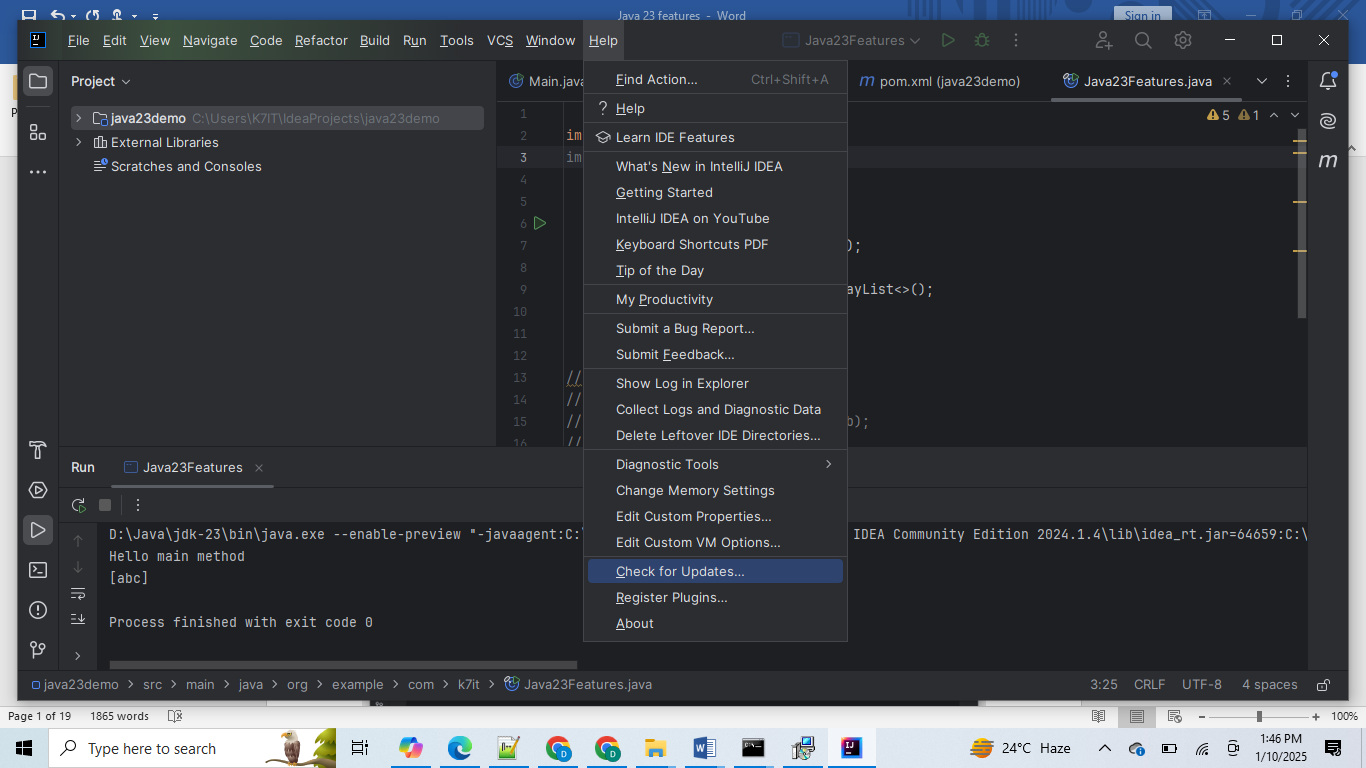
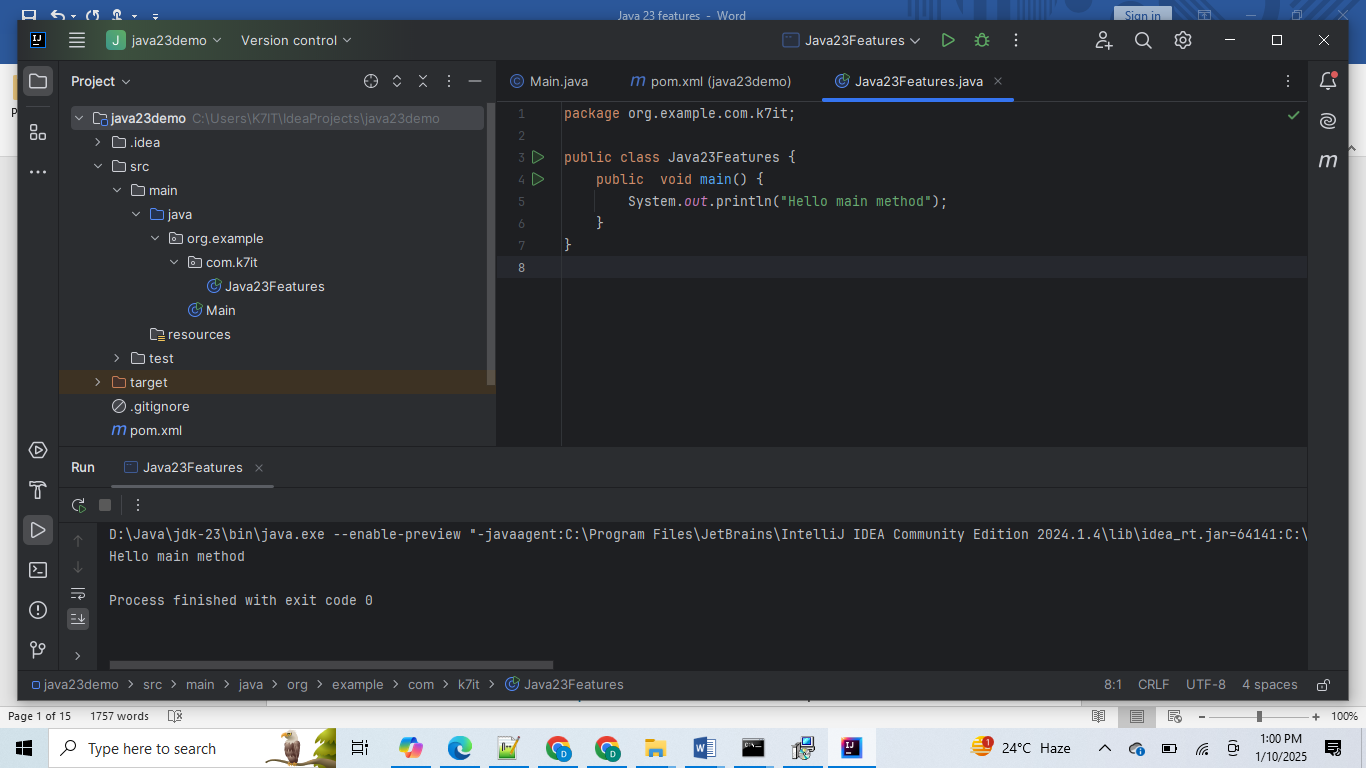
**Java 23 features:**

