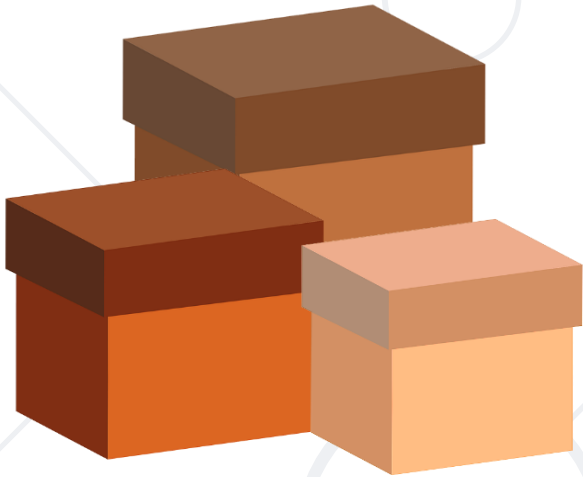


# Data Types and Variables

Numerical Types, Text Types and Type Conversion



**SoftUni Team**  
Technical Trainers



**SoftUni**



Software University

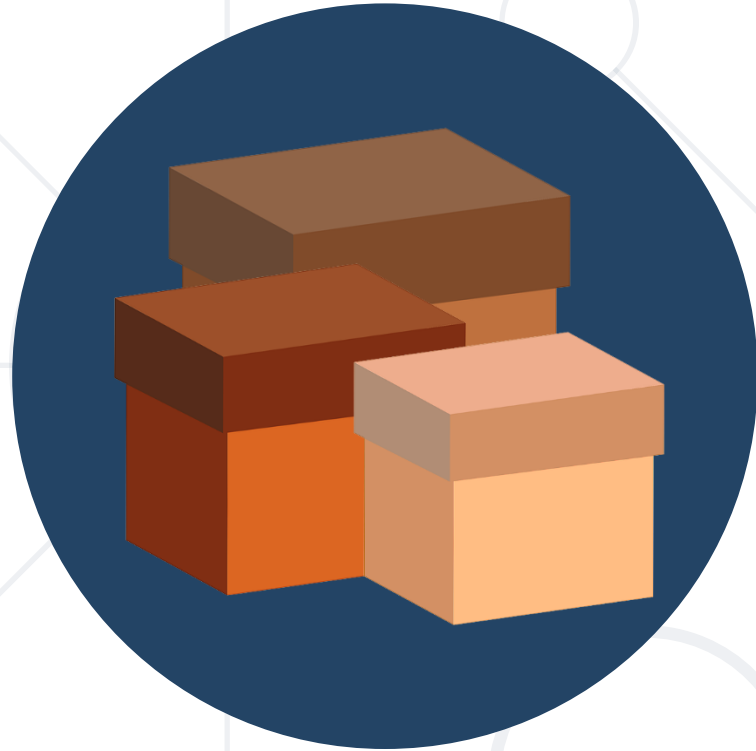
<https://softuni.bg>

[sli.do](https://sli.do)

**#fund-java**

1. **Data Types** and **Variables**
2. **Integer** and **Real Number Type**
3. Type **Conversion**
4. **Boolean** Type
5. **Character** and **String Type**

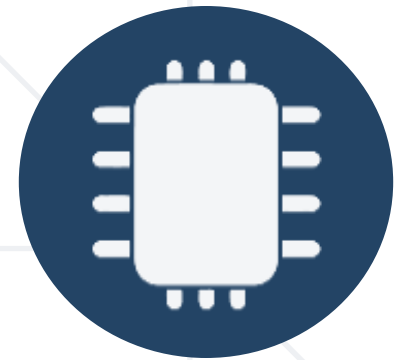




**Data Types**

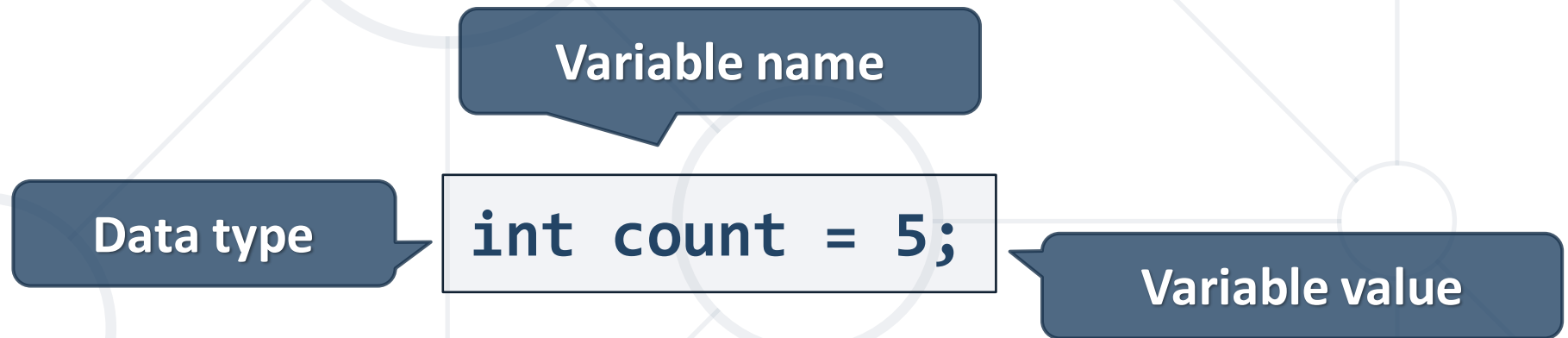
# How Computing Works?

- Computers are machines that process data
  - Instructions and data are stored in the computer memory



# Variables

- Variables have name, data type and value
  - Assignment is done by the operator "="
  - Example of variable definition and assignment



- When processed, data is stored back into variables

# What is a Data Type?

- A **data type**
  - Is a **domain of values** of similar characteristics
  - Defines the type of information stored in the computer memory (in a **variable**)
- Examples:
  - Positive integers: **1, 2, 3, ...**
  - Alphabetical characters: **a, b, c, ...**
  - Days of week: **Monday, Tuesday, ...**

- A data type has:
  - **Name** (Java keyword)
  - **Size** (how much memory is used)
  - **Default value**
- Example:
  - Name: **int**
  - Size: **32 bits** (4 bytes)
  - Default value: **0**

int: sequence  
of 32 bits in  
the memory




int: 4 sequential bytes in  
the memory





# Naming Variables

- 
- Always refer to the naming **conventions** of a programming language
    - **camelCase** is used in Java
  - Preferred form: **[Noun]** or **[Adjective] + [Noun]**
  - Should explain the purpose of the variable (Always ask "**What does this variable contain?**")



`firstName, report, config, userList, fontSize`



`foo, bar, p, p1, populate, LastName, last_name`

# Variable Scope and Lifetime

- **Scope** - where you can access a variable (global, local)
- **Lifetime** - how long a variable stays in memory

Accessible in the **main()**

```
String outer = "I'm inside the Main()";  
for (int i = 0; i < 10; i++) {  
    String inner = "I'm inside the loop";  
}  
System.out.println(outer);  
// System.out.println(inner); Error
```

Accessible only in the loop

- Variable span is how long before a variable is called
- Always declare a variable as late as possible (e.g. shorter span)

```
static void main(String[] args) {  
    String outer = "I'm inside the main()";  
    for (int i = 0; i < 10; i++)  
        String inner = "I'm inside the loop";  
        System.out.println(outer);  
        //System.out.println(inner); Error  
}
```

"outer"  
variable span

# Keep Variable Span Short

- Shorter span simplifies the code
  - Improves its **readability** and **maintainability**

```
for (int i = 0; i < 10; i++) {  
    String inner = "I'm inside the loop";  
}  
String outer = "I'm inside the main()";  
System.out.println(outer);  
// System.out.println(inner); Error
```

**"outer"** variable  
span – reduced



**int**

**Integer Types**

# Integer types

Type	Default Value	Min Value	Max Value	Size
byte	0	-128 ( $-2^7$ )	127 ( $2^7-1$ )	8 bit
short	0	-32768 ( $-2^{15}$ )	32767 ( $2^{15} - 1$ )	16 bit
int	0	-2147483648 ( $-2^{31}$ )	2147483647 ( $2^{31} - 1$ )	32 bit
long	0	-9223372036854775808 ( $-2^{63}$ )	9223372036854775807 ( $2^{63}-1$ )	64 bit

# Centuries – Example

- Depending on the unit of measure we can use different data types

```
byte centuries = 20;  
short years = 2000;  
int days = 730484;  
long hours = 17531616;
```

```
System.out.printf("%d centuries = %d years = %d days = %d hours.",  
                  centuries, years, days, hours)  
//20 centuries = 2000 years = 730484 days = 17531616 hours.
```

# Beware of Integer Overflow!

- Integers have **range** (minimal and maximal value)
- Integers could overflow → this leads to incorrect values

```
byte counter = 0;  
for (int i = 0; i < 130; i++) {  
    counter++;  
    System.out.println(counter);  
}
```

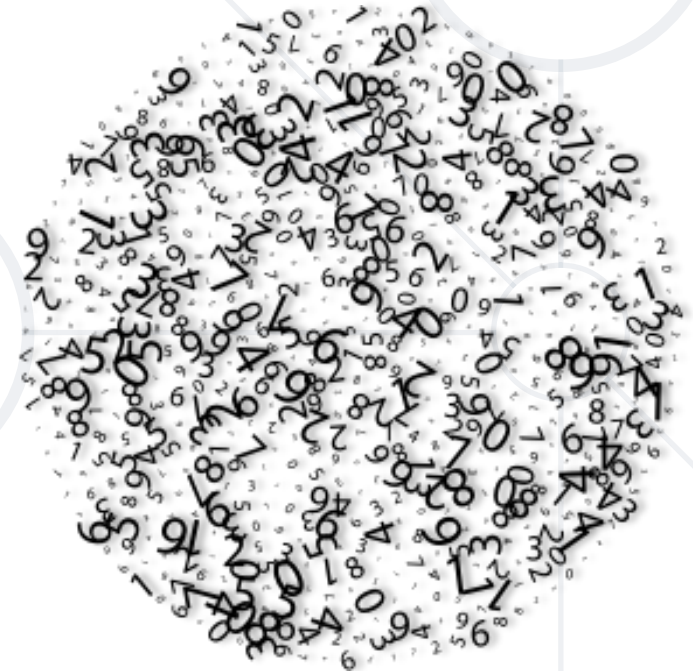


1  
2  
...  
127  
-128  
-127



- Examples of integer literals:
  - The '**0x**' and '**0X**' prefixes mean a hexadecimal value
    - E.g. **0xFE**, **0xA8F1**, **0xFFFFFFFF**
  - The '**l**' and '**L**' suffixes mean a **long**
    - E.g. **9876543L**, **0L**

```
int hexa = 0xFFFFFFFF; //-1  
long number = 1L;      //1
```



# Problem: Convert Meters to Kilometres

- Write a program that converts meters to kilometers formatted to the second decimal point
- Examples: 

1852
------

 → 

1.85
------

798
-----

 → 

0.80
------

```
Scanner scanner = new Scanner(System.in);  
  
int meters = Integer.parseInt(scanner.nextLine());  
double kilometers = meters / 1000.0;  
System.out.printf("%.2f", kilometers);
```



**float**

**Real Number Types**

# What Are Floating-Point Types?

- **Floating-point** types:
  - Represent real numbers, e.g. **1.25**, **-0.38**
  - Have range and precision depending on the memory used
  - Sometimes behave abnormally in the calculations
  - May hold very small and very big values like **0.000000000000000001** and **100.0**



# Floating-Point Numbers

- Floating-point types are:
  - **float** ( $\pm 1.5 \times 10^{-45}$  to  $\pm 3.4 \times 10^{38}$ )
    - 32-bits, the precision of 7 digits
  - **double** ( $\pm 5.0 \times 10^{-324}$  to  $\pm 1.7 \times 10^{308}$ )
    - 64-bits, the precision of 15-16 digits
- The default value of floating-point types:
  - Is **0.0F** for the **float** type
  - Is **0.0D** for the **double** type



- Difference in precision when using **float** and **double**:

```
float floatPI = 3.141592653589793238f;  
double doublePI = 3.141592653589793238;  
System.out.println("Float PI is: " + floatPI);  
System.out.println("Double PI is: " + doublePI);
```

3.1415927

- NOTE: The "**f**" suffix in the first statement!

3.141592653589793

- Real numbers are by default interpreted as **double**
- One should explicitly convert them to **float**

# Problem: Pound to Dollars

- Write a program that converts British pounds to US dollars formatted to 3rd decimal point
- 1 British Pound = 1.36 Dollars



```
double num = Double.parseDouble(scanner.nextLine());  
double result = num * 1.36;  
System.out.printf("%.3f", result);
```

- Floating-point numbers can use scientific notation, e.g.
  - **1e+34, 1E34, 20e-3, 1e-12, -6.02e28**

```
double d = 1000000000000000000000000000000000000000000000000.0;  
System.out.println(d); // 1.0E34  
double d2 = 20e-3;  
System.out.println(d2); // 0.02  
double d3 = Double.MAX_VALUE;  
System.out.println(d3); //1.7976931348623157E308
```



- Integral division and floating-point division are different:


```
System.out.println(10 / 4);           // 2 (integral division)
System.out.println(10 / 4.0);         // 2.5 (real division)
System.out.println(10 / 0.0);         // Infinity
System.out.println(-10 / 0.0);        // -Infinity
System.out.println(0 / 0.0);          // NaN (not a number)
System.out.println(8 % 2.5);          // 0.5 (3 * 2.5 + 0.5 = 8)
System.out.println(10 / 0);           // ArithmeticException
```

- Sometimes floating-point numbers work incorrectly!
- Read more about **IEEE 754**

```
double a = 1.0f;
double b = 0.33f;
double sum = 1.33d;
System.out.printf("a+b=%f sum=%f equal=%b",
                  a+b, sum, (a + b == sum));
// a+b=1.33000001311302 sum=1.33 equal = false
double num = 0;
for (int i = 0; i < 10000; i++) num += 0.0001;
System.out.println(num); // 0.99999999999999062
```

# BigDecimal

- Built-in Java Class
- Provides arithmetic operations
- Allows calculations with very **high precision**
- Used for financial calculations



```
BigDecimal number = new BigDecimal(0);  
number = number.add(BigDecimal.valueOf(2.5));  
number = number.subtract(BigDecimal.valueOf(1.5));  
number = number.multiply(BigDecimal.valueOf(2));  
number = number.divide(BigDecimal.valueOf(2));
```

# Problem: Exact Sum of Real Numbers

- Write program to enter **n** numbers and print their exact sum:

2  
1000000000000000000000000  
5



1000000000000000000000005

2  
0.0000000000003  
33333333333333.3



33333333333333.300000000003

Check your solution here: <https://judge.softuni.org/Contests/1227/>

# Solution: Exact Sum of Real Numbers

```
int n = Integer.parseInt(sc.nextLine());
BigDecimal sum = new BigDecimal(0);
for (int i = 0; i < n; i++) {
    BigDecimal number = new BigDecimal(sc.nextLine());
    sum = sum.add(number);
}
System.out.println(sum);
```



**Type Conversion**

- Variables hold values of a certain type
- Type can be **changed** (**converted**) to another type
  - **Implicit** type conversion (**lossless**): variable of the bigger type (e.g. **Double**) takes a smaller value (e.g. **float**)

```
float heightInMeters = 1.74f;  
double maxHeight = heightInMeters;
```

**Implicit** conversion

- **Explicit** type conversion (**lossy**) – when precision can be lost:

```
double size = 3.14;  
int intSize = (int) size;
```

**Explicit** conversion

# Problem: Centuries to Minutes

- Write a program to enter an integer number of centuries and convert it to years, days, hours, and minutes

1



1 centuries = 100 years = 36524 days  
= 876581 hours = 52594877 minutes

5



5 centuries = 500 years = 182621 days  
= 4382906 hours = 262974384 minutes

The output is  
on one row

Check your solution here: <https://judge.softuni.org/Contests/1227/>



# Solution: Centuries to Minutes

```
int centuries = Integer.parseInt(sc.nextLine());  
double years = centuries * 100;  
double days = years * 365.2422;  
double hours = 24 * days;  
double minutes = 60 * hours;  
System.out.printf(  
"%d centuries = %.0f years = %.0f days = %.0f hours = %.0f minutes"  
, centuries, years, days, hours, minutes);
```

Tropical year has  
**365.2422** days



**Boolean Type**

- Boolean variables (**boolean**) hold **true** or **false**:

```
int a = 1;
int b = 2;
boolean greaterAB = (a > b);
System.out.println(greaterAB);    // False
boolean equalA1 = (a == 1);
System.out.println(equalA1);      // True
```

# Problem: Special Numbers

- A number is special when its sum of digits is 5, 7 or 11
  - For all numbers **1...n** print the number and if it is special

20



1 -> false	8 -> false	15 -> false
2 -> false	9 -> false	16 -> true
3 -> false	10 -> false	17 -> false
4 -> false	11 -> false	18 -> false
5 -> true	12 -> false	19 -> false
6 -> false	13 -> false	20 -> false
7 -> true	14 -> true	

Check your solution here: <https://judge.softuni.org/Contests/1227/>

# Solution: Special Numbers

```
int n = Integer.parseInt(sc.nextLine());
for (int num = 1; num <= n; num++) {
    int sumOfDigits = 0;
    int digits = num;
    while (digits > 0) {
        sumOfDigits += digits % 10;
        digits = digits / 10;
    }
    // TODO: check whether the sum is special
}
```

Check your solution here: <https://judge.softuni.org/Contests/1227/>



**Character Type**

- The character data type
  - Represents symbolic information
  - Is declared by the **char** keyword
  - Gives each symbol a corresponding integer code
  - Has a '**\0**' default value
  - Takes 16 bits of memory (from **U+0000** to **U+FFFF**)
  - Holds a single Unicode character (or part of character)

- Each **character** has an unique **Unicode** value (**int**):

```
char ch = 'a';  
System.out.printf("The code of '%c' is: %d\n", ch, (int) ch);  
ch = 'b';  
System.out.printf("The code of '%c' is: %d\n", ch, (int) ch);  
ch = 'A';  
System.out.printf("The code of '%c' is: %d\n", ch, (int) ch);  
ch = 'щ'; // Cyrillic letter 'sht'  
System.out.printf("The code of '%c' is: %d\n", ch, (int) ch);
```



# Problem: Reversed Chars

- Write a program that takes 3 lines of characters and prints them in reversed order with a space between them
- Examples:

A  
B  
C



C B A

1  
L  
&



& L 1

# Solution: Reversed Chars

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

char firstChar = scanner.nextLine().charAt(0);
char secondChar = scanner.nextLine().charAt(0);
char thirdChar = scanner.nextLine().charAt(0);

System.out.printf("%c %c %c",
    thirdChar, secondChar, firstChar);
```

- Escaping sequences are:
  - Represent a special character like ' , " or \n (new line)
  - Represent system characters (like the [TAB] character \t)
- Commonly used escaping sequences are:
  - \' → for single quote      \" → for double quote
  - \\ → for backslash      \n → for a new line
  - \uXXXX → for denoting any other Unicode symbol

# Character Literals – Example

```
char symbol = 'a'; // An ordinary character
symbol = '\u006F'; // Unicode character code in a
                    // hexadecimal format (Letter 'o')
symbol = '\u8449'; // 葉 (Leaf in Traditional Chinese)
symbol = '\''; // Assigning the single quote character
symbol = '\\'; // Assigning the backslash character
symbol = '\n'; // Assigning new line character
symbol = '\t'; // Assigning TAB character
symbol = "a"; // Incorrect: use single quotes!
```



**"ABC"**

**String**

Sequence of Letters

# The String Data Type

- The string data type
  - Represents a sequence of characters
  - Is declared by the **String** keyword
  - Has a default value **null** (no value)

- Strings are enclosed in quotes:

```
String s = "Hello, JAVA";
```

- Strings can be concatenated
  - Using the **+** operator



- Strings are enclosed in quotes "":

```
String file = "C:\\Windows\\win.ini";
```

The backslash \ is  
escaped by \\

- Format strings insert variable values by pattern:

```
String firstName = "Svetlin";  
String lastName = "Nakov";  
String fullName = String.format(  
    "%s %s", firstName, lastName);
```

- Combining the names of a person to obtain the full name:

```
String firstName = "Ivan";  
String lastName = "Ivanov";  
String fullName = String.format(  
    "%s %s", firstName, lastName);  
System.out.printf("Your full name is %s.", fullName);
```

- We can concatenate strings and numbers by the **+** operator:

```
int age = 21;  
System.out.println("Hello, I am " + age + " years old");
```



# Problem: Concat Names

- Read first and last name and delimiter
- Print the first and last name joined by the delimiter

John  
Smith  
->



John->Smith

Linda  
Terry  
=>



Linda=>Terry

Jan  
White  
<->



Jan<->White

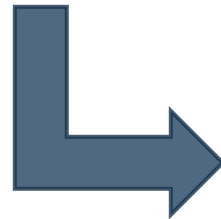
Lee  
Lewis  
---



Lee---Lewis

# Solution: Concat Names

```
String firstName = sc.nextLine();  
String lastName = sc.nextLine();  
String delimiter = sc.nextLine();  
  
String result = firstName + delimiter + lastName;  
System.out.println(result);
```

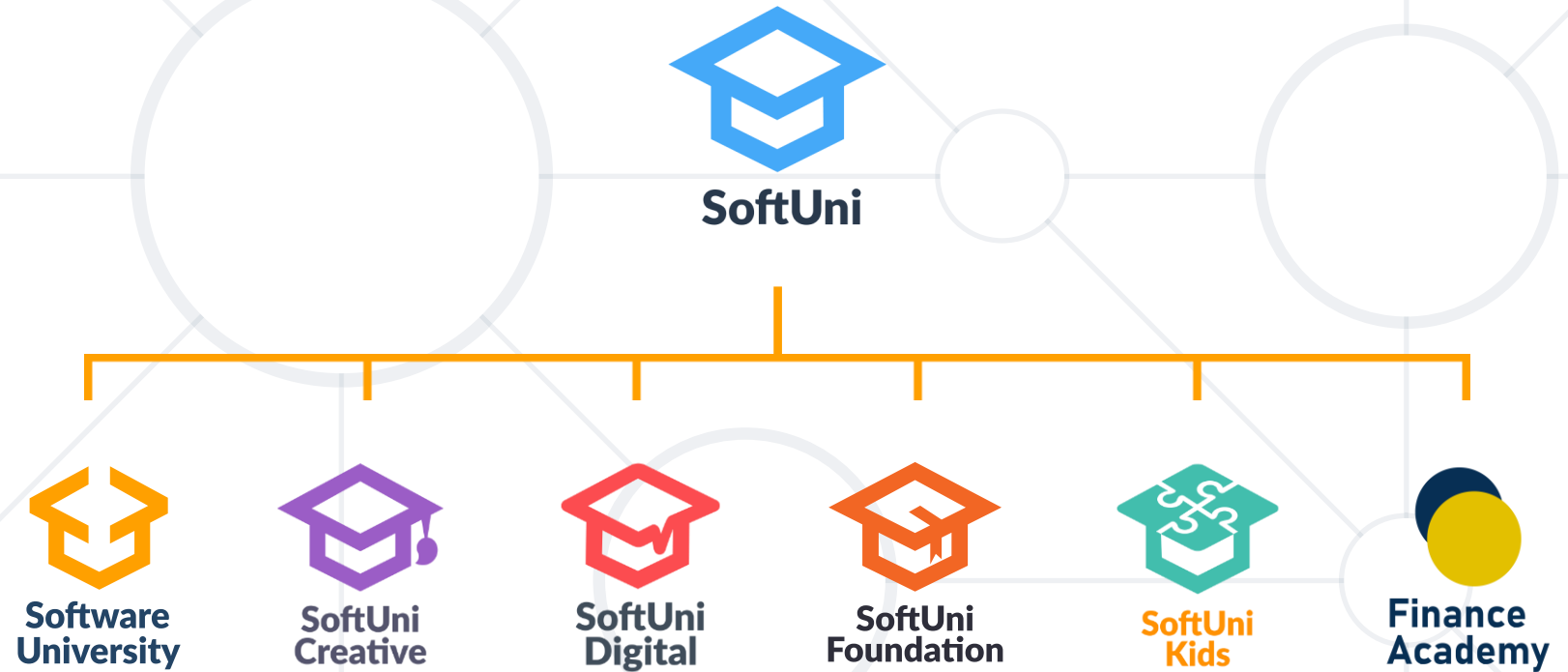


Jan<->White

- **Variables** – store data
- Numeral types:
  - Represent **numbers**
  - Have **specific ranges** for every type
- String and text types:
  - Represent **text**
  - **Sequences of Unicode characters**
- Type conversion: **implicit** and **explicit**



# Questions?



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

