



CS 213 – Programming Language 2

Dr. Ahmed Hesham Mostafa

Lectures 1 – Java Basics

Java Program Components

- **Class name**
- **Main method**
- **Statements**
- **Statement terminator**
- **Reserved words**
- **Comments**
- **Blocks**

Class Name

- Every Java program must have at least one class.
- Each class has a name.
- By convention, class names start with an uppercase letter.
- In this example, the class name is Welcome.

```
// This program prints Welcome to Java!  
public class Welcome {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Main Method

- Line 2 defines the main method.
- In order to run a class, the class must contain a method named main.
- The program is executed from the main method.

```
// This program prints Welcome to Java!  
public class Welcome {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Statement

- A statement represents an action or a sequence of actions.
- The statement `System.out.println("Welcome to Java!")` in the program displays the greeting "Welcome to Java!".

```
// This program prints Welcome to Java!  
public class Welcome {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Statement Terminator

Every statement in Java ends with a semicolon (;).

```
// This program prints Welcome to Java!  
public class Welcome {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Keywords

- Keywords are words that have a specific meaning to the compiler and cannot be used for other purposes in the program.
- For example, when the compiler sees the word class, it understands that the word after class is the name for the class.

```
// This program prints Welcome to Java!  
public class Welcome {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Blocks

- A pair of braces in a program forms a block that groups components of a program.

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Class block

Method block

Special Symbols

Character Name	Description
{ }	Opening and closing braces Denotes a block to enclose statements.
()	Opening and closing parentheses Used with methods.
[]	Opening and closing brackets Denotes an array.
//	Double slashes Precedes a comment line.
" "	Opening and closing quotation marks Enclosing a string (i.e., sequence of characters).
;	Semicolon Marks the end of a statement.

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

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

Example

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in);
        System.out.println("Enter name, age and salary:");
        String name = myObj.nextLine();
        int age = myObj.nextInt();
        double salary = myObj.nextDouble();
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}
```

Example Output

```
Enter name, age and salary:
```

```
ahmed
```

```
25
```

```
1200.25
```

```
Name: ahmed
```

```
Age: 25
```

```
Salary: 1200.25
```

Identifiers

An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs (\$).

An identifier must start with a letter, an underscore (_), or a dollar sign (\$). It cannot start with a digit.

An identifier cannot be a reserved word. (See Appendix A, “Java Keywords,” for a list of reserved words).

An identifier cannot be true, false, or null.

An identifier can be of any length.

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:

- **Primitive Data Types** store data values.
- **Non- Primitive Data Types** do not themselves store data values but are used to refer to objects. such as Strings, Arrays, and Classes

Data type sizes in Java are fixed

- to ensure portability across implementations of the language.

Primitive Data Types

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Data types

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

```
typeName variableName = value;
```

```
int cansPerPack = 6;
```

-Case sensitive
Do not
-use Reserved words
-start with number

Data types

- Constants “you can not change the value”

```
final typeName variableName = expression;
```

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 '<'.

Example

```
public class Main {  
    public static void main(String[] args) {  
        double doubleVal=2.8;  
        int intVal= (int) doubleVal;  
        System.out.println("intVal = "+intVal);  
  
        int a=35,b=12,d=13;  
        char valC=(char)(a+b+d);  
        System.out.println("valC = "+valC);  
    }  
}
```

Output


intVal = 2

valC = <

Data types

- 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!
```



```
double doubleVal=2.8;  
int intVal=doubleVal;
```

Not Allowed assign smaller
integer with big Double

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**

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 a length of zero.

String `concatenation` (+)

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

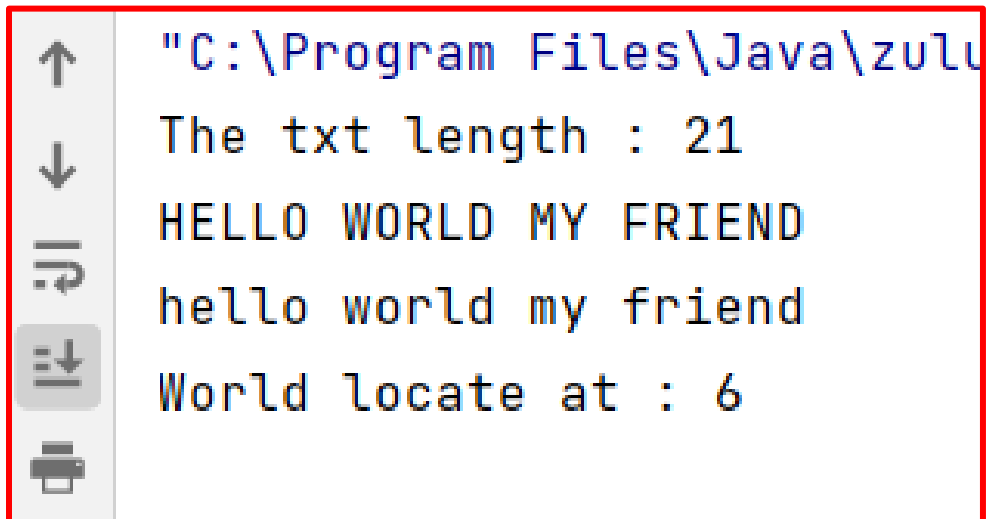
Strings

More string methods

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

Example

```
public class Main {  
    public static void main(String[] args) {  
  
        String txt = "Hello World My Friend";  
        System.out.println("The txt length : " + txt.length());  
        System.out.println(txt.toUpperCase());  
        System.out.println(txt.toLowerCase());  
        System.out.println("World locate at : "+txt.indexOf("World"));  
    }  
}
```



```
"C:\Program Files\Java\zulu  
The txt length : 21  
HELLO WORLD MY FRIEND  
hello world my friend  
World locate at : 6
```

Reading a String from the Console

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter First String: ");
        String First = input.nextLine();
        System.out.println("Enter Second String: ");
        String Second = input.next();
        System.out.println("First String is : "+First);
        System.out.println("First Second is : "+Second);
    }
}
```

```
Enter First String:
ahmed hesham mostafa
Enter Second String:
ibrahim ayman said
First String is : ahmed hesham mostafa
First Second is : ibrahim
```

Reading a Character from the Console

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter a character: ");
        String s = input.nextLine();
        char ch = s.charAt(0);
        System.out.println("The character entered is " + ch);
    }
}
```

Enter a character:

z

The character entered is z

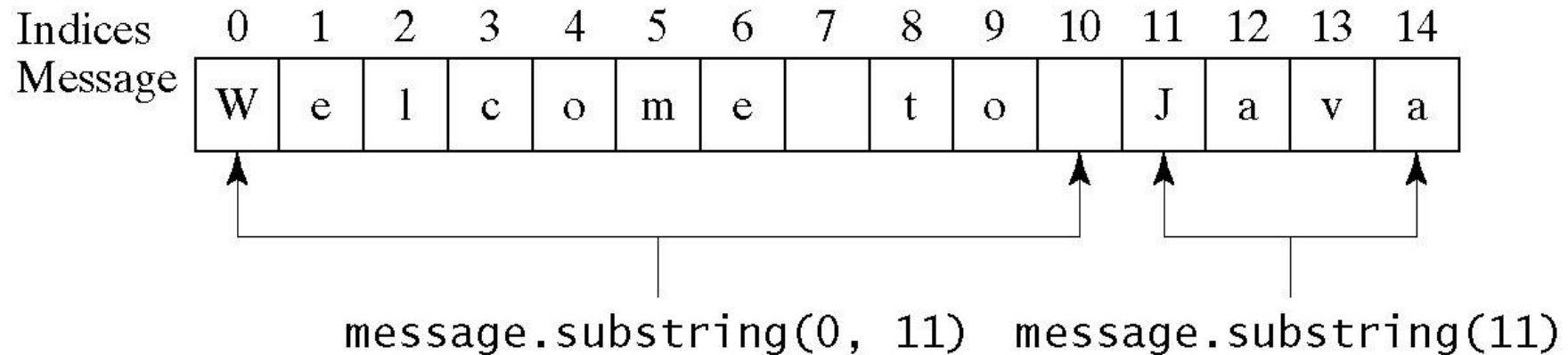
Comparing Strings

Method	Description
<code>equals(s1)</code>	Returns true if this string is equal to string <code>s1</code> .
<code>equalsIgnoreCase(s1)</code>	Returns true if this string is equal to string <code>s1</code> ; it is case insensitive.
<code>compareTo(s1)</code>	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than <code>s1</code> .
<code>compareToIgnoreCase(s1)</code>	Same as <code>compareTo</code> except that the comparison is case insensitive.
<code>startsWith(prefix)</code>	Returns true if this string starts with the specified prefix.
<code>endsWith(suffix)</code>	Returns true if this string ends with the specified suffix.

Note : the `==` operator is used to compare two object references to see whether they refer to the same instance.

Obtaining Substrings

Method	Description
<code>substring(beginIndex)</code>	Returns this string's substring that begins with the character at the specified <code>beginIndex</code> and extends to the end of the string.
<code>substring(beginIndex, endIndex)</code>	Returns this string's substring that begins at the specified <code>beginIndex</code> and extends to the character at index <code>endIndex - 1</code> . Note that the character at <code>endIndex</code> is not part of the substring.

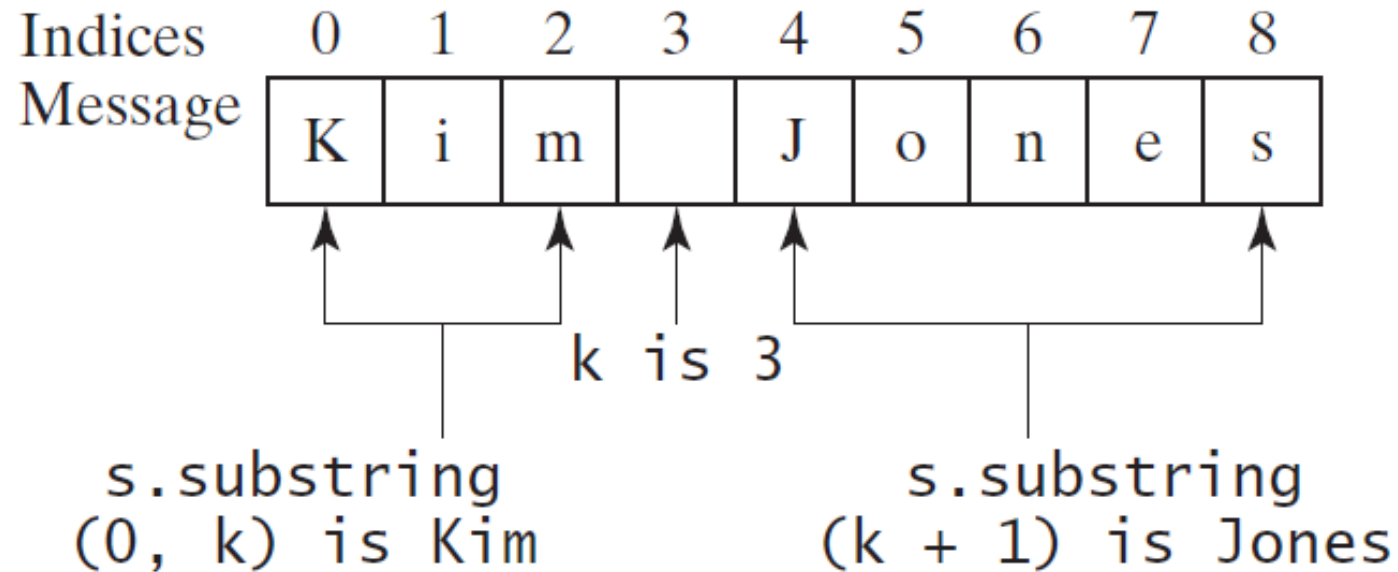


Finding a Character or a Substring in a String

Method	Description
<code>indexOf(ch)</code>	Returns the index of the first occurrence of <code>ch</code> in the string. Returns <code>-1</code> if not matched.
<code>indexOf(ch, fromIndex)</code>	Returns the index of the first occurrence of <code>ch</code> after <code>fromIndex</code> in the string. Returns <code>-1</code> if not matched.
<code>indexOf(s)</code>	Returns the index of the first occurrence of string <code>s</code> in this string. Returns <code>-1</code> if not matched.
<code>indexOf(s, fromIndex)</code>	Returns the index of the first occurrence of string <code>s</code> in this string after <code>fromIndex</code> . Returns <code>-1</code> if not matched.
<code>lastIndexOf(ch)</code>	Returns the index of the last occurrence of <code>ch</code> in the string. Returns <code>-1</code> if not matched.
<code>lastIndexOf(ch, fromIndex)</code>	Returns the index of the last occurrence of <code>ch</code> before <code>fromIndex</code> in this string. Returns <code>-1</code> if not matched.
<code>lastIndexOf(s)</code>	Returns the index of the last occurrence of string <code>s</code> . Returns <code>-1</code> if not matched.
<code>lastIndexOf(s, fromIndex)</code>	Returns the index of the last occurrence of string <code>s</code> before <code>fromIndex</code> . Returns <code>-1</code> if not matched.

Finding a Character or a Substring in a String

```
int k = s.indexOf(' ');  
String firstName = s.substring(0, k);  
String lastName = s.substring(k + 1);
```



Example

```
java
welcome to
First =welcome
last =to java
```

```
public class Main {
    public static void main(String[] args) {
        String mes="welcome to java";
        System.out.println(mes.substring(11));
        System.out.println(mes.substring(0,11));
        int k = mes.indexOf(' ');
        String first = mes.substring(0, k);
        String last = mes.substring(k + 1);
        System.out.println("First =" + first);
        System.out.println("last =" + last);
    }
}
```


Conversion between Strings and Numbers

