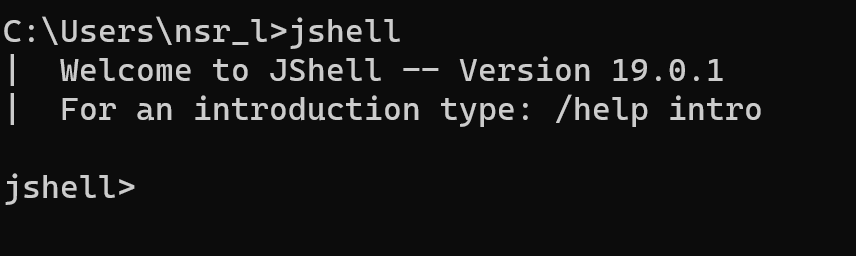
**Java 17 Masterclass: Start Coding in 2023**

**Section 1: Getting Started**

**Section 2: Programming Tools**

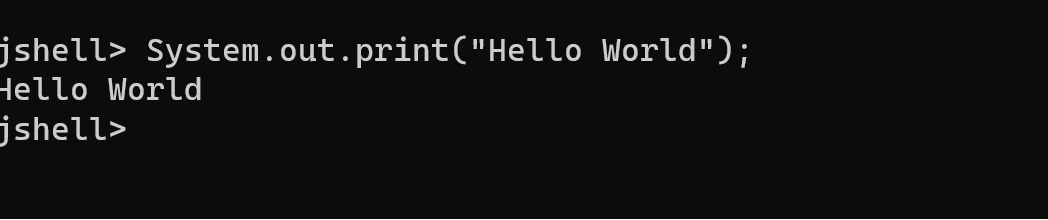
**Section 3: First Steps**



**=>Hello World**

System.out.print(“Hello World”);

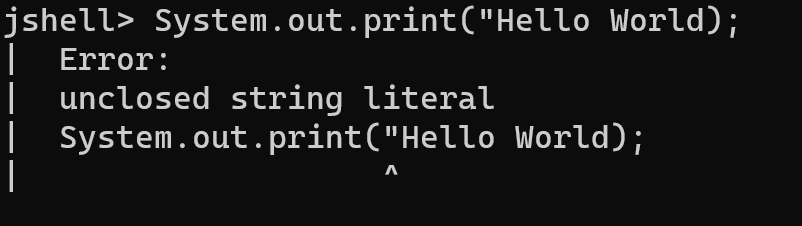
This is a command to print some information to the screen.



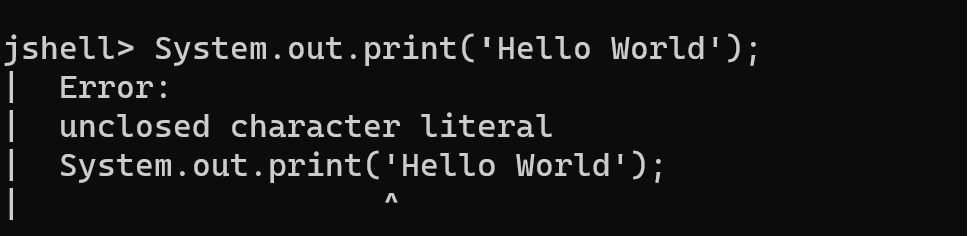
To modify existing line you can press the up arrow key-remember that in Jshell. You can see history of the lines you’ve previously typed using up arrows and down arrows.

**Errors:**

**1.Unclosed string literal**

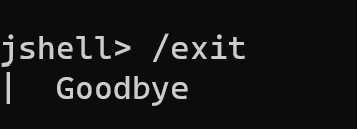
****

**2.Unclosed character literal**



Typing forward slash and the word ‘exit’, or forward slash with the shortcut text ex, will end your Jshell session,if you get stuck.

An example would be /exit or /ex



**=>Variables**

In programming, a variable is a named storage location that holds data, and its value can be changed during the execution of a program. Variables are fundamental to programming as they allow you to manipulate and store data in memory. Here are some key points about variables:

1. **Declaration:**
   * Before using a variable, you need to declare it. The declaration includes the data type of the variable and its name.
   * Example: **int age;**
2. **Initialization:**
   * After declaring a variable, you typically initialize it by assigning an initial value.
   * Example: **age = 25;**
3. **Assignment:**
   * You can change the value of a variable through assignment statements.
   * Example: **age = 30;**
4. **Data Types:**
   * Variables have data types that define the type of data they can hold (e.g., **int**, **double**, **String**).
   * Example: **double salary = 50000.50;**
5. **Naming Rules:**
   * Variable names must follow certain rules, such as starting with a letter, being case-sensitive, and not using reserved keywords.
   * Example: **String firstName;**
6. **Scope:**
   * The scope of a variable defines where in the code it can be accessed. For example, local variables are only accessible within the method or block where they are declared.
   * Example:

javaCopy code

void exampleMethod() { int localVar = 10; // localVar is only accessible within this method }

1. **Constants:**
   * You can use the **final** keyword to declare constants, which are variables whose values cannot be changed.
   * Example: **final double PI = 3.14159;**
2. **Instance and Class Variables:**
   * In object-oriented programming, variables can be instance variables (associated with an instance of a class) or class variables (associated with a class itself).
   * Example:

javaCopy code

public class MyClass { int instanceVar; // instance variable static int classVar; // class variable }

1. **Local Variables:**
   * Variables declared inside a method, constructor, or block are called local variables.
   * Example:

javaCopy code

void exampleMethod() { int localVar = 10; // local variable }

1. **Null and Default Values:**
   * Variables that are not explicitly initialized might have default values (e.g., **0** for numeric types, **false** for boolean, **null** for reference types).

These concepts are general to many programming languages, including Java. Understanding how to use variables is fundamental to writing effective and dynamic programs.

In Java, variables are used to store and manipulate data. They are essentially named storage locations that can hold values of a particular data type. Here are the basic types of variables in Java:

1. **Primitive Data Types:**
   * **byte:** 8-bit integer.
   * **short:** 16-bit integer.
   * **int:** 32-bit integer.
   * **long:** 64-bit integer.
   * **float:** 32-bit floating-point.
   * **double:** 64-bit floating-point.
   * **char:** 16-bit Unicode character.
   * **boolean:** Represents true or false.

Example:

javaCopy code

int age = 25; double salary = 50000.50; char grade = 'A'; boolean isStudent = true;

1. **Reference Data Types:**
   * **Objects:** Instances of classes.
   * **Arrays:** Homogeneous collections of elements.

Example:

javaCopy code

String name = "John"; Scanner scanner = new Scanner(System.in); int[] numbers = {1, 2, 3, 4, 5};

1. **Constants:**
   * Variables declared with the **final** keyword are constants, meaning their values cannot be changed once assigned.

Example:

javaCopy code

final double PI = 3.14159;

1. **Local Variables:**
   * Declared within a method, constructor, or block.
   * They must be initialized before use.

Example:

javaCopy code

void exampleMethod() { int localVar = 10; // Do something with localVar }

1. **Instance Variables (Non-Static Fields):**
   * Declared within a class but outside any method, constructor, or block.
   * Each instance of the class has its own copy of the instance variable.

Example:

javaCopy code

public class MyClass { int instanceVar = 5; }

1. **Class Variables (Static Fields):**
   * Declared with the **static** keyword.
   * Belongs to the class, not to instances of the class.

Example:

javaCopy code

public class MyClass { static int classVar = 10; }

It's important to note that Java is a statically-typed language, meaning that you must declare the type of a variable before using it. Additionally, Java follows the camelCase naming convention for variables, where the first word starts with a lowercase letter, and subsequent concatenated words begin with an uppercase letter.

In Java, keywords are reserved words that have special meanings and cannot be used as identifiers (such as variable names or class names). Here is a list of some of the keywords in Java:

1. **Primitive Data Type Keywords:**
   * **boolean**: Represents true or false values.
   * **byte**: 8-bit integer.
   * **short**: 16-bit integer.
   * **int**: 32-bit integer.
   * **long**: 64-bit integer.
   * **float**: 32-bit floating-point.
   * **double**: 64-bit floating-point.
   * **char**: 16-bit Unicode character.
2. **Flow Control Keywords:**
   * **if**: Conditional statement.
   * **else**: Alternative statement for if.
   * **switch**: Multi-way branch statement.
   * **case**: Defines a branch in a switch statement.
   * **default**: Defines the default branch in a switch statement.
   * **while**: Looping construct with a condition.
   * **do**: Looping construct that executes at least once.
   * **for**: Looping construct with initialization, condition, and iteration expressions.
   * **break**: Exits from a loop or a switch statement.
   * **continue**: Skips the rest of the loop and starts the next iteration.
3. **Modifiers:**
   * **public**: Access modifier indicating visibility to all classes.
   * **private**: Access modifier indicating visibility only within the declaring class.
   * **protected**: Access modifier indicating visibility to subclasses.
   * **static**: Modifier indicating that a variable or method belongs to the class, not instances of the class.
   * **final**: Modifier indicating that a variable, method, or class cannot be changed or extended.
   * **abstract**: Modifier indicating that a class or method is incomplete and must be implemented by a subclass.
   * **synchronized**: Modifier indicating that a method can be accessed by only one thread at a time.
4. **Class, Object, and Package Keywords:**
   * **class**: Declares a class.
   * **extends**: Indicates that a class is a subclass of another class.
   * **implements**: Indicates that a class implements an interface.
   * **interface**: Declares an interface.
   * **package**: Declares a package.
   * **import**: Imports a class or entire package.
5. **Exception Handling Keywords:**
   * **try**: Encloses a block of code that may throw exceptions.
   * **catch**: Catches and handles exceptions.
   * **finally**: Encloses a block of code that is executed regardless of whether an exception is thrown or not.
   * **throw**: Throws an exception manually.
   * **throws**: Specifies that a method might throw exceptions.
6. **Miscellaneous Keywords:**
   * **this**: Refers to the current instance of the class.
   * **super**: Refers to the superclass.
   * **new**: Creates a new object.
   * **return**: Returns a value from a method.
   * **void**: Indicates that a method does not return any value.
   * **instanceof**: Checks whether an object is an instance of a particular class or interface.

