Variables and Datatypes

Faculty of Information Technology, Hanoi University

What You Are About To Achieve

❖ By the end of this lecture, we are going to:

- ❑ **Understand** the process of declaring, initializing, and assigning values to variables.
- ❑ **Explore** the various numeric data types and their uses.
- ❑ **Apply** coding conventions to maintain readability and consistency in code.
- ❑ **Utilize** operators effectively in expressions and calculations.
- ❑ **Perform** numeric type conversions.
- ❑ **Identify** common errors and pitfalls.
- ❑ **Outline** the software development process.

- **Variable**
- Numeric Datatypes
- Coding Conventions
- Operators
- Numeric Type Conversions
- Software Development Process
- Common Errors and Pitfalls
- Final Touches

Variables

- In our previous lecture, we wrote some basic programs to perform tasks like population projection and solving algebraic equations.

```
3 public class Algebra {
4     public static void main(String[] args) {
5         System.out.println("3.4x + 50.2y = 44.5");
6         System.out.println("2.1x + 0.55y = 5.9");
7         System.out.println("The result is: ");
8         System.out.println("x = " + (44.5 * 0.55 - 50.2 * 5.9) / (3.4 * 0.55 - 50.2 * 2.1));
9         System.out.println("y = " + (3.4 * 5.9 - 44.5 * 2.1) / (3.4 * 0.55 - 50.2 * 2.1));
10    }
11 }
```

```
3 public class PopulationProjection {
4     public static void main(String[] args) {
5         System.out.println("Population after 1 year: "
6                 + (312032486 + 365 * 24 * 60 * 60 / 7 - 365 * 24 * 60 * 60 / 13
7         System.out.println("Population after 2 years: "
8                 + (312032486 + 365 * 2 * 24 * 60 * 60 / 7 - 365 * 2 * 24 * 60 *
9         System.out.println("Population after 3 years: "
10                + (312032486 + 365 * 3 * 24 * 60 * 60 / 7 - 365 * 3 * 24 * 60 *
11        System.out.println("Population after 4 years: "
12                + (312032486 + 365 * 4 * 24 * 60 * 60 / 7 - 365 * 4 * 24 * 60 * 60 / 13 + 365 * 4 * 24 * 60 * 60 / 4
13        System.out.println("Population after 5 years: "
14                + (312032486 + 365 * 5 * 24 * 60 * 60 / 7 - 365 * 5 * 24 * 60 * 60 / 13 + 365 * 5 * 24 * 60 * 60 / 45))
15    }
16 }
```

Variables


```
3 public class PopulationProjection {
4     public static void main(String[] args) {
5         System.out.println("Population after 1 year: "
6                 + (312032486 + 365 * 24 * 60 * 60 / 7 - 365 * 24 * 60 * 60 / 13 + 365 * 24 * 60 * 60 / 45));
7         System.out.println("Population after 2 years: "
8                 + (312032486 + 365 * 2 * 24 * 60 * 60 / 7 - 365 * 2 * 24 * 60 * 60 / 13 + 365 * 2 * 24 * 60 * 60 / 45));
9         System.out.println("Population after 3 years: "
10                + (312032486 + 365 * 3 * 24 * 60 * 60 / 7 - 365 * 3 * 24 * 60 * 60 / 13 + 365 * 3 * 24 * 60 * 60 / 45));
11        System.out.println("Population after 4 years: "
12                + (312032486 + 365 * 4 * 24 * 60 * 60 / 7 - 365 * 4 * 24 * 60 * 60 / 13 + 365 * 4 * 24 * 60 * 60 / 45));
13        System.out.println("Population after 5 years: "
14                + (312032486 + 365 * 5 * 24 * 60 * 60 / 7 - 365 * 5 * 24 * 60 * 60 / 13 + 365 * 5 * 24 * 60 * 60 / 45));
15    }
16 }
```

For example, we had to manually enter the population values or equation directly into the code. This approach is not flexible - it requires us to change the code each time we want to use different data.

Variables


```
3 public class PopulationProjection {
4     public static void main(String[] args) {
5         System.out.println("Population after 1 year: "
6                 + (312032486 + 365 * 24 * 60 * 60 / 7 - 365 * 24 * 60 * 60 / 13 + 365 * 24 * 60 * 60 / 45));
7         System.out.println("Population after 2 years: "
8                 + (312032486 + 365 * 2 * 24 * 60 * 60 / 7 - 365 * 2 * 24 * 60 * 60 / 13 + 365 * 2 * 24 * 60 * 60 / 45));
9         System.out.println("Population after 3 years: "
10                + (312032486 + 365 * 3 * 24 * 60 * 60 / 7 - 365 * 3 * 24 * 60 * 60 / 13 + 365 * 3 * 24 * 60 * 60 / 45));
11        System.out.println("Population after 4 years: "
12                + (312032486 + 365 * 4 * 24 * 60 * 60 / 7 - 365 * 4 * 24 * 60 * 60 / 13 + 365 * 4 * 24 * 60 * 60 / 45));
13        System.out.println("Population after 5 years: "
14                + (312032486 + 365 * 5 * 24 * 60 * 60 / 7 - 365 * 5 * 24 * 60 * 60 / 13 + 365 * 5 * 24 * 60 * 60 / 45));
15    }
16 }
```

The long and complex expressions made the code harder to read and understand. Also, the code was quite repetitive. We had to write similar lines of code over and over again, which isn't efficient or easy to maintain.

To overcome these challenges, we'll dive into elementary programming concepts, including primitive data types, variables, constants, operators, expressions, and input/output handling.

Variables

❖ The algorithm for calculating the area of a circle can be described as follows:

1. Read in the circle's radius.
2. Compute the area using the following formula:

   area = radius * radius * p
3. Display the result.

Let's first consider the simple problem of computing the area of a circle. How do we write a program for solving this problem?

Variables

❖ You already know that every Java program begins with a class definition in which the keyword **class** is followed by the class name. Assume that you have chosen **ComputeArea** as the class name. The outline of the program would look like this:

```
public class ComputeArea {
  public static void main(String[] args) {
    // Step 1: Read in radius

    // Step 2: Compute area

    // Step 3: Display the area
  }
}
```

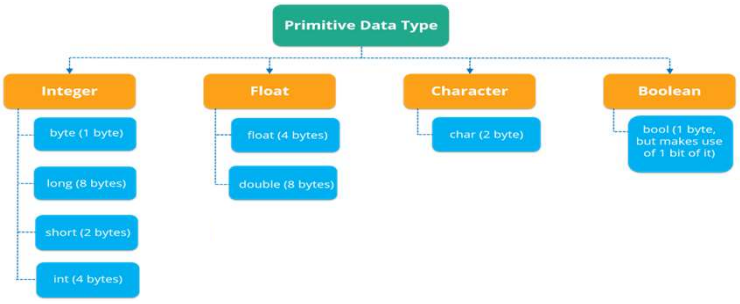
Look at step 1, the program needs to read the radius entered by the user from the keyboard. This raises two important issues:

- ❑ **Reading the radius.**
- ❑ **Storing the radius in the program.**

Variables

❖ Let's address the second issue first. In order to store the radius, the program needs to declare a symbol called a *variable*. A variable represents a value stored in the computer's memory.

❖ Rather than using **x** and **y** as variable names, please choose descriptive names: in this case, **radius** for radius, and **area** for area. To let the compiler know what **radius** and **area** are, just specify their data types. That is the kind of data stored in a variable, whether integer, real number, or something else. This is known as *declaring variables*. Java provides simple data types for representing integers, real numbers, characters, and Boolean types. These types are known as *primitive data types* or *fundamental types*.



Variables

- For now, to learn how variables work, you can assign a fixed value to **radius** in the program as you write the code; later, you'll modify the program to prompt the user for this value.
- The second step is to compute **area** by assigning the result of the expression **radius * radius * 3.14159** to **area**.
- In the final step, the program will display the value of **area** on the console by using the **System.out.println** method.

```
2
3 public class ComputeArea {
4     public static void main(String[] args) {
5         double radius; // Declare radius
6         double area; // Declare area
7
8         // Assign a radius
9         radius = 20; // New value is radius
10
11        // Compute area
12        area = radius * radius * 3.14159;
13
14        // Display results
15        System.out.println("The area for the circle of radius " + radius + " is " + area);
16    }
17 }
```

```
Problems  Javadoc  Declaration  Console
<terminated> ComputeArea [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Sep 2, 2024, 10:26:19 AM – 10:26:19 AM) [pid: 22212]
The area for the circle of radius 20.0 is 1256.636
```

Let's see how it works!

Variables


```
2
3 public class ComputeArea {
4     public static void main(String[] args) {
5         double radius; // Declare radius
6         double area; // Declare area
7
8         // Assign a radius
9         radius = 20; // New value is radius
10
11        // Compute area
12        area = radius * radius * 3.14159;
13
14        // Display results
15        System.out.println("The area for the circle of radius " + radius + " is " + area);
16    }
17 }
```

```
Problems  Javadoc  Declaration  Console
<terminated> ComputeArea [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Sep 2, 2024, 10:26:19 AM – 10:26:19 AM) [pid: 22212]
The area for the circle of radius 20.0 is 1256.636
```

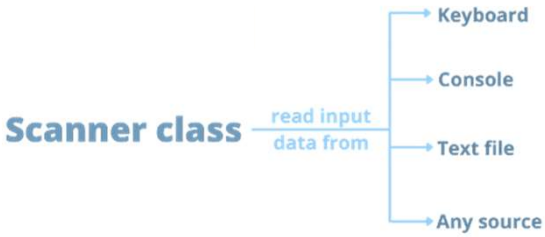
line#	radius	area
5	no value	
6		no value
9	20	
12		1256.636

- Variables such as **radius** and **area** correspond to memory locations. Every variable has a name, a type, a size, and a value.
- Line 5 declares that **radius** can store a **double** value. The value is not defined until you assign a value. Line 9 assigns **20** into variable **radius**.
- Similarly, line 6 declares variable **area**, and line 12 assigns a value into **area**.