**Installation of java 23 and set up projects in intellij**

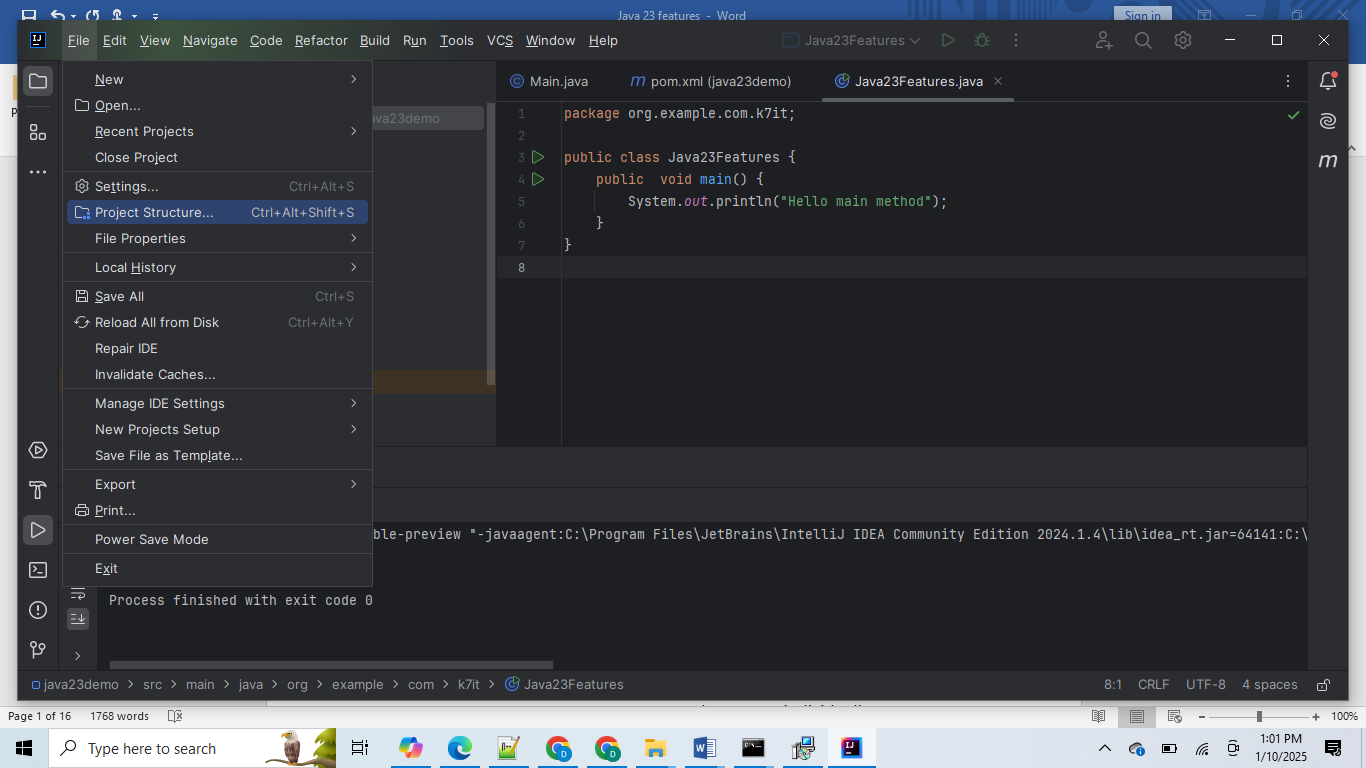
**Help -> check for updates**