```
public class Main {  
    public static void main(String[] args) {  
        String intString = "5";  
        int intValue = Integer.parseInt(intString);  
        String doubleString = "5.0";  
        double doubleValue = Double.parseDouble(doubleString);  
        System.out.println("intValue= "+intValue);  
        System.out.println("doubleValue= "+doubleValue);  
    }  
}
```

```
intValue= 5  
doubleValue= 5.0
```

Example

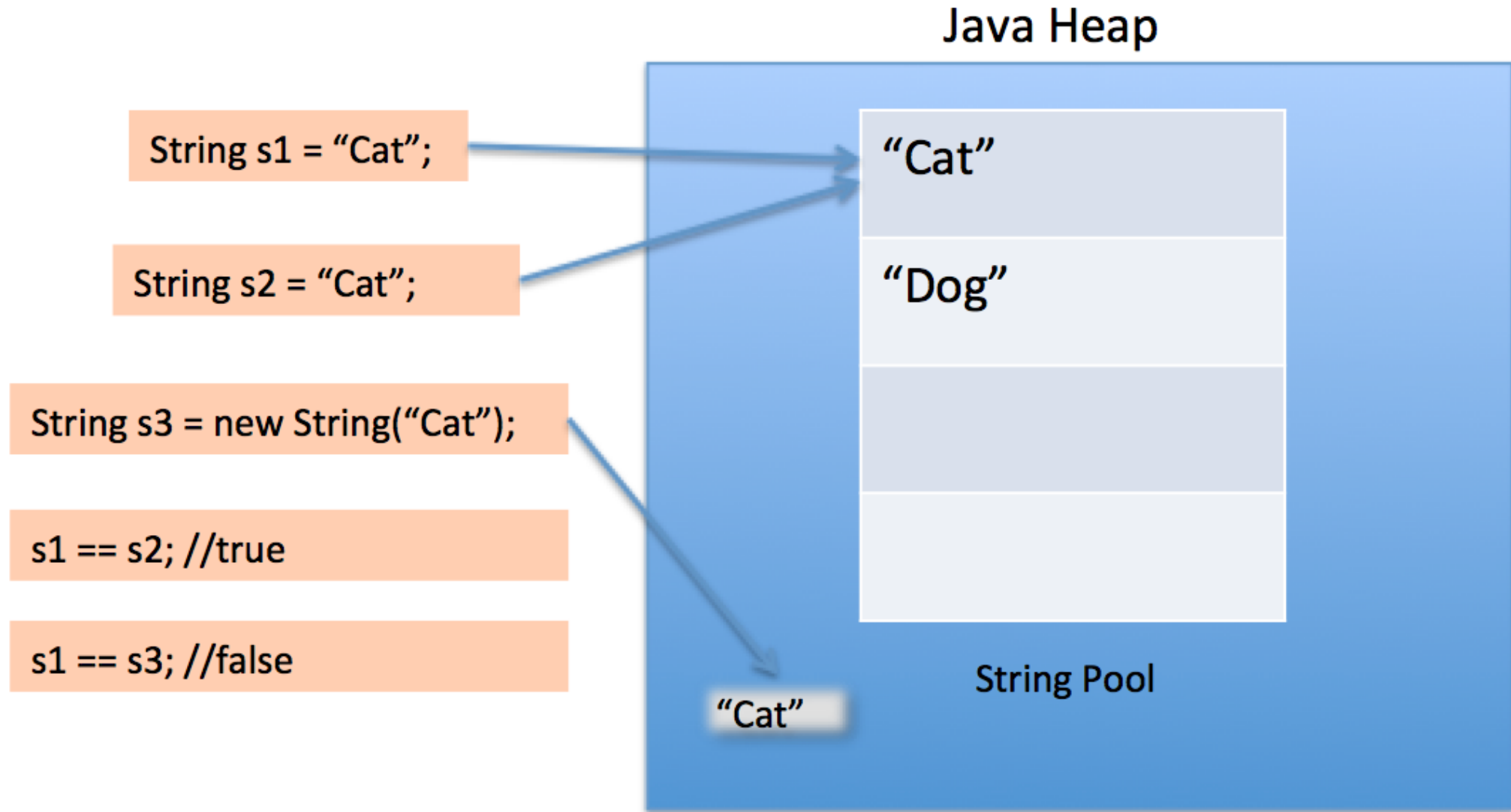
```
public class Main {  
    public static void main(String[] args) {  
        String str1 = "Cat";  
        String str2 = "Cat";  
        String str3 = new String(str1);  
        String str4 = "CAT";  
        System.out.println("str1.equals(str2): "+str1.equals(str2));  
        System.out.println("str1==str2 : "+(str1==str2));  
        System.out.println("str1==str3 : "+(str1==str3));  
        System.out.println("str1.equalsIgnoreCase(str4) : "+str1.equalsIgnoreCase(str4));  
        System.out.println("str1.compareTo(str2) : "+str1.compareTo(str2));  
        System.out.println("str1.compareTo(str3) : "+str1.compareTo(str4));  
    }  
}
```