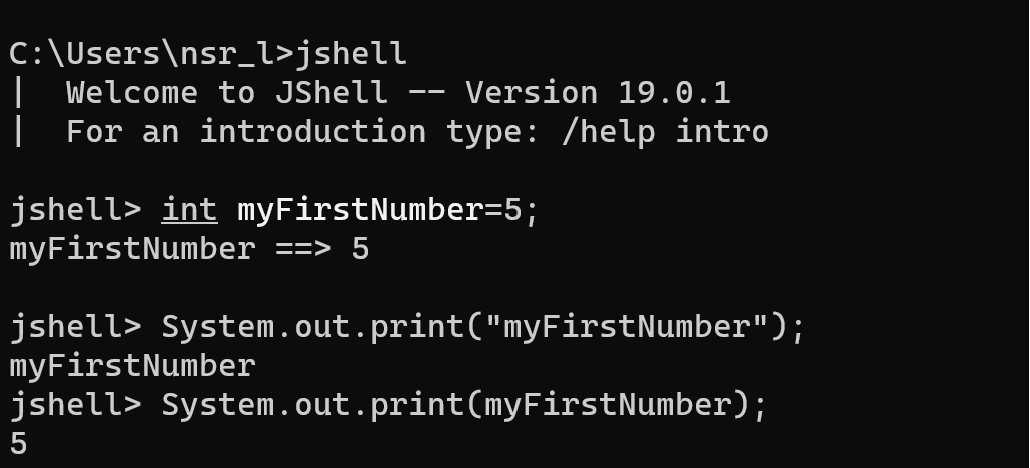
These are just some of the keywords in Java. It's essential to understand and use them correctly to write effective Java programs.

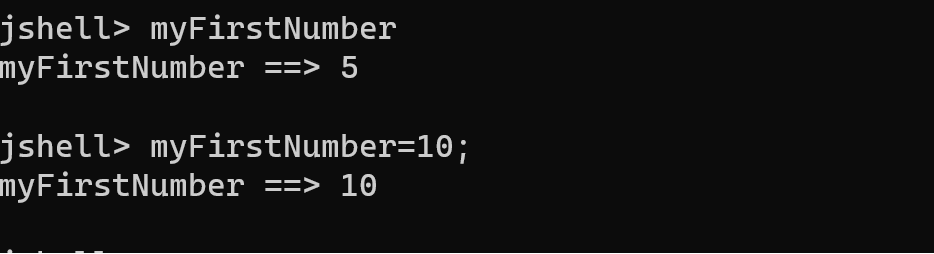
Here is a list of Java keywords:

1. **abstract**
2. **assert**
3. **boolean**
4. **break**
5. **byte**
6. **case**
7. **catch**
8. **char**
9. **class**
10. **const** (not used, reserved for future use)
11. **continue**
12. **default**
13. **do**
14. **double**
15. **else**
16. **enum**
17. **extends**
18. **final**
19. **finally**
20. **float**
21. **for**
22. **if**
23. **implements**
24. **import**
25. **instanceof**
26. **int**
27. **interface**
28. **long**
29. **native**
30. **new**
31. **package**
32. **private**
33. **protected**
34. **public**
35. **return**
36. **short**
37. **static**
38. **strictfp**
39. **super**
40. **switch**
41. **synchronized**
42. **this**
43. **throw**
44. **throws**
45. **transient**
46. **try**
47. **void**
48. **volatile**
49. **while**

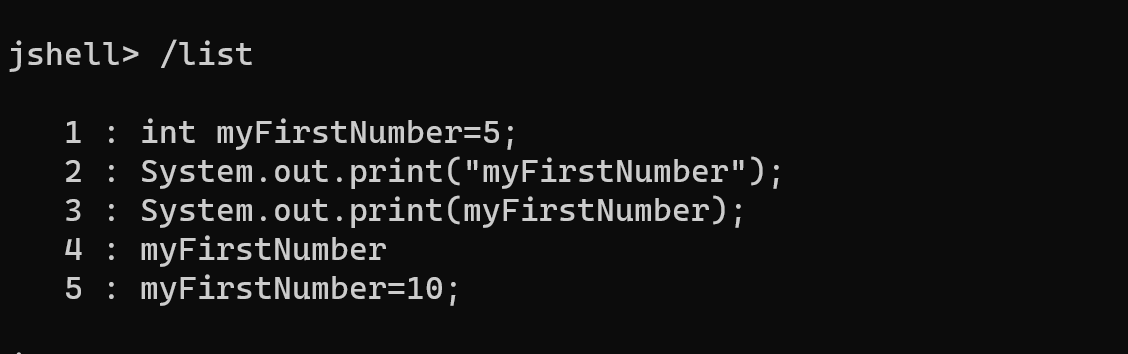
Note that **const** and **goto** are reserved keywords but not used in the language. They are reserved for potential future use. Additionally, starting from Java 9, **var** became a keyword for local variable type inference.

When we put something in double quotes it is **string literal**





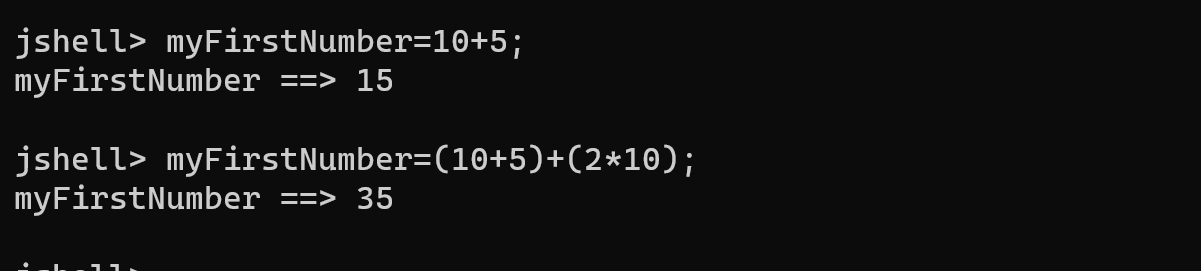
Command**: jshell>/list**



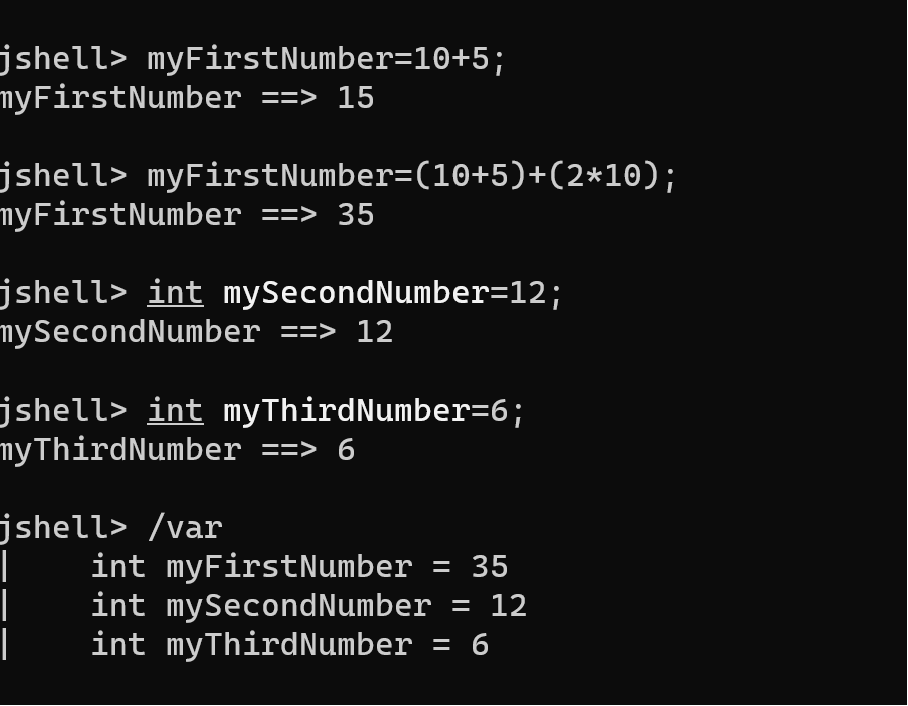
By declaring a variable again,we are efficiently re-declaring a variable,and in normal java programming,that would not be allowed ,and would throw an error

**Redeclaring a variable in java is not allowed**

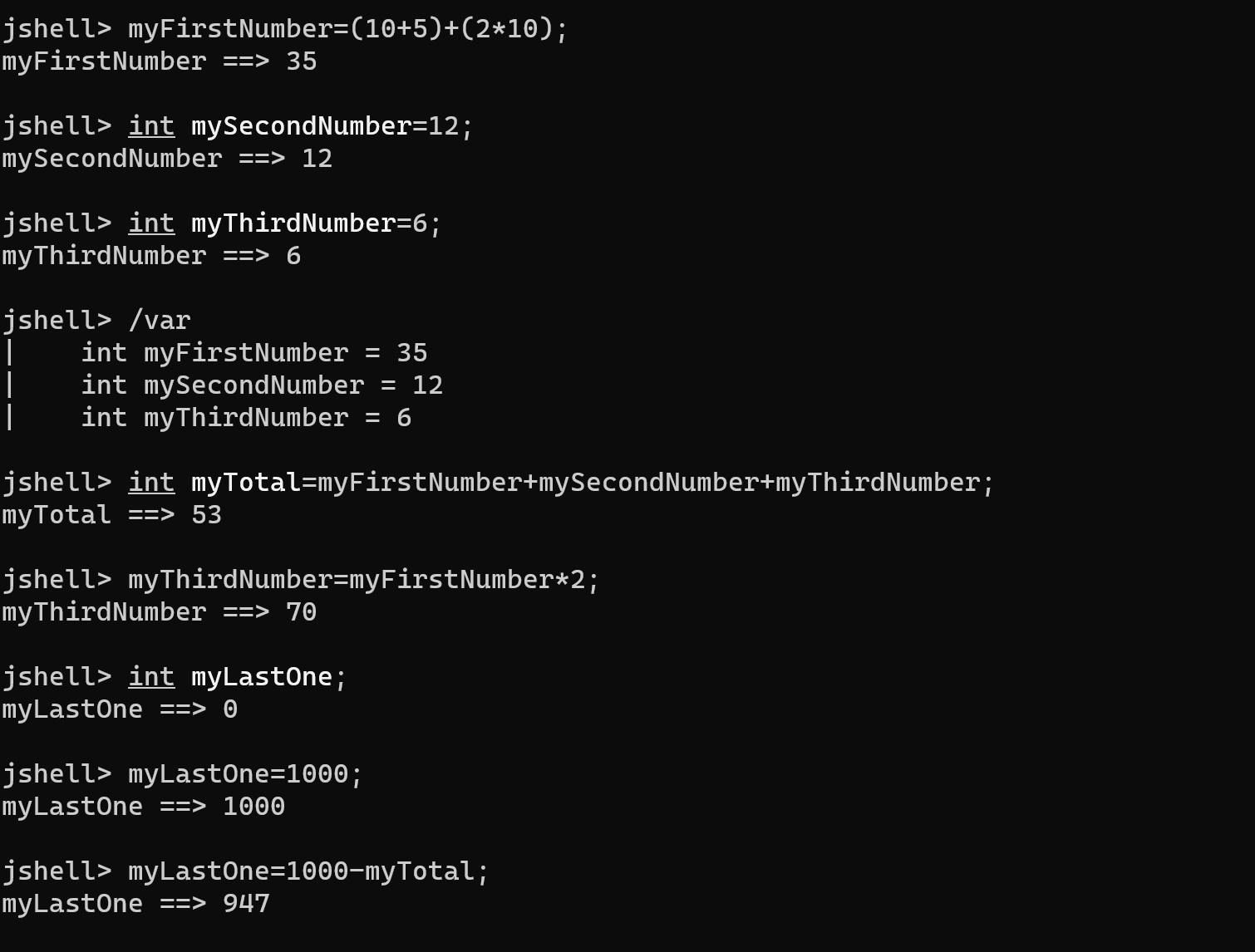
**Operators**



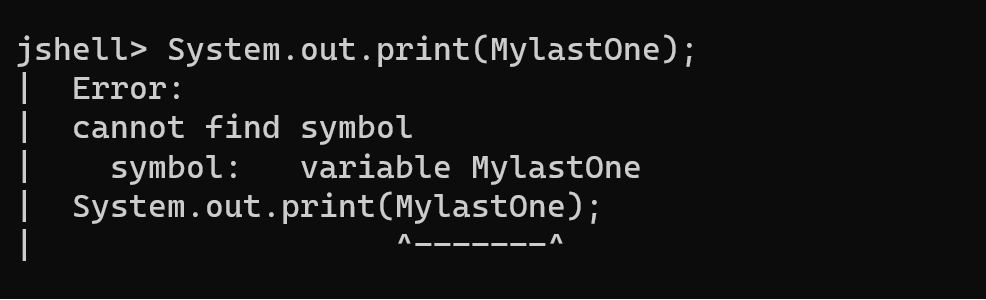
**=>Starting out with expressions**



Command:jshell>/var



Error:**3.cannot find symbol**



**Java code is case sensitive**

Int in lowercase,is not the same as Int with first letter capital or INT all in uppercase

Keywords need to be in lowercase

The **/vars** command in Jshell can you to identify any misspellings you have made.