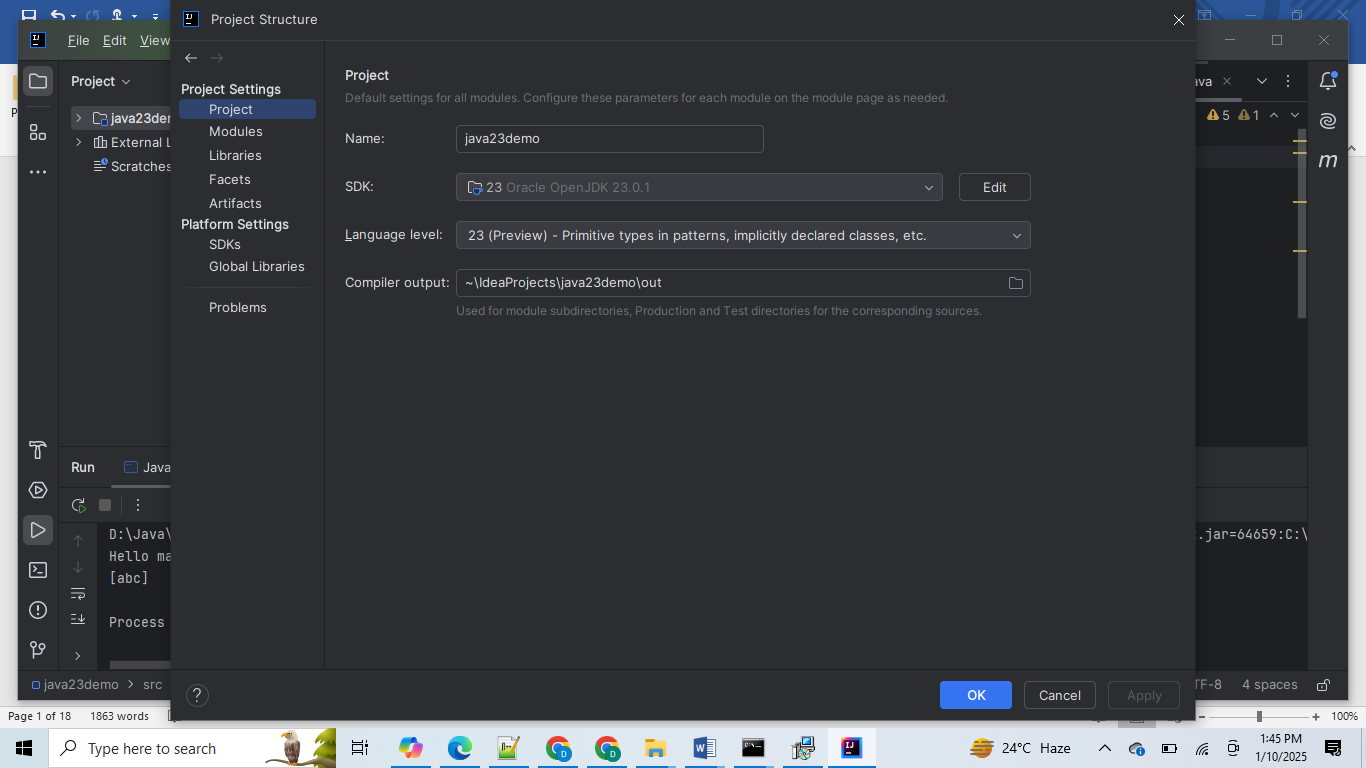
**If we want to use java23 in intellij we should upgrade our intellij with 2024.3.1.1 latest version**