```
str1.equals(str2): true  
str1==str2 : true  
str1==str3 : false  
str1.equalsIgnoreCase(str3) : true  
str1.compareTo(str2) : 0  
str1.compareTo(str3) : 32
```

String Pool

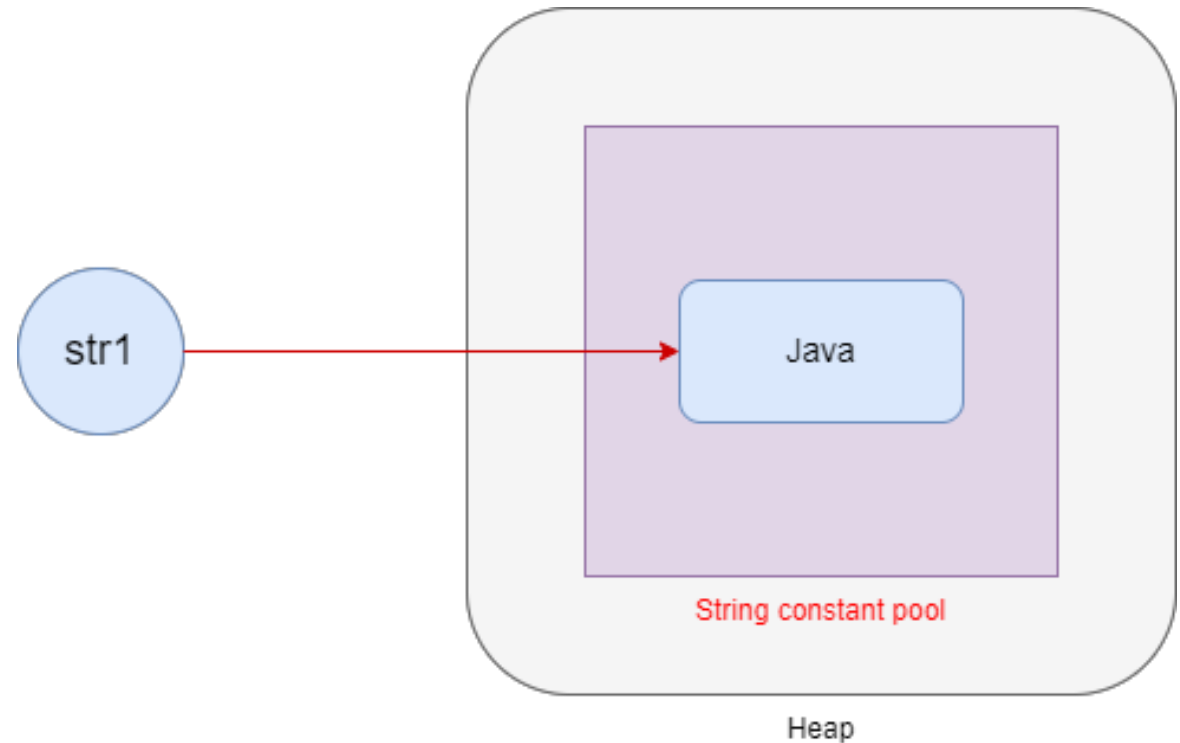
- String str1 = "Cat";
String str2 = "Cat";
String str3 = new String(str1);
- So, the question here is why **str1==str2 is true** while **str1==str3 is false**
- **This is because String Pool in java Heap memory (part of JVM)**
- **String pool** helps in saving a lot of space for Java Runtime although it takes more time to create the String. When we use double quotes to create a String, it first looks for a String with the same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.
- However, using *new* operator, we force String class to create a new String object in heap space.

String Pool



Java Strings are immutable

- The string is immutable means once we created a string variable we cannot modify it.
- Objects in Java are created in heap memory. So String objects are also created in the heap. But here the thing is, there is a special memory area to store strings in Java, known as **String Constant Pool**. Assume we create a String variable as below.
- `String str1 = "Java";`
- This is how it is stored inside the memory.

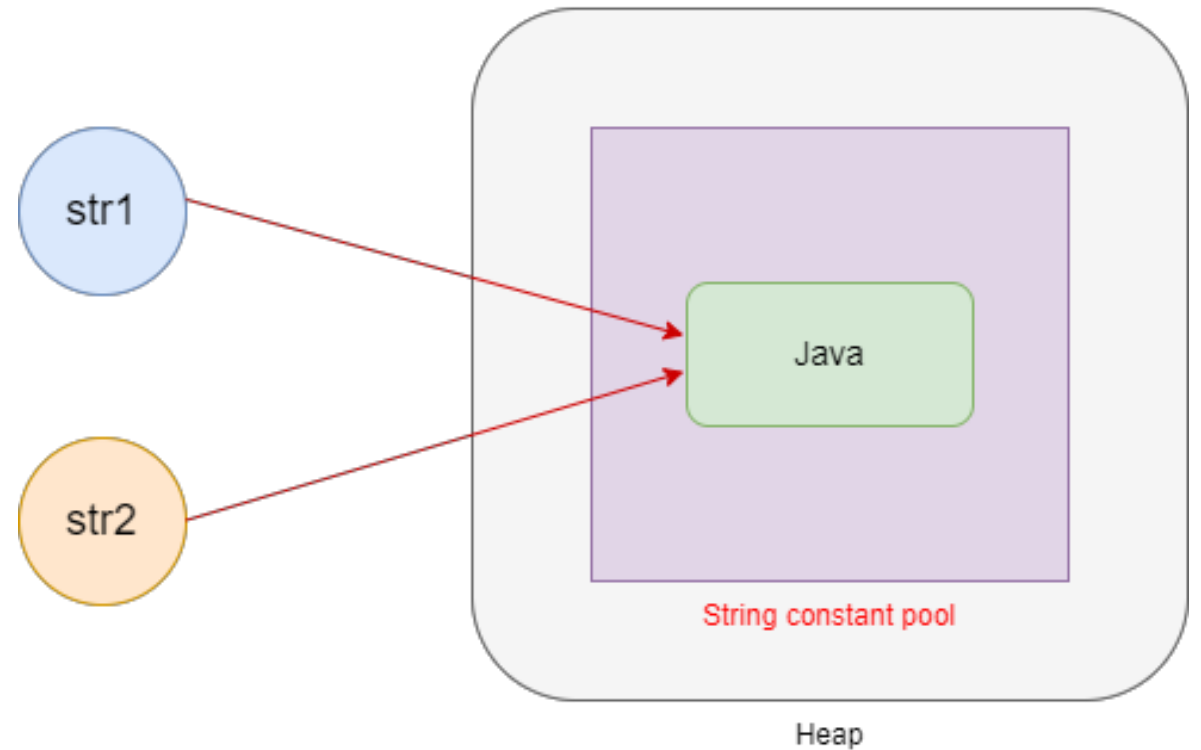


Java Strings are immutable

- References are stored in a memory area called stack. So “str ”will be placed in the stack and “Java” will be stored in the string constant pool. And str1 refers to the string object Java.
- Now, what will happen if we add the str2 variable with the same value “Java”. Will it create another object as “Java” to refer to the str2?
- `String str1 = "Java";`
- `String str2 = "Java";`

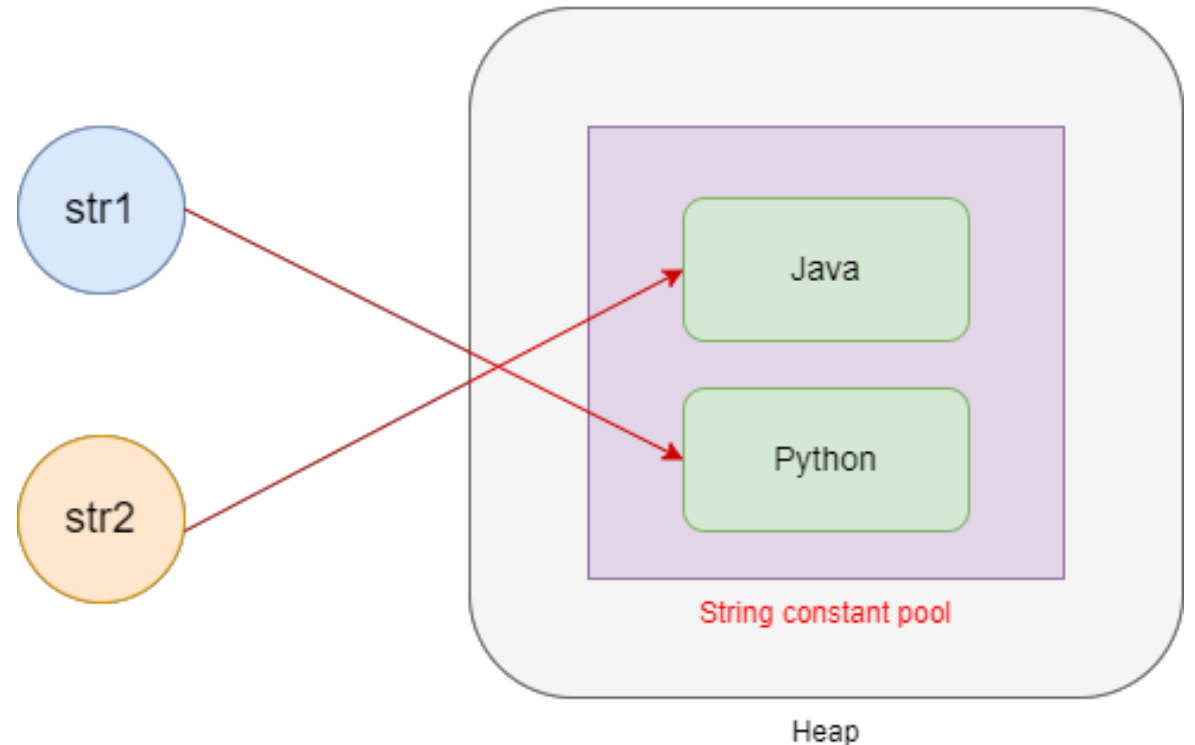
Java Strings are immutable

- Not actually. str2 will also be referring to the same string object “**Java**” inside the heap memory.
- No matter how many variables are created with the same string object that is already contained in the string constant pool, it will always refer to the same String object.



Java Strings are immutable

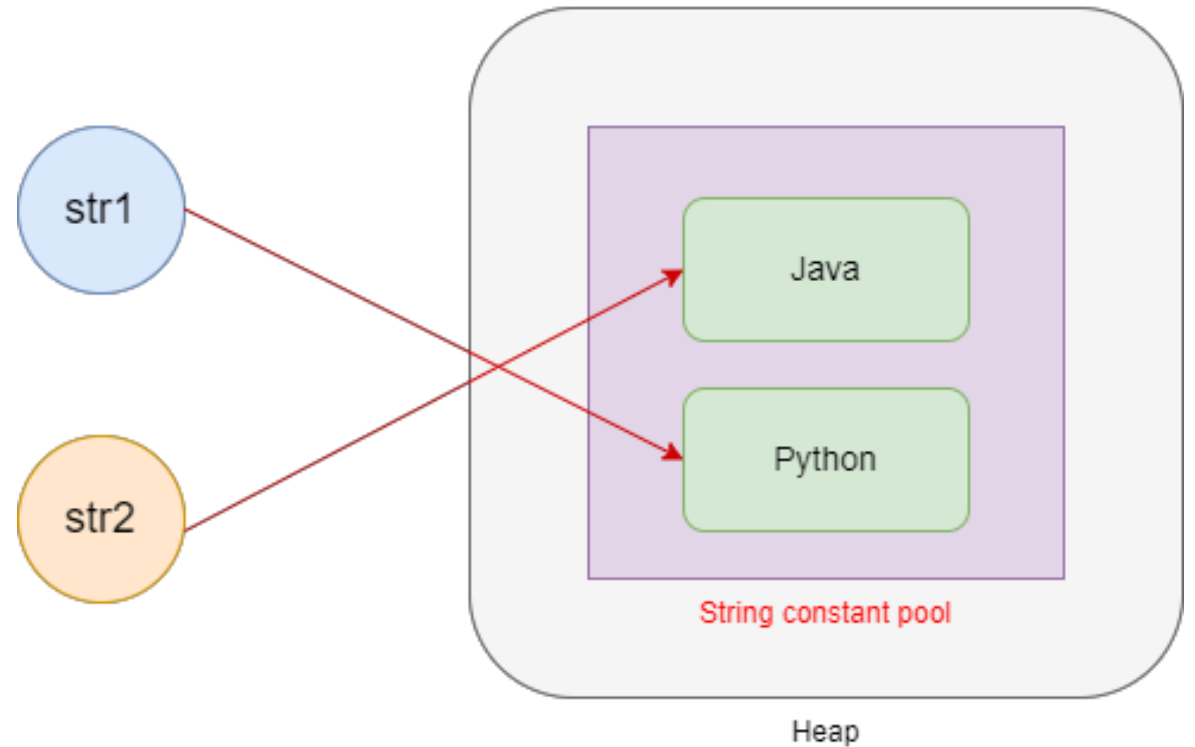
- Then how about this code
- ```
String str1 = "Java";
String str2 = "Java";
Str1 = "Python";
```
- A new String object “**Python**” will be created and **str1** will be referred to it. **str2** is still referring to “**Java**”.





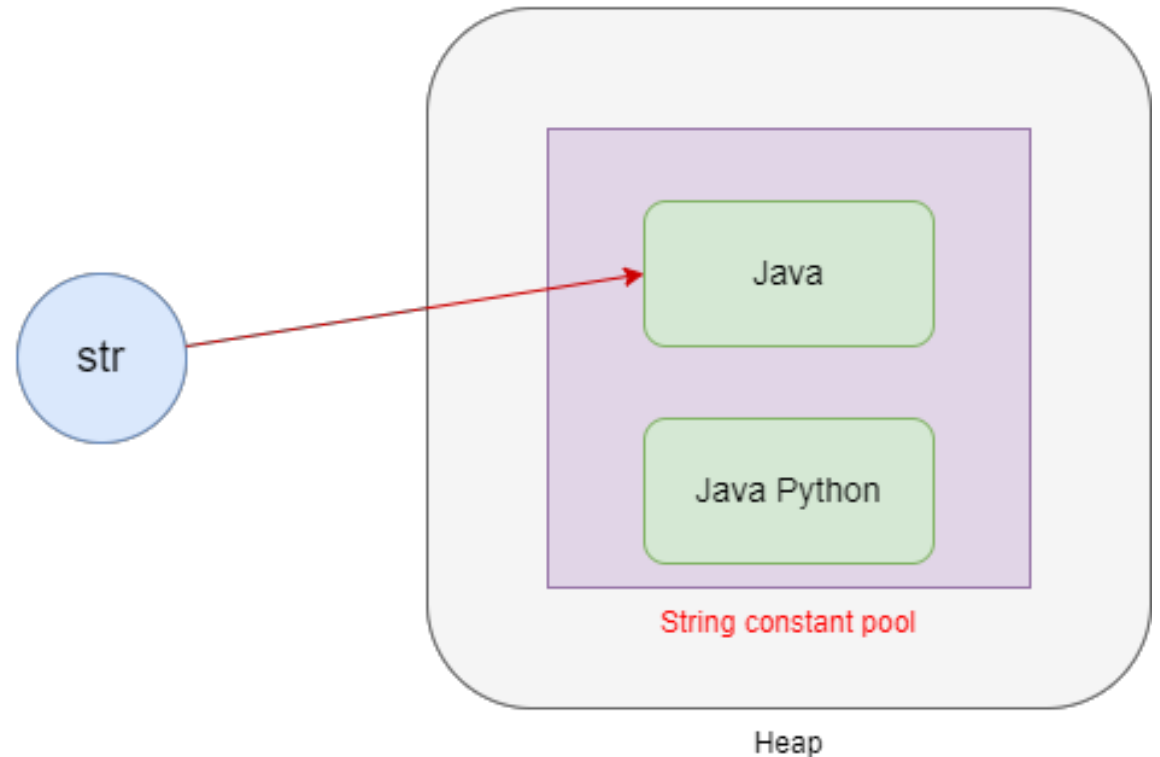
# Java Strings are immutable

- Just think about this. The string “**Java**” was not replaced by the new string “**Python**”. So, here it is changing the reference, not the actual object. That’s why strings are immutable. Once we created a string object, we cannot change it.



# Java Strings are immutable

- Then how about this code
- ```
String str = "Java";  
str.concat(" Python")  
System.out.println(str);
```
- A new string object "**Java Python**" will be created in the string constant pool. But Nothing has happened to the "Java" string object and still **str** is referring to that. So the output will be "**Java**" not "**Java Python**".



Java Strings are immutable

- We can have **mutable** strings with the aid of **StringBuffer** and **StringBuilder** in Java.
- The biggest benefit is that the same string object will be shared for multiple references. This allows for saving a lot of memory which makes efficient memory usage.

StringBuffer

```
public class Main {  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("Hello ");  
        sb.append("Java"); // now original string is changed  
        System.out.println(sb);    }  
}
```

- The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

StringBuilder

```
public class Main {  
    public static void main(String[] args) {  
        StringBuilder str = new StringBuilder(" Hello ");  
        str.append("Java"); // now original string is changed  
        System.out.println(str);    }  
}
```

- The StringBuilder class in java is same as String class except it is mutable i.e. it can be changed.

StringBuffer VS StringBuilder

StringBuffer	StringBuilder
Thread Safe	Not Thread Safe
Synchronized	Not Synchronized
Since Java 1.0	Since Java 1.5
Slower	Faster

Operators

- **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

Operators

- **Relational operators**

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

Operators

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:

- 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.

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

Augmented Assignment Operators

<i>Operator</i>	<i>Name</i>	<i>Example</i>	<i>Equivalent</i>
<code>+=</code>	Addition assignment	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	Subtraction assignment	<code>i -= 8</code>	<code>i = i - 8</code>
<code>*=</code>	Multiplication assignment	<code>i *= 8</code>	<code>i = i * 8</code>
<code>/=</code>	Division assignment	<code>i /= 8</code>	<code>i = i / 8</code>
<code>%=</code>	Remainder assignment	<code>i %= 8</code>	<code>i = i % 8</code>

Increment and Decrement Operators

<i>Operator</i>	<i>Name</i>	<i>Description</i>	<i>Example (assume i = 1)</i>
++var	preincrement	Increment var by 1 , and use the new var value in the statement	int j = ++i; // j is 2, i is 2
var++	postincrement	Increment var by 1 , but use the original var value in the statement	int j = i++; // j is 1, i is 2
--var	predecrement	Decrement var by 1 , and use the new var value in the statement	int j = --i; // j is 0, i is 0
var--	postdecrement	Decrement var by 1 , and use the original var value in the statement	int j = i--; // j is 1, i is 0

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**.

```
if (condition)
{
    statements
}

if (condition) { statements1 }
else { statements2 }
```

- 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;
}
```

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

Conditional processing

```
if (number % 2 == 0)
    even = true;
else
    even = false;
```

(a)

Equivalent

```
boolean even
    = number % 2 == 0;
```

(b)

```
if (even == true)
    System.out.println(
        "It is even.");
```

(a)

Equivalent

```
if (even)
    System.out.println(
        "It is even.");
```

(b)

Multiple Alternative if Statements

```
if (score >= 90.0)
    System.out.print("A");
else
    if (score >= 80.0)
        System.out.print("B");
    else
        if (score >= 70.0)
            System.out.print("C");
        else
            if (score >= 60.0)
                System.out.print("D");
            else
                System.out.print("F");
```

(a)

Equivalent

```
if (score >= 90.0)
    System.out.print("A");
else if (score >= 80.0)
    System.out.print("B");
else if (score >= 70.0)
    System.out.print("C");
else if (score >= 60.0)
    System.out.print("D");
else
    System.out.print("F");
```

This is better

(b)

Multiple Alternative if Statements

The else clause matches the most recent if clause in the same block.

```
int i = 1, j = 2, k = 3;

if (i > j)
    if (i > k)
        System.out.println("A");
else
    System.out.println("B");
```

(a)

Equivalent

This is better
with correct
indentation

```
int i = 1, j = 2, k = 3;

if (i > j)
    if (i > k)
        System.out.println("A");
    else
        System.out.println("B");
```

(b)

Conditional Operators

```
if (x > 0)
```

```
    y = 1
```

```
else
```

```
    y = -1;
```

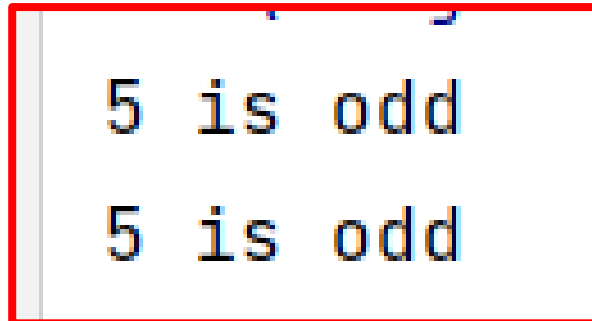
is equivalent to

```
y = (x > 0) ? 1 : -1;
```

(boolean-expression) ? expression1 : expression2

Conditional Operator

```
public class Main {  
    public static void main(String[] args) {  
        int num=5;  
        if (num % 2 == 0)  
            System.out.println(num + " is even");  
        else  
            System.out.println(num + " is odd");  
        System.out.println((num % 2 == 0)? num + " is even" : num + " is odd");  
    }  
}
```



```
5 is odd  
5 is odd
```

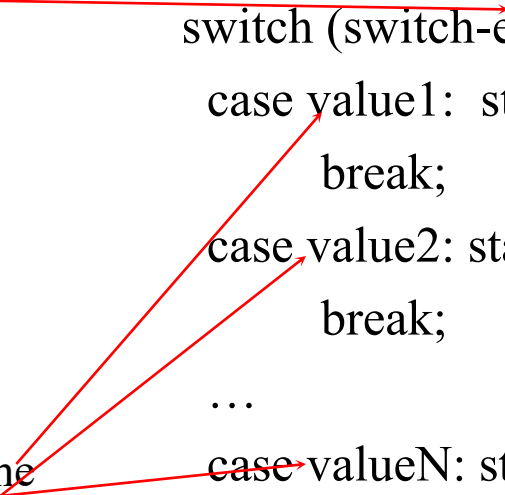
switch Statements

```
switch (status) {  
    case 0: stmat;  
        break;  
    case 1: stmat;  
        break;  
    case 2: stmat;  
        break;  
    case 3: stmat;  
        break;  
    default: System.out.println("Errors: invalid status");  
        System.exit(1);  
}
```

switch Statement Rules

The switch expression must yield a value of char, byte, short, or int type. It also works with enumerated types (discussed in Enum Types), the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer and must always be enclosed in parentheses.

```
switch (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-default;  
}
```



The value1, ..., and valueN must have the same data type as the value of the switch expression. The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch expression. Note that value1, ..., and valueN are constant expressions, meaning that they cannot contain variables in the expression, such as $1 + \underline{x}$.

switch Statement Rules

The keyword break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement. If the break statement is not present, the next case statement will be executed.