Variables - Reading Input From the Console

❖ In this program, the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. Obviously, this is not convenient, so instead you can use the **Scanner** class for console input.

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



Variables - Reading Input From the Console

❖ You can invoke the **nextDouble()** method to read a **double** value as follows:

```
double radius = input.nextDouble();
```

❖ This statement reads a number from the keyboard and assigns the number to **radius**. The program become:

```
3  import java.util.Scanner; // Scanner is in the java.util package
4
5  public class ComputeAreaWithConsoleInput {
6      public static void main(String[] args) {
7          // Create a Scanner object
8          Scanner input = new Scanner(System.in);
9
10         // Prompt the user to enter a radius
11         System.out.print("Enter a number for radius: ");
12         double radius = input.nextDouble();
13
14         // Compute area
15         double area = radius * radius * 3.14159;
16
17         // Display result
18         System.out.println("The area for the circle of radius " + radius + " is " + area);
19     }
20 }
```

```
Problems  Javadoc  Declaration  Console
<terminated> ComputeAreaWithConsoleInput [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Sep 2, 2024, 10:47:08 AM – 10:47:17 AM) [pid: 23932]
Enter a number for radius: 2.5
The area for the circle of radius 2.5 is 19.6349375
```

Variables

❖ Recall the Population Project:

```
public class PopulationProjection {
    public static void main(String[] args) {
        System.out.println("Population after 1 year: "
                + (312032486 + 365 * 24 * 60 * 60 / 7 - 365 * 24 * 60 * 60 / 13 + 365 * 24 * 60 * 60 / 45));
        System.out.println("Population after 2 years: "
                + (312032486 + 365 * 2 * 24 * 60 * 60 / 7 - 365 * 2 * 24 * 60 * 60 / 13 + 365 * 2 * 24 * 60 * 60 / 45));
        System.out.println("Population after 3 years: "
                + (312032486 + 365 * 3 * 24 * 60 * 60 / 7 - 365 * 3 * 24 * 60 * 60 / 13 + 365 * 3 * 24 * 60 * 60 / 45));
        System.out.println("Population after 4 years: "
                + (312032486 + 365 * 4 * 24 * 60 * 60 / 7 - 365 * 4 * 24 * 60 * 60 / 13 + 365 * 4 * 24 * 60 * 60 / 45));
        System.out.println("Population after 5 years: "
                + (312032486 + 365 * 5 * 24 * 60 * 60 / 7 - 365 * 5 * 24 * 60 * 60 / 13 + 365 * 5 * 24 * 60 * 60 / 45));
    }
}
```

```
public class PopulationProject {
    public static void main(String[] args) {
        int population = 312032486;
        long secondInOneYear = 365 * 24 * 60 * 60;
        long secondInTwoYears = secondInOneYear * 2;
        long secondInThreeYears = secondInOneYear * 3;
        long secondInFourYears = secondInOneYear * 4;
        long secondInFiveYears = secondInOneYear * 5;
        System.out.println("Population after 1 year: "
                + (population + secondInOneYear / 7 + secondInOneYear / 13 + secondInOneYear / 45));
        System.out.println("Population after 2 years: "
                + (population + secondInTwoYears / 7 - secondInTwoYears / 13 + secondInTwoYears / 45));
        System.out.println("Population after 3 years: "
                + (population + secondInThreeYears / 7 - secondInThreeYears / 13 + secondInThreeYears / 45));
        System.out.println("Population after 4 years: "
                + (population + secondInFourYears / 7 - secondInFourYears / 13 + secondInFourYears / 45));
        System.out.println("Population after 5 years: "
                + (population + secondInFiveYears / 7 - secondInFiveYears / 13 + secondInFiveYears / 45));
    }
}
```

Now that we've learned how to declare, initialize, and assign values to variables, it's important to consider how we name these variables. Let's recall a potential problem: the names we choose can either make our code clear and understandable or confusing and hard to follow.

For example, if we use generic names like **x**, **y**, or **value1**, it might not be immediately clear what these variables represent. This can lead to confusion, especially when working on larger projects or collaborating with others.

Imagine trying to debug or update your code months later, or someone else trying to understand it - poorly named variables could make that task unnecessarily difficult.

To avoid these issues, we use **identifiers** - the names that identify the elements such as classes, methods, and variables in a program.

Seen

Variables - Identifiers

- As you see in the program, PopulationProject, main, population, secondInOneYear, radius, input, and so on are the names of things that appear in the program. In programming terminology, such names are called *identifiers*. All identifiers must obey the following rules:
  - 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.
  - An identifier cannot be true, false, or null.
  - An identifier can be of any length.

Fun fact: This is only the common rules, the convention would be provided later, in this lecture.

The following fifty keywords are reserved for use by the Java language:

abstract	double	int	super
assert	else	interface	switch
boolean	enum	long	synchronized
break	extends	native	this
byte	final	new	throw
case	finally	package	throws
catch	float	private	transient
char	for	protected	try
class	goto	public	void
const	if	return	volatile
continue	implements	short	while
default	import	static	
do	instanceof	strictfp*	

Variables

Now that we've discussed identifiers and how important it is to name our variables clearly, let's turn our attention back to the variables themselves. We already know how to declare, initialize, and assign values to variables. But what exactly are variables, and why are they so crucial in programming?

Variables

- At their core, variables are used to store values to be used later in a program. They are called variables because their values can be changed.
- In the program ComputeArea.java, radius and area are variables of the double type. You can assign any numerical value to radius and area, and the values of radius and area can be reassigned. For example, in the following code, radius is initially 1.0 (line 2) and then changed to 2.0 (line 7), and area is set to 3.14159 (line 3) and then reset to 12.56636 (line 8).

```
1  // Compute the first area
2  radius = 1.0;                                          radius:  1.0
3  area = radius * radius * 3.14159;                        area:  3.14159
4  System.out.println("The area is " + area + " for radius " + radius);
5
6  // Compute the second area
7  radius = 2.0;                                          radius:  2.0
8  area = radius * radius * 3.14159;                        area:  12.56636
9  System.out.println("The area is " + area + " for radius " + radius);
```

Variables

- Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler its name as well as what type of data it can store. The *variable declaration* tells the compiler to allocate appropriate memory space for the variable based on its data type. The syntax for declaring a variable is

  datatype variableName;

- Here are some examples of variable declarations:

```
int count;              // Declare count to be an integer variable
double radius;          // Declare radius to be a double variable
double interestRate;    // Declare interestRate to be a double variable
```

Variables

- After a variable is declared, you can assign a value to it by using an *assignment statement.* In Java, the equal sign (=) is used as the *assignment operator.* The syntax for assignment statements is as follows:

variable = expression;

- An *expression* represents a computation involving values, variables, and operators that, taking them together, evaluates to a value. For example, consider the following code:

```
int y = 1;                    // Assign 1 to variable y
double radius = 1.0;          // Assign 1.0 to variable radius
int x = 5 * (3 / 2);          // Assign the value of the expression to x
x = y + 1;                    // Assign the addition of y and 1 to x
double area = radius * radius * 3.14159; // Compute area
```

You can use a variable in an expression. A variable can also be used in both sides of the = operator. For example,

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

Sound interesting! But sometimes we need to store values that should remain unchanged, how can we do that?

Then we can use a *named constant,* or simply *constant,* representing permanent data that never changes…

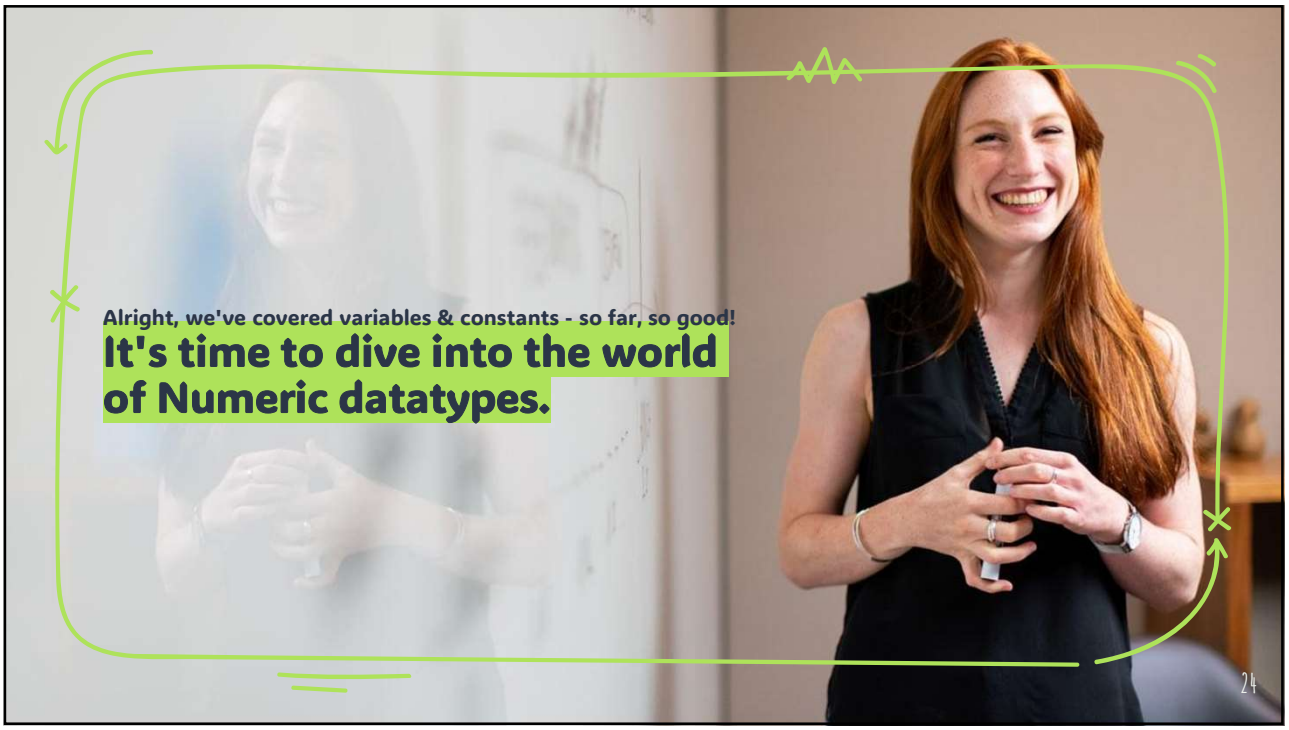
Variables - Constant

- In our ComputeArea program, Pi is a constant. If you use it frequently, you don't want to keep typing 3.14159; instead, you can declare a constant for Pi. Here is the syntax for declaring a constant:

  final datatype CONSTANT_NAME = value;

- A constant must be declared and initialized in the same statement. The word final is a Java keyword for declaring a constant. For example, you can declare Pi as a constant:

```
1  import java.util.Scanner; // Scanner is in the java.util package
2
3  public class ComputeAreaWithConstant {
4    public static void main(String[] args) {
5      final double PI = 3.14159; // Declare a constant
6
7      // Create a Scanner object
8      Scanner input = new Scanner(System.in);
9
10     // Prompt the user to enter a radius
11     System.out.print("Enter a number for radius: ");
12     double radius = input.nextDouble();
```



- Variable
- **Numeric Datatypes**
- Coding Conventions
- Operators
- Numberic Type Conversions
- Software Development Process
- Common Errors and Pitfalls
- Final Touches

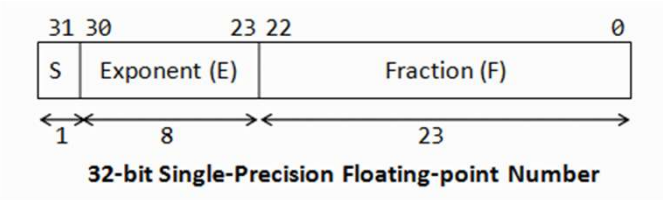
Numeric Datatypes

- Each data type in Java has a specific range of values and determines how much memory is allocated for it. Java offers eight primitive data types, including types for numeric values, characters, and Boolean values. Among these, six are numeric types used for handling integers and floating-point numbers. You can use operators like +, -, *, /, and % with these types. In this section, we'll dive into the numeric data types and explore how to use them effectively in your programs.

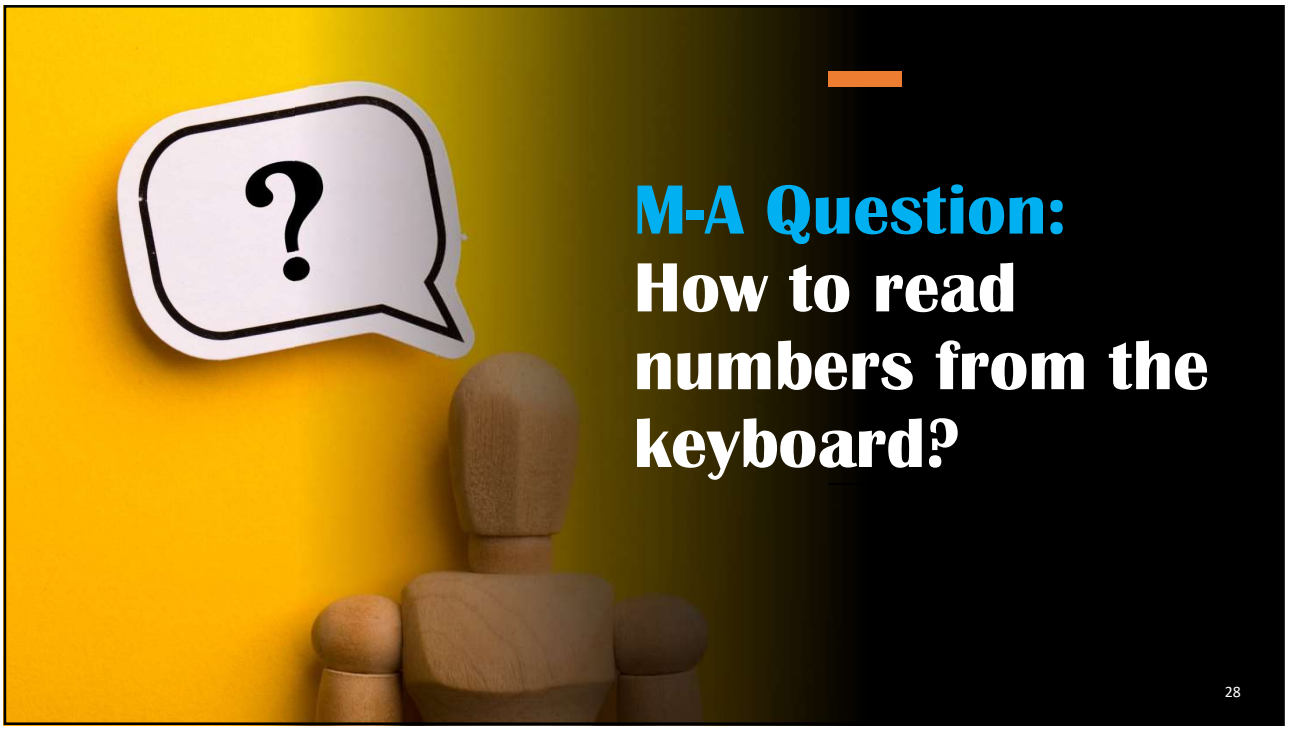
Name	*Range*	*Storage Size*	
`byte`	-2^7 to $2^7 - 1$ (−128 to 127)	8-bit signed	`byte` type
`short`	-2^{15} to $2^{15} - 1$ (−32768 to 32767)	16-bit signed	`short` type
`int`	-2^{31} to $2^{31} - 1$ (−2147483648 to 2147483647)	32-bit signed	`int` type
`long`	-2^{63} to $2^{63} - 1$	64-bit signed	`long` type
	(i.e., −9223372036854775808 to 9223372036854775807)		
`float`	Negative range: −3.4028235E + 38 to −1.4E − 45	32-bit IEEE 754	`float` type
	Positive range: 1.4E − 45 to 3.4028235E + 38		
`double`	Negative range: −1.7976931348623157E + 308 to −4.9E − 324	64-bit IEEE 754	`double` type
	Positive range: 4.9E − 324 to 1.7976931348623157E + 308		

Numeric Datatypes

- Java uses four types for integers: **byte**, **short**, **int**, and **long**. Choose the type that is most appropriate for the variable. For example, if you know an integer stored in a variable is within a range of a byte, declare the variable as a **byte**. For simplicity and consistency, we will use **int** for integers most of the time in this course.
- Java uses two types for floating-point numbers: **float** and **double**. The **double** type is twice as big as **float**, so the **double** is known as *double precision* and **float** as *single precision.* Normally, you should use the **double** type, because it is more accurate than the **float** type.



32-bit Single-Precision Floating-point Number



Numeric Datatypes

❖ You've already known how to use the `nextDouble()` method in the `Scanner` class to read a double value from the keyboard. You can also use the methods listed below to read a number of the `byte`, `short`, `int`, `long`, and `float` type.

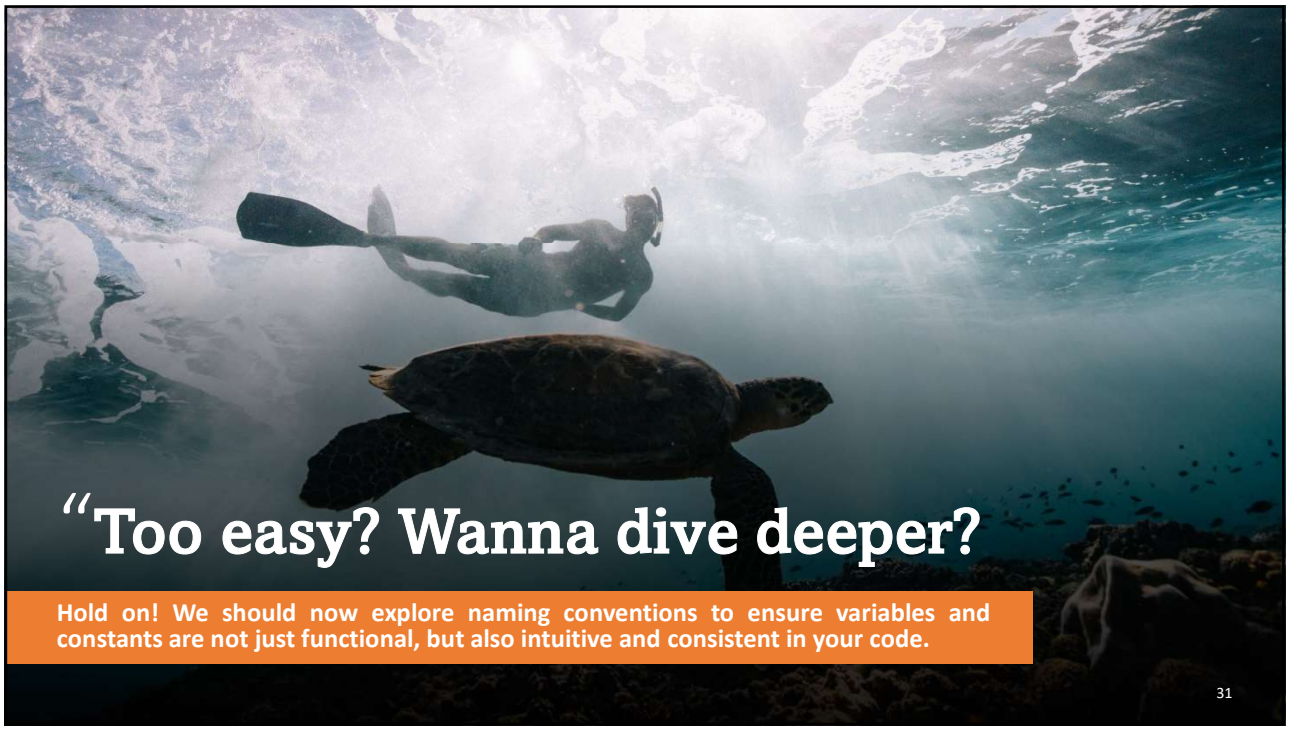
Method	*Description*
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

Numeric Datatypes

❖ Here are examples for reading values of various types from the keyboard:

```
1  Scanner input = new Scanner(System.in);
2  System.out.print("Enter a byte value: ");
3  byte byteValue = input.nextByte();
4
5  System.out.print("Enter a short value: ");
6  short shortValue = input.nextShort();
7
8  System.out.print("Enter an int value: ");
9  int intValue = input.nextInt();
10
11 System.out.print("Enter a long value: ");
12 long longValue = input.nextLong();
13
14 System.out.print("Enter a float value: ");
15 float floatValue = input.nextFloat();
```

Note that if you enter a value with an incorrect range or format, a runtime error would occur. For example, you enter a value `128` for line 3, an error would occur because `128` is out of range for a `byte` type integer.

