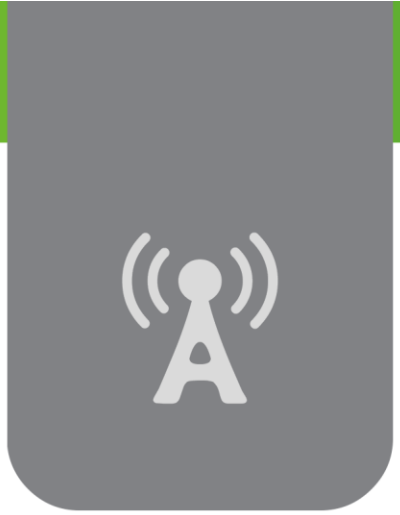


Chapter 1:
Java Basics

Title:

Java Basics

Key Words:

Variables, Data Types, Parameters, Conditional Statement, Loops, Arrays.

Summary:

This unit provides an overview of Java programming language basics.

Outcomes:

Student will learn in this unit:

- Using different data types.
- Passing parameters and returning values.
- Using conditional statements.
- Using loops.
- Dealing with arrays.

Plan:

2 Learning Objects:

1. Java Basics: Data Types
2. Java Basics: Statements

1. Data Types

Learning outcomes:

Using variables and data types.

Java's Variable

- **Variable**: A piece of the computer's memory that is given a name and type, and can store a value.
- A variable can be declared / initialized in one statement.
- Syntax:
  ```
  type name = value;
  ```
- Example:

```
double myGPA = 3.95;
int x = (11 % 3) + 12; // x will be 14
```

Java's primitive types:

- **primitive types**: 8 simple types for numbers, text, etc.
  - Java also has **object types**, which we'll talk about later

Name	Description	Examples
`int`	integers	`42, -3, 0, 926394`
`double`	real numbers	`3.1, -0.25, 9.4e3`
`char`	single text characters	`'a', 'X', '?', '\n'`
`boolean`	logical values	`true, false`

Type casting:

- **Type cast**: A conversion from one type to another.
  - To promote an int `int` a `double` to get exact division from /
  - To truncate a `double` from a real number to an integer
- Syntax:
  `(type) expression`
- Examples:

```
double result = (double) 19 /5;  // 3.8
int result2 = (int) result;      //   3
int x = (int) Math.pow(10, 3);   //   1000
```

Increment and decrement:

shortcuts to increase or decrease a variable's value by 1

Shorthand	Equivalent longer version
variable++;	**variable** = **variable** + 1;
variable−−;	**variable** = **variable** − 1;

- Examples:

```
int x = 2;
x++;                        // x = x + 1;
                            // x now stores 3

double gpa = 2.5;
gpa--;                      // gpa = gpa - 1;
                            //gpa now stores 1.5
```

Precedence:

- **Precedence**: Order in which operators are evaluated.
- Generally, operators evaluate left-to-right

```
1-2-3 is (1-2) -3 which is -4
```

- But * / % have a higher level of precedence than + -

```
1 + 3 * 4         is 13
6 + 8 / 2* 3
6 + 4 * 3
6 + 12       is 18
```

- Parentheses can force a certain order of evaluation

```
(1 + 3)*4         is 16
```

- Spacing does not affect order of evaluation

```
1 + 3 * 4 - 2     is 11
```

String concatenation:

- string concatenation: Using + between a string and another value to make a longer string.

```
"hello" + 42 is "hello42"
1 + "abc" + 2 is "1abc2"
"abc" + 1 + 2 is "abc12"
1 + 2 + "abc" is "3abc"
"abc" + 9 * 3 is "abc27"
"1" + 1 is "11"
4 - 1 + "abc" is "3abc"
```

- Use + to print a string and an expression's value together.
- Example:

```
System.out.println ("Grade:" + (95.1 + 71.9) / 2)
```

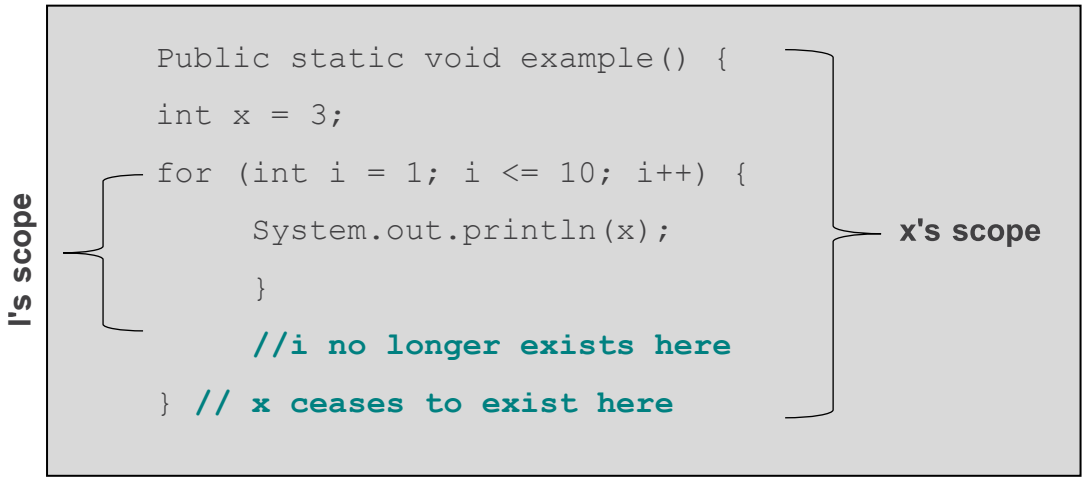
  - Output: `Grade: 83.5`

Variable scope:

- **Scope**: The part of a program where a variable exists.

  From its declaration to the end of the { } braces

  - A variable declared in a for loop exists only in that loop.
  - A variable declared in a method exists only in that method.



Class constants:

- **class constant**: A value visible to the whole program.
  - value can only be set at declaration
  - value can't be changed while the program is running
- Syntax:

  `public static final` **`type name = value;`**
  - name is usually in ALL_UPPER_CASE
  - Examples:

```
public static final int DAYS_IN_WEEK = 7;
public static final double INTEREST_RATE = 3.5
public static final int SSN = 658234569;
```

Passing parameters:

- Declaration:

```
public void name (type  name, ..., type  name) {
        statement(s);
}
```

- Call:

```
methodName (value, value, ..., value);
```

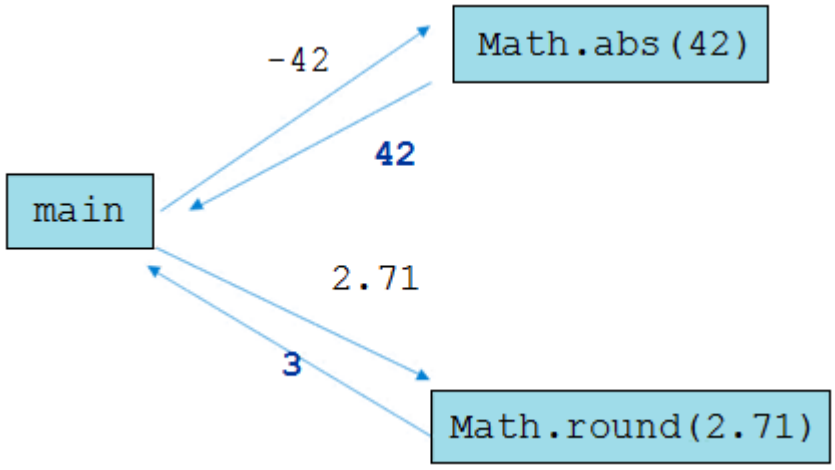
- Example:

```
public static void main(String[] args) {
     sayPassword(42);    //The password is: 42
     sayPassword(12345); //The password is: 12345
}


public static void sayPassword(int code) {
     System.out.println("The password is: " +
     code);
}
```

Return:

- **return**: To send out a value as the result of a method.
- The opposite of a parameter:
  - Parameters send information in from the caller to the method.
  - Return values send information out from a method to its caller.



Java's Math class:

Method name	Description
`Math.abs(value)`	absolute value
`Math.round(value)`	nearest whole number
`Math.ceil(value)`	rounds up
`Math.floor(value)`	rounds down
`Math.log10(value)`	logarithm, base 10
`Math.max(value1, value2)`	larger of two values
`Math.min(value1, value2)`	smaller of two values
`Math.pow(base, exp)`	base to the exp power
`Math.sqrt(value)`	square root
`Math.sin(value)`	sine of an angle in radians
`Math.cos(value)`	cosine of an angle in radians
`Math.tan(value)`	tangent of an angle in radians
`Math.toDegrees(value)`	convert degrees to radians and back
`Math.toRadians(value)`	random double between 0 and 1
`Math.random()`	sine of an angle in radians

Returning a value:

- Syntax:

```
public type name(parameters) { statements;
...
return expression;
}
```

- Example:

```
// Returns the slope of the line between the given
points.
public double slope(int x1, int y1, int x2, int y2)
{
    double dy = y2 - y1;
    double dx = x2 - x1;
    return dy/dx;
}
```

Strings:

- **string**: An object storing a sequence of text characters.

```
String name = "text";
String name = expression;
```

  - Characters of a string are numbered with 0-based indexes:

```
String name = "P. Diddy";
```

Index	0	1	2	3	4	5	6	7
Char	P	.		D	i	d	d	y

- The first character's index is always 0
- The last character's index is 1 less than the string's length
- The individual characters are values of type `char`

`String` methods:

Method name	Description
`indexOf(`**`str`**`)`	index where the start of the given string appears in this string (−1 if it is not there)
`length()`	number of characters in this string
`substring(`**`index1`**`, `**`index2`**`)` or `substring(`**`index1`**`)`	number of characters in this string the characters in this string from *index1* (inclusive) to *index2* (exclusive); if *index2* omitted, grabs till end of string
`indexOf(`**`str`**`)`	index where the start of the given string appears in this string (−1 if it is not there)
`toLowerCase()`	a new string with all lowercase letters
`toUpperCase()`	a new string with all uppercase letters

- **These methods are called using the dot notation**

```
String gangsta = "Dr. Dre";
System.out.println (gangsta.length()); // 7
```

String test methods:

Method	Description
`equals(str)`	whether two strings contain the same characters
`equalsIgnoreCase(str)`	whether two strings contain the same characters, ignoring upper vs. lower case
`startsWith(str)`	whether one contains other's characters at start
`endsWith(str)`	whether one contains other's characters at end
`contains(str)`	whether the given string is found within this one

```
String name = console.next();
if (name.startsWith("Dr.")) {
   System.out.println("Are you single?");
} else if (name.equalsIgnoreCase("LUMBERG")) {
   System.out.println("I need your TPS
   reports.");
}
```

The equals method:

- **Objects are compared using a method named** `equals`.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
     System.out.println("I love you, you love me,");
     System.out.println("We're a happy family!");
}
```

Technically this is a method that returns a value of type `boolean`, the type used in logical tests.

Type char:

- `char`: **A primitive type representing single characters**.
  - Each character inside a `String` is stored as a `char` value.
  - Literal char values are surrounded with apostrophe (single-quote) marks, such as 'a' or '4' or '\n' or '\"'
  - It is legal to have variables, parameters, returns of type `char`

```
char letter = 'S';
System.out.println(letter); // S
```

- `char` **values can be concatenated with strings**.

```
char initial = 'P';
System.out.println(initial + " Diddy");// P Diddy
```

char vs. String:

- "h" is a String
- 'h' is a `char` (the two behave differently)
- `String` is an object; it contains methods

```
String s = "h";                    // 'H'
s = s.toUpperCase();               // 1
int len = s.length();        // 1
char first = s.charAt(0);          // 'H'
```

- `Char`: **is primitive; you can't call methods on it**

```
char c = 'h';
c = c.toUpperCase();// ERROR: "cannot be dereferenced"
```

2. Java Basics: Statements

Learning outcomes:

conditional statements, loops, and arrays

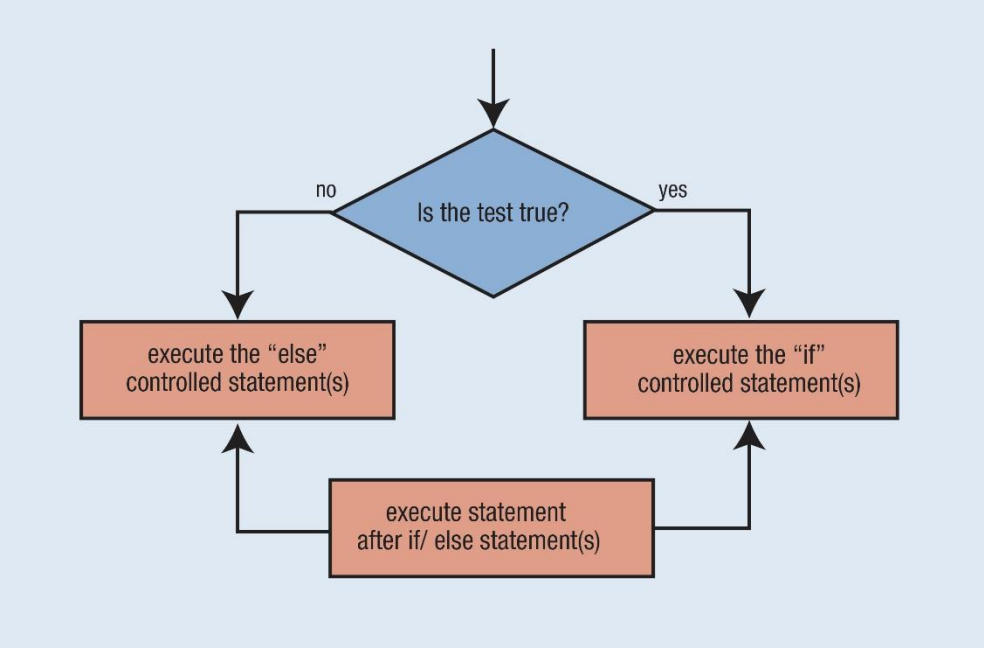
Relational expressions:

if/else

Executes one block if a test is true, another if false.

- Syntax:

```
if (test) {
        statement(s);
} else {
        statement(s);
}
```



- Example:

```
double gpa = console.nextDouble();
if (gpa >= 2.0) {
    System.out.println ("Welcome to Mars
    University!");
} else {
      System.out.println ("Application
      denied.");
}
```

- **A test** in an if is the same as in a for loop.

```
for (int i = 1; i <= 10; i++) { ...
if (i <= 10) { ...
```

  - **These are** `boolean` expressions.

- Tests use *relational operators:*

Operator	Meaning	Example	Value
==	equals	`1 + 1 == 2`	`true`
!=	does not equal	`3.2 != 2.5`	`true`
<	less than	`10 < 5`	`false`
>	greater than	`10 > 5`	`true`
<=	less than or equal to	`126 <= 100`	`false`
>=	greater than or equal to	`5.0 >= 5.0`	`true`

Logical operators: &&, ||, !

Conditions can be combined using *logical operators*:

Operator	Description	Example	Result
`&&`	and	`(2 == 3) && (-1 < 5)`	`false`
`||`	or	`(2 == 3) || (-1 < 5)`	`true`
`!`	not	`!(2 == 3)`	`true`

Truth table:

p	q	p && q	p || q	! p
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Type boolean:

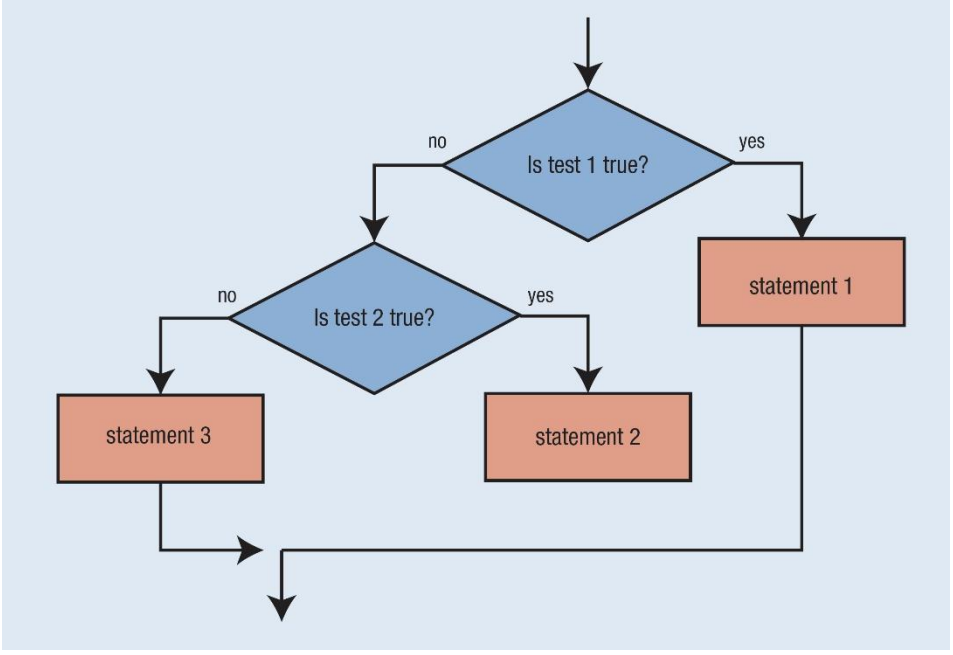
- **boolean**: A logical type whose values are true and false.
- **A test** in an `if`, `for`, or while is a `boolean` expression.
  - You can create `boolean` variables, pass `boolean` parameters, return boolean values from methods, ...
- Example

```
boolean minor = (age < 21);
boolean expensive = iPhonePrice > 200.00;
boolean iLoveCS = true;
if (minor) {
   System.out.println("Can't purchase alcohol!");
}
if (iLoveCS || !expensive) {
System.out.println("Buying an iPhone");
 }
```

If / else Structures:

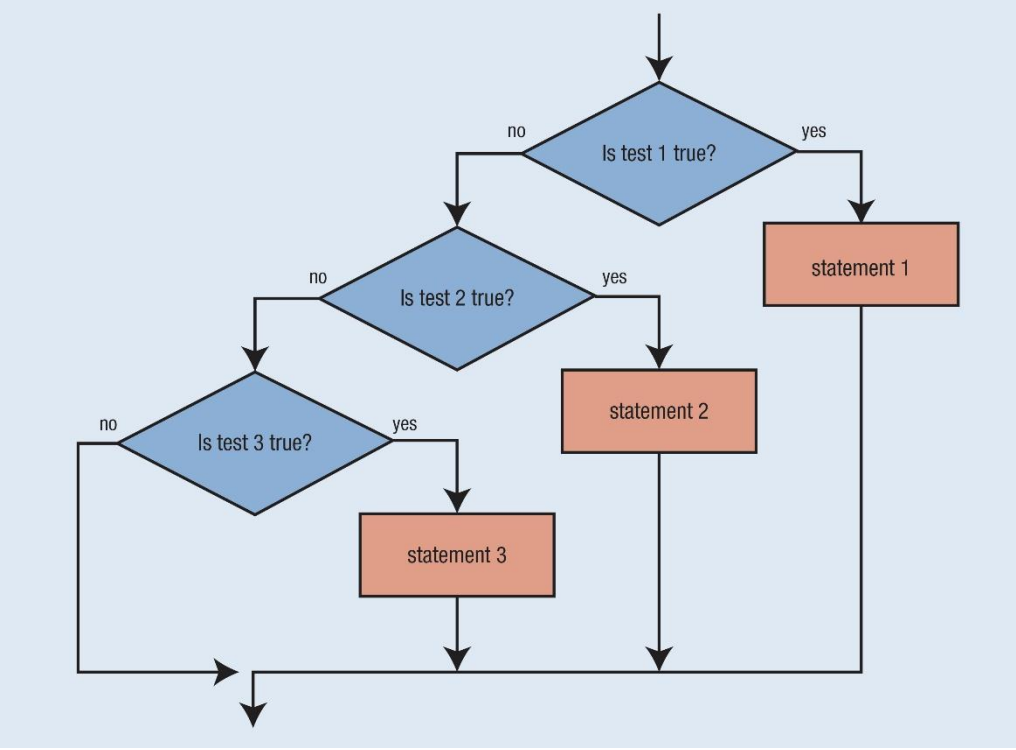
- Exactly 1 path: (mutually exclusive)
  - Syntax:

```
if (test) {
      statement(s); statement(s);
} else if (test) {
      statement(s);
} else {
      statement(s);
      }
```



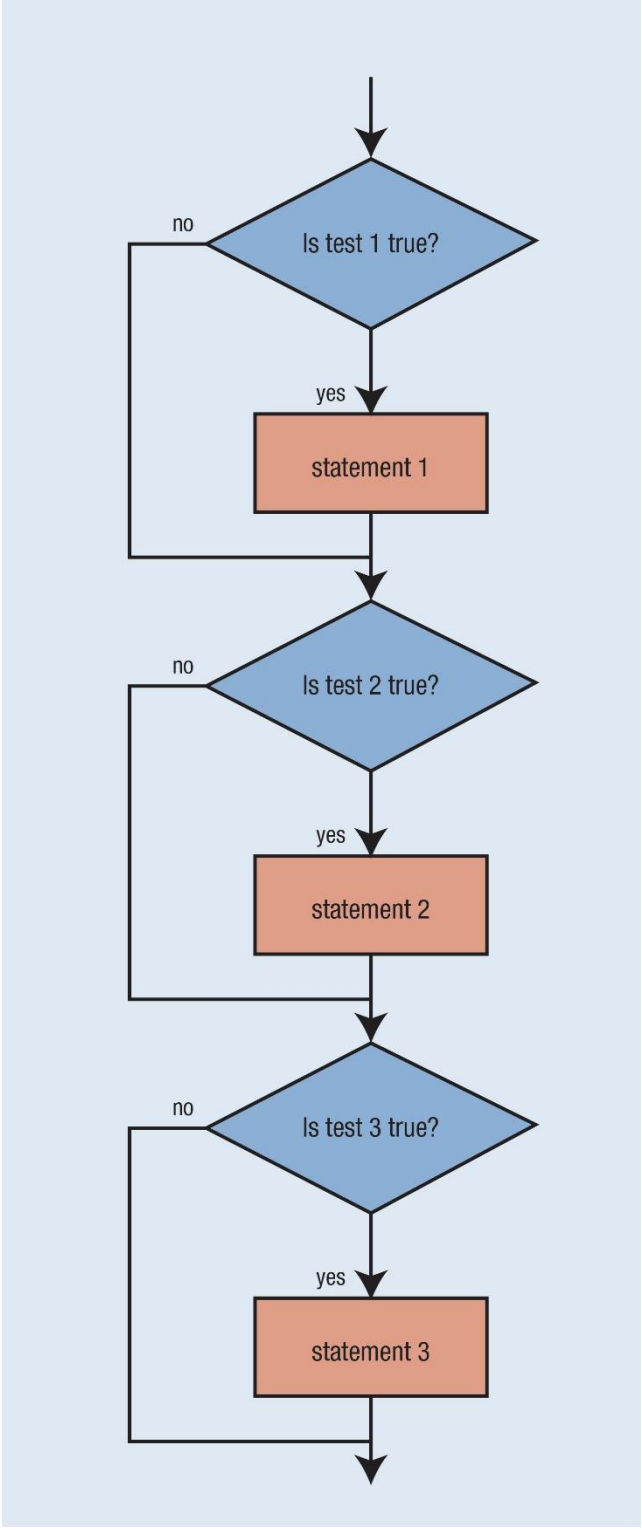
- 0 or 1 path:
  - Syntax:

```
if (test) {
statement(s);
} else if (test) {
statement(s);
} else if (test) {
statement(s);
}
```



- 0, 1, or many paths: (independent tests, not exclusive)
  - Syntax:

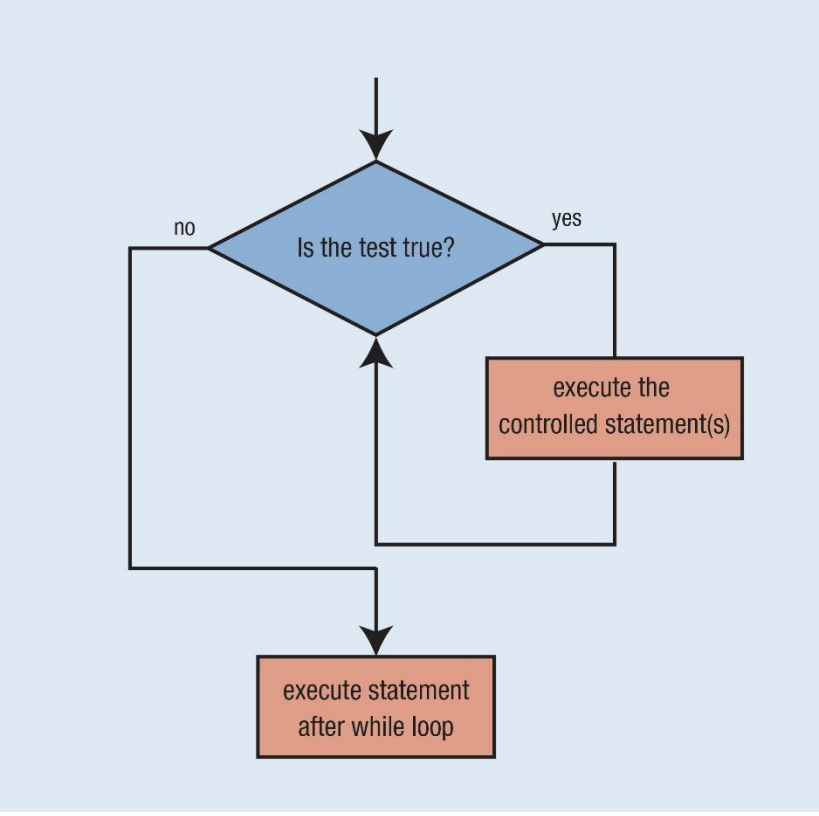
```
if (test) {
      statement(s);
}
if (test) {
      statement(s);
}
if (test) {
      statement(s);
}
```


no
Is test 1 true?
yes
statement 1
no
Is test 2 true?
yes
statement 2
no
Is test 3 true?
yes
statement 3

while loops:

- **`while` loop**: Repeatedly executes its body as long as a logical test is true.
  - Syntax:

```
while (test) {
        statement(s);
}
```



- Example:

```
int num=1;                      // initialization
while (num <=200) {             // test
   System.out.print(num + " ");
   num = num  * 2;              // update
}
```

  - Output: 1 2 4 8 16 32 64 128

The `Random` class:

- **A `Random` object generates pseudo-random* numbers.**
  - **Class `Random` is found in the `java.util` package.**

```
import java.util.*;
```

Method name	Description
`nextInt()`	returns a random integer
`nextInt(max)`	returns a random integer in the range $[0, \text{max})$ in other words, 0 to max-1 inclusive
`nextDouble()`	returns a random real number in the range $[0.0, 1.0)$

- Example:

```
Random rand = new Random();
int randomNumber = rand.nextInt(10);     // 0-9
```

break:

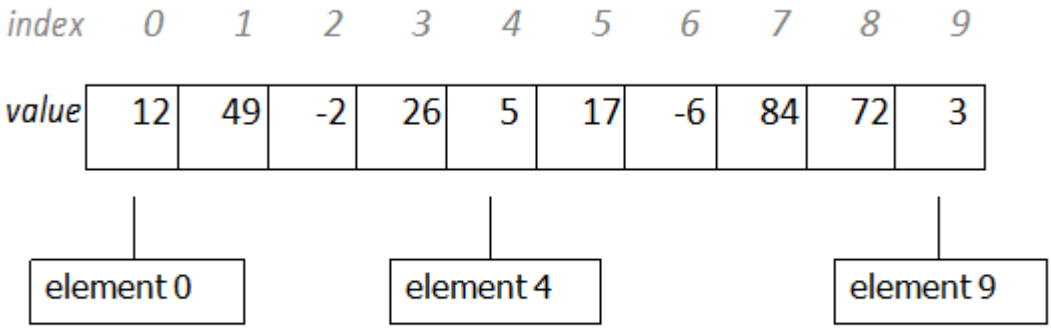
- **`break`** statement: Immediately exits a loop.
  - Can be used to write a loop whose test is in the middle.
  - Such loops are often called *"forever" loops* because their header's boolean test is often changed to a trivial `true`.
  - Syntax:

```
while (true) {
  statement(s);
  if (test) {
    break;
  }
  statement(s);
}
```

  - Some programmers consider `break` to be bad style.

Arrays:

- **array**: object that stores many values of the same type.
  - **element**: One value in an array.
  - **index**: A 0–based integer to access an element from an array.



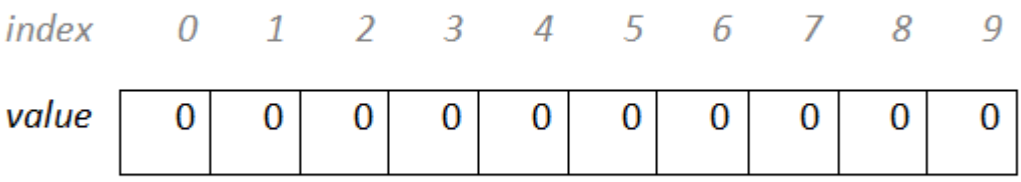
Array declaration:

- Syntax:

```
type[] name = new type[length];
```

- Example:

```
int[] numbers = new int[10];
```



Accessing elements:

- Syntax:

```
name[index]                    // access
name[index] = value;           // modify
```

- Example:

```
numbers[0] = 27;
numbers[3] = -6;
System.out.println (numbers[0]);
if (numbers[3] < 0) {
     System.out.println("Element 3 is negative.");
     }
```

Index	1	2	3	4	5	6	7	8	9
value	27	0	0	-6	0	0	0	0	0

Out-of-bounds:

- Legal indexes: between **0** and the **array's length** - **1**.
  - Reading or writing any index outside this range will throw an `ArrayIndexOutOfBoundsException`.
- Example:

```
int[] data = new int[10];
System.out.println(data[0]);     // okay
System.out.println(data[9]);     // okay
System.out.println(data[-1]);         // exception
System.out.println(data[10]);         // exception
```

The length field:

- An array's `length` field stores its number of elements.
  - Syntax:
    `name``.length`
- Example:

```
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
// output: 0 2 4 6 8 10 12 14
```

- It does not use parentheses like a String's `.length()`.

Quick array initialization:

```
type[] name = {value, value, … value};
```

- Example:

```
int[] numbers = {12, 49, -2, 26, 5, 17, -6};
```

Index	0	1	2	3	4	5	6
value	12	49	−2	26	5	17	−6

- Useful when you know what the array's elements will be.
- The compiler figures out the size by counting the values.

The `Arrays` class:

- Class `Arrays` in package `java.util` has useful static methods for manipulating arrays:

Method name	Description
`binarySearch(`**`array, key`**`)`	returns the index of the given value in a
`equals(`**`array1, array2`**`)`	returns `true` if the two arrays contain the same elements in the same order
`fill(`**`array, value`**`)`	sets every element in the array to have the
`sort(`**`array`**`)`	arranges the elements in the array into
`String toString (`**`array`**`)`	returns a string representing the array, such

Arrays as parameters:

- Declaration:
  `public type methodName(type[] name) {`
- Example:

```
public double average(int[] numbers) {
}
```

- Call:
  `methodName(arrayName);`
- Example:

```
int[] scores = {13, 17, 12, 15, 11};
double avg = average(scores);
```

Arrays as return:

- Declaring:

```
public type[] methodName(parameters) {
```

- Example:

```
public int[] countDigits(int n) {
    int[] counts = new int[10];
    ...
    return counts;
}
```

- Calling:

```
type[] name = methodName(parameters);
```

- Example:

```
public static void main(String[] args) {
    int[] tally = countDigits(229231007);
    System.out.println(Arrays.toString(tally));
}
```

Value semantics (primitives):

- **value semantics**: Behaviour where values are copied when assigned to each other or passed as parameters.
  - When one primitive 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;
```

Reference semantics (objects):

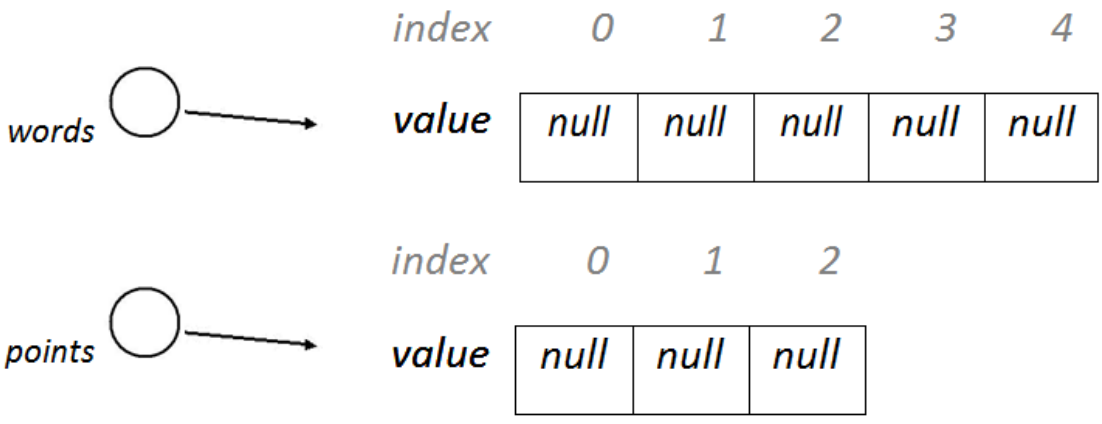
- **reference semantics**: Behaviour where variables actually store the address of an object in memory.
  - When one reference variable is assigned to another, the object is not copied; both variables refer to the same object.
  - Modifying the value of one variable *will* affect others.
- Example:

```
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1; // refer to same array as a1
a2[0] = 7;
System.out.println (a1[0]);       // 7
```

Null:

- **null**: A reference that does not refer to any object.
  - Fields of an object that refer to objects are initialized to `null`.
  - The elements of an array of objects are initialized to `null`.
- Example:

```
String[] words = new String[5];
Point[] points = new Point[3];
```



Null pointer exception:

- **dereference**: To access data or methods of an object with the dot notation, such as `s.length()`.
  - It is illegal to dereference null (causes an exception).
  - `null` is not any object, so it has no methods or data.

```
String[] words = new String[5];
System.out.println("word is: " + words[0]);
words[0] = words[0].toUpperCase();
```

  - Output:

```
word is: null

Exception in thread "main"
java.lang.NullPointerExceptin at Example.main
(Example.java:8)
```

Throwing exceptions:

- Syntax

```
throw new ExceptionType();
throw new ExceptionType("message");
```

- Generates an exception that will crash the program, unless it has code to handle (catch) the exception.
- Common exception types:

  ArithmeticException, ArrayIndexOutOfBoundsException, FileNotFoundException, IllegalArgumentException, IllegalStateException, IOException, NoSuchElementException, NullPointerException, RuntimeException, UnsupportedOperationException