Create project with sdk 23 with project name is java23demo  


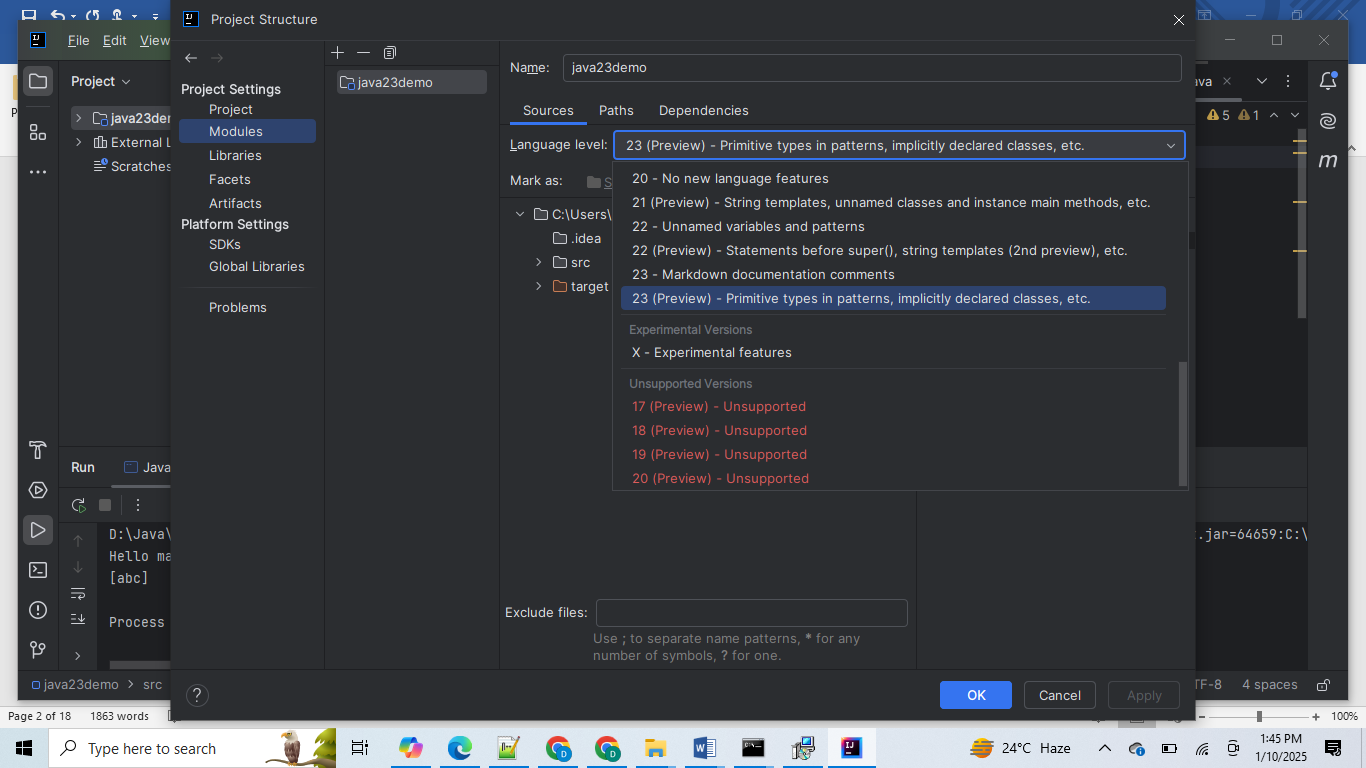
Then update language levels and sdK under project and modules



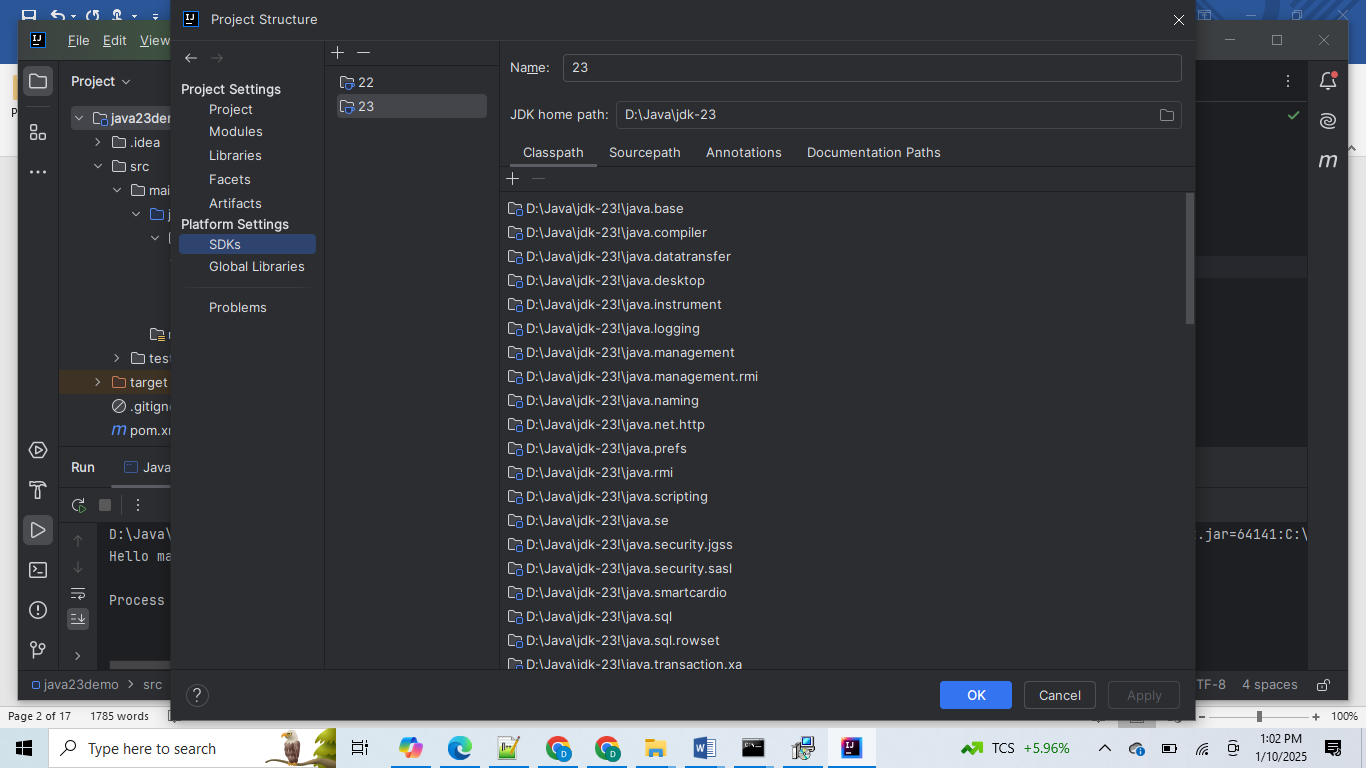
Under Project change SDK and language level like bellow :



Next change same things under modules as well



Then go to SDKS : select java 23



1. **Module Import Declarations (Preview)**

Since Java 1.0, all classes of the *java.lang* package are automatically imported into every .java file. That’s why we can use classes like *Object*, *String*, *Integer*, *Exception*, *Thread*, etc. without *import* statements.

We have also always been able to import complete packages. For example, importing *java.util.\** means that we do not have to import classes such as *List*, *Set*, *Map*, *ArrayList*, *HashSet* and *HashMap* individually.

[JDK Enhancement Proposal 476](https://openjdk.org/jeps/476) now allows us to import complete modules – more precisely, all classes in the packages exported by the module.

For example, we can import the complete *java.base* module as follows and use classes from this module (in the example *List*, *Map*, *Collectors*, *Stream*) without further imports:

import module java.base;

public static Map<Character, List<String>> groupByFirstLetter(String... values) {

return Stream.of(values).collect(

Collectors.groupingBy(s -> Character.toUpperCase(s.charAt(0))));

}

import static java.io.IO.*println*;  
import module java.base;  
  
  
 void main() {  
 *println*("Hello main method");  
  
 List<String> list = new ArrayList<>();  
 list.add("abc");  
 *println*(list);  
  
 }

To use *import module*, the importing class itself doesn't need to be in a module.

#### **Ambiguous Class Names**

If there are two imported classes with the same name, such as *Date* in the following example, a compiler error occurs:

import module java.base;

import module java.sql;

. . .

Date date = new Date(); // Compiler error: "reference to Date is ambiguous"

. . .Code language: Java (java)

The solution is simple: we also have to import the desired Date class directly:

import module java.base;

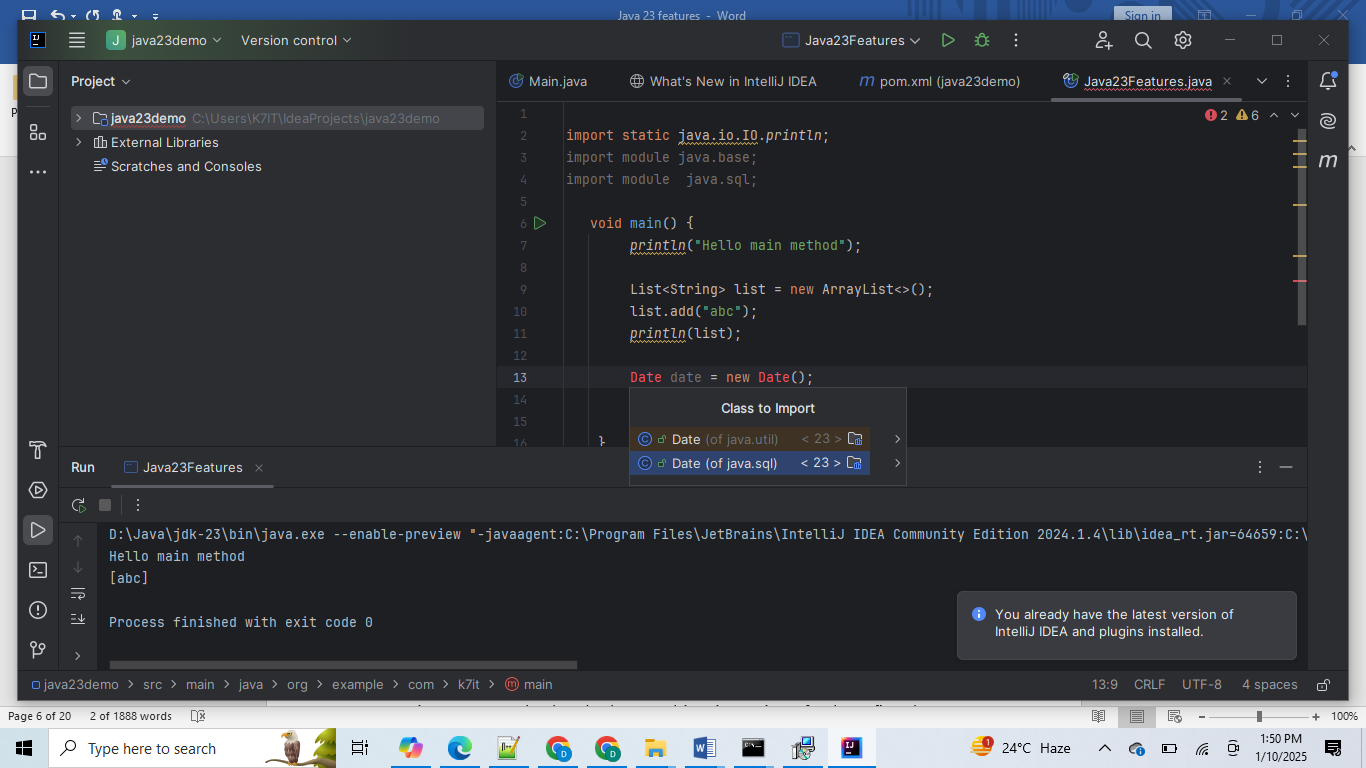
import module java.sql;