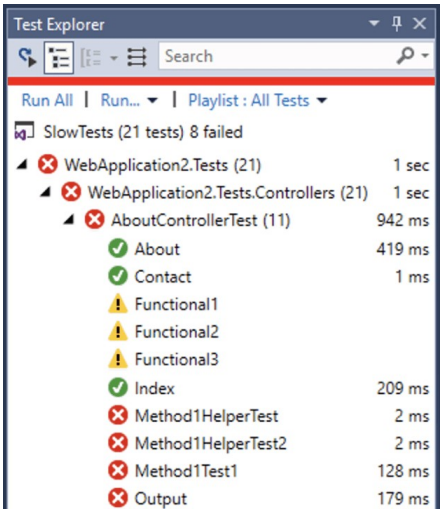


32

- Variable
- Numeric Datatypes
- **Coding Conventions**
- Operators
- Numeric Type Conversions
- Software Development Process
- Common Errors and Pitfalls
- Final Touches

Naming Conventions

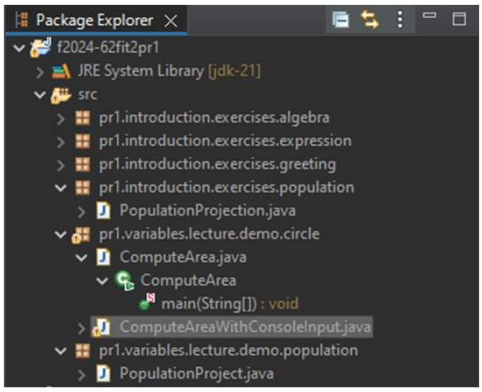
❖ Have a look at this picture, what can you see?



Can you understand what each method does just by looking at this picture, without having to read the code?

Naming Conventions

❖ Have a look at this picture, what can you see?



What about this? Can you easily understand what these classes do just by looking at their names? **Exactly!**

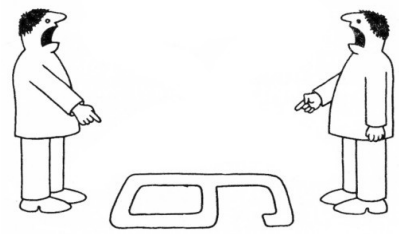
Naming Conventions

You see, while variables and constants are fundamental to programming, how we name them can significantly impact the readability and maintainability of our code.

Naming Conventions

The examples we've looked at illustrate this clearly. By selecting descriptive names that convey the purpose of the variable or constant, we make our code more understandable to ourselves and others.

Naming Conventions



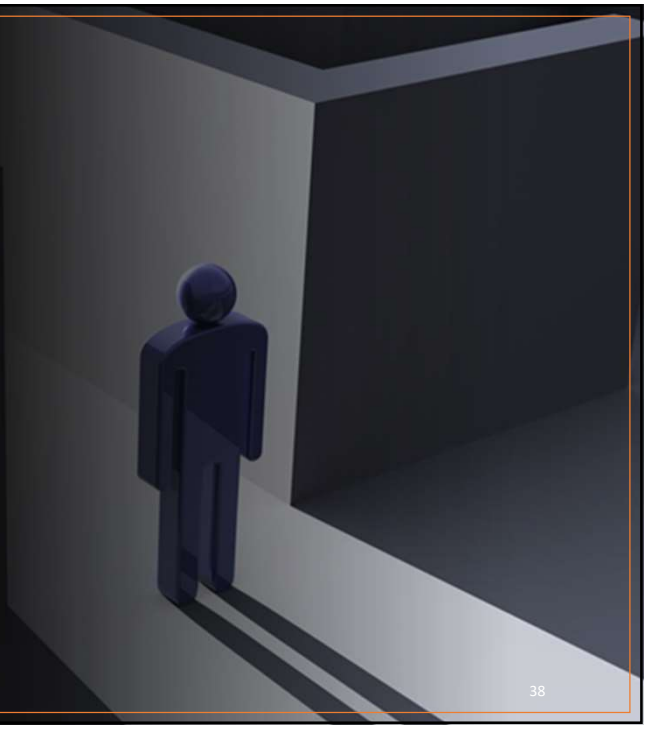
Proposition

- Imagine you're in a situation when a coworker and you looked at the same code and had different opinions about its correctness and cleanness.
- Sounds familiar, doesn't it? Still, there are some time-tested principles that should be adhered to. In the end, they will be advantageous for you, because if you leave your code in the state in which you yourself would like to receive it, then the world would become a little happier and cleaner.

Or if you have to dig into someone else's code? Instead of two hours, you may spend two days to simply understand the logic of what is happening. The funny thing is that for the person who wrote the code, everything is clear and entirely transparent. This is not surprising: after all, perfect code is an unclear concept, because each developer has the own vision of the world and of the code, too.

Feeling stuck or unsure what can you do?

Naming conventions can guide you out of the confusion.



Naming Conventions

❖ Naming conventions are rules or guidelines that help us choose meaningful names for variables, constants, methods, classes, and other elements in our code. Adhering to these conventions ensures that our code is readable and understandable, not just to ourselves but also to other developers who may work with it in the future.

While there are numerous guidelines available for programming languages, the Oracle coding conventions are a trusted standard you can follow.

Naming Conventions

❖ Here are some key naming conventions in Java. Please note that this information was last reviewed on April 20, 1999.

Identifier Type	Rules for Naming	Examples
Packages	**Prefix in Package Name:** ❑ Always written in all-lowercase ASCII letters. ❑ Should be a top-level domain name (e.g., com, edu, gov, net, org, etc.) or use English two-letter country codes as per ISO Standard 3166, 1981. **Subsequent Components:** ❑ Vary based on the organization's internal naming conventions. ❑ May include directory name components such as division, department, project, machine, or login names.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese java.util.Scanner ; java.io.*; pr1.variables.demo.shape

Refs: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

Naming Conventions

❖ Here are some key naming conventions in Java. Please note that this information was last reviewed on April 20, 1999.

Identifier Type	Rules for Naming	Examples
Classes	❑ Should be nouns and should clearly describe what the class represents, making it intuitive and easy to understand. ❑ Follows the **PascalCase** convention, meaning each word in the name starts with a capital letter, without spaces or underscores. ❑ Avoid using acronyms or abbreviations in class names unless they are widely recognized.	class **Raster** class **ImageSprite** class **Student** class **UserController** class **UserDetails**
Methods	❑ Should always be verbs because they represent actions or behaviors that objects can perform. ❑ Follows the **camelCase** convention, where the first letter of the name is lowercase, and the first letter of each subsequent internal word is capitalized. ❑ Avoid using special characters or spaces in method names. Stick to letters and numbers, with the exception of the underscore (_)	sendEmail() getUserInfo() updateRecord()

Refs: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

Naming Conventions

❖ Here are some key naming conventions in Java. Please note that this information was last reviewed on April 20, 1999.

Identifier Type	Rules for Naming	Examples
Variables	❑ Should be nouns and should clearly describe what it represents, making it intuitive and easy to understand. ❑ Follows the **camelCase** convention, meaning each word in the name starts with a lowercase letter, and each subsequent word should start with an uppercase letter. ❑ Except for temporary or loop variables, single-character names should generally be avoided as they do not convey meaning. Common exceptions include i, j, k, m, and n for loop counters or indices, and c, d, and e for characters.	int[] marks; int i; char c; float rectWidth; String firstName; int totalAmount; boolean isFinished;
Constants	❑ Should be nouns and should clearly describe what it represents. ❑ Follows **SCREAMING_SNAKE_CASE**, where all letters are uppercase, and words are separated by underscores. ❑ ANSI constants should be avoided in Java programming to ease debugging.	final int DEFAULT_TIMEOUT = 4; final double PI = 3.1415;

Refs: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

44

- Variable
- Numeric Datatypes
- Coding Conventions
- **Operators**
- Numeric Type Conversions
- Software Development Process
- Common Errors and Pitfalls
- Final Touches

Operators

❖ In this section, we are going to discuss about:

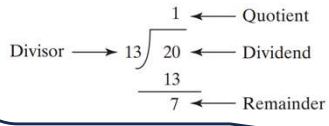
- ❑ Numeric Operators
- ❑ Exponent Operations
- ❑ Numeric Literals
- ❑ Evaluating Expressions and Operator Precedence
- ❑ Augmented Assignment Operators
- ❑ Increment and Decrement Operators

Operators - Numeric Operators

❖ The operators for numeric data types include the standard arithmetic operators: addition (+), subtraction (–), multiplication (*), division (/), and remainder (%), as shown below. The *operands* are the values operated by an operator.

Name	*Meaning*	*Example*	*Result*
+	Addition	34 + 1	35
-	Subtraction	34.0 – 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

The % operator, known as *remainder* or *modulo* operator, yields the remainder after division. The operand on the left is the dividend and the operand on the right is the divisor.



Proposition

❑ When both operands of a division are integers, the result of the division is the quotient and the fractional part is truncated. For example, 5 / 2 yields 2, not 2.5, and –5 / 2 yields -2, not –2.5. To perform a float-point division, one of the operands must be a floating-point number. For example, 5.0 / 2 yields 2.5.

Operators - Exponent Operations

❖ In case you want to calculate a^b, the **Math** class provides a method called **pow**. The **Math.pow(a, b)** method can be used to compute a^b. You invoke the method using the syntax **Math.pow(a, b)** (e.g., **Math.pow(2, 3)**), which returns the result of $a^b(2^3)$. Here, **a** and **b** are parameters for the **pow** method and the numbers **2** and **3** are actual values used to invoke the method.

❖ For example,

```
System.out.println(Math.pow(2, 3));      // Displays 8.0
System.out.println(Math.pow(4, 0.5));    // Displays 2.0
System.out.println(Math.pow(2.5, 2));    // Displays 6.25
System.out.println(Math.pow(2.5, -2));   // Displays 0.16
```

Proposition

❑ In future lecture, we will introduce more details on methods. For now, all you need to know is how to invoke the **pow** method to perform the exponent operation.

Nobita

Conan, I've been using various numeric operators in my code, but I'm curious - how does the Java compiler know what specific numbers we're working with?

Good question! The numbers you use in your code are called numeric literals. They represent actual values, and they come in different forms. For example, integers and floating-point numbers are both types of numeric literals. Each type has its own characteristics and usage rules.

Delivered

Nobita

So, the way we write these numbers can affect how Java interprets them?

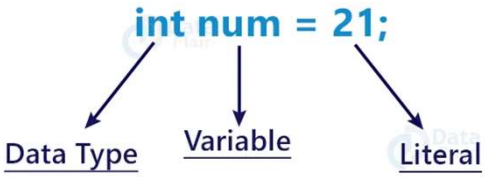
Exactly! Numeric literals can vary in format, such as decimal, octal, hexadecimal, or even scientific notation. Understanding these literals helps in writing accurate and efficient code. Let's see how they fit into the world of Java programming.

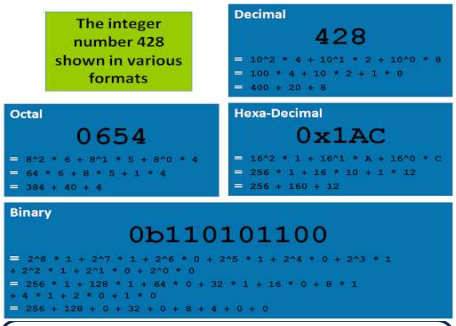
Delivered

Operators - Numeric Literals

❖ Numeric literals are the actual numbers you write in your code to represent values. They come in various forms, such as:

- ❑ Integers
- ❑ Floating-point Numbers
- ❑ Scientific Notation





As once said numeric literals represent actual values, and they come in different forms.

Operators - Numeric Literals (Integers)

❖ An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compile error will occur if the literal is too large for the variable to hold. The statement **byte b = 128**, for example, will cause a compile error, because **128** cannot be stored in a variable of the **byte** type. (Note that the range for a byte value is from **–128** to **127**.)

❖ An integer literal is assumed to be of the **int** type, whose value is between -2^{31} (-2147483648) and $2^{31} - 1$ (2147483647). To denote an integer literal of the **long** type, append the letter **L** or **l** to it. For example, to write integer **2147483648** in a Java program, you have to write it as **2147483648L** or **2147483648l**, because **2147483648** exceeds the range for the **int** value. **L** is preferred because **l** (lowercase **L**) can easily be confused with 1 (the digit one).

Proposition

By default, an integer literal is a decimal integer number. To denote a binary integer literal, use a leading *0b* or *0B* (zero B), to denote an octal integer literal, use a leading *0* (zero), and to denote a hexadecimal integer literal, use a leading *0x* or *0X* (zero X). For example,

```
System.out.println(0B1111);     // Displays 15
System.out.println(07777);      // Displays 4095
System.out.println(0XFFFF);     // Displays 65535
```

Operators - Numeric Literals (Floating-point) 51

❖ A Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a **double** type value. For example, **5.0** is considered a **double** value, not a **float** value. You can make a number a **float** by appending the letter **f** or **F**, and you can make a number a **double** by appending the letter **d** or **D**. For example, you can use **100.2f** or **100.2F** for a **float** number, and **100.2d** or **100.2D** for a **double** number.

Proposition

The **double** type values are more accurate than the **float** type values. For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays `1.0 / 3.0 is 0.3333333333333333`

16 digits

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays `1.0F / 3.0F is 0.33333334`

8 digits

Operators - Numeric Literals (Scientific Notation) 52

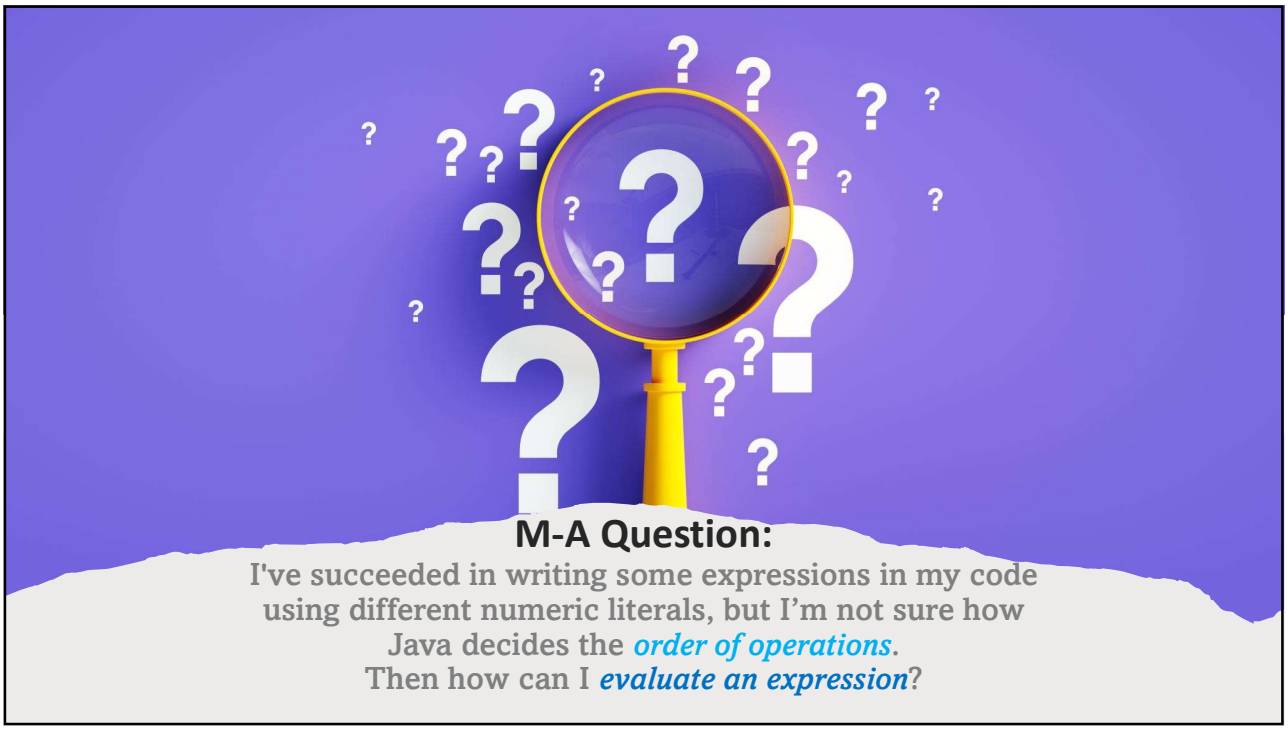
❖ Floating-point literals can be written in scientific notation in the form of $a * 10^b$. For example, the scientific notation for 123.456 is $1.23456 * 10^2$ and for 0.0123456 is $1.23456 * 10^{-2}$. A special syntax is used to write scientific notation numbers. For example, $1.23456 * 10^2$ is written as **1.23456E2** or **1.23456E+2** and $1.23456 * 10^{-2}$ as **1.23456E-2**. **E** (or **e**) represents an exponent and can be in either lowercase or uppercase.

Proposition

- The **float** and **double** types are used to represent numbers with a decimal point. Why are they called *floating-point numbers*? These numbers are stored in scientific notation internally. When a number such as **50.534** is converted into scientific notation, such as **5.0534E+1**, its decimal point is moved (i.e., floated) to a new position.
- To improve readability, Java allows you to use underscores between two digits in a number literal. For example, the following literals are correct.

```
long ssn = 232_45_4519;
long creditCardNumber = 2324_4545_4519_3415L;
```

However, **45_** or **_45** is incorrect. The underscore must be placed between two digits.



M-A Question:

I've succeeded in writing some expressions in my code using different numeric literals, but I'm not sure how Java decides the *order of operations*.
Then how can I *evaluate an expression*?

Operators - Numeric Literals (Operator Precedence) 54

❖ Writing a numeric expression in Java involves a straightforward translation of an arithmetic expression using Java operators. For example, the arithmetic expression

$$\frac{3 + 4x}{5} - \frac{10(y - 5)(a + b + c)}{x} + 9\left(\frac{4}{x} + \frac{9 + x}{y}\right)$$

can be translated into a Java expression as:

```
(3 + 4 * x) / 5 – 10 * (y - 5) * (a + b + c) / x + 9 * (4 / x + (9 + x) / y)
```

Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression is the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression.

Operators - Numeric Literals (Operator Precedence) 55

❖ You can combine operators to form complicated expressions. When you do, the order in which the operations are carried out is determined by the precedence of each operator in the expression. The order of precedence for the arithmetic operators:

1. Operators contained within pairs of parentheses are evaluated first.
2. Increment (++) and decrement (--) operators are evaluated.
3. Next, sign operators (+ or -) are applied.
4. Then, multiplication (*), division (/), and remainder (%) operators are evaluated.
5. Finally, addition (+) and subtraction (-) operators are applied.

Proposition

- ❑ If an expression includes two or more operators at the same order of precedence, the operators are evaluated **left to right**.
- ❑ If you want, you can use parentheses to change the order in which operations are performed. If an expression has two or more sets of parentheses, the operations in the innermost set are performed first.

Operators - Numeric Literals (Operator Precedence) 56

❖ Here is an example of how an expression is evaluated:

```
3 + 4 * 4 + 5 * (4 + 3) - 1
                (1) inside parentheses first
3 + 4 * 4 + 5 * 7 - 1
            (2) multiplication
3 + 16 + 5 * 7 - 1
             (3) multiplication
3 + 16 + 35 - 1
        (4) addition
19 + 35 - 1
     (5) addition
  54 - 1
       (6) subtraction
  53
```

Operators - Augmented Assignment Operators

❖ You know, the operators +, -, *, /, and % can also be combined with the assignment operator to form augmented operators. Very often the current value of a variable is used, modified, and then reassigned back to the same variable. For example, the following statement increases the variable count by 1:

```
count = count + 1;
```

❖ Java allows you to combine assignment and addition operators using an augmented (or compound) assignment operator. For example, the preceding statement can be written as

```
count += 1;
```

❖ The += is called the *addition assignment operator.*

Operators - Augmented Assignment Operators

❖ There are some other augmented assignment operators.

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

Proposition

❑ The augmented assignment operator is performed last after all the other operators in the expression are evaluated. For example,

```
x /= 4 + 5.5 * 1.5;
```

is same as

```
x = x / (4 + 5.5 * 1.5);
```

59

Nobita

These augmented assignment operators are really useful! But I often find myself just adding or subtracting 1 from a variable. Is there a quicker way to do that?

You're in luck! One of the most common operations in computer programming is adding or subtracting 1 from a variable. This is known as incrementing or decrementing the variable. Instead of writing something like a = a + 1 or a += 1, Java provides a shorthand for this task.

Delivered

Nobita

Really? How does that work?

Java uses special operators called increment (++) and decrement (--) operators. These operators are designed specifically for adding or subtracting 1 from a variable quickly and efficiently. Let's dive into how they work.

Delivered

Operators - Increment and Decrement Operators

60

❖ The ++ and —— are two shorthand operators for incrementing and decrementing a variable by 1. These are handy because that's often how much the value needs to be changed in many programming tasks. For example, the following code increments i by 1 and decrements j by 1.

```
int i = 3, j = 3;
i++;    // i becomes 4
j--;    // j becomes 2
```

❖ i++ is pronounced as i plus plus and i—— as i minus minus. These operators are known as *postfix increment* (or postincrement) and *postfix decrement* (or postdecrement), because the operators ++ and —— are placed after the variable. These operators can also be placed before the variable. For example,

```
int i = 3, j = 3;
++i;    // i becomes 4
 --j;   // j becomes 2
```

Operators - Increment and Decrement Operators

❖ As you see, the effect of `i++` and `++i` or `i——` and `——i` are the same in the preceding examples. However, their effects are different when they are used in statements that do more than just increment and decrement. Table below describes their differences and gives examples.

Operator	*Name*	*Description*	*Example (assume i = 1)*
`++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`

Operators - Increment and Decrement Operators

❖ Here are additional examples to illustrate the differences between the prefix form and the postfix form. Consider the following code:

```
int i = 10;
int newNum = 10 * i++;

System.out.print("i is " + i
  + ", newNum is " + newNum);
```

Same effect as →

```
int newNum = 10 * i;
i = i + 1;
```

```
i is 11, newNum is 100
```

```
int i = 10;
int newNum = 10 * (++i);

System.out.print("i is " + i
  + ", newNum is " + newNum);
```

Same effect as →

```
i = i + 1;
int newNum = 10 * i;
```

```
i is 11, newNum is 110
```

Operators - Increment and Decrement Operators

❖ Here is another example, do you know the result?

```
double x = 1.0;
double y = 5.0;
double z = x-- + (++y);
```

Yeah, after all three lines are executed:
y becomes 6.0,
z becomes 7.0,
and x becomes 0.0.

Feeling a bit bored? Let's try to outsmart the compiler with this tricky quiz!

```
int x = 5;
int y = x++ + ++x - --x + x--;
System.out.println("Final value of y: " + y);
```

Yeah, the answer is 12

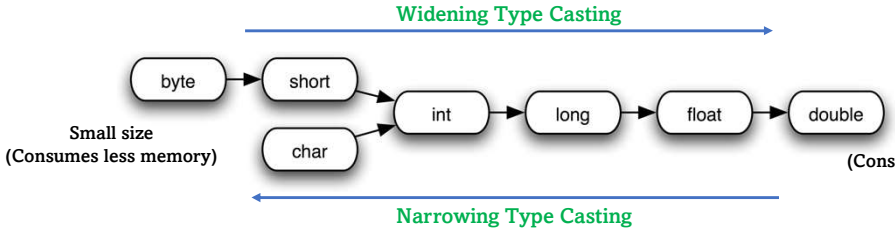
Well, Java offers a variety of operators, but it is quite strict with type handling. Can you perform binary operations with operands of different types?

Yeah, let me explain a simple way to do that!

- Variable
- Numeric Datatypes
- Coding Conventions
- Operators
- **Numeric Type Conversions**
- Software Development Process
- Common Errors and Pitfalls
- Final Touches

Numeric Type Conversions

❖ The process of converting a value from one data type to another is known as type conversion in Java. You can always assign a value to a numeric variable whose type supports a larger range of values; thus, for instance, you can assign a **long** value to a **float** variable. You cannot, however, assign a value to a variable of a type with a smaller range unless you use *type casting*.



Numeric Type Conversions

❖ If two data types are compatible with each other, Java will perform such conversion ***automatically*** or ***implicitly***. We can easily convert a primitive data type into another primitive data type by using type casting. Look at this example:

```
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
}
```

Numeric Type Conversions

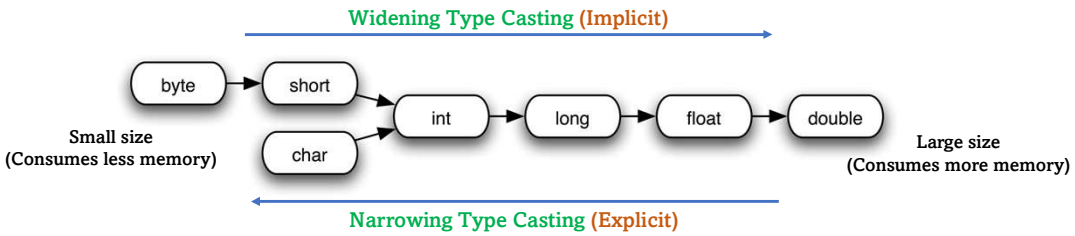
❖ Similarly, it is also possible to convert a non-primitive data type (referenced data type) into another non-primitive data type by using type casting. But we cannot convert a primitive data type into an advanced (referenced) data type by using type casting. For this case, we will have to use methods of Java Wrapper classes.

```
public static void main(String[] args) {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt);
    System.out.println(myDouble);
    System.out.println(myChar);
}
```

myInt, myDouble, and myChar ave now been converted into objects that store the values of primitive types.

Numeric Type Conversions

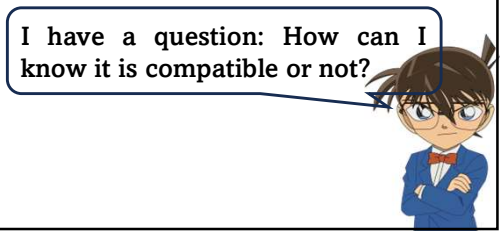
❖ There are two types of casting possible in Javas:
- ❑ **Implicit** type casting (also known as automatic type conversion)
- ❑ **Explicit** type casting



Numeric Type Conversions - Implicit

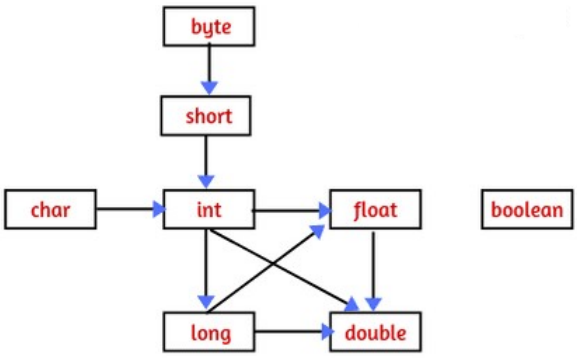
- Automatic conversion (casting) done by Java compiler internally is called implicit conversion or implicit type casting in Java. It is performed to convert a **lower data** type into a **higher data** type.
- For example, if we assign an int value to a long variable, it is compatible with each other, but an int value cannot be assigned to a byte variable.

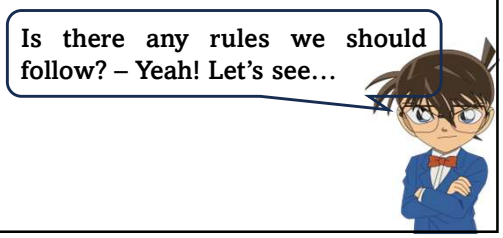
```
int x = 20;
long y = x;   // Automatic conversion
byte z = x;   // Type mismatch: cannot convert from int to byte.
```



Numeric Type Conversions - Implicit

- You know, when performing arithmetic operations with operands of different types, the Java compiler **automatically converts** the **'lower' type** to the **'higher' type**. Lower types consume *less memory* and store *fewer digits*, while higher types use *more memory* and store *more digits*.
- The below figure shows which conversion is allowed by Java.





Numeric Type Conversions - Implicit

❖ There are some Promotion Rules in Expression, just keep in your mind...

1. If byte, short, and int are used in a mathematical expression, Java always converts the result into an int.
2. If a single long is used in the expression, the whole expression is converted to long.
3. If a float operand is used in an expression, the whole expression is converted to float.
4. If any operand is double, the result is promoted to double.
5. Boolean values cannot be converted to another type.
6. Conversion from float to int causes truncation of the fractional part which represents the loss of precision. Java does not allow this.
7. Conversion from double to float causes rounding of digits that may cause some of the value's precision to be lost.
8. Conversion from long to int is also not possible. It causes the dropping of the excess higher order bits.

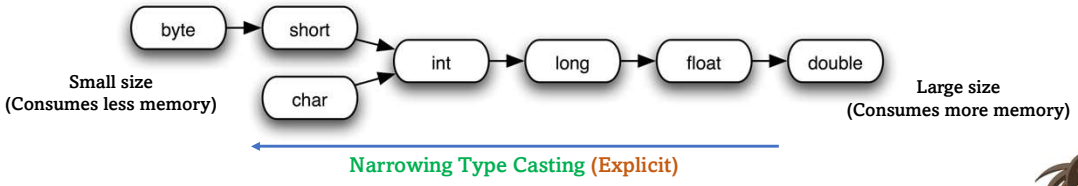
Numeric Type Conversions - Implicit

❖ Let's take one more example program to understand automatic type promotion better.

```
public class AutoTypeConversion {
    int x = 20;
    double y = 40.5;
    long p = 30;
    float q = 10.60f;
    void sum() {
        // int z = x + y; (1)
        // Error; cannot convert from double to int.
        double z = x + y;
        System.out.println("Sum of two numbers: " +z);
    }
    void sub() {
        // long r = p - q; // (2)
        // Error; cannot convert from float to long.
        float r = p - q;
        System.out.println("Subtraction of two numbers: " +r);
    }
    public static void main(String[] args) {
        AutoTypeConversion obj = new AutoTypeConversion();
        obj.sum(); // Calling sum method.
        obj.sub(); // Calling sub method.
    }
}
```

Numeric Type Conversions - Explicit

- As we know that automatic conversion is very helpful, and safe, but it cannot fulfill all needs. For example, if you assign a double value to an int variable, this conversion cannot be performed automatically because an int is smaller than a double.
- In this case, we must use explicit type casting to create a conversion between two incompatible types. This kind of conversion is known as ***narrowing conversion*** in Java, where a **higher data type** is converted into a **lower data type**.



Numeric Type Conversions - Explicit

- The syntax for casting a type is to specify the target type in *parentheses*, followed by the variable's name or the value to be cast.

Proposition
The formula: (**target datatype**) literal or expression;

Example 01.
The following statement

```
System.out.println( (int) 1.7);    // displays 1.
```

Reason: When a double value is cast into an int value, the fractional part is truncated

Example 02.
On the other hand, the following statement

```
System.out.println( (double) 1 / 2);    // displays 0.5.
```

Reason: 1 is cast to 1.0 first, then 1.0 is divided by 2.

Example 03.
The following statement

```
System.out.println( 1 / 2);    // displays 0.
```

Reason: 1 and 2 are both integers and the resulting value should also be an integer.

Numeric Type Conversions - Explicit

❖ Though explicit conversion offers significant advantages, but it does come with certain drawbacks:

1. **Data Loss:** Converting from a larger data type to a smaller one (e.g., double to int) can result in data loss, as the precision or magnitude of the original value might be truncated.
2. **Precision Loss:** When converting *floating-point* numbers to *integers*, any fractional part is lost, leading to potential inaccuracies in calculations.
3. **Risk of Errors:** Manual type conversions increase the risk of runtime errors, such as **ClassCastException**, if the conversion is not handled properly.

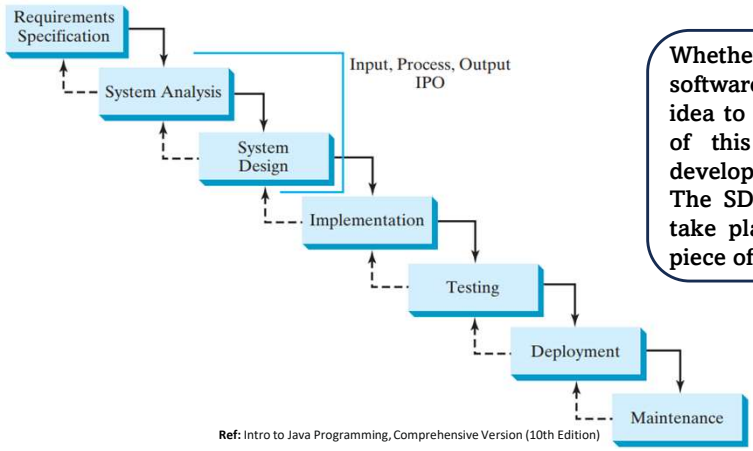
Now that we've covered data type casting and its implications, do you have any questions?

Actually, you've learned the basics of building a Java program, but developing a software product involves more than just writing code in an IDE. It's a multi-stage process that called...

- Variable
- Numeric Datatypes
- Coding Conventions
- Operators
- Numeric Type Conversions
- **Software Development Process**
- Common Errors and Pitfalls
- Final Touches

Software Development Process

- Developing a software product is an engineering process. Software products, no matter how large or how small, have the same life cycle: Requirements specification, Analysis, Design, Implementation, Testing, Deployment, and Maintenance.

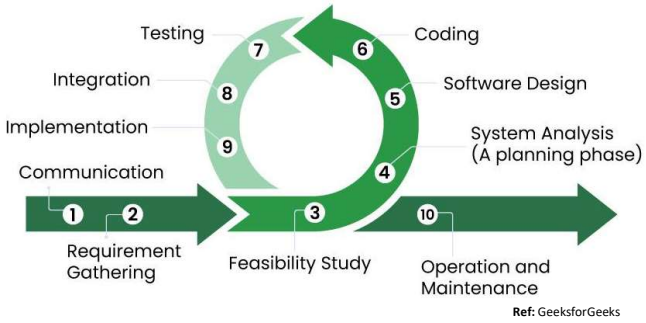


Ref: Intro to Java Programming, Comprehensive Version (10th Edition)

Whether you plan it or not, every piece of software goes through a similar path from idea to launch day. Collectively, the steps of this path are called the software development lifecycle (or SDLC for short). The SDLC is the sequence of steps that take place during the development of a piece of software.

Software Development Process

❖ If you're a project manager, you're probably already familiar with the different steps in the SDLC. As the shepherd for a digital project, you have to think about everything from requirements to stakeholder communication, development, and ongoing maintenance.

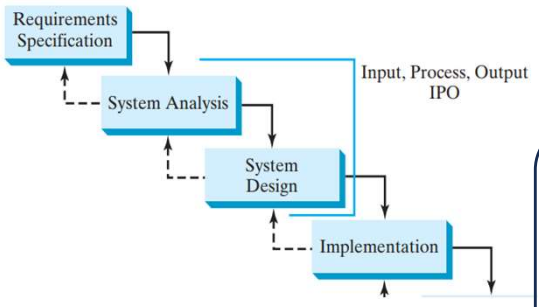


Ref: GeeksforGeeks

These steps are all pretty much the same across any software development process you use. However, as we'll discuss later, the order and sequence they occur in can change depending on your needs, goals, and project and team size (for example, some steps might be combined, duplicated, or run in parallel).

Software Development Process

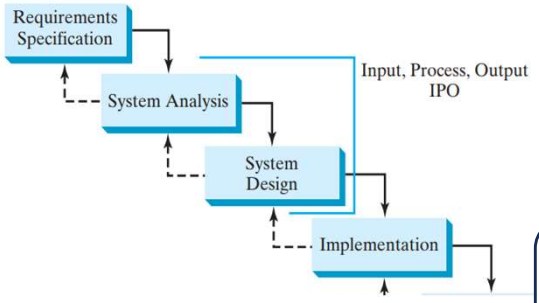
❖ To better understand software development, let's explore each stage of the process:



Requirements Specification is a formal process to understand and document what the software needs to do. It involves close interaction between users and developers to define and address problems. While examples in some books may have clear requirements, real-world problems are often less well-defined, requiring careful collaboration with customers to determine software needs.

Software Development Process

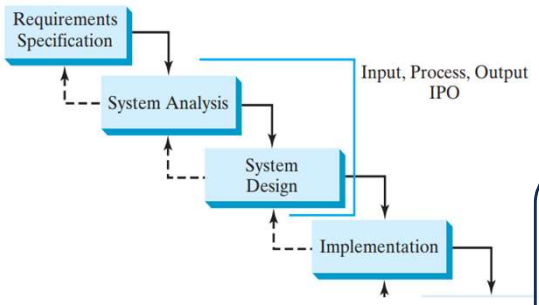
❖ To better understand software development, let's explore each stage of the process:



System Analysis seeks to analyze the data flow and to identify the system's input and output. When you do analysis, it helps to identify what the output is first, and then figure out what input data you need in order to produce the output.

Software Development Process

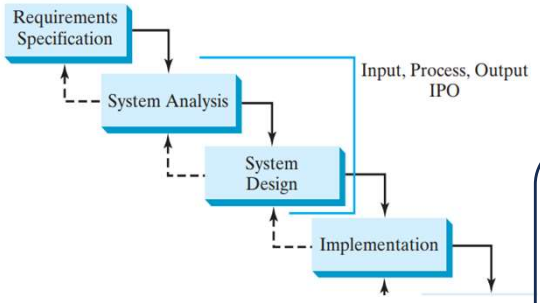
❖ To better understand software development, let's explore each stage of the process:



System Design involves creating a process to convert input into output. This phase breaks down the problem into manageable components, each performing a specific function. It uses various levels of abstraction to design strategies for implementing each component, focusing on the core concepts of input, process, and output (IPO).

Software Development Process

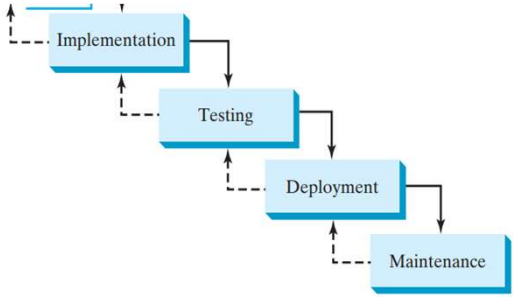
❖ To better understand software development, let's explore each stage of the process:



Implementation involves translating the system design into programs. Separate programs are written for each component and then integrated to work together. This phase requires the use of a programming language such as Java. The implementation involves coding, selftesting, and debugging (that is, finding errors, called *bugs*, in the code).

Software Development Process

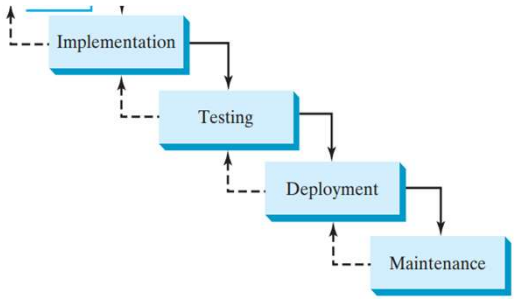
❖ To better understand software development, let's explore each stage of the process:



Testing ensures that the code meets the requirements specification and weeds out bugs. An independent team of software engineers not involved in the design and implementation of the product usually conducts such testing.

Software Development Process

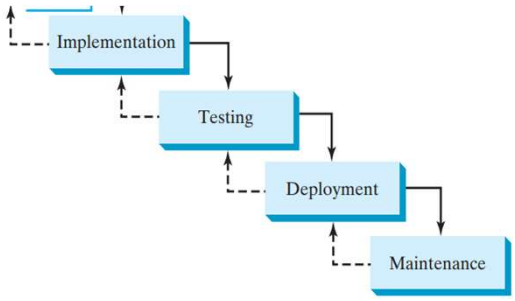
❖ To better understand software development, let's explore each stage of the process:



Deployment makes the software available for use. Depending on the type of software, it may be installed on each user's machine or installed on a server accessible on the Internet

Software Development Process

❖ To better understand software development, let's explore each stage of the process:



Maintenance is concerned with updating and improving the product. A software product must continue to perform and improve in an ever-evolving environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

Software Development Process

❖ To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on Requirements Specification, Analysis, Design, Implementation, and Testing.

Stage 1: Requirements Specification

The program must satisfy the following requirements:

- ❑ It must let the user enter the interest rate, the loan amount, and the number of years for which payments will be made.
- ❑ It must compute and display the monthly payment and total payment amounts.

Software Development Process

❖ To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on Requirements Specification, Analysis, Design, Implementation, and Testing.

Stage 2: System Analysis

The output is the monthly payment and total payment, which can be obtained using the following formulas:

$$monthlyPayment = \frac{loanAmount \times monthlyInterestRate}{1 - \frac{1}{(1 + monthlyInterestRate)^{numberOfYears \times 12}}}$$

$$totalPayment = monthlyPayment \times numberOfYears \times 12$$

So, the input needed for the program is the monthly interest rate, the length of the loan in years, and the loan amount.

Software Development Process

❖ To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on Requirements Specification, Analysis, Design, Implementation, and Testing.

Stage 3: System Design

During system design, you identify the steps in the program.

- ❑ **Step 1.** Prompt the user to enter the annual interest rate, the number of years, and the
- ❑ loan amount.
- ❑ **Step 2.** The input for the annual interest rate is a number in percent format, such as
- ❑ 4.5%. To convert an annual interest rate to a monthly rate in decimal format, divide by 1200. For example, an annual rate of 4.5% becomes 4.5 / 1200 = 0.00375.
- ❑ **Step 3.** Compute the monthly payment using the preceding formula.
- ❑ **Step 4.** Compute the total payment, which is the monthly payment multiplied by 12 and multiplied by the number of years.
- ❑ **Step 5**. Display the monthly payment and total payment.

Software Development Process

❖ To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on Requirements Specification, Analysis, Design, Implementation, and Testing.

Stage 4: Implementation

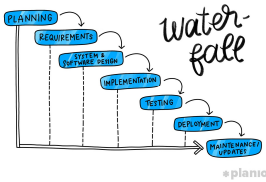
Implementation is also known as coding (writing the code). Note to choose the most appropriate data type for the variable.

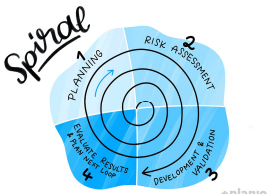
Stage 5: Testing

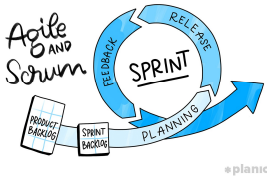
After the program is implemented, test it with some sample input data and verify whether the output is correct. Some of the problems may involve many cases, as you will see in later lectures. For these types of problems, you need to design test data that cover all cases.

Software Development Process

- While the SDLC we outlined above might seem like a step-by-step plan for building software, it's really more of a guideline. You need to check each box to ensure you're shipping and maintaining great software. But how you check them, when, and in what order is up to you. Over the years, a number of different software development processes have been formalized to tackle more and more complex projects.
- Ultimately, which process you use will come down to your goals, the size of the project and your team, and other factors. To help you decide, here are 3 of the best software development processes:







Fun fact: For more details about these, please read it hereeeeee....

While working on software development, you may encounter some issues or pitfalls that are common among developers. How to avoid this?

Let's explore these frequent challenges to better prepare for them and improve our coding practices.

- Variable
- Numeric Datatypes
- Coding Conventions
- Operators
- Numeric Type Conversions
- Software Development Process
- **Common Errors and Pitfalls**
- Final Touches

Common Errors and Pitfalls

- *Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.*

Common Error 1: Undeclared/Uninitialized Variables and Unused Variables

- A variable must be declared with a type and assigned a value before using it. A common error is not declaring a variable or initializing a variable. Consider the following code:

```
double interestRate = 0.05;
double interest = interestrate * 45;
```

This code is wrong, because `interestRate` is assigned a value `0.05`; but `interestrate` has not been declared and initialized. Java is case sensitive, so it considers `interestRate` and `interestrate` to be two different variables.

Common Errors and Pitfalls

❖ *Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.*

Common Error 2: Integer Overflow

❖ Numbers are stored with a limited numbers of digits. When a variable is assigned a value that is too large (*in size*) to be stored, it causes *overflow*. For example, executing the following statement causes overflow, because the largest value that can be stored in a variable of the int type is 2147483647:

```
int value = 2147483647 + 1;
// value will actually be -2147483648
```

Common Errors and Pitfalls

❖ *Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.*

Common Error 3: 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. For example,

`System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);` displays 0.5000000000000001, not 0.5,

`System.out.println(1.0 - 0.9);` displays 0.09999999999999998, not 0.1.

Common Errors and Pitfalls

❖ *Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.*

Common Error 4: Unintended Integer Division

❖ 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 truncated. To force two integers to perform a floating-point division, make one of the integers into a floating-point number. For example, the code in (a) displays that average is `1` and the code in (b) displays that average is `1.5`.

```
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)

Common Errors and Pitfalls

❖ *Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.*

Common Pitfall 1: Redundant Input Objects

❖ New programmers often write the code to create multiple input objects for each input. For example, the following code reads an integer and a double value.

```
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();
```

BAD CODE

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();
System.out.print("Enter a double value: ");
double v2 = input.nextDouble();
```

GOOD CODE

- Variable
- Numeric Datatypes
- Coding Conventions
- Operators
- Numeric Type Conversions
- Software Development Process
- Common Errors and Pitfalls
- **Final Touches**

Final Touches

1. *Identifiers* are names for naming elements such as variables, constants, methods, classes, packages in a program.

2. An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($). An identifier must start with a letter or an underscore. It cannot start with a digit. An identifier cannot be a reserved word. An identifier can be of any length.

3. *Variables* are used to store data in a program. To declare a variable is to tell the compiler what type of data a variable can hold.

4. There are two types of **import** statements: *specific import* and *wildcard import*. The specific import specifies a single class in the import statement; the wildcard import imports all the classes in a package.

Final Touches

5. In Java, the equal sign (**=**) is used as the *assignment operator.*

6. A variable declared in a method must be assigned a value before it can be used.

7. A *named constant* (or simply a *constant*) represents permanent data that never changes.

8. A named constant is declared by using the keyword **final**.

9. Java provides four integer types (**byte**, **short**, **int**, and **long**) that represent integers of four different sizes.

10. Java provides two *floating-point types* (**float** and **double**) that represent floatingpoint numbers of two different precisions.

Final Touches

11. Java provides *operators* that perform numeric operations: **+** (addition), **–** (subtraction), (multiplication), **/** (division), and **%** (remainder).

12. Integer arithmetic (**/**) yields an integer result.

13. The numeric operators in a Java expression are applied the same way as in an arithmetic expression.

14. Java provides the augmented assignment operators **+=** (addition assignment), **–=** (subtraction assignment), ***=** (multiplication assignment), **/=** (division assignment), and **%=** (remainder assignment).

15. The *increment operator* (**++**) and the *decrement operator* (**--**) increment or decrement a variable by **1**

Final Touches

16. When evaluating an expression with values of mixed types, Java automatically converts the operands to appropriate types.

17. You can explicitly convert a value from one type to another using the **(type) value** notation.

18. *Casting* a variable of a type with a small range to a variable of a type with a larger range is known as *widening a type.* Casting a variable of a type with a large range to a variable of a type with a smaller range is known as *narrowing a type.*

19. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.

20. In computer science, midnight of January 1, 1970, is known as the *UNIX epoch.*

Thanks!

Any questions?

For an in-depth understanding of Java, I highly recommend referring to the textbooks. This slide provides a brief overview and may not cover all the details you're eager to explore!