The default case, which is optional, can be used to perform actions when none of the specified cases matches the switch expression.

```
switch (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-default;  
}
```

When the value in a **case** statement matches the value of the **switch-expression**, the statements *starting from this case* are executed until either a **break** statement or the end of the **switch** statement is reached.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        System.out.println("Enter Daya : ");
        Scanner inp=new Scanner(System.in);
        int day= inp.nextInt();
        switch (day) {
            case 1: System.out.println("1");
            case 2: System.out.println("2");
            case 3: System.out.println("3");
            case 4: System.out.println("4");
            case 5: System.out.println("Weekday"); break;
            case 0: System.out.println("0");
            case 6: System.out.println("Weekend");
        }
    }
}
```

Example

Enter Daya :

2

2

3

4

Weekday

Enter Daya :

0

0

Weekend

Enter Daya :

5

Weekday

Repetitive processing

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

```
while (condition)
{
    statements
}
```

Example:

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

Repetitive processing

```
for (initialization; condition; update)
{
    statements
}
```

Examples:

```
for (int i=2*start; i<=endvalue; i=i+2)
{
    System.out.println(i);
}
```

↓ Can you figure out how
this loop is executed??

Example

```
public class Main {  
    public static void main(String[] args) {  
        int sum = 0;  
        int n = 1000;  
        for (int i = 1; i <= n; ++i) {  
            sum += i;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

Sum = 500500

Repetitive processing

- The initial action in a for loop can be a list of zero or more comma-separated expressions.
- The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements.
- But the control statement in for loop is only one statement
- Therefore, the following two for loops are correct. They are rarely used in practice, however.
- `for (int i = 1; i < 100; System.out.println(i++));`
- `for (int i = 0, j = 0; (i + j < 10); i++, j++) {`
`// Do something}`

```
public class Main {  
    public static void main(String[] args) {  
        for(int i=1;i<10;System.out.println(i++));  
    }  
}
```

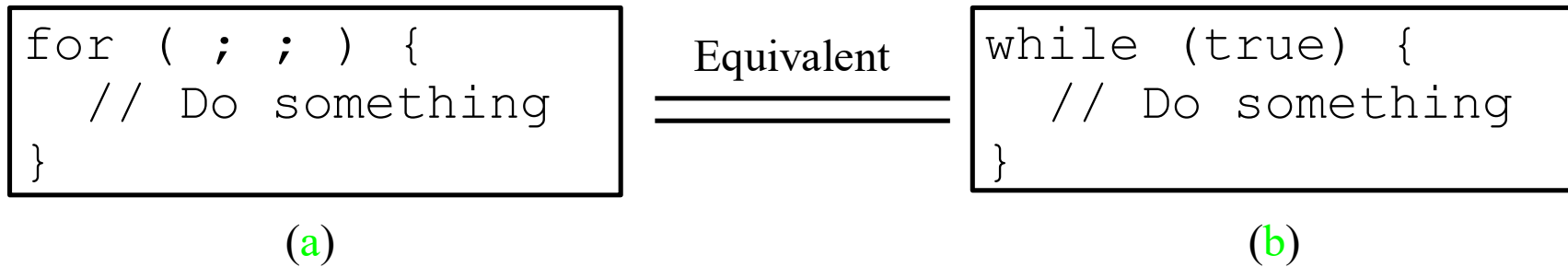
1
2
3
4
5
6
7
8
9

```
public class Main {  
    public static void main(String[] args) {  
        for(int i=0,j=0;(i+j<5);i++,j++){  
            System.out.println("i= "+i+" and j= "+j);  
        }  
    }  
}
```

i= 0 and j= 0
i= 1 and j= 1
i= 2 and j= 2

Repetitive processing

- If the loop-continuation condition in a for loop is omitted, it is implicitly true.
- Thus the statement given below in (a), which is an infinite loop, is correct. Nevertheless, it is better to use the equivalent loop in (b) to avoid confusion:



```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        int sum = 0;
        Scanner input = new Scanner(System.in);
        System.out.println("Enter a number");
        int number = input.nextInt();
        while (number >= 0) {
            sum += number;
            System.out.println("Enter a number");
            number = input.nextInt();
        }
        System.out.println("Sum = " + sum);
        input.close();
    }
}
```

Example

```
Enter a number
25
Enter a number
9
Enter a number
5
Enter a number
-3
Sum = 39
```

Example

```
public class Main {  
    public static void main(String[] args) {  
        int i = 1, n = 5;  
        do {  
            System.out.println(i);  
            i++;  
        } while(i <= n);  
    }  
}
```



1
2
3
4
5

Programming Errors

Syntax Errors

- Detected by the compiler

Runtime Errors

- Causes the program to abort

Logic Errors

- Produces incorrect result

Syntax Errors

```
public class ShowSyntaxErrors {  
    public static main(String[] args) {  
        System.out.println("Welcome to Java");  
    }  
}
```

//miss the right " double quotation

Runtime Errors

```
public class ShowRuntimeErrors {  
    public static void main(String[] args) {  
        System.out.println(1 / 0);  
    }  
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Main.main(Main.java:3)

Division on zero

Logic Errors

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Celsius 35 is Fahrenheit degree ");  
        System.out.println((9 / 5) * 35 + 32);  
    }  
}
```

```
Celsius 35 is Fahrenheit degree  
67
```

