

Java Variables, Statements, Loops

Dr. Krishnendu Guha

Assistant Professor/ Lecturer

School of Computer Science and Information Technology

University College Cork

Email: kguha@ucc.ie

Conditions (Recap)

The Java if statement:

```
if (condition1) {  
    statement1  
}  
  
else if (condition2) {  
    statement2  
}  
  
else {  
    statement3  
}
```

boolean variable: `true`, `false`, `isOver21`

- method with boolean result: `exists()`, `isSent(email)`
- operand-comparison operator-operand: age `>= 5`, speed `== 0`, `firstLetter == 'a'`
- boolean expressions combined with logical operator: `x < 5 || x > 10`, `!exists()`

Condition is true

```
int number = 5;  
  
if (number > 0) {  
    // code  
}  
  
else {  
    // code  
}  
  
// code after if...else
```

Condition is false

```
int number = 5;  
  
if (number < 0) {  
    // code  
}  
  
else {  
    // code  
}  
  
// code after if...else
```

If/ else if/ else (Recap)

Python

```
if x < y:  
    print('x is smaller than y')  
elif x == y:  
    print('x is equal to y')  
else:  
    print('x is larger than y')
```

Java

```
if (x < y) {  
    System.out.println("x is smaller than y");  
}  
else if (x == y) {  
    System.out.println("x is equal to y");  
}  
else {  
    System.out.println("x is larger than y");  
}
```

Statements

- **Declaration statements**

```
int x;
```

- **Expression statements:** assignment, calling a method,..., e.g.:

```
x = 3;
```

```
double y = 12.3;
```

```
x = x + 1;
```

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

```
System.out.println(x);
```

```
String z = sc.nextLine();
```

```
System.out.println(z);
```

- **Control flow statements:** if, while, for, break,...
- In Java, **most statements are terminated by semi-colons**

Block

A block in Java is a **group of one or more statements enclosed in *braces***.

- A block begins with an opening brace ({) and ends with a closing brace (}) - curly brackets.
- In some cases, you can omit the braces - you probably should not!
- Indentation does not change the functionality.
- No semi-colons after the closing brace

While loops

```
while (condition) {  
    // code block to be executed  
}
```

The **do/while** loop is a variant of the while loop.

This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

```
do {  
    // code block to be executed  
}  
while (condition);
```

While- comparison to Python

```
while x < y:  
    x = x + 1  
print(x)
```

```
while(x < y) {  
    x = x + 1;  
    System.out.println(x);  
}
```

Question: What are the main differences?

- the round parentheses for the Boolean expression;
- the curly braces for the blocks
- semicolons

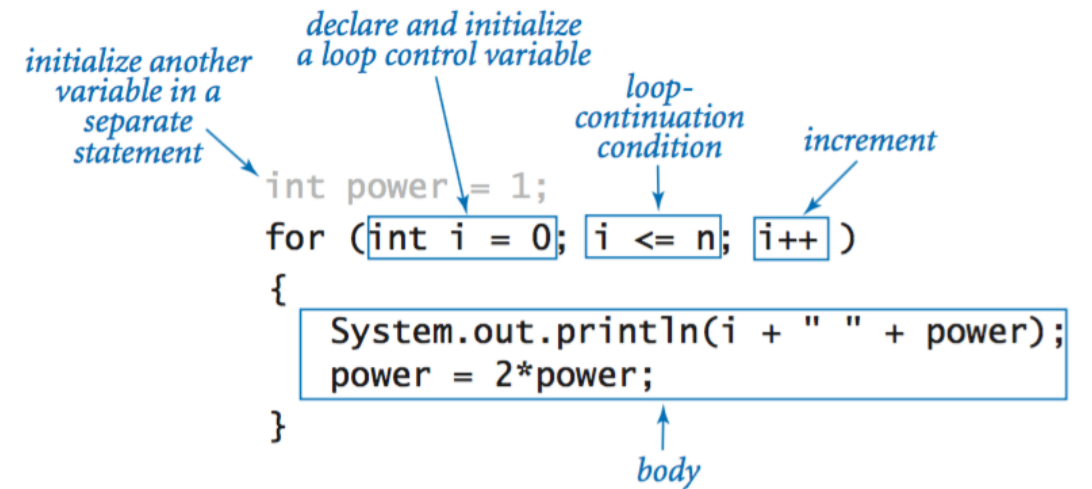
For loops

```
for (statement 1; statement 2; statement 3) {  
    // code block to be executed  
}
```