import java.util.Date; // ⟵ This resolves the ambiguity

. . .

Date date = new Date();

. . .



import static java.io.IO.*println*;  
import module java.base;  
import module java.sql;  
  
import java.sql.Date;  
  
void main() {  
 *println*("Hello main method");  
  
 List<String> list = new ArrayList<>();  
 list.add("abc");  
 *println*(list);  
  
 Date date = new Date(122323);  
  
  
 }

1. **Primitive Types in Patterns, instanceof, and switch (Preview)**

With *instanceof* and *switch*, we can check whether an object is of a particular type, and if so, bind this object to a variable of this type, execute a specific program path, and use the new variable in this program path.

The following code block, for example, which has been permitted since [Java 16](https://www.happycoders.eu/java/java-16-features/#Pattern_Matching_for_instanceof), checks whether an object is a string of at least five characters and, if so, prints it in upper case. If the object is an integer, the number is squared and printed. Otherwise, the object is printed as it is.

if (obj instanceof String s && s.length() >= 5) {

System.out.println(s.toUpperCase());

} else if (obj instanceof Integer i) {

System.out.println(i \* i);

} else {

System.out.println(obj);

}

Since [Java 21](https://www.happycoders.eu/java/java-21-features/#Record_Patterns_JEP_440), we can do the same much more clearly using *switch*:

switch (obj) {

case String s when s.length() >= 5 -> System.out.println(s.toUpperCase());

case Integer i -> System.out.println(i \* i);

case null, default -> System.out.println(obj);

}

So far, however, this only works with objects. *instanceof* cannot be used with primitive data types at all, *switch* only to the extent that it can match variables of the primitive types *byte*, *short*, *char*, and *int* against constants, e.g., like this:

int x = 10;

switch (x) {

case 1, 2, 3 -> System.out.println("Low");

case 4, 5, 6 -> System.out.println("Medium");

case 7, 8, 9 -> System.out.println("High");

}

 two changes in Java 23:

* Firstly, *all* primitive types may now be used in *switch* expressions and statements, including *long*, *float*, *double*, and *boolean*.
* Secondly, we can also use all primitive types in pattern matching – both for *instanceof* and *switch*.

#### **Primitive Types in Pattern Matching**

With primitive patterns, the exact meaning is different than when using objects – because there is no inheritance with primitive types:

Be *a* a variable of a primitive type (i.e., *byte*, *short*, *int*, *long*, *float*, *double*, *char*, or *boolean*) and *B* one of these primitive types. Then, *a instanceof B* results in *true* if the precise value of *a* can also be stored in a variable of type *B*.

To help you better understand what is meant by this, here is a simple example:

int a = ...

if (a instanceof byte b) {

System.out.println("b = " + b);

}

The code should be read as follows: If the value of the variable *a* can also be stored in a *byte* variable, then assign this value to the *byte* variable *b* and print it.

This would be the case for *a = 5*, for example, but not for *a = 1000*, as *byte* can only store values from -128 to 127.

Just as with objects, for primitive types, you can also add further checks directly in the *instanceof* check using *&&*. The following code, for example, only prints positive byte values (i.e., 1 to 127):

int a = ...

if (a instanceof byte b && b > 0) {

System.out.println("b = " + b);

}

int a=10;  
  
if(a instanceof byte b){  
  
}else if(a instanceof short s){  
  
}else if(a instanceof int i){  
  
}else if( a instanceof long l){  
  
}else if(a instanceof float f){  
  
}else if(a instanceof double d){  
  
}

#### **Primitive Type Pattern with switch**

We can use primitive patterns not only in instanceof but also in switch:

double value = ...

switch (value) {

case byte b -> System.out.println(value + " instanceof byte: " + b);

case short s -> System.out.println(value + " instanceof short: " + s);

case char c -> System.out.println(value + " instanceof char: " + c);

case int i -> System.out.println(value + " instanceof int: " + i);

case long l -> System.out.println(value + " instanceof long: " + l);

case float f -> System.out.println(value + " instanceof float: " + f);

case double d -> System.out.println(value + " instanceof double: " + d);

}

double value = 10.25;  
  
switch (value){  
 case byte b -> System.*out*.println(value + " instanceof byte: " + b);  
 case short s-> System.*out*.println(value + " instanceof byte: " + s);  
 case int i -> System.*out*.println(value + " instanceof byte: " + i);  
 case long l -> System.*out*.println(value + " instanceof byte: " + l);  
 case float f -> System.*out*.println(value + " instanceof byte: " + f);  
 case double d -> System.*out*.println(value + " instanceof double: " + d);  
 // default -> System.out.println(value + " instanceof byte: ");  
 };

here if we use double type then no need to use default case,

double result = switch (value){  
 case byte b -> b;  
 case short s-> s;  
 case int i -> i;  
 case long l ->l;  
 case float f -> f;  
 case double d -> d;  
 // default -> System.out.println(value + " instanceof byte: ");  
};

1. **Implicitly Declared Classes and Instance Main Methods (Third Preview)**

When Java developers write their first program, it usually looks like this (until now):

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello world!");

}

}

Java beginners are confronted with numerous new concepts at once:

* with classes,
* with the visibility modifier *public*,
* with static methods,
* with unused method arguments,
* with *System.out*.

As of Java 23, the following code is a valid and complete Java program:

void main() {

println("Hello world!");

}

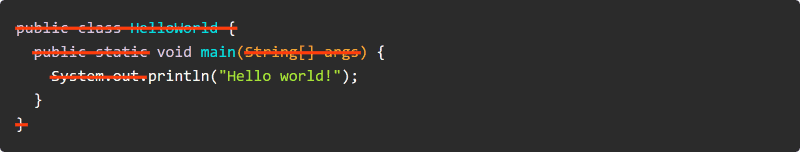
package org.example.com.k7it;  
  
import static java.io.IO.*println*;  
  
public class Java23Features {  
 void main() {  
 *println*("Hello main method");  
 }  
}

or

import static java.io.IO.*println*;  
  
  
 void main() {  
 *println*("Hello main method");  
 }

Here println() is available under java.io.IO class it introduced in java 23 , just add static import and use the println() directly without using System.out.println()

Here old to new difference:



1. **Flexible Constructor Bodies**

Its same as java 22 about usage of super() and this() inside constructor body

Let’s assume you have a class like the following:

public class ConstructorTestParent {

private final int a;

public ConstructorTestParent(int a) {

this.a = a;

printMe();

}

void printMe() {

System.out.println("a = " + a);

}

}Code language: Java (java)

And let’s assume you have a second class that extends this class:

public class ConstructorTestChild extends ConstructorTestParent {

private final int b;

public ConstructorTestChild(int a, int b) {

super(a);

this.b = b;

}

}Code language: Java (java)

Now, you want to ensure that a and b are not negative in the ConstructorTestChild constructor before calling the super constructor.

It was previously not permitted to place a corresponding check *before* the constructor. That's why we had to make do with contortions like the following:

public class ConstructorTestChild extends ConstructorTestParent {

private final int b;

public ConstructorTestChild(int a, int b) {

super(verifyParamsAndReturnA(a, b));

this.b = b;

}

private static int verifyParamsAndReturnA(int a, int b) {

if (a < 0 || b < 0) throw new IllegalArgumentException();

return a;

}

}

This is neither very elegant nor easy to read.

These changes now allow the code to be rewritten as follows:

*public class ConstructorTestChild extends ConstructorTestParent {*

*private final int b;*

*public ConstructorTestChild(int a, int b) {*

*if (a < 0 || b < 0) throw new IllegalArgumentException(); // ⟵ Now allowed!*

*this.b = b; // ⟵ Now allowed!*

*super(a);*

*}*

*@Override*

*void printMe() {*

*super.printMe();*

*System.out.println("b = " + b);*

*}*

*}Code language: Java (java)*

*A call to new ConstructorTestChild(1, 2) now results in the expected output:*

*a = 1*

*b = 2*

1. **Markdown Documentation Comments**

To format JavaDoc comments, we have always had to use HTML. This was undoubtedly a good choice in 1995, but nowadays, Markdown is much more popular than HTML for writing documentation.

[JDK Enhancement Proposal 467](https://openjdk.org/jeps/467) allows us to write JavaDoc comments in Markdown from Java 23 onwards.