**Primitive Types**

The eight primitive data types in Java are:

Byte

Short

Int

Long

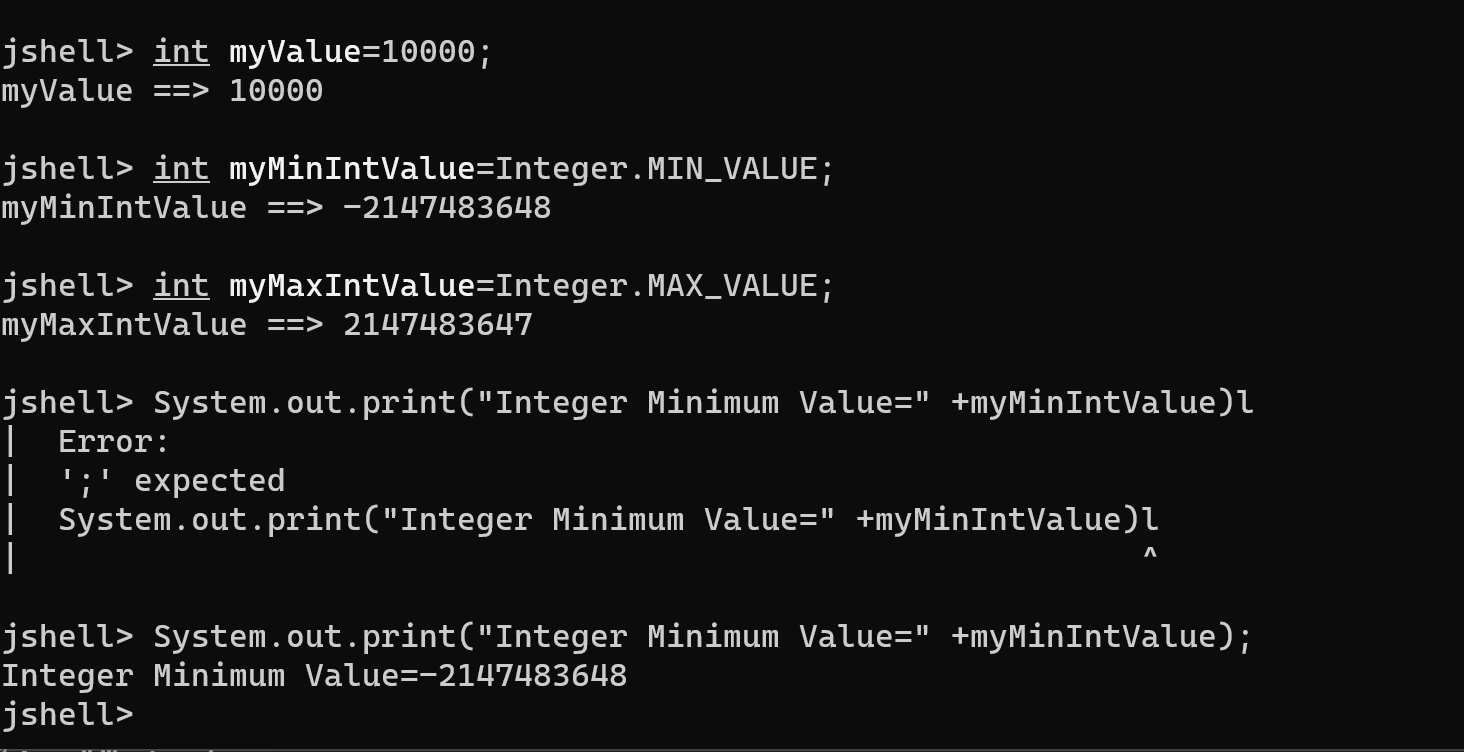
Float

Double

Char

Boolean

**Int**



**Using the + sign in System.out.print**

The plus sign,+,when used in System.out.print will print different data types togarther as a single line of text

System.out.print("Integer Minimum Value=" +myMinIntValue);

Whatever follows the plus sign in System.out.print here is converted to a String by Java and concatenated to String before it.

System.out.print(“Integer Value Range (“ +Integer.MIN\_VALUE+” to “ +Integer.MAX\_VALUE+”)”);

Integer Value Range(-2147483648 to 2147483647)

**What is a Class?**

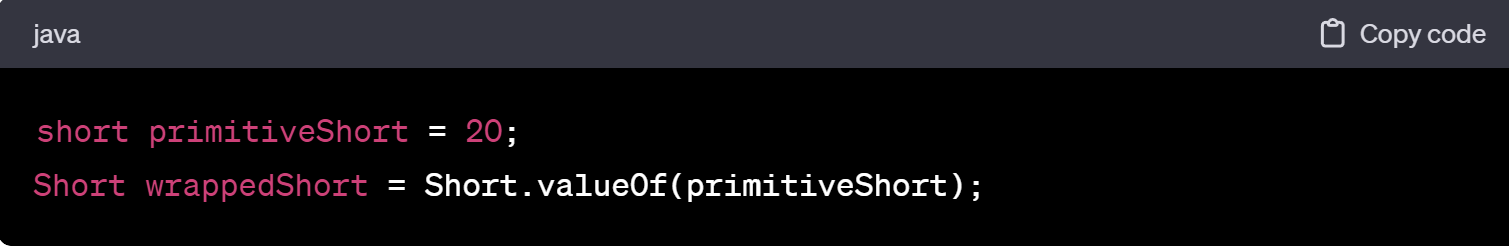
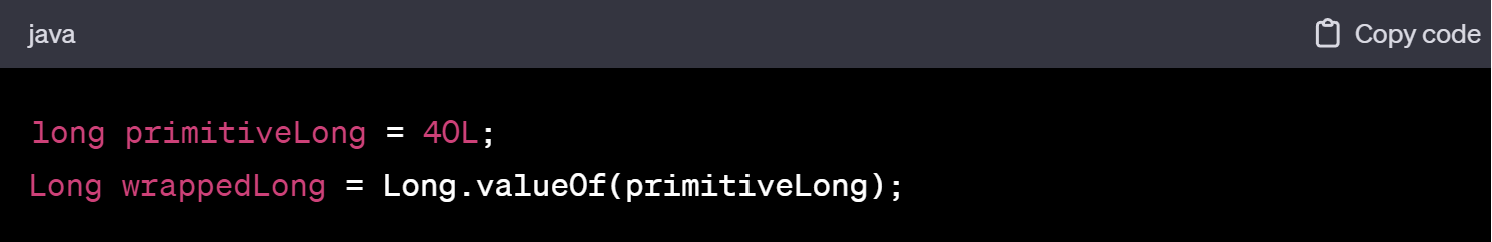
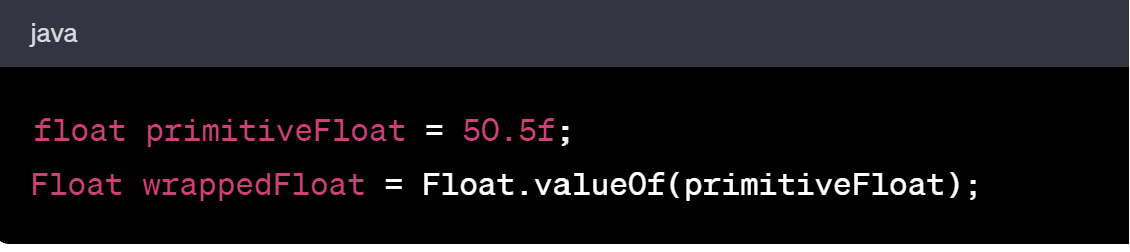
A class is a building block for object oriented programming and allows us to build custom data types.

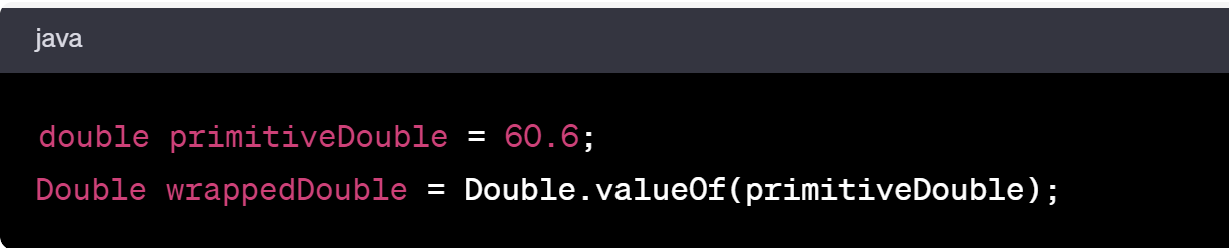
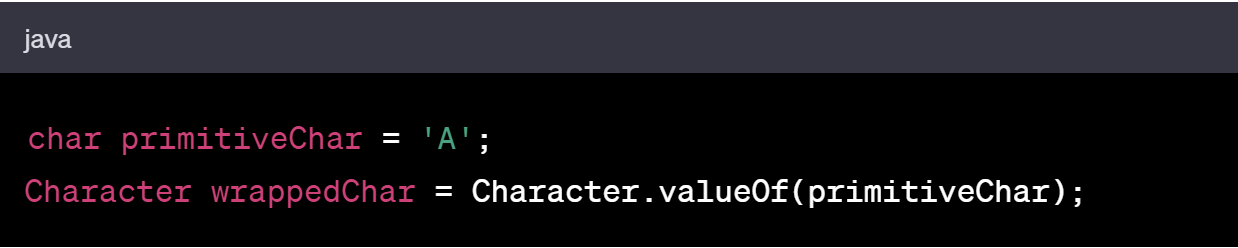
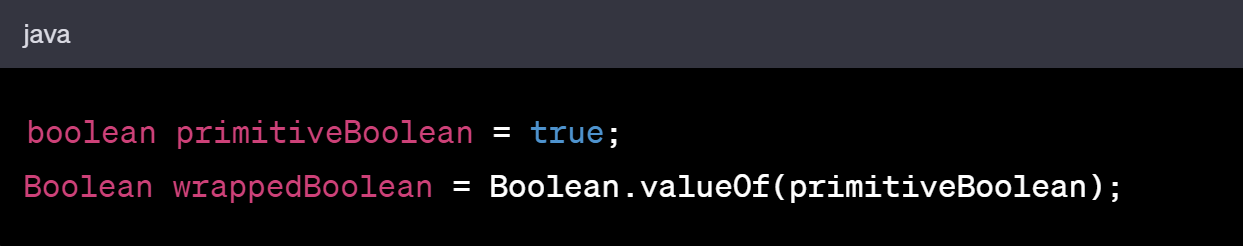
In Java, wrapper classes are used to represent primitive data types as objects. While Java primarily uses primitive data types (e.g., int, double, boolean) for efficiency, there are situations where an object representation is necessary. Wrapper classes provide a way to convert primitive data types into objects. The following are the wrapper classes in Java:

1. **Byte:**
   * Primitive type: byte
   * Wrapper class: **Byte**

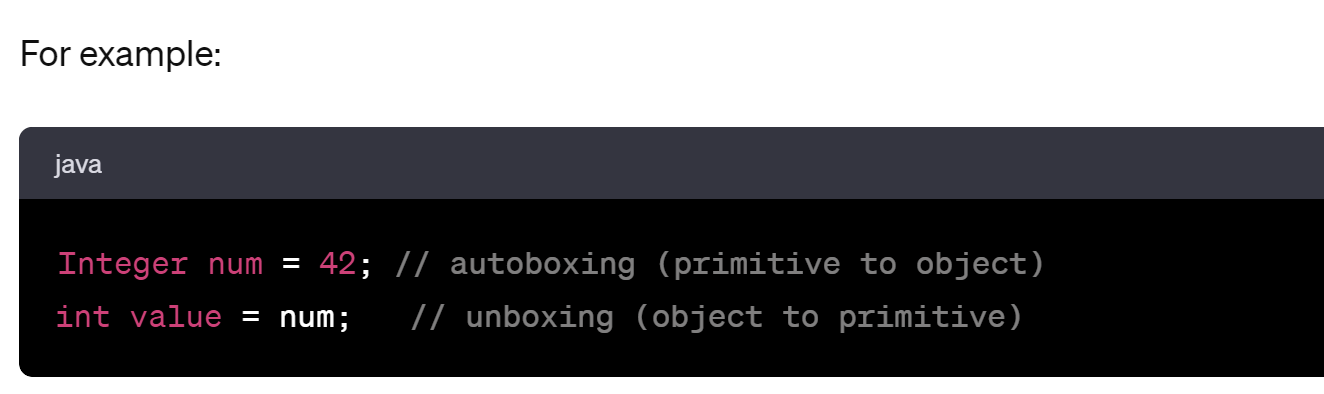
byte primitiveByte = 10;

* + Byte wrappedByte = Byte.valueOf(primitiveByte);

1. **Short:**
   * Primitive type: short
   * Wrapper class: **Short**
   * 
2. **Integer:**
   * Primitive type: int
   * Wrapper class: **Integer**
   * 
3. **Long:**
   * Primitive type: long
   * Wrapper class: **Long**
   * 
4. **Float:**
   * Primitive type: float
   * Wrapper class: **Float**
   * 

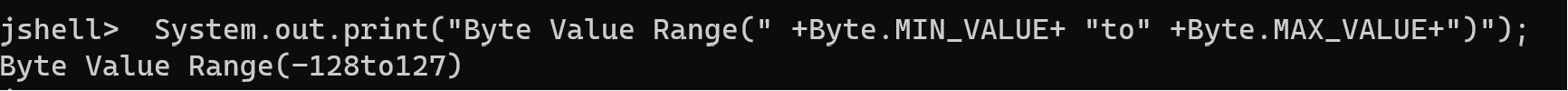
1. **Double:**
   * Primitive type: double
   * Wrapper class: **Double**
   * 
2. **Character:**
   * Primitive type: char
   * Wrapper class: **Character**
   * 
3. **Boolean:**
   * Primitive type: boolean
   * Wrapper class: **Boolean**
   * 

These wrapper classes provide methods to convert between primitive data types and their corresponding object representations. They also provide utility methods for tasks such as parsing and conversion. Additionally, Java autoboxing and unboxing allow automatic conversion between primitive types and their corresponding wrapper classes without explicit method calls.



These wrapper classes are particularly useful when working with collections (e.g., **List**, **Set**, **Map**) that require objects rather than primitives.

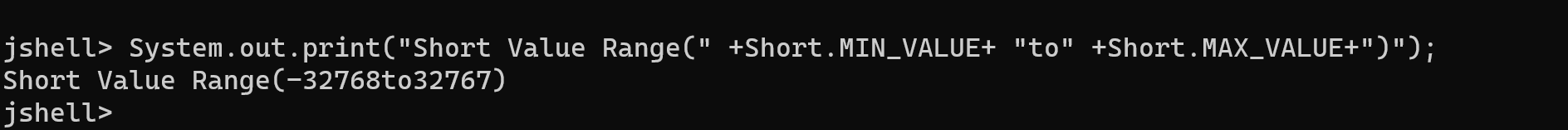
**Byte**



Byte Value Range:-128 to 127

The byte wrapper class is the byte with capital B

**Short**



Short Value Range: -32768 to 32767

The short wrapper class is the Short with a capital S.

In Java, the sizes of primitive data types are specified by the language and are not platform-dependent. The sizes are defined in the Java Language Specification (JLS). Here are the sizes of primitive data types in Java:

1. **Integral Types:**
   * **byte**: 8 bits
   * **short**: 16 bits
   * **int**: 32 bits
   * **long**: 64 bits
   * **char**: 16 bits
2. **Floating-Point Types:**
   * **float**: 32 bits
   * **double**: 64 bits
3. **Boolean Type:**
   * **boolean**: Not precisely defined; JVM-dependent, but often considered as one byte.

Keep in mind that the sizes of these data types are constant regardless of the platform or architecture.

Additionally, the "width" of a data type is often used to refer to the number of bits it occupies. In that context:

* **byte**, **char**, and **short** have a width of 8, 16, and 16 bits, respectively.
* **int** and **float** have a width of 32 bits.
* **long** and **double** have a width of 64 bits.

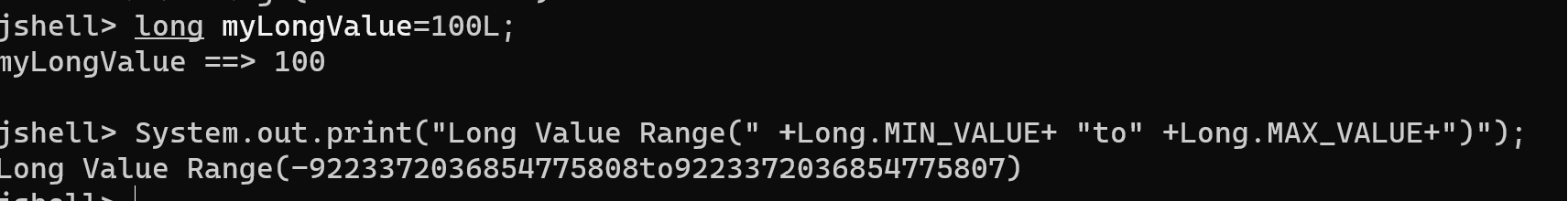
It's important to note that the actual storage size of these data types can be influenced by factors such as the JVM implementation and the use of Java Virtual Machine (JVM) options. However, the sizes mentioned here are the standard sizes defined by the Java language.

If you need to work with specific bit sizes or ensure a particular level of precision, you might need to use classes from the **java.nio** package, such as **ByteBuffer**, to handle data at the byte level.

**Using a numeric literal character suffix**

The number 100,by default is an int

The long is one of these types and it’s suffix is an ‘l’ or ‘L’

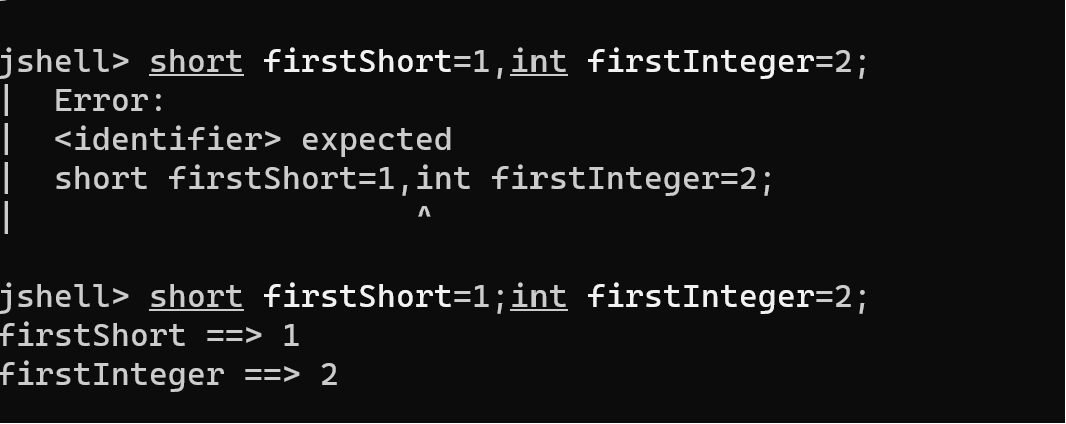


**=>Casting in Java**

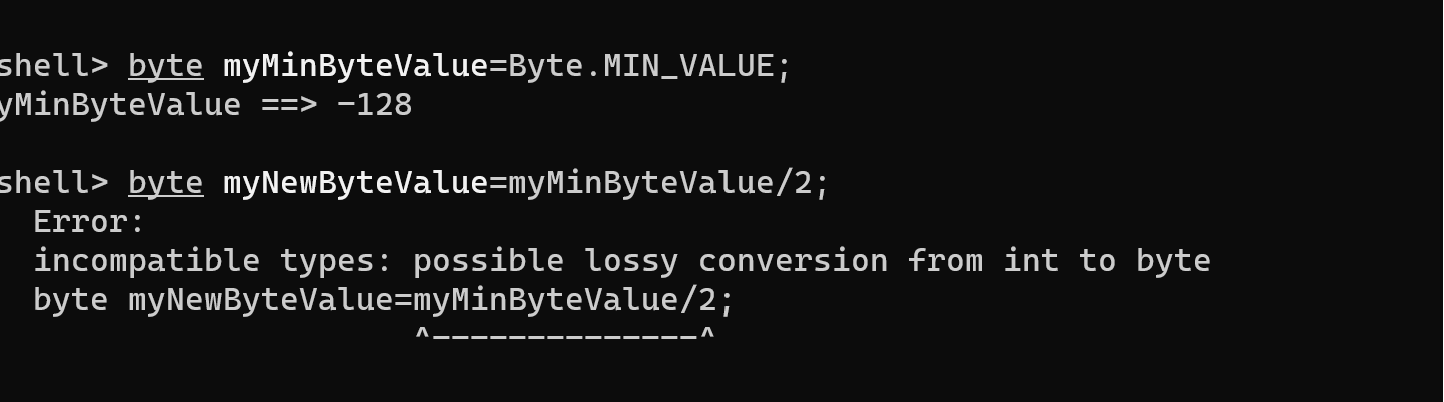
**Rules for declaring multiple variables in one statement**

If we want to declare multiple variables of same data type in a single statement,we need to specify the data type only once before any variable names

**Error 4:**

****

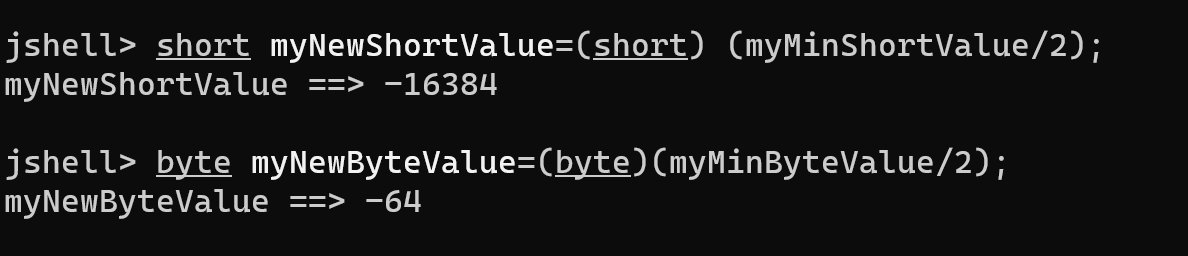
**Error 5:**

****

**Casting in Java**

Casting means to treat or convert a number, from one type to another. We put the type we want the number to be,in parenthesis like this

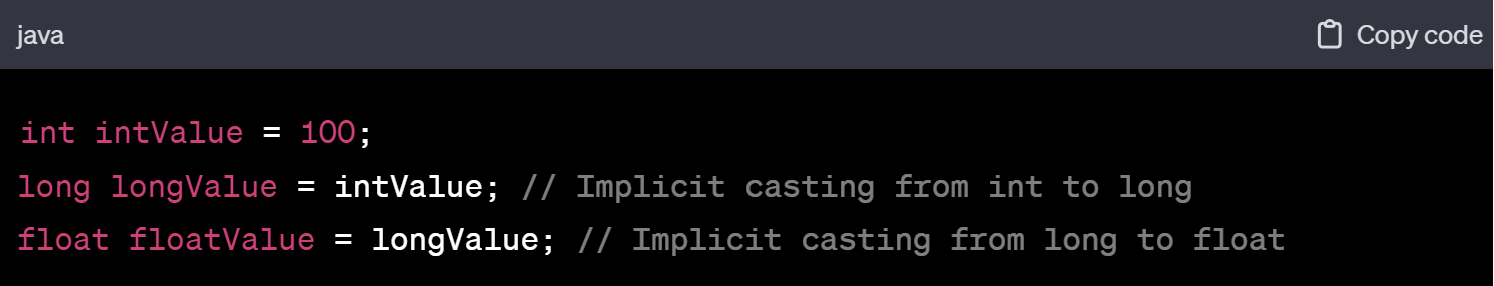
(byte)(myMinByteValue/2);



Casting in Java is the process of converting a value from one data type to another. There are two types of casting: implicit (automatic) casting and explicit (manual) casting.

**1. Implicit Casting (Widening):**

Implicit casting, also known as widening or automatic casting, occurs when you convert a smaller data type to a larger data type. Java performs implicit casting automatically when there is no loss of information.

from long to float

In the above example, the values are implicitly cast from **int** to **long**, and then from **long** to **float**. No explicit casting is required because widening conversions are done automatically.

**2. Explicit Casting (Narrowing):**

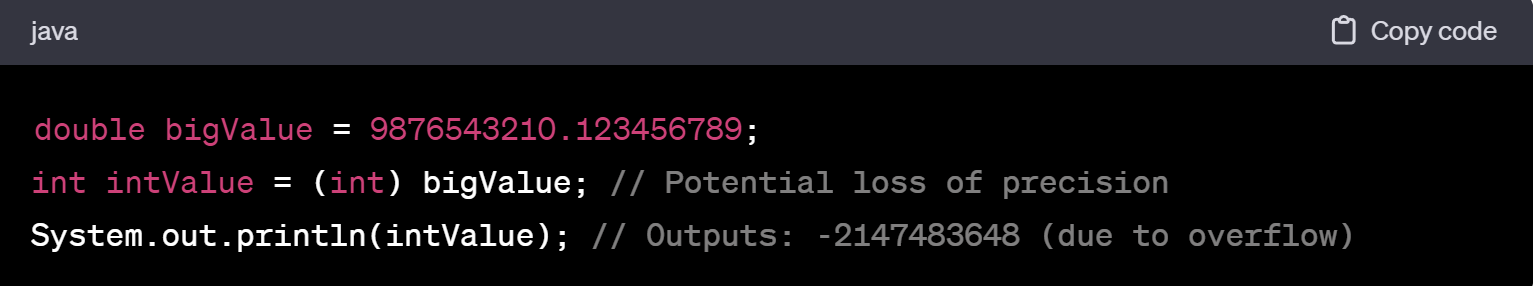
Explicit casting, also known as narrowing or manual casting, is required when you convert a larger data type to a smaller data type. This process may result in loss of information, and therefore, explicit casting is necessary to inform the compiler that you are aware of the potential loss.



In this example, the value is explicitly cast from **double** to **int**. Note the use of parentheses and the **(int)** syntax to indicate the explicit casting operation.

**Caution with Explicit Casting:**

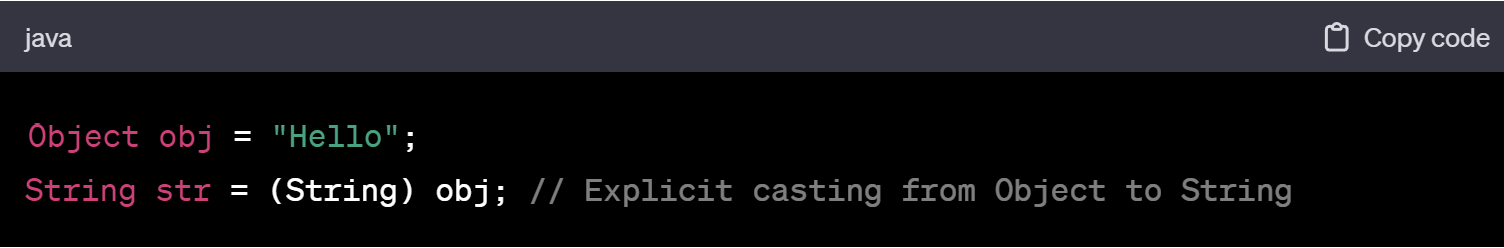
When narrowing, be aware that you might lose precision or encounter overflow issues, especially when converting from floating-point types to integers.



In the above example, the **double** value is explicitly cast to an **int**, resulting in a loss of precision and an overflow issue, leading to the minimum integer value.

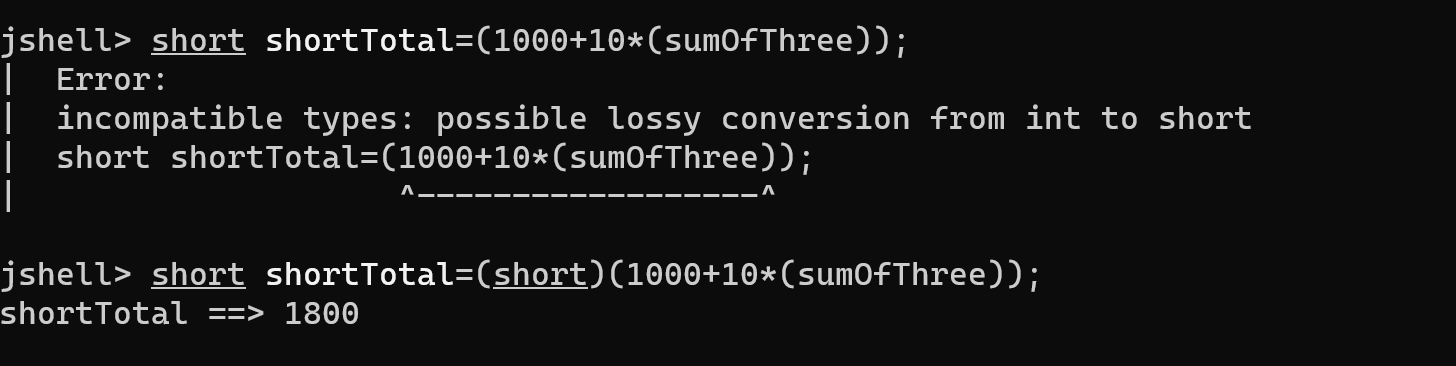
**Casting with Objects:**

For object types, casting is used to convert between different types of objects. This can be essential when working with inheritance and polymorphism.



Here, the **Object** reference is explicitly cast to a **String** reference.

Remember to use casting judiciously and be aware of potential issues, especially when narrowing, to avoid unintended consequences.



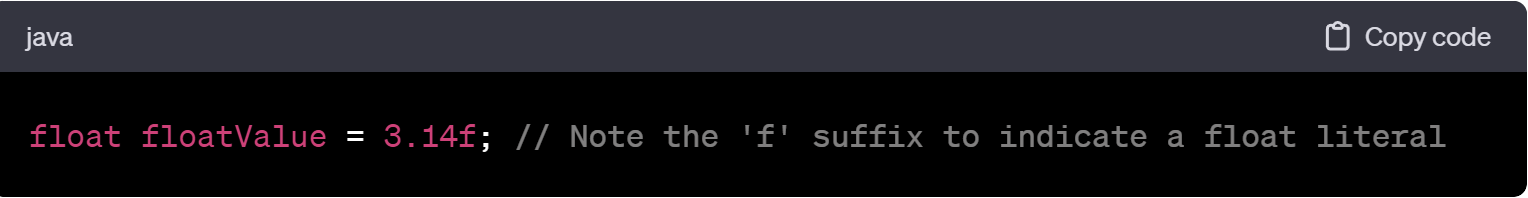
**Float and Double Primitives in Java**

In Java, **float** and **double** are primitive data types used to represent floating-point numbers, which are numbers that have a fractional part. Here are the key characteristics of **float** and **double**:

**1. float:**

* **Size:** 32 bits (4 bytes)
* **Range:** Float Value Range: 1.4E-45 to 3.4028235E38
* **Precision:** 7 decimal digits

Example:

the 'f' suffix to indicate a float literal

**2. double:**

* **Size:** 64 bits (8 bytes)
* Double Value Range:4.9E-324 to 1.7976931348623157E308
* **Precision:** 15 decimal digits



**Choosing Between float and double:**

* **double** is the default choice for representing floating-point numbers in Java.
* Use **float** only when memory is a critical concern (e.g., in large arrays) or if you are working with a system or library that specifically uses **float**.
* Operations involving **double** are generally faster than those involving **float** on modern hardware.