There is also a for-each loop, which is used exclusively to loop through elements in an array

We will see Arrays later.

```
for (type variableName : arrayName) {  
    // code block to be executed  
}
```



The diagram illustrates the components of a for loop with the following code and annotations:

```
int power = 1;  
for (int i = 0; i <= n; i++)  
{  
    System.out.println(i + " " + power);  
    power = 2*power;  
}
```

Annotations:

- initialize another variable in a separate statement* points to `int power = 1;`
- declare and initialize a loop control variable* points to `int i = 0` in the for loop header.
- loop-continuation condition* points to `i <= n` in the for loop header.
- increment* points to `i++` in the for loop header.
- body* points to the code block inside the for loop: `System.out.println(i + " " + power); power = 2*power;`

For comparison to Python

```
for i in range(10):  
    print(i)
```

```
for (int i = 0; i < 10; i = i + 1) {  
    System.out.println(i);  
}
```

- **variable i** is referred to as the **loop variable** (or **loop counter**)
- the expression **int i = 0** **declares and initialises the loop variable**
- the expression **i < 10** is called the **loop test** (or **loop condition**)
- the **block** is called the *loop body*

Other control flow statements

Ones that are common to Java and Python

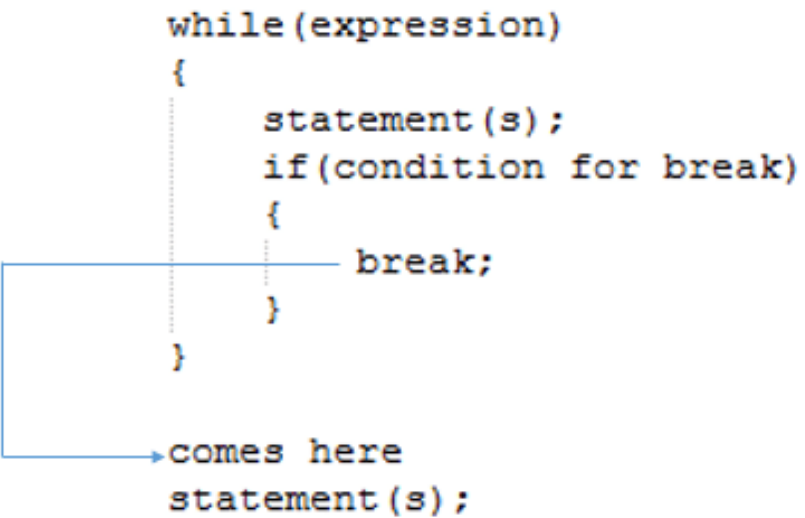
- `break`, `return`, `import`
- `try...except...else...finally` (Python), `try...catch...finally` (Java)
- `continue`
- `raise` (Python), `throw`

- Ones that **Java has and Python doesn't**: `switch`, `do...while`
- Ones that **Python has that Java doesn't**: `pass`, `yield`,...

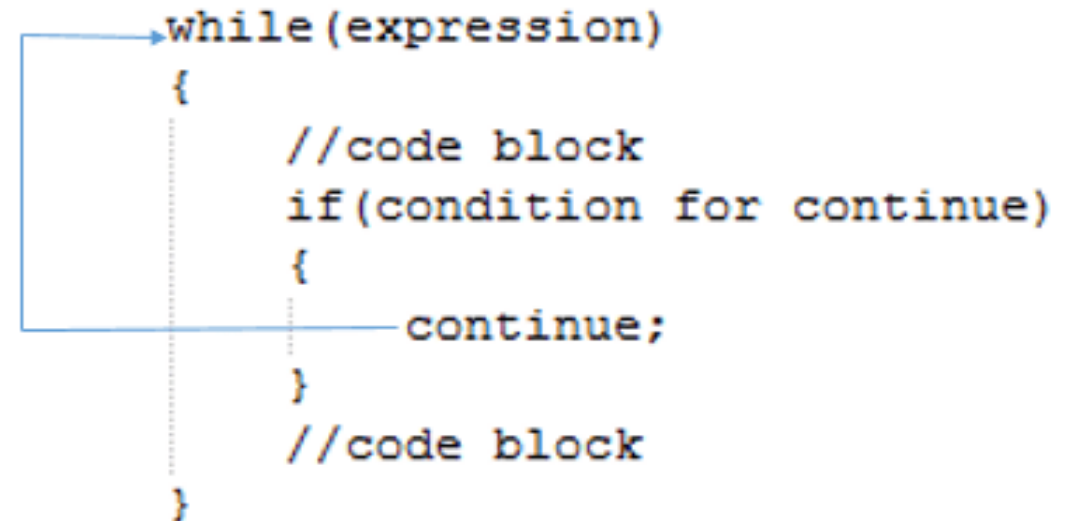
While, break, continue

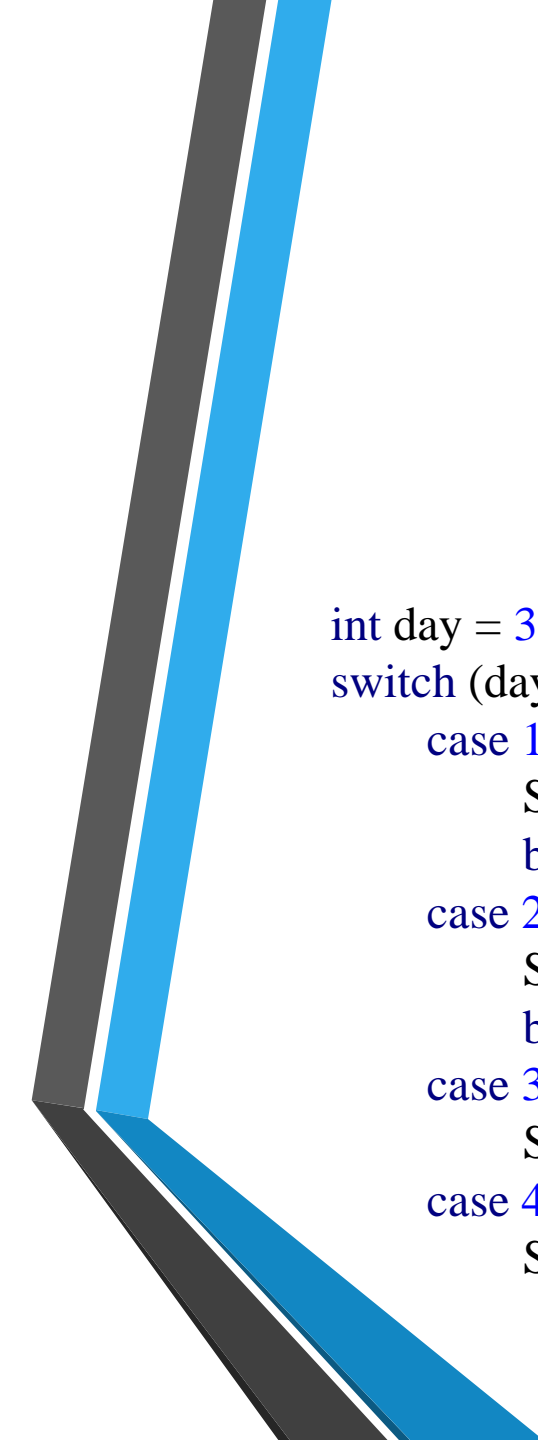
```
while(expression)
{
    statement(s);
    if(condition for break)
    {
        break;
    }
}
```

comes here
statement(s);



```
while(expression)
{
    //code block
    if(condition for continue)
    {
        continue;
    }
    //code block
}
```





```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday"); break;
    case 4:
        System.out.println("Thursday"); break;
```

```
    case 5:
        System.out.println("Friday"); break;
    case 6:
        System.out.println("Saturday"); break;
    case 7:
        System.out.println("Sunday"); break;
    default:
        System.out.println("Invalid"); break;
}
```

Java print

Display Variables

The `println()` method is often used to display variables.

To combine both text and a variable, we use the `+` character:

Example

```
String name = "UCC";  
System.out.println("Hello " + name);
```

You can also use the `+` character to add a variable to another variable:

Example

```
String firstName = "UCC ";  
String lastName = "CSIT";  
String fullName = firstName + lastName;  
System.out.println(fullName);
```

For numeric values, the `+` character works as a mathematical [operator](#) (notice that we use `int` (integer) variables here):

Example

```
int x = 1;
```

```
int y = 2;
```

```
System.out.println(x + y); // Print the value of x + y
```

From the example above, we can expect:

- x stores the value 1
- y stores the value 2
- Then we use the `println()` method to display the value of `x + y`, which is **3**

Declaring Multiple Variables

Declare Many Variables

To declare more than one variable of the **same type**, you can use a comma-separated list:

Example

Instead of writing:

```
int x = 1;  
int y = 2;  
int z = 3;  
System.out.println(x + y + z);
```

We can simply write:

```
int x = 5, y = 6, z = 50;  
System.out.println(x + y + z);
```

One Value to Multiple Variables

You can also assign the **same value** to multiple variables in one line:

Example

```
int x, y, z;  
x = y = z = 50;  
System.out.println(x + y + z);
```

Java Identifiers

What are Identifiers?

All Java **variables** must be **identified** with **unique names**.

These **unique names** are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

Example

// Good

```
int minutesPerHour = 60;
```

// OK, but not so easy to understand what **m** actually is

```
int m = 60;
```

The general rules for naming variables are:

- Names can contain **letters, digits, underscores, and dollar signs**
- Names must **begin with a letter**
- Names should **start with a lowercase letter** and it **cannot contain whitespace**
- Names can also **begin with \$ and _**
- Names are **case sensitive** ("myVar" and "myvar" are different variables)
- Reserved words (like **Java keywords**, such as **int** or **boolean**) cannot be used as names

JAVA Data Types

A [variable](#) in Java must be a specified data type:

Example

```
int myNum = 5; // Integer (whole number)
```

```
float myFloatNum = 5.99f; // Floating point number
```

```
char myLetter = 'D'; // Character Data Type
```

```
boolean myBool = true; // Boolean Data Type
```

```
String myText = "Hello"; // String Data Type
```

Data types are divided into two groups:

- **Primitive data types** - includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`
- **Non-primitive data types** - such as [String](#), [Arrays](#) and [Classes](#)

Primitive Data Types

A primitive data type **specifies the size and type of variable values**, and it has no additional methods. There are eight primitive data types in Java:

<u>Data Type</u>	<u>Size</u>	<u>Description</u>
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



Numbers

Primitive number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals
Valid types are **byte**, **short**, **int** and **long**. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. There are two types: **float** and **double**.

Even though there are many numeric types in Java, the most used for numbers are
int (for whole numbers) and
double (for floating point numbers)

Integer Types

Byte

The **byte** data type can store whole numbers from -128 to 127

This can be used instead of **int** or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example

```
byte myNum = 100;  
System.out.println(myNum);
```

Short

The **short** data type can store whole numbers from -32768 to 32767:

Example

```
short myNum = 5000;  
System.out.println(myNum);
```

Int

The **int** data type can store whole numbers from -2147483648 to 2147483647

In general, the **int** data type is the **preferred data type** when we create variables with a numeric value.

Example

```
int myNum = 100000;  
System.out.println(myNum);
```

Long

The **long** data type can store whole numbers from -9223372036854775808 to 9223372036854775807
This is used **when int is not large enough** to store the value
Note that you should end the value with an "L":

Example

```
long myNum = 15000000000L;  
System.out.println(myNum);
```

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.
The **float** and **double** data types **can store fractional numbers**.
Note that you should end the value with an "f" for floats and "d" for doubles:

Float Example

```
float myNum = 5.75f;  
System.out.println(myNum);
```

Double Example

```
double myNum = 19.99d;  
System.out.println(myNum);
```

Use **float** or **double**?

The **precision** of a floating point value **indicates how many digits the value can have after the decimal point**.

The precision of **float** is only **six or seven decimal digits**, while **double** variables have a precision of about **15 digits**.

Therefore, it is safer to use **double** for most calculations.

Scientific Numbers

A floating point number can also be a **scientific number with an "e" to indicate the power of 10**

Example

```
float f1 = 35e3f;
```

```
double d1 = 12E4d;
```

```
System.out.println(f1);
```

```
System.out.println(d1);
```

Boolean Data Types

Very often in programming, you will need a data type that **can only have one of two values**, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, Java has a **boolean** data type, which can only take the values **true** or **false**:

Example

```
boolean isJavaFun = true;  
boolean isHotNice = false;  
System.out.println(isJavaFun); // Outputs true  
System.out.println(isHotNice); // Outputs false
```

Boolean values are mostly used for conditional testing.

Characters

The **char** data type is used to store a **single** character

The character must be surrounded by single quotes, like 'A' or 'c':

Example

```
char myGrade = 'A';  
System.out.println(myGrade);
```

Alternatively, you can ASCII values to display certain characters:

Example

```
char myVar1 = 65, myVar2 = 66, myVar3 = 67;  
System.out.println(myVar1); System.out.println(myVar2); System.out.println(myVar3);
```

Strings

The **String** data type is used to **store a sequence of characters (text)**

String values must be surrounded by double quotes:

Example

```
String greeting = "Hello World";  
System.out.println(greeting);
```

The String type is so much used and integrated in Java, that some call it "the special **ninth** type".

A String in Java is actually a **non-primitive** data type, because it refers to an object.

The String object has methods that are used to perform certain operations on strings.

JAVA Non primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- **Primitive types** are **predefined (already defined)** in Java.
- **Non-primitive types** are **created by the programmer** and is **not defined by Java** (except for **String**).
- **Non-primitive types** can be **used to call methods to perform certain operations**, while primitive types cannot.
- A **primitive type** has **always a value**, while **non-primitive types** can be **null**.
- A **primitive type** starts with a **lowercase letter**, while **non-primitive types** starts with an **uppercase letter**.
- The **size** of a **primitive type** depends on the data type, while **non-primitive types** have all the same size.

Examples of non-primitive types are [Strings](#), [Arrays](#), [Classes](#), [Interface](#), etc.

Java Type Casting

What is Type Casting?

Type casting is when you **assign a value of one primitive data type to another type**.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
`byte -> short -> char -> int -> long -> float -> double`

Narrowing Casting (manually) - converting a larger type to a smaller size type
`double -> float -> long -> int -> char -> short -> byte`

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example

```
public class Main { public static void main(String[] args)
{
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double
    System.out.println(myInt); // Outputs 9
    System.out.println(myDouble); // Outputs 9.0
}
}
```

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

Example

```
public class Main { public static void main(String[] args)
{
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int
    System.out.println(myDouble); // Outputs 9.78
    System.out.println(myInt); // Outputs 9
}
}
```

Java Operators

Operators are used to perform operations on variables and values.

In the example below, we use the **+** **operator** to add together two values:

Example

```
int x = 100 + 50;
```

Although the **+** operator is often used to add together two values, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1 = 100 + 50; // 150 (100 + 50)
```

```
int sum2 = sum1 + 250; // 400 (150 + 250)
```

```
int sum3 = sum2 + sum2; // 800 (400 + 400)
```

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations

<u>Operator</u>	<u>Name</u>	<u>Description</u>	
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$++x$
--	Decrement	Decreases the value of a variable by 1	$--x$

Java Strings

Strings are used for storing text.

A **String** variable contains a collection of characters surrounded by double quotes:

Example

Creating a variable of type **String** and assigning it a value:

```
String greeting = "Hello";
```

String Length

A String in Java is actually an object,

which **contain methods** that can **perform certain operations on strings**.

For example, the length of a string can be found with the **length()** method:

Example

```
String text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
System.out.println("The length of the text string is: " + text.length());
```

More String Methods

There are many string methods available, for example **toUpperCase()** and **toLowerCase()**:

Example

```
String text = "Hello World";  
System.out.println(text.toUpperCase()); // Outputs "HELLO WORLD"  
System.out.println(text.toLowerCase()); // Outputs "hello world"
```

Finding a Character in a String

The `indexOf()` method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

Example

```
String txt = "Please locate where 'locate' occurs!";  
System.out.println(txt.indexOf("locate")); // Outputs 7
```

Java counts positions from zero.

0 is the first position in a string, 1 is the second, 2 is the third ...

JAVA String Concatenation

The `+` operator can be used between strings to combine them
This is called **concatenation**:

Example

```
String firstName = "Java";  
String lastName = "Oracle";  
System.out.println(firstName + " " + lastName);
```

Note that we have added an empty text (" ") to create a space between firstName and lastName on print.

You can also use the `concat()` method to concatenate two strings:

Example

```
String firstName = "Java ";  
String lastName = "Oracle";  
System.out.println(firstName.concat(lastName));
```

Adding Numbers and Strings

WARNING!

Java uses the `+` operator for both **addition** and **concatenation**.

Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

Example

```
int x = 10;  
int y = 20;  
int z = x + y; // z will be 30 (an integer/number)
```

If you add two strings, the result will be a string concatenation:

Example

```
String x = "10";  
String y = "20";  
String z = x + y; // z will be 1020 (a String)
```


Strings - Special Characters

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

```
String txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

The sequence \" inserts a double quote in a string:

Example

```
String txt = "We are the so-called \"Vikings\" from the north.";
```

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

The sequence \' inserts a single quote in a string:

Example

```
String txt = "It\'s alright.";
```

The sequence `\\` inserts a single backslash in a string:

Example

```
String txt = "The character \\ is called backslash.";
```

Other common escape sequences that are valid in Java are:

<u>Code</u>	<u>Result</u>
<code>\n</code>	New Line
<code>\r</code>	Carriage Return
<code>\t</code>	Tab
<code>\b</code>	Backspace
<code>\f</code>	Form Feed



The Java Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.max(x,y)

The **Math.max(x,y)** method can be used to find the highest value of x and y :

Example

```
System.out.println(Math.max(5, 10));
```

Math.min(x,y)

The **Math.min(x,y)** method can be used to find the lowest value of x and y :

Example

```
System.out.println(Math.min(5, 10));
```

Math.sqrt(x)

The **Math.sqrt(x)** method returns the square root of x :

Example

```
System.out.println(Math.sqrt(64));
```

Math.abs(x)

The **Math.abs(x)** method returns the absolute (positive) value of x :

Example

```
System.out.println(Math.abs(-4.7));
```

Random Numbers

Math.random() returns a random number between 0.0 (inclusive), and 1.0 (exclusive):

Example

```
System.out.println(Math.random());
```

To get more control over the random number, for example, **if you only want a random number between 0 and 100**, you can use the following formula:

Example

```
int randomNum = (int)(Math.random() * 101); // 0 to 100
```

Errors

