SmartCliff
CAREER MOBILITY SOLUTIONS

SDE Readiness Training

Empowering Tomorrow's Innovators

Module I

Java Software Development: Effective Problem Solving

SmartCliff
Career Mobility Solutions

Introduction to Java

Learning Level: Basic

DATE : 13.02.2025

Contents

01 Introduction to Java

02 Java Tokens

03 Read User Input

Introduction to Java

What is Java?

- Java is a **general-purpose, object-oriented, concurrent, and secured** programming language.

- It is a **widely used robust technology**.
- It is a **platform** – Has a **runtime environment (JRE)** and a **comprehensive set of APIs** that allow developers to **build and run Java applications** across **different hardware and operating systems**.

- **James Gosling –** the father of Java.

History

- Java's initial work started in **1990 by Sun Microsystems** engineer **Patrick Naughton** as a part of the **Stealth Project.**
- The Stealth Project soon changed to the **Green Project**, with **Mike Sheridan and James Gosling** joining the ranks, and the group began developing new technology for programming next-generation smart appliances.
- **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in **June 1991.**
- Gosling attempted to modify and **extend C++** but quickly abandoned this approach in favor of creating an entirely new language.

History

- Firstly, it was called "**Greentalk**" by James Gosling, and file extension was .gt. After that, it was called **Oak**, named after the tree that stood outside his office.
- Originally **designed for small, embedded systems in electronic appliances** like set-top boxes. Then incorporate some changes based on emergence of **World Wide Web**, which demanded **portable programs**.
- In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.
- The first publicly available version of **Java (Java 1.0)** was released in 1995.
- In 2006 Sun started to make Java available under the **GNU General Public License** (GPL). Sun Microsystems was acquired by the **Oracle Corporation in 2010**. Oracle continues this project called OpenJDK.

Where it is used?

- Java is **almost everywhere** and more than **60 billion devices** run Java.
- It is used create desktop Applications, Web Applications, Enterprise Applications such as banking applications, mobile applications.
- Web services and Cloud-based applications.
- Embedded System
- Games etc.,

Editions

J2SE (JavaPlatform, Standard Edition)	J2ME (Java Platform, Micro Edition)	J2EE (Java Platform, Enterprise Edition)
• **Java 2 Standard Edition** • Used to create applications for Desktop Environment	• **Java 2 Micro Edition** • Used to create applications for mobile Devices	• **Jakarta EE/Java 2 Enterprise Edition** • Used to create applications for Enterprise. • Community driven Edition.

Version

Version	Class File Format Version[7]	Release date	End of Public Updates (Free)	End of Extended Support (Paid)
JDK 1.0	45	23rd January 1996	May 1996	—
JDK 1.1	45	18th February 1997	October 2002	—
J2SE 1.2	46	4th December 1998	November 2003	—
J2SE 1.3	47	8th May 2000	March 2006	—
J2SE 1.4	48	13th February 2002	October 2008	—
J2SE 5.0	49	30th September 2004	October 2009	—
Java SE 6	50	11th December 2006	April 2013	December 2016 for Red Hat[8] October 2018 for Oracle[9] March 2026 for BellSoft Liberica[10] December 2027 for Azul[11]
Java SE 7	51	28th July 2011	July 2015	June 2020 for Red Hat[8] July 2022 for Oracle[12] March 2026 for BellSoft Liberica[10] December 2027 for Azul[11]
Java SE 8 (LTS)	52	18th March 2014	April 2019 for Oracle November 2026 for Eclipse Temurin[13] November 2026 for Red Hat[8] July 2027 for Amazon Corretto[14] December 2030 for Azul[11] March 2031 for BellSoft Liberica[10]	December 2030 for Oracle[4]

Version

Java SE 9	53	21st September 2017	March 2018	—
Java SE 10	54	20th March 2018	September 2018	—
Java SE 11 (LTS)	55	25th September 2018	April 2019 for Oracle October 2024 for Red Hat[8] March 2027 for BellSoft Liberica[10] October 2027 for Eclipse Temurin[13] October 2027 for Amazon Corretto[14] January 2032 for Azul[11]	January 2032 for Oracle[4]
Java SE 12	56	19th March 2019	September 2019	—
Java SE 13	57	17th September 2019	March 2020	—
Java SE 14	58	17th March 2020	September 2020	—
Java SE 15	59	16th September 2020	March 2021	—
Java SE 16	60	16th March 2021	September 2021	—
Java SE 17 (LTS)	61	14th September 2021	September 2024 for Oracle[4] October 2027 for Eclipse Temurin[13] October 2027 for Red Hat[8] October 2029 for Amazon Corretto[14] September 2029 for Azul[11] March 2030 for BellSoft Liberica[10]	September 2029 for Oracle[4]

Version

Java SE 18	62	22nd March 2022	September 2022	—
Java SE 19	63	20th September 2022	March 2023	—
Java SE 20	64	21st March 2023	September 2023	—
Java SE 21 (LTS)	65	19th September 2023	September 2026 for Oracle[4] December 2029 for Red Hat[8] December 2029 for Eclipse Temurin[13] October 2030 for Amazon Corretto[14] September 2031 for Azul[11] March 2032 for BellSoft Liberica[10]	September 2031 for Oracle[4]
Java SE 22	66	19th March 2024	September 2024	—
Java SE 23	67	September 2024	March 2025	—
Java SE 24	68	March 2025	September 2025	—
Java SE 25 (LTS)	69	September 2025	September 2028 for Oracle[4]	September 2033 for Oracle[4]

Legend: Old version | Older version, still maintained | **Latest version** | Future release

Environment

- Java includes many development tools, classes, and methods.
- **Development tools** are part of Java Development Kit **(JDK)**
- The classes and methods are part of Java Standard Library **(JSL),** also known as Application Programming Interface **(API).**
- It includes **hundreds of classes** and **methods grouped** into **several packages** according to their functionality.

Environment - JDK

- **Java Development Kit(JDK) :** It is a software **development kit** used by **Java developers** for building Java applications.
- Includes **tools, libraries, and documentation** necessary for Java development.
- **JDK** consists of :
  - **Java Compiler (javac):** Translates Java source code into bytecode.
  - **Java Runtime Environment (JRE):** Includes a version of the JRE, which provides the **runtime environment for executing Java applications**. The JRE consists of the **JVM and libraries** required for running Java bytecode.

Environment - JDK

- **JDK** consists of :
  - **Java Development Tools:**
    - **Java Debugger (jdb):** A tool for debugging Java programs.
    - **Java Archive (JAR) tool:** A utility for creating and managing JAR files, which are compressed archives that contain Java classes and resources.
    - **JavaDoc tool:** A tool for generating HTML documentation from Java source code comments

JDK = Java Runtime Environment (JRE) + Development Tool

Environment - JRE

- **JRE (Java Runtime Environment)** is a software package that provides **Java class libraries, Java Virtual Machine (JVM),** and other components that are required **to run Java applications**.
- **JRE** is the **superset of JVM.**

JRE = Java Virtual Machine (JVM) + Library Classes

Environment - JVM

- **Java Virtual Machine(JVM) :** It's a **virtual machine** that allows a computer **to run Java programs.**
- JVM is a **runtime environment that loads, verifies, and executes Java bytecode**, and **converts bytecode to machine-specific code.**
- JVM consists of **three distinct components**:

    1.Class Loader

    2.Runtime Memory/Data Area

    3.Execution Engine

Environment - JVM

- **Class loader:**
  - Responsible for **loading classes into the JVM**.
  - When you run a Java program, the class loader **searches for the necessary classes** and **loads** them into the **runtime data area**.
- **Runtime data area:**
  - **JVM stores data in the runtime data area** during program execution.
  - It consists of several different areas:
    - **Method area:** Where class and method information is stored
    - **Heap:** Where objects are stored.
    - **Stack:** Where method calls and local variables are stored.

Environment - JVM

- **Execution engine** :
    - Responsible for **executing the bytecode (compiled Java code)** with the help of **Java interpreter and JIT compiler** that is loaded into the JVM.

Java Interpreter + Just-In-Time Compiler(JIT)

- The JVM also includes a **garbage collector**, which automatically frees up memory that is no longer being used by the program.

Environment - JVM

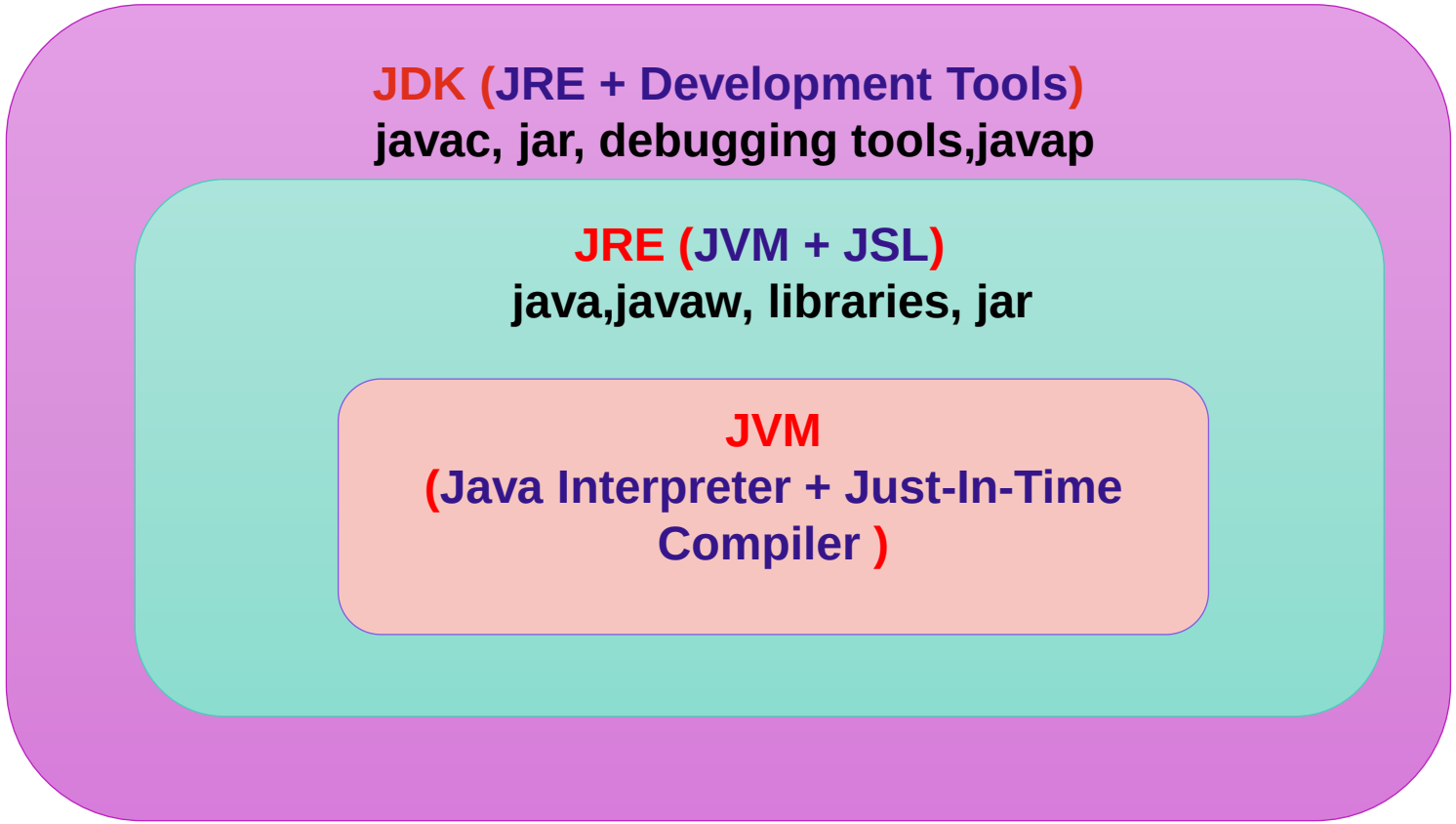
Interpreter

- The interpreter **reads and executes the bytecode instructions line by line**. Due to the line-by-line execution, the interpreter is **comparatively slower.**
- Another **disadvantage** of the interpreter is that when **a method is called multiple times**, every time a **new interpretation is required.**

JIT Compiler

- The **JIT Compiler overcomes the disadvantage of the interpreter**. The **Execution Engine** first uses the **interpreter to execute the byte code**, but when it **finds some repeated code**, it **uses the JIT compiler**.
- The **JIT compiler** then **compiles** the entire **bytecode and changes it to native machine code**. This native machine code is **used directly for repeated method calls**, which **improves the performance of the system.**

Environment



Note

- JVM, JRE and JDK are **ported to different platforms** to provide **hardware** and **operating system-independence.**

Difference - JDK vs JRE vs JVM

	JDK	JRE	JVM
Full From	Java Development Kit	Java Runtime Environment	Java Virtual Machine
Definition	Software development kit provided for developing Java applications	Software package that provides the runtime environment for executing Java applications.	Abstract machine that provides an environment for the execution of Java Bytecodes.
Tools	Includes tools and libraries for developing, debugging, and profiling Java applications.	Includes the JVM, class libraries, and other runtime components required to run Java applications	Does not include any software tools
Implementation	**JDK :** JRE + Development tools	**JRE :** JVM + Class libraries	**JVM :** provides a runtime environnent.

Features

- **Simple –** Java syntax based on earlier languages like C and C++. Designed considering the **pitfalls** of earlier languages
- