Faculty of Computer Science

Programming Language 2
Object oriented design using JAVA

Dr. Ayman Ezzat
Email: ayman@fcih.net
Web: www.fcih.net/ayman

Introduction

Unit 2: Java Fundamentals

1. Data types

- Java is a **strongly typed** language
  - This means that every **variable** and **expression has a type** when the program is compiled and that you cannot arbitrarily assign values of one type to another.

- There are two categories of type in Java:
  1. **Primitive variables** store data values.
  2. **Reference variables** do not themselves store data values, but are used to refer to objects. (I.e Pointers in C++)

- Data type sizes in Java are fixed
  - to ensure portability across implementations of the language.

1. Data types

Primitive data types

- They include:
  - numeric integers, numeric non-integers, characters, and logical types.

Java Primitive Data Types				
Type	Values	Default	Size	Range
byte	signed integers	0	8 bits	-128 to 127
short	signed integers	0	16 bits	-32768 to 32767
int	signed integers	0	32 bits	-2147483648 to 2147483647
long	signed integers	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	+/-1.4E-45 to +/-3.4028235E+38, +/-infinity, +/-0, NAN
double	IEEE 754 floating point	0.0	64 bits	+/-4.9E-324 to +/-1.79769313486231570E+308, +/-infinity, +/-0, NaN
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
boolean	`true, false`	`false`	1 bit used in 32 bit integer	NA

1. Data types

Primitive data types (Cont'd)

- **Declaration and initialization:**

  we can reserve memory locations for storing values of any primitive type. Those memory locations can then be made to contain values that are legal for the particular type. This involves two steps.

  1) **Declaration**: Create a variable of the type you want to use.

  2) I**nitialization:** This refers to when you first store something useful in a variable's memory location.

Example: Declare two variables of type **int** and initialize them

```
int myInt;
int myOtherInt;
myInt = 888;
myOtherInt = -2;
```

1. Data types

Primitive data types (Cont'd)

- **Escape sequence**:
  - It is a **sequence of characters** standing in place of a single value of the **character type**.
    - Example of a printable character: \u00A9 = @
    - Example of non-printable character: \n = NewLine

Character Escape Sequences	
Escape Sequence	**Description**
\uxxxx	Hexadecimal UNICODE character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line
\f	Form feed
\t	Tab
\b	Backspace

1. Data types

Casting

- There are occasions where we want to be able to convert from one type to another. This can be done either:
  - **Automatically:** Java does it for us.
  - **By casting**: we have to specify a type conversion.
    - By writing the desired type of an expression in parentheses in front of the expression; for example, write (int) in front of an expression if we wanted it to be converted to an int.
  - Note that the fractional parts of a float data type are lost if casting to an integral type.

Ex. Casting a double to an int.

```
double doubVal;
int intVal;
doubVal = 2.8;
intVal = (int) doubVal; // casting to an int
```

intVal will have the value of 2

Example: Casting an int to a char

```
int a = 35, b = 12, d = 13;
char c;
c = (char) (a + b + d);
```

c will have the value 60, which is the 61st character in the Unicode set. This turns out to be the character '<'.

1. Data types

Casting (Cont'd)

- When is a cast required?
  - A cast is required if the type you are assigning (**on the right-hand side of an equals sign) occupies a larger space** in memory than the type you are assigning to (on the left-hand side of an equals sign).
  - If you have not specified a cast in a case where a cast is required, the compiler will produce a compilation error with the message that there is a 'possible loss of precision'.

```
double doubVal;
int intVal;
intVal = 2;
doubVal = (double) intVal;    // this is allowed
doubVal = intVal;             // this is also allowed!
```

1. Data types

Statements and Scope

- **A statement** is a unit of executable code, usually terminated by a semicolon.
- **Code block** includes a group of statements in enclosed in curly brackets.

```
{
    int i;
    int j = 24;
    i = j;
}
```

- **Local variables**: They are variables that are declared inside a code block, and they are valid only inside the brackets. We say that the **variables' scope** is the region enclosed by the brackets

```
{
    int k; // k valid from here
    {
        // k is valid, so this is okay
        int j = k;
        // j valid until next closing curly bracket
    }
    // j no longer valid, k still valid
}
```

Outline

1. Data types
2. Operators
3. Strings
4. Arrays
5. Conditional processing
6. Repetitive processing
7. Developing some methods

2. Operators

1. Arithmetic operators

Operator	Name	Example expression	Meaning
*	Multiplication	a * b	a times b
/	Division	a / b	a divided by b
%	Remainder (modulus)	a % b	the remainder after dividing a by b
+	Addition	a + b	a plus b
–	Subtraction	a – b	a minus b

2. The negation operator → unary operator

```
int a = 2;
int b = -a;
```

3. Increment (++) and decrement (--) operators

```
int myInt = 0;
myInt++;      // postfix increment
++myInt;      // prefix increment
```

myInt would have the value 1 then 2

```
int myInt = 10;
int x = myInt++; // postfix form
```

myInt would have the value 11
and **x** would have the value 10.

2. Operators

SAQ 3

Give the values of the following expressions or indicate if there is an error:

(a) `1 + (2 * 3)`

(b) `(1 + 2) * 3`

(c) `6 % 2`

(d) `17 % 12`

(e) `5 % 0`

(f) `2 % 3`

(g) `5 / 2`

(h) `5 / 0`

(i) `0 / 5`

(j) `5 / 2.0`

ANSWERS .....

(a) 7

(b) 9

(c) 0 (2 goes into 6 three times, with no remainder).

(d) 5

(e) This would cause a run-time error, due to division by zero. (Our compiler does not catch this.)

(f) 2 (3 goes into 2 zero times and leaves a remainder of 2).

(g) 2 (the answer is truncated to an integer).

(h) Run-time error, due to division by zero. (Our compiler does not catch this.)

(i) 0

(j) 2.5 (because at least one argument is floating-point, the result is floating-point).

2. Operators

4. Relational operators

Operator	Name	Example expression	Meaning
==	Equal to	x == y	true if x equals y, otherwise false
!=	Not equal to	x != y	true if x is not equal to y, otherwise false
>	Greater than	x > y	true if x is greater than y, otherwise false
<	Less than	x < y	true if x is less than y, otherwise false
>=	Greater than or equal to	x >= y	true if x is greater than or equal to y, otherwise false
<=	Less than or equal to	x <= y	true if x is less than or equal to y, otherwise false

- An expression involving a relational operator is a logical expression, so has a boolean value. The following expressions would all evaluate to true:

1 < 2	0.5 > 0.0	2 == 2
true != false	false == false	'a' < 'c'

2. Operators

5. Logical operators

Operator	Name	Example expression	Meaning
`&&`	Logical AND	`a && b`	returns `true` if both `a` and `b` are `true`, otherwise `false`
`||`	Logical OR	`a || b`	returns `false` if both `a` and `b` are `false`, otherwise `true`
`!`	Logical NOT	`!a`	returns `false` if `a` is `true`; returns `true` if `a` is `false`

Example:

```
int a, b;
boolean eitherPositive, bothNegative;
a = 22;
b = -33;
eitherPositive = (a > 0) || (b > 0);
bothNegative = (a < 0) && (b < 0);
```

- The variable eitherPositive is assigned the value true, because a > 0 is true and therefore the whole expression is true (even though b > 0 is false).
- The variable bothNegative is assigned the value false because a < 0 is false, and
- therefore the whole expression is false. Even though b < 0 is true.

Outline

1. Data types
2. Operators
3. **Strings**
4. **Arrays**
5. **Conditional processing**
6. **Repetitive processing**
7. **Developing some methods**

3. Strings

- A **string** is a sequence of characters.
  - For example, a string could be used to store a person's name
- Strings are represented using the **reference type** called **`String`**

```
String name, name2;    // Figure 1
name = "David Jones";  // Figure 2
name2 = name;          // Figure 3
name = "Roderick";     // Figure 4
```

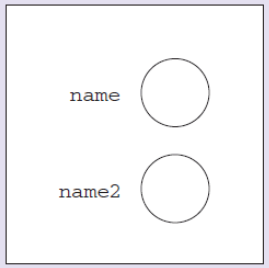


Figure 1 Two uninitialized reference variables

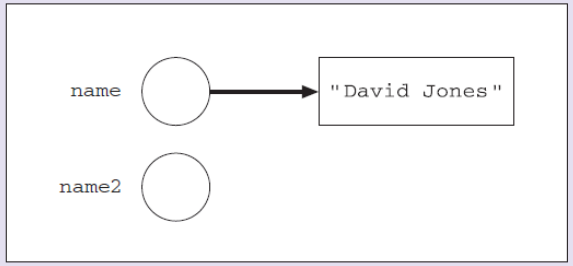


Figure 2 The **`name`** variable referencing a **`String`** object

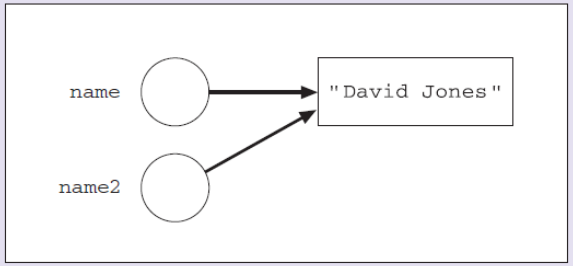


Figure 3 Two variables referencing the same object

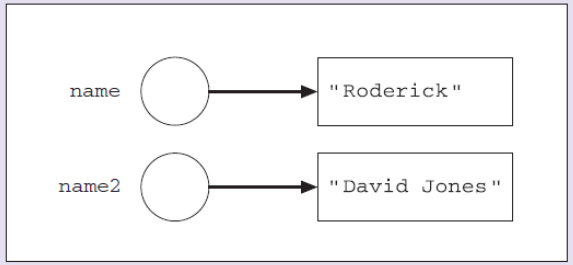


Figure 4 The **`name`** variable referencing a new object

3. Strings

The `length` method

- For example: `name.length()`
  - This evaluates to the number of characters in the string referenced by name.
  - If this string were "Roderick", the length method would return the value 8.
  - The empty string has length zero.

String concatenation (+)

- The concatenation operator, +, will automatically turn any arguments into strings before carrying out a concatenation.
- Example:

```
int occupants = 2;
String address = "Filby Lane";
String printout;
printout = address + " has " + occupants + " occupants";
```

The output is "Filby Lane has 2 occupants".

3. Strings

More string methods

Method	Meaning	Returns
`equals(str)`	returns `true` if the passed `String str` has the same characters as the receiver string object	a `boolean`
`length()`	returns the number of characters in a `String`	an `int`
`charAt(i)`	returns the character at the `int` index `i` in the `String`; the index begins at zero	a `char` (It is an error if `i` is out of range.)
`indexOf(str)`	returns the starting index of the string `str` within the receiver object, or –1 if not found	an `int`
`substring(int1, int2)`	returns the substring of the receiver string, starting at `int1` and finishing at `int2 – 1`	a `String` (It is an error if the indexes are out of range.)

3. Strings

The `StringBuffer` type

- A more typical class type in Java, in terms of object creation, is **`StringBuffer`**, which can be used to handle sequences of characters.
- A **`StringBuffer`** object is **mutable**: it can be changed once it has been created.

```
StringBuffer userName;
userName = new StringBuffer("Jones21");
```

```
StringBuffer userName;
userName = "Jones21";  // illegal!
```

- **`String`** vs. **`StringBuffer`**: The significant performance difference between these two classes is that **`StringBuffer`** is faster than **`String`** when performing simple concatenations.

```
String str = new String ("Stanford  ");
str += "Lost!!";
```

```
StringBuffer str = new StringBuffer ("Stanford ");
str.append("Lost!!");
```

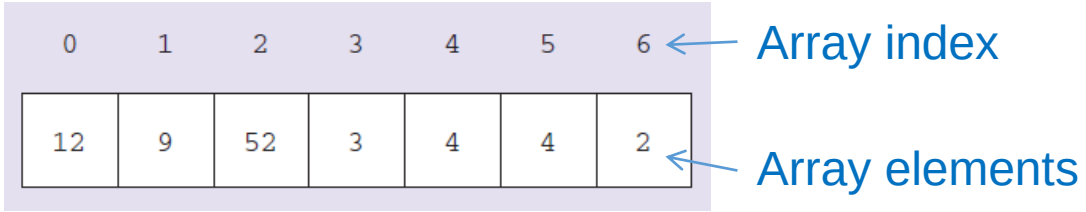
Runs faster

Outline

1. Data types
2. Operators
3. Strings
4. Arrays
5. Conditional processing
6. Repetitive processing
7. Developing some methods

4. Arrays

An array is a collection of items of the **same type**.



Array declaration

- In Java, array variables are references, not primitives.
- Two ways to declare arrays **of integers**:

```
int[] holder;      → preferable
```

or

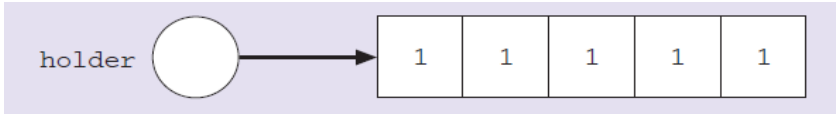
```
int holder[];
```

- other types of arrays are `boolean[]`, `char[]`, `double[]`, and `String[]`.

Array initializers

- Example of declaration and assignment:

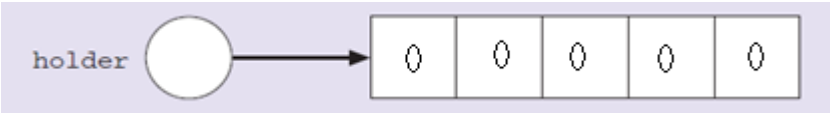
```
int[] holder = {1,1,1,1,1};
```



4. Arrays

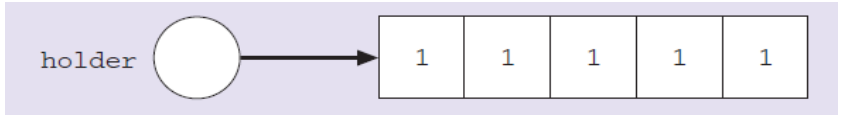
- **Another way of creating an array is as follows:**

```
int[] holder = new int[5];
```



Here, we have only stated that there should be room for four integers to be stored. We can initialize these elements (they will default to the value 0) using assignment to individual locations:

```
holder[0]= 1;
holder[1]= 1;
holder[2]= 1;
holder[3]= 1;
holder[4]= 1;
```



- This is equivalent to the following:

```
int []GirlsStudents={22,34,56,33}; //Declare and initializ
Students[0]=10; //Direct initialize item by item.
Students[1]=2;
Students[2]=30;
Students[3]=4;
Students[4]=5;
```

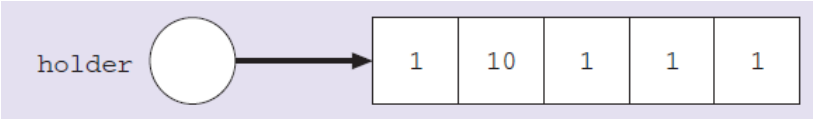
```
int[] myArray = {1,1,1,1,1};
```

4. Arrays

Assigning values to individual elements

- Example of changing the value stored in holder's **second** location to 10:

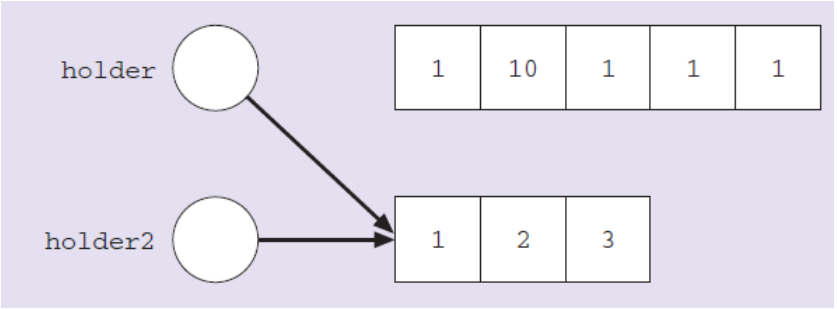
```
holder [1]= 10;
```



Changing an array reference

- As with strings (and it is true of all references), we can make an array reference variable refer to a different object.

```
int[] holder2 = {1, 2, 3};
holder = holder2;
```



4. Arrays

The array instance variable `length`

- Arrays have an instance variable **`length`**, which gives the capacity of the array. The expression:

  `holder.length`

  represents the number of locations in the array **`holder`**.

- Note that **`length`** is an **instance variable** associated with **arrays**, while **`length()`** is a **method** associated with **strings**.

Array examples Inverse

Method 1 inverse in same array

```
int ctr1=0;
int ctr2=4;
for (int i=0;i<5/2;i++) // First way to inverse array in itsel
{
    temp=Students[ctr1];
    Students[ctr1]=Students[ctr2];
    Students[ctr2]=temp;
    ctr1++;
    ctr2--;
}
```

Method 2 inverse in another array

```
int ctr3=4; // second way to inverse arra
for (int i=0;i<5;i++)
{
    TempArray[ctr3]=Students[i];
    ctr3--;
}
for (int i=0;i<5;i++)
{
    Students[i]=TempArray[i];

}
```

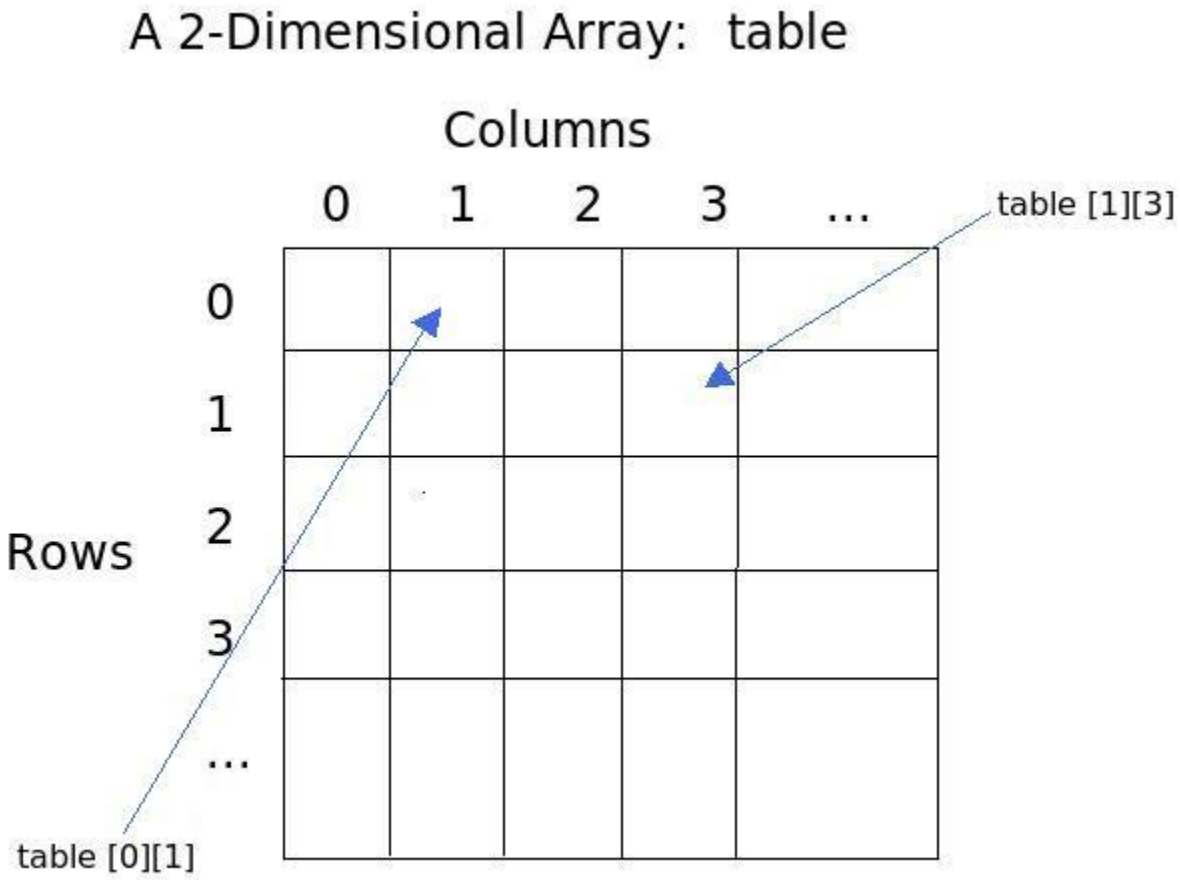
Which is better you think ?

2D arrays

```
//2 D arrays
int [][] TwoDimArray=new int [5][5];
int counter=0;
for (int i=0;i<5;i++)
{
  for (int j=0;j<5;j++)
    {
        TwoDimArray[i][j]=counter;
        counter++;
        System.out.println(counter);
    }
 }
```

Loop 1

Loop 2



- A 1 dimension array uses 1 for loop.
- A 2 dimension array uses 2 for loops
- A 3 Dimension array use ….. loop

Outline

1. Data types
2. Operators
3. Strings
4. Arrays
5. **Conditional processing**
6. **Repetitive processing**
7. **Developing some methods**

5. Conditional processing

- The **flow control structures** determine the way in which a Java program is executed, allowing **different segments** of code to be executed under **different circumstances**.

The `if` statement

```
if (logical_expression)
{
    statements;
}
```

- If the logical_expression within the parentheses evaluates to true, then any statements in the following code block are executed. If the logical_expression is false, then any statements in the body of the if statement are not executed.

Example:

```
if (a == b)
{
    b = 23;
    signal = true;
    a = 0;
}
```

5. Conditional processing

The if ... else statement

```
if (logical_expression)
{
    statementsA;
}
else
{
    statementsB;
}
```

- statementsA are executed if the logical_expression is true and statementsB are executed if the expression is false.
- Example:

```
boolean novice, verbose;
// novice gets set to true or false
if (novice)
{
    verbose = true;
}
else
{
    verbose = false;
}
```

5. Conditional processing

Nesting if statements

- We can write if statements within if statements, which we call nesting. An example of this is shown below:
- Example:

```
if (a == 34)
{
    s = 23;
    i++;
    if (i < 23)
    {
        b++;
        c++;
    }
}
```

5. Conditional processing

The `switch ... case` statement

Example:

```
char control;
// control is assigned a value
switch (control)
{
    case 'a':
    {
        videoId = 3;
        break;
    }
    case 'b':
    {
        videoId = 19;
        soundId = 12;
        break;
    }
    case 'c':
    {
        videoId = 11;
        link++;
        break;
    }
}
// after a break, or if no cases match,
// execution resumes here
```

```
switch (control)
{
    case 'a':
        videoId = 3;
        break;
    case 'b':
        videoId = 19;
        soundId = 12;
        break;
    case 'c':
        videoId = 11;
        link++;
    case 'x' :
        // code for new case
}
```

5. Conditional processing

The `switch ... case` statement

The argument is an expression of type **int**, **char**, **short** or **byte** (usually just the **name of a variable**).

Each selector is a constant value (usually **a literal**) compatible with the argument type.

A code block enclosing the statements in each case is optional.

The statements are performed if the case selector is logically **equal to** the argument ; in other words, **if argument == selector**.

The `break` keyword causes the switch to terminate. **Failure to use a break** statement results in control 'falling through' to the next case.

A default case may be given to indicate processing to take place when no selector is matched.

```
switch (argument)
{
    case selector:
        statements;
        break;
    case selector:
        statements;
        break;
    case selector:
        statements;
        break;
    // as many cases as required
    case selector:
        statements;
        break;
    default:
        statements;
        break;
}
```

Outline

1. Data types
2. Operators
3. Strings
4. Arrays
5. Conditional processing
6. Repetitive processing
7. Developing some methods

6. Repetitive processing

- Java allow sections of code to be executed **repeatedly** while some condition is satisfied → iteration!

```
while (logical_expression)
{
    statements;
}
```

Meaning: While the logical_expression is true, the statements are executed. The truth of the logical_expression is rechecked after each execution of the body of the while.

Example:

```
int [ ] vals = {10, 20, 30, 40, 50};
int i = 0, sum = 0;
while (i < 3)
{
    sum = sum + vals [i];
    i++;
}
```

6. Repetitive processing

The `for` statement

```
for (control_initializer; logical_expression; control_adjustment)
{
    statements;
}
```

Meaning: The control variable is created and given its initial value in control_initializer. Next, the logical_expression determines if the loop body is executed. After each time the statements are executed, control_adjustment defines the new value of the control variable, and the logical_expression is checked again. The loop terminates when the logical_expression becomes false.

Note: The control_initializer is only executed once and is the first statement to be executed by the for.

Examples:

```
int [ ] a = {1,2,3,4,5,6,7,8,9,10};
for (int i = 0; i < a.length; i++)
{
    a [i] = 10;
}
```

This sets all elements of a[] to 10

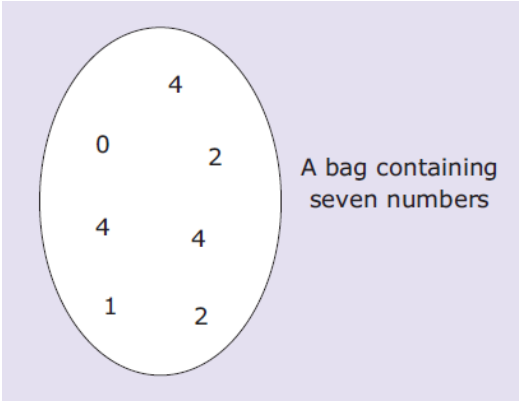
```
for (int i = 2 * start; i <= endValue; i = i + 2)
{
    statements;
}
```

Can you figure out how this loop is executed??

Outline

1. Data types
2. Operators
3. Strings
4. Arrays
5. Conditional processing
6. Repetitive processing
7. **Developing some methods**

7. Developing some methods



Example 1: a bag

- This example shows the code for methods associated with a simple bag that keeps track of the number of times the integers ranging from 0 to 9 are stored in it.
- A way of storing the items in this bag would be an array that has 10 locations corresponding to the 10 possible integers that it could contain

  bagVals: `int []bagVals = {0,0,0,0,0,0,0,0,0,0}`

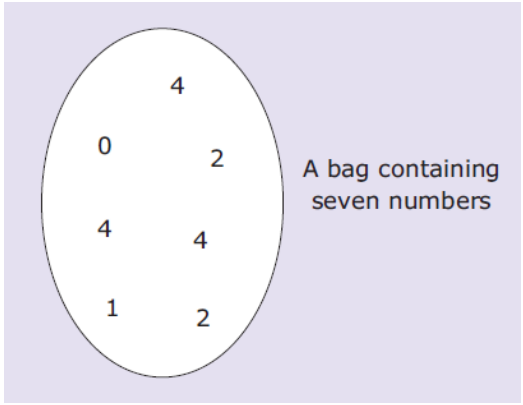
  By initializing the array locations to zero we are indicating that the bag is empty.
- We show two methods:

  1) a method that adds an integer to a bag, `addToBag`:
  - The method takes one argument, an integer to be added. This integer must from 0 to 9.

```
public void addToBag (int toBeAdded){
++bagVals [toBeAdded];}
```

  - Thus, the nth array location contains the number of times that the value n has been added to the bag. For example, if this method were called with the argument 1, then bagVals [1] would be incremented from 0 to 1. If this were to happen a second time, bagVals [1] would be incremented from 1 to 2.

7. Developing some methods



Example 1: a bag (Cont'd)

2) a method that returns the number of occurrences of a particular integer within a bag, `findNum`.

- It has one argument, the integer whose number of occurrences we want to know.

```
++bagVals public
int findNum (int lookedFor){
        return bagVals [lookedFor];
    }
```

- For example, if the argument lookedFor is 1, then we return the value stored in
- bagVals [1]. Remember that the keyword public specifies that any other class can use this method and the keyword int indicates that an integer value is returned from the method.

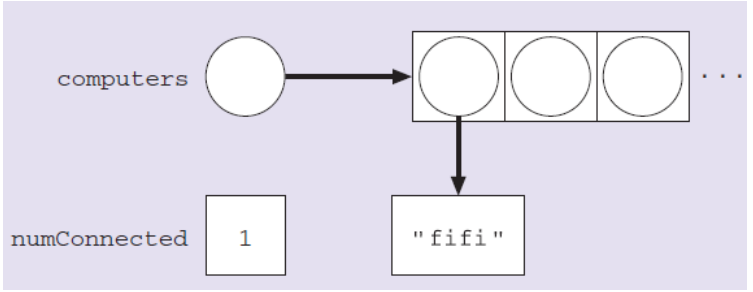
7. Developing some methods

Example 2: a collection of computer names

- In this example, we develop a set of methods for an **object** that contains a collection of computer names. Each name in the collection represents a computer currently connected to a network.

2 instance variables associated with this object:

computers, which is an array of strings storing the names of computers, and **numConnected**, which is an integer variable that contains the current number of computers connected together in the network.



Assume that four methods are required:

1) a method that returns the number of computers in the network, getNumConnected;

2) a method that adds a named computer to the network, addComputer;

3) a method that returns the index at which a named computer is found in the collection or -1 if the computer is not found in the collection, indexOf;

4) a method that removes a named computer from the network, removeComputer.

7. Developing some methods

Example 2: a collection of computer names (Cont'd)

Assumptions:

1) we will not be adding a computer with a name that is the same as a computer already in the network;

2) we will have enough room in the computers array to hold all the computers we want to add to network.

```
public int getNumConnected ()
{
    return numConnected;
}
```

```
public void addComputer (String addedComputer)
{
    computers [numConnected] = addedComputer;
    numConnected++;
}
```

```
public int indexOf (String wantedComputer)
{
    boolean found = false;
    int i = 0;
    while (!found && i < numConnected)
    {
        found = wantedComputer.equals(computers [i]);
        i++;
    }
    if (found)
    {
        // i was incremented after the find
        return i - 1;
    }
    else
    {
        return -1;
    }
}
```

```
public void removeComputer (String unwantedComputer)
{
    // find the index of the computer
    int i = indexOf(unwantedComputer);
    // if it is not -1, the computer was found
    if (i > -1)
    {
        /* Move the computers after this position to the left by one.
           The body of this loop is not executed if the computer
           is last in the collection. */
        for (int j = i; j < numConnected - 1; j++)
        {
            computers [j] = computers [j + 1];
        }
        numConnected--;
    }
}
```

Reading Input from the Console

1. Import Scanner class

```
import java.util.Scanner;
```

2. Create a Scanner object

```
Scanner input = new Scanner(System.in);
```

3. Use the methods next(), nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble(), or nextBoolean() to obtain to a string, byte, short, int, long, float, double, or boolean value. For example,

```
System.out.print("Enter a double value: ");
Scanner input = new Scanner(System.in);
double d = input.nextDouble();
```

Methods for Scanner Objects

TABLE 2.1 Methods for `Scanner` Objects

Method	*Description*
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.
`next()`	reads a string that ends before a whitespace character.
`nextLine()`	reads a line of text (i.e., a string ending with the *Enter* key pressed).