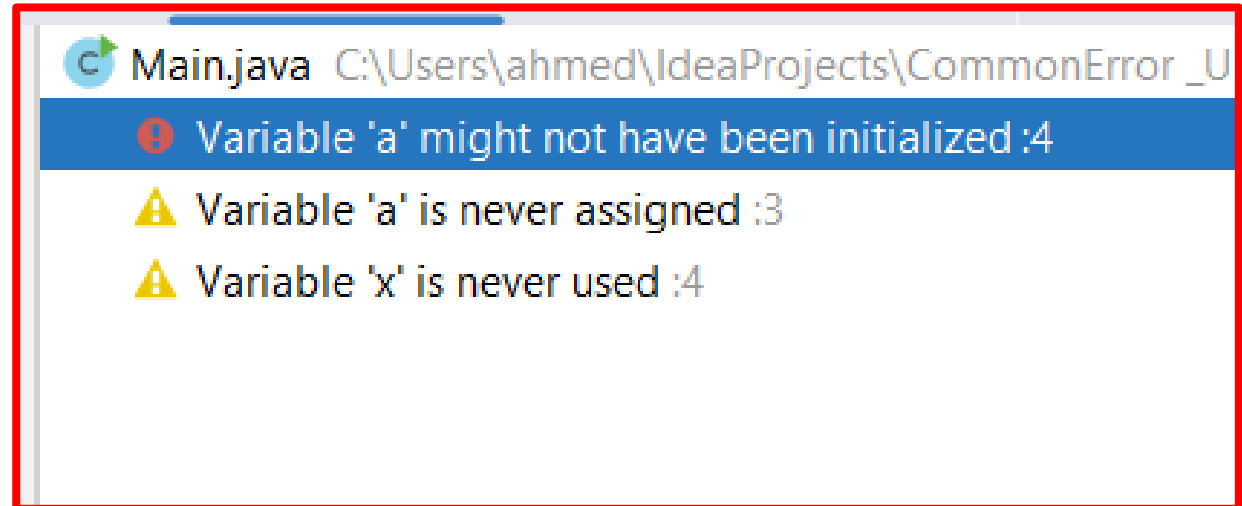
Integer division will give wrong total value

If replaced with → `System.out.println((9.0 / 5.0) * 35 + 32);`

```
Celsius 35 is Fahrenheit degree  
95.0
```

Common Error : Undeclared/Uninitialized Variables and Unused Variables

```
public class Main {  
    public static void main(String[] args) {  
        double a;  
        double x= 6*a;  
    }  
}
```



Common Error : Integer Overflow

```
int value = 2147483647 + 1;
```

```
// value will actually be -2147483648
```

Int is 4 byte so max number can be represented by
int is $[(2^{31}) - 1] = \mathbf{2147483647}$

Common Error : Round-off Errors

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);  
        System.out.println(1.0 - 0.9);  
    }  
}
```

0.500000000000000001

0.099999999999999998

Common Error : Round-off Errors

- A round-off error, also called a rounding error, is the difference between the calculated approximation of a number and its exact mathematical value.
- For example, $1/3$ is approximately 0.333 if you keep three decimal places, and is 0.3333333 if you keep seven decimal places.
- Since the number of digits that can be stored in a variable is limited, round-off errors are inevitable.
- Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy.

Common Error : Unintended Integer Division

```
int number1 = 1;  
int number2 = 2;  
double average = (number1 + number2) / 2;  
System.out.println(average);
```

(a)

```
int number1 = 1;  
int number2 = 2;  
double average = (number1 + number2) / 2.0;  
System.out.println(average);
```

(b)

- Java uses the same divide operator, namely /, to perform both integer and floating-point division.
- When two operands are integers, the / operator performs an integer division. The result of the operation is an integer. The fractional part is dropped.
- To force two integers to perform a floating-point division, make one of the integers into a floating-point number.

Common Errors

- Adding a semicolon at the end of an if clause is a common mistake.

```
if (radius >= 0); ← Wrong
{
    area = radius*radius*PI;
    System.out.println("The area for the circle of radius " + radius + " is " + area);
}
```

- This mistake is hard to find, because it is not a compilation error or a runtime error, it is a logic error.
- This error often occurs when you use the next-line block style.

Common Errors

```
public class Main {  
    public static void main(String[] args) {  
        double radius=-1,area,PI= 3.14;  
        if (radius >= 0);  
        {  
            area = radius*radius*PI;  
            System.out.println("The area for the circle of radius " + radius + " is " + area);  
        }  
    }  
}
```



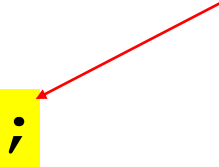
```
The area for the circle of radius -1.0 is 3.14
```


Common Errors

Adding a semicolon at the end of the for clause before the loop body is a common mistake, as shown below (similar to previous example:

```
for (int i=0; i<10; i++) ;  
{  
    System.out.println("i is " + i);  
}
```

Logic Error



Common Errors

```
public class Main {  
    public static void main(String[] args) {  
        int i=0;  
        for (i=0; i<10; i++);  
        {  
            System.out.println("i is " + i);  
        }  
    }  
}
```



i is 10

```
public class Main {  
    public static void main(String[] args) {  
        for (int i=0; i<10; i++);  
        {  
            System.out.println("i is " + i);  
        }  
    }  
}
```



What is the Output from this code?

Common Errors

- Similarly, the following loop is also wrong:

```
int i=0;
while (i < 10);
{
    System.out.println("i is " + i);
    i++;
}
```

Logic Error

What is the Output from this code?

- In the case of the do loop, the following semicolon is needed to end the loop.

```
int i=0;
do {
    System.out.println("i is " + i);
    i++;
} while (i<10);
```

Correct

Common Pitfall : Redundant Input Objects

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

```
System.out.print("Enter an integer: ");
```

```
int v1 = input.nextInt();
```

```
Scanner input1 = new Scanner(System.in);
```

```
System.out.print("Enter a double value: ");
```

```
double v2 = input1.nextDouble();
```

The code is not good. It creates two input objects unnecessarily and may lead to some errors.

Thanks



MCQ Questions

- Solve Ch 1,2,3,4,5
- https://media.pearsoncmg.com/ph/esm/ecs_liang_iip_12/cw/#selftest

References

- Java: How To Program, Early Objects, 11th edition
- Cay S. Horstmann, Big Java: Late Objects
- Setup IntelliJ IDEA (2021) for JavaFX & SceneBuilder and Create Your First JavaFX Application – YouTube
- Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang
- Java documentation <https://docs.oracle.com/javase>
- <https://www.youtube.com/watch?v=LTgClBqDank&feature=youtu.be>
- <https://medium.com/java-for-beginners/understanding-java-virtual-machine-jvm-architecture-e68d1c611026>
- <https://medium.com/@mohamedathan3/architecture-of-java-virtual-machine-c628c6b034f4>

References

- <https://medium.com/javarevisited/how-java-code-compiled-and-run-e4702fb83ffa>
- <https://simplesnippets.tech/execution-process-of-java-program-in-detail-working-of-just-it-time-compiler-jit-in-detail/>
- [JVM Architecture - MindScripts Tech](#)
- [compilation - Is Java a Compiled or an Interpreted programming language ? - Stack Overflow](#)
- [What is Java String Pool? | DigitalOcean](#)
- <https://blog.devgenius.io/java-string-is-immutable-what-does-it-actually-mean-6c7e9194a007>
- [StringBuffer class in Java – GeeksforGeeks](#)
- [StringBuilder Class in Java with Examples – GeeksforGeeks](#)
- [String vs StringBuffer vs StringBuilder | DigitalOcean](#)