The following example shows the documentation of the *Math.ceilMod(...)* method in the conventional notation:

/\*\*

\* Returns the ceiling modulus of the {@code long} and {@code int} arguments.

\* <p>

\* The ceiling modulus is {@code r = x – (ceilDiv(x, y) \* y)},

\* has the opposite sign as the divisor {@code y} or is zero, and

\* is in the range of {@code -abs(y) < r < +abs(y)}.

\*

\* <p>

\* The relationship between {@code ceilDiv} and {@code ceilMod} is such that:

\* <ul>

\* <li>{@code ceilDiv(x, y) \* y + ceilMod(x, y) == x}</li>

\* </ul>

\* <p>

\* For examples, see {@link #ceilMod(int, int)}.

\*

\* @param x the dividend

\* @param y the divisor

\* @return the ceiling modulus {@code x – (ceilDiv(x, y) \* y)}

\* @throws ArithmeticException if the divisor {@code y} is zero

\* @see #ceilDiv(long, int)

\* @since 18

\*/

The example contains formatted code, paragraph marks, a bulleted list, a link, and JavaDoc-specific information such as *@param* and *@return*.

To use Markdown, we need to start all lines of a JavaDoc comment with three slashes. The same comment in Markdown would look like this:

/// Returns the ceiling modulus of the `long` and `int` arguments.

///

/// The ceiling modulus is `r = x – (ceilDiv(x, y) \* y)`,

/// has the opposite sign as the divisor `y` or is zero, and

/// is in the range of `-abs(y) < r < +abs(y)`.

///

/// The relationship between `ceilDiv` and `ceilMod` is such that:

///

/// – `ceilDiv(x, y) \* y + ceilMod(x, y) == x`

///

/// For examples, see [#ceilMod(int, int)].

///

/// @param x the dividend

/// @param y the divisor

/// @return the ceiling modulus `x – (ceilDiv(x, y) \* y)`

/// @throws ArithmeticException if the divisor `y` is zero

/// @see #ceilDiv(long, int)

/// @since 18

This is both easier to write and easier to read.

What has changed in detail?

* Source code is marked with *`...`* instead of *{@code ...}*.
* The HTML paragraph character *<p>* has been replaced by a blank line.
* The enumeration items are introduced by hyphens.
* Instead of *{@link ...}*, links are marked with *[...]*.
* The JavaDoc-specific details, such as *@param* and *@return*, remain unchanged.

The following text formatting is supported:

/// \*\*This text is bold.\*\*

/// \*This text is italic.\*

/// \_This is also italic.\_

/// `This is source code.`

///

/// ```

/// This is a block of source codex.

/// ```

///

/// Indented text

/// is also rendered as a code block.

///

/// ~~~

/// This is also a block of source code

/// ~~~

Enumerated lists and numbered lists are supported:

/// This is a bulleted list:

/// – One

/// – Two

/// – Three

///

/// This is a numbered list:

/// 1. One

/// 1. Two

/// 1. Three

package org.example.com.k7it;  
  
import static java.io.IO.*println*;  
import module java.base;  
import module java.sql;  
  
import java.sql.Date;  
*/// this class is using for explanation about java 23 new features  
///  
/// @author K7  
/// @since 2025  
/// 1. java  
/// 2. oracle  
/// 3. mongodb  
///*class Java23Features {  
 void main() {  
 *println*("Hello main method");  
  
 List<String> list = new ArrayList<>();  
 list.add("abc");  
 *println*(list);  
  
 Date date = new Date(122323);  
  
 int a = 10;  
  
 if (a instanceof byte b) {  
  
 } else if (a instanceof short s) {  
  
 } else if (a instanceof int i) {  
  
 } else if (a instanceof long l) {  
  
 } else if (a instanceof float f) {  
  
 } else if (a instanceof double d) {  
  
 }  
  
 switch (a) {  
 case 1, 2, 3:  
 *println*("low priority");  
 case 4, 5, 6:  
 *println*("medium priority");  
 case 7, 8, 9:  
 *println*("high priority");  
 }  
  
 double value = 10.25;  
  
 switch (value) {  
 case byte b -> System.*out*.println(value + " instanceof byte: " + b);  
 case short s -> System.*out*.println(value + " instanceof short: " + s);  
 case int i -> System.*out*.println(value + " instanceof int: " + i);  
 case long l -> System.*out*.println(value + " instanceof long: " + l);  
 case float f -> System.*out*.println(value + " instanceof float: " + f);  
 case double d -> System.*out*.println(value + " instanceof double: " + d);  
 // default -> System.out.println(value + " instanceof byte: ");  
 }  
 ;  
  
 double result = switch (value) {  
 case byte b -> b;  
 case short s -> s;  
 case int i -> i;  
 case long l -> l;  
 case float f -> f;  
 case double d -> d;  
 // default -> System.out.println(value + " instanceof byte: ");  
 };  
  
  
 }  
  
}

