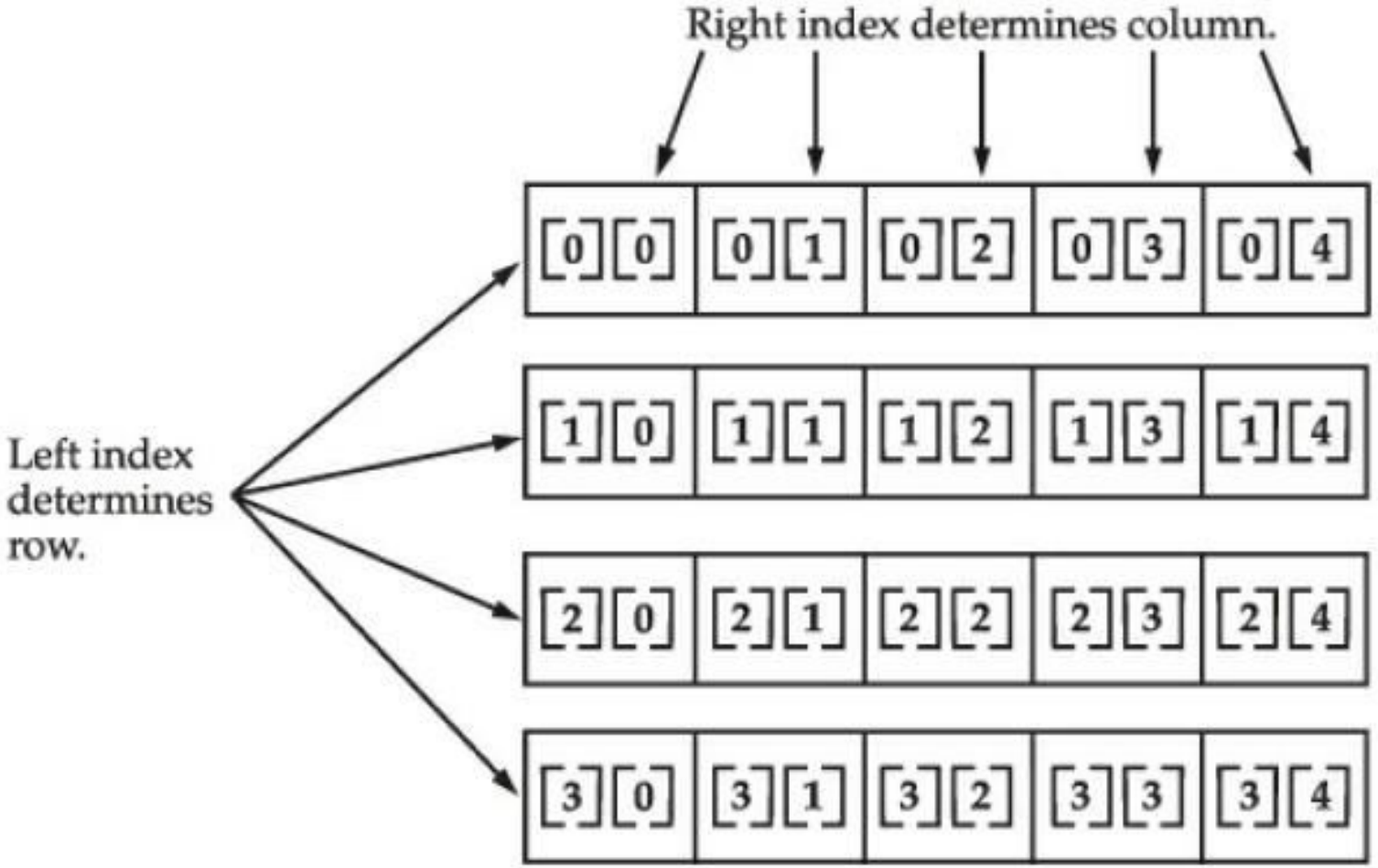
```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

# Multidimensional Arrays

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension.
- You can allocate the remaining dimensions separately.
- For example, this following code allocates memory for the first dimension of **twoD when it is declared. It allocates the second dimension manually.**

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

# Multidimensional Arrays




Given: int twoD [ ] [ ] = new int [4] [5] ;

# Multidimensional Arrays

- when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension.

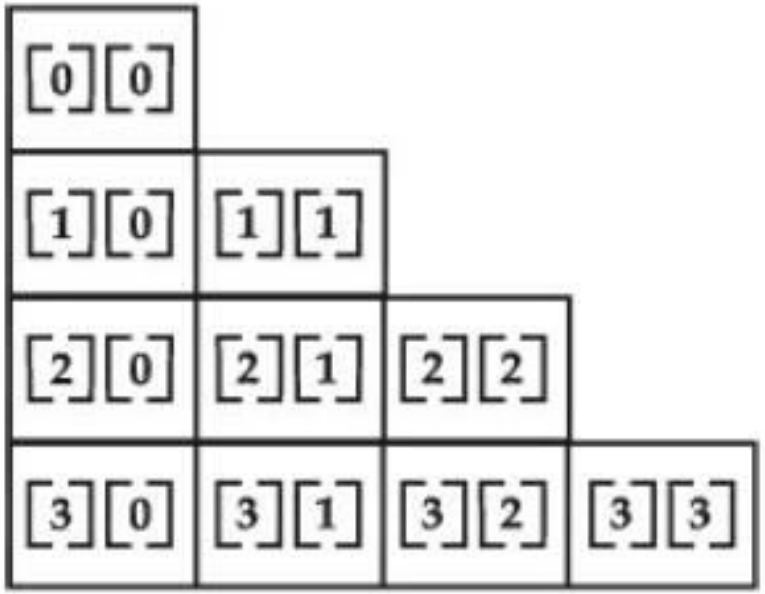
- Since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two dimensional array in which the sizes of the second dimension are unequal:

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
      twoD[i][j] = k;
      k++;
    }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
0
1 2
3 4 5
6 7 8 9
```

# Multidimensional Arrays

- The array created by this program looks like this:




- The use of uneven (or irregular) multidimensional arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, irregular arrays can be used effectively in some situations.

# Multidimensional Arrays

- It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix where each element contains the product of the row and column indexes.

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

# Alternative Array Declaration Syntax

- There is a second form that may be used to declare an array:

  *type*[ ] *var-name*;

- Here, the square brackets follow the type speci□er, and not the name of the array variable.

- For example, the following two declarations are equivalent:

```
int al[ ] = new   int[3];
int[ ] a2 = new  int[3];
```

- The following declarations are also equivalent:

```
char twod1[ ][ ] = new char[3][4];
char[ ][ ] twod2 = new char[3][4];
```

# Alternative Array Declaration Syntax

- This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3; // create three arrays
```

- creates three array variables of type **int**. It is the same as writing

```
int nums[], nums2[], nums3[]; // create three arrays
```

# What is the class name of java array?

- In java, array is an object. For array object, an proxy class is created whose name can be obtained by **getClass().getName()** method on the object.

```
class Testarray4{
public static void main(String args[]){

int arr[]={4,4,5};

Class c=arr.getClass();
String name=c.getName();

System.out.println(name);

}}
```

Output: I

# Call by value in java

- **There is only call by value in java, not call by reference.**
- If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

```
class Operation{
 int data=50;
 void change(int data){
 data=data+100;//changes will be in the local var
iable only
 }
 public static void main(String args[]){
   Operation op=new Operation();
   System.out.println("before change "+op.data);
   op.change(500);
   System.out.println("after change "+op.data);
 }
}
```

Output:
before change 50
after change 50

In case of call by value original value is not changed.

# Call by reference in java

```
class Operation2{
 int data=50;

 void change(Operation2 op){
 op.data=op.data+100;//changes will be in the i
nstance variable
 }
 public static void main(String args[]){
   Operation2 op=new Operation2();

   System.out.println("before change "+op.data);

   op.change(op);//passing object
   System.out.println("after change "+op.data);

 }
}
```

In case of call by reference original value is changed if we made changes in the called method.

If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value.

# A Few Words About Strings

- The **String** type is used to declare string variables. You can also declare arrays of strings. A quoted string constant can be assigned to a **String** variable.

```
String str = "this is a test";
System.out.println(str);
```

- Here, **str** is an object of type **String**. It is assigned the string “this is a test”.

# Thank You