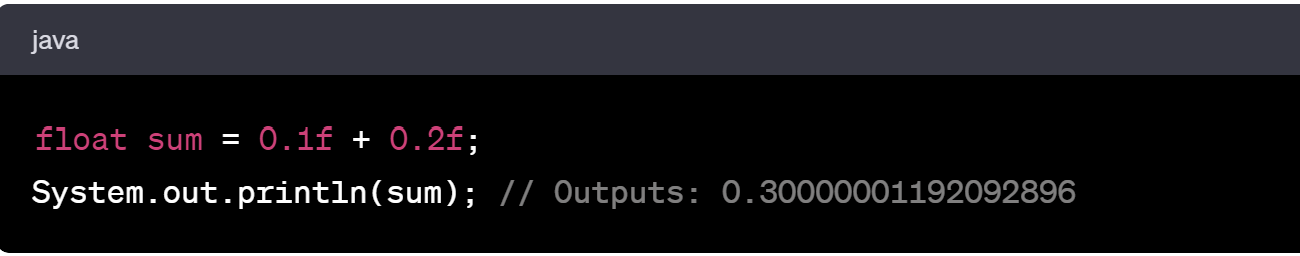
**Suffixes:**

* When using literals, it's common to include a suffix to indicate the data type explicitly.
* For **float**, use **f** or **F** (e.g., **3.14f**).
* For **double**, no suffix is needed, but you can use **d** or **D** if you prefer (e.g., **3.14** or **3.14d**).

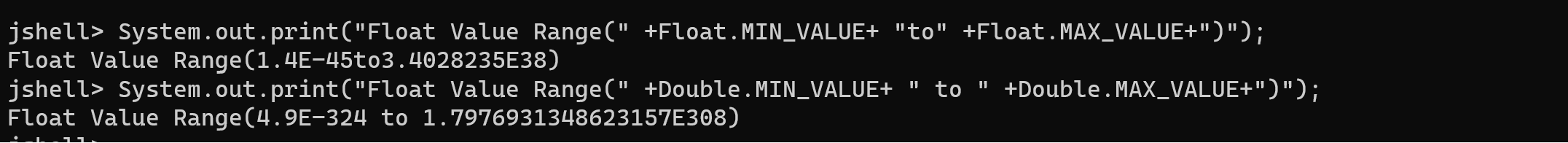
**Floating-Point Precision Issues:**

* Floating-point numbers have limited precision due to the finite number of bits available for representation.
* This can lead to rounding errors and unexpected behavior, especially when performing operations that involve a large number of calculations.
* Consider using **BigDecimal** for precise arithmetic in situations where exact decimal representation is crucial.

Example of Precision Issue:



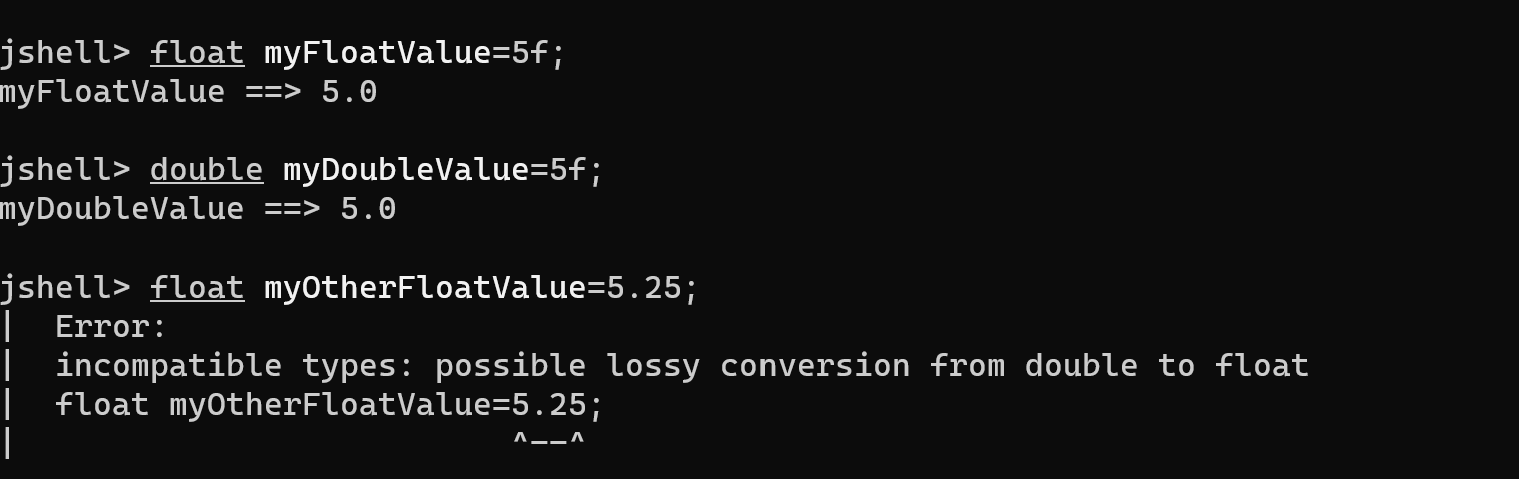
In this example, the limited precision of **float** results in a small rounding error. If precise decimal representation is essential, consider using **double** or **BigDecimal** depending on your requirements.

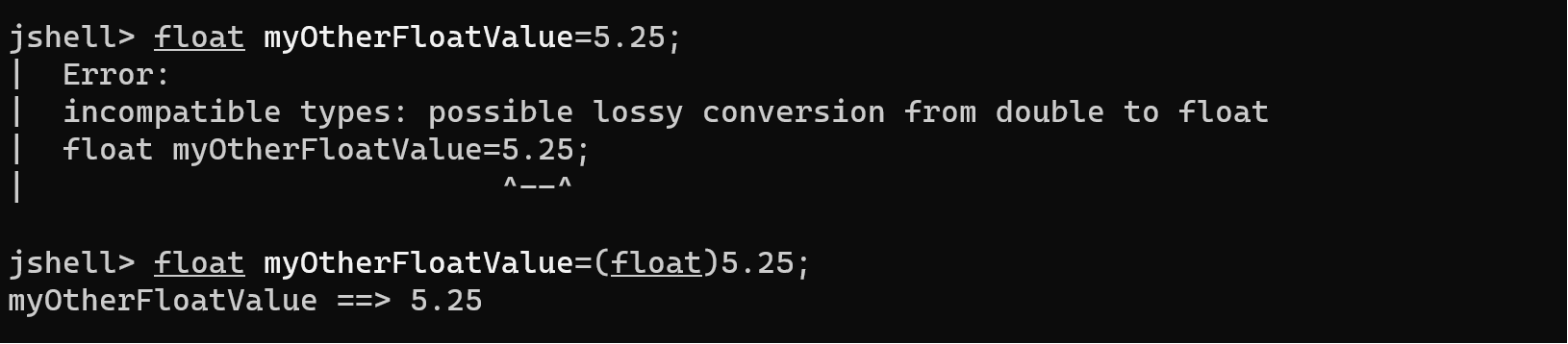


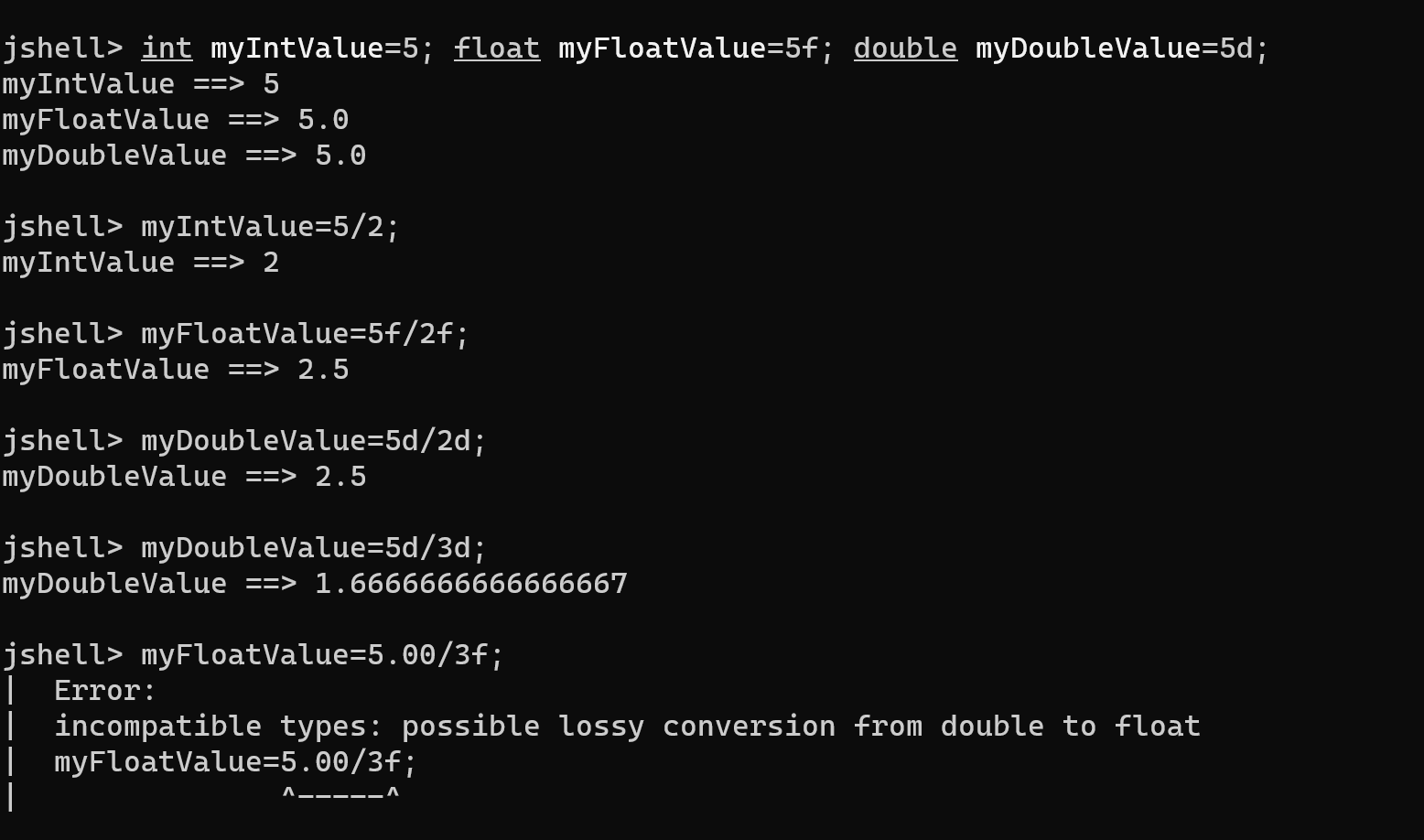
**Float and double and numeric literal suffixes**

The double data type can be specified as a numeric literal with a suffix of either lowercase’d’ or uppercase ‘D’, but because doubles are the default in Java

The float data type can be specified as a numeric literal with a suffix of either lowercase ’f’ or uppercase ‘F’, but because float are the default in Java







**The char and Boolean primitive data types**

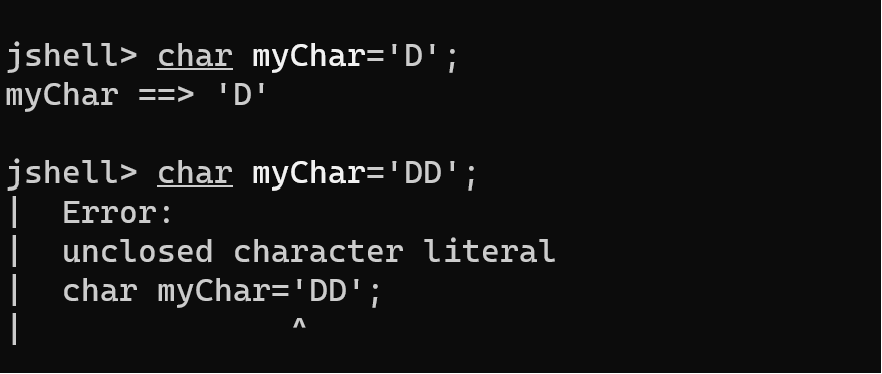
A char occupies two bytes of memory,or 16bits,and thus has a width of 16

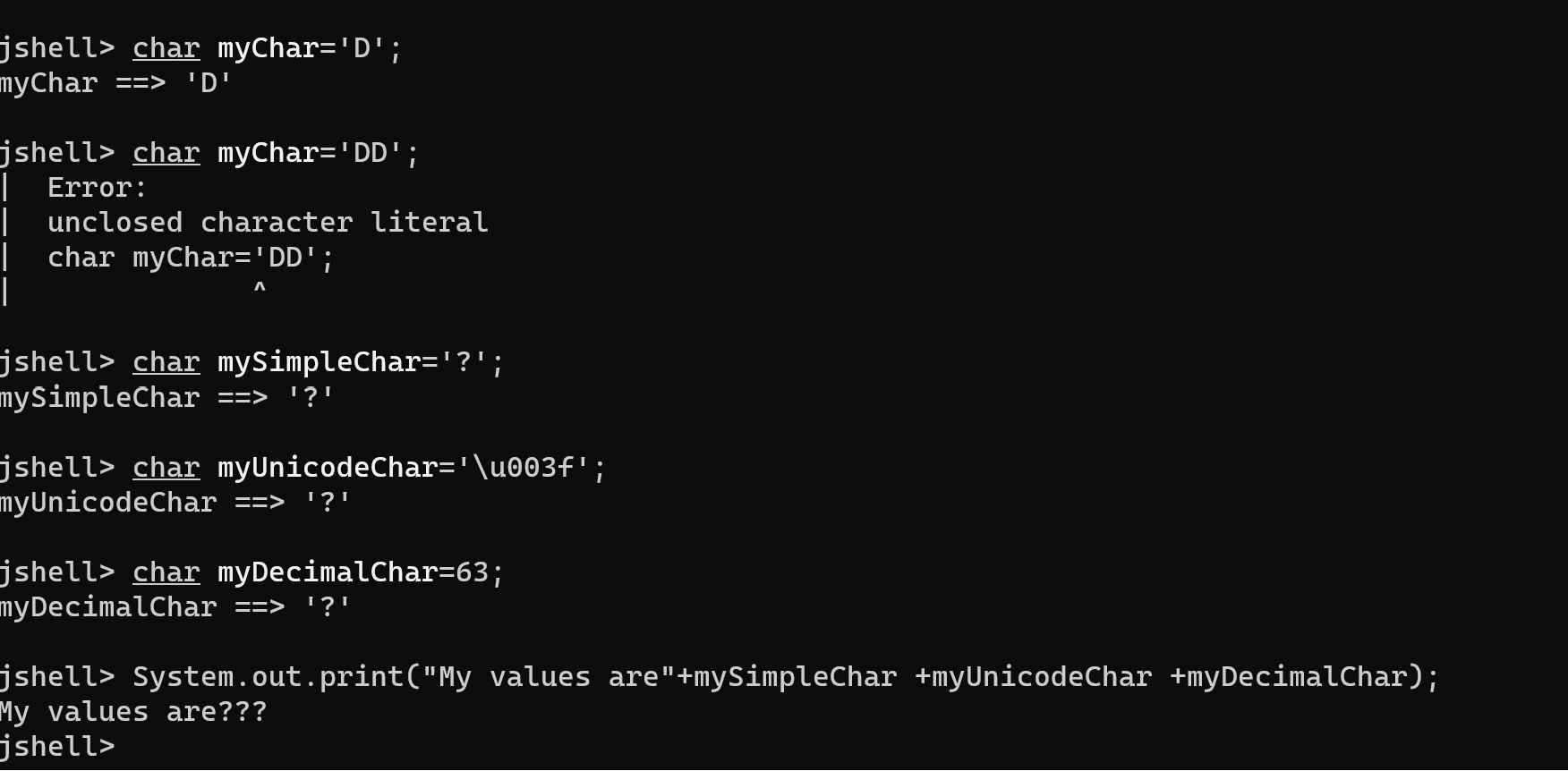
You can use single quotes and a character literal to assign a value to a char

Unicode

Unicode is an international encoding standard for use with different languages and scripts by which each letter,digit or symbol is assigned a unique numeric value that applies across different platforms and programs

**Error 5:**





**Java’s 8 Primitive Data types**

**Whole number**

Byte

Short

Int

Long

**Real Number(floating point or decimal)**

Float

Double

**Single character**

Char

**Boolean value**

Bool

**Java’s Buit-in Classes**

Wrappers(Boolean,Byte,Character,etc.)

BigDecimal

String

String is a class that contains a sequence of characters

**Executing multiple lines of code in Jshell**

jshell > { //start with a curly opening brace

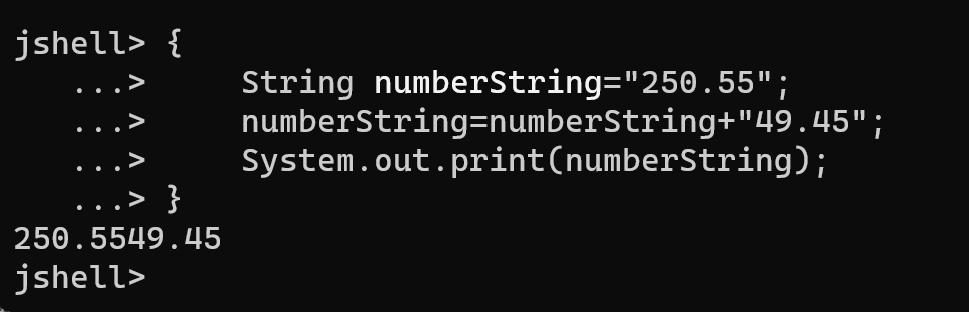
first statement;

second statement;

third statement;

} //end with a curly closing brace

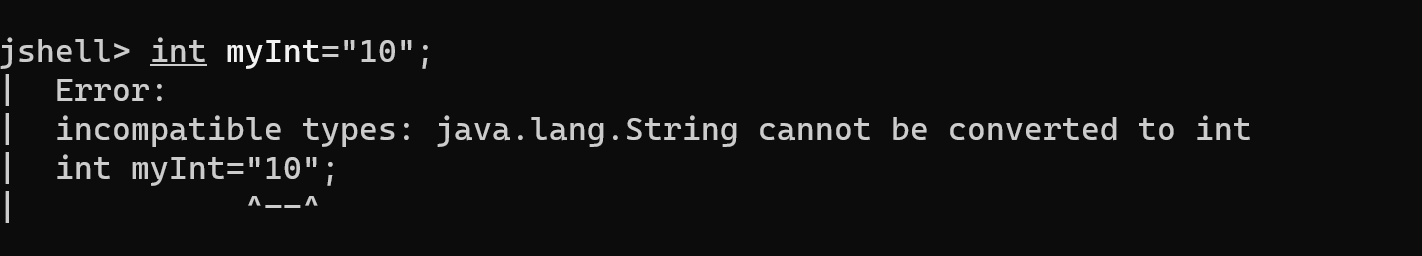
Jshell will display an alternate prompt as you can see ,three dots and a greater than sign.



There are two ways to execute multiple statements in JShell:

* Put your statements on a single line
* Or,enclose your statements in a set of curly braces{}

Error 6:



**String concatenation**

In Java,the + symbol is an operator which mean addition,if used for numbers

Strings are immutable

Immutable means you can’t change the string after it is created

**String Vs StringBuilder**

The String class is immutable,but can be used much like primitive data type.

The StringBuilder class is mutable,but doesn’t share the string’s special features,such as being able to assign it a String literal or use the + operator on it.

Both are classes,but the String Class is in a special category in the Java language.

**Operators,Operands and Expressions**

Operators in java are special symbols that perform specific operations on one,two,or three operands,and then return a result.

Eg:

Long longTotal=5000L+ 10L\*(byteValue+shortValue+intValue);

Operand

Expression is formed by combining variables,literals,method return values and operators

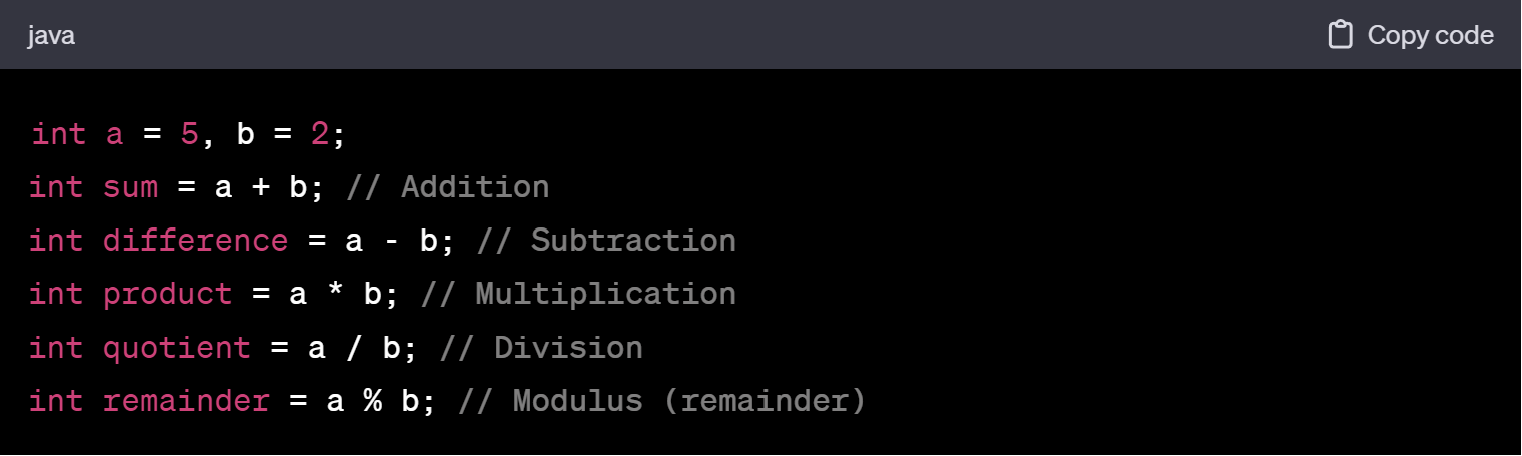
Comments are ignored by the computer,and are added to a program,to help describe something.

Eg://1+2=3

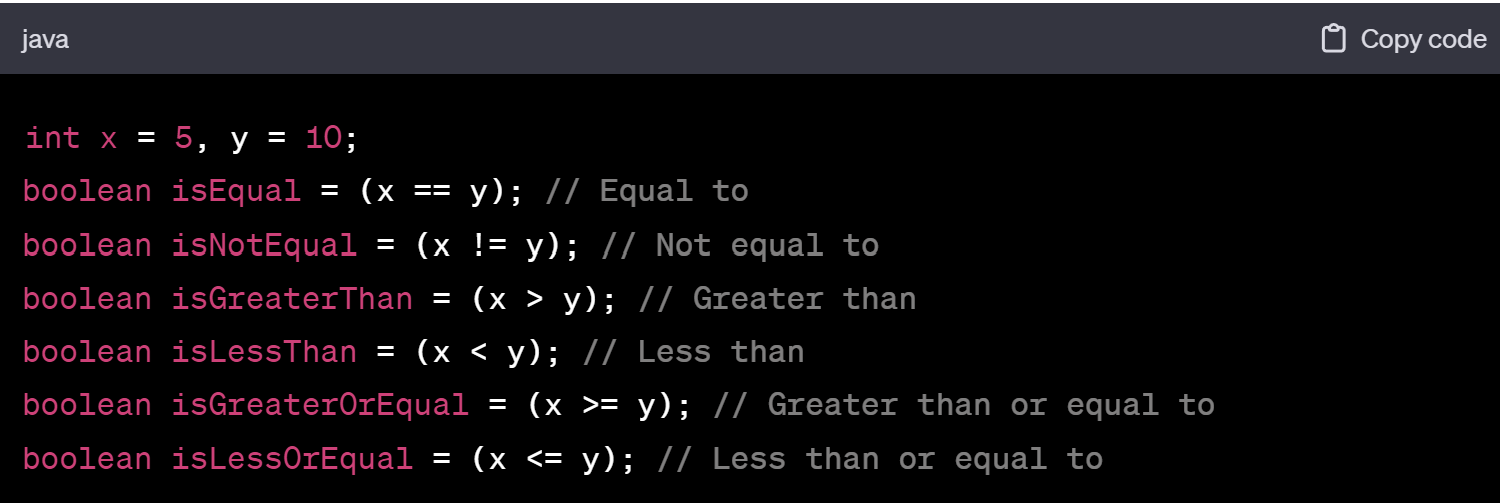
In Java, operators are special symbols or keywords used to perform operations on operands. Operands can be variables, constants, or expressions. Java supports a wide range of operators, which can be classified into several categories:

**1. Arithmetic Operators:**

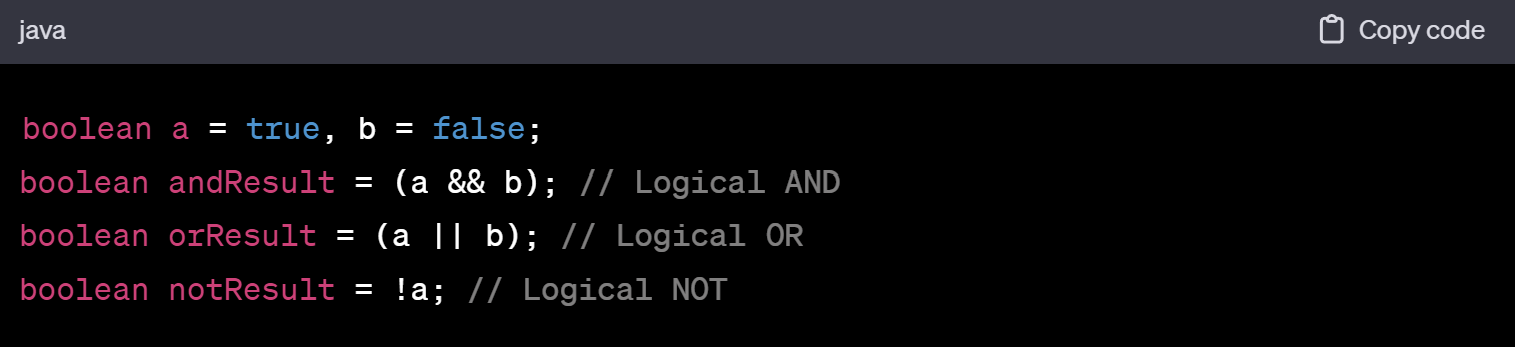
* Used to perform basic mathematical operations.



**2. Comparison Operators:**

* Used to compare two values and return a boolean result.
* 

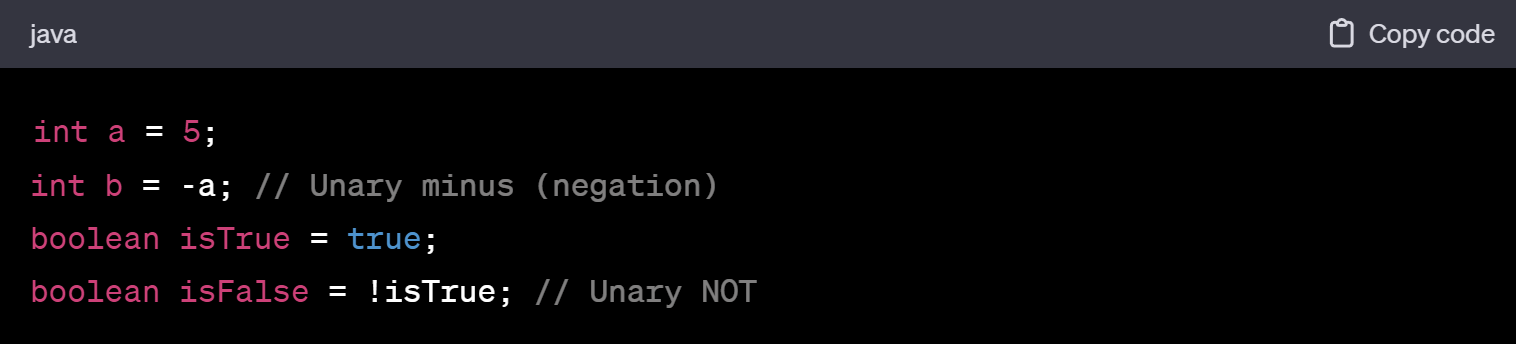
**3. Logical Operators:**

* Used to perform logical operations and return a boolean result.
* 

**4. Assignment Operators:**

* Used to assign values to variables.
* 

**5. Unary Operators:**

* Operate on a single operand.
* 

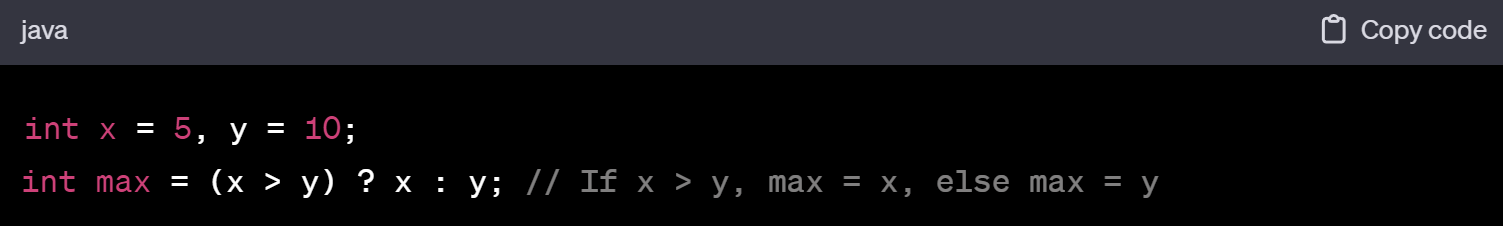
**6. Increment and Decrement Operators:**

* Used to increase or decrease the value of a variable by 1.



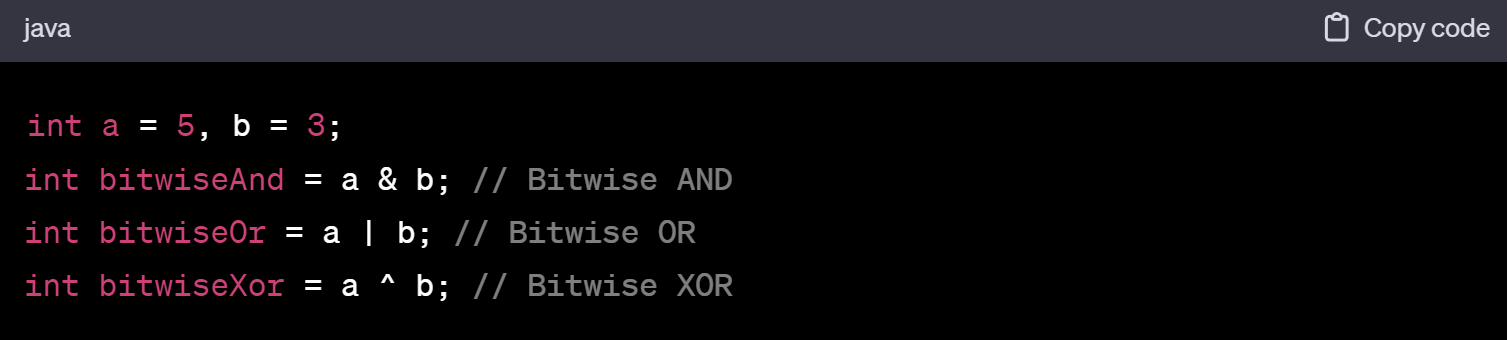
**7. Conditional (Ternary) Operator:**

* Provides a concise way to implement a simple conditional expression.



**8. Bitwise Operators:**

* Perform operations at the bit level.

These are some of the fundamental operators in Java. Understanding how to use them is crucial for writing effective and efficient code.

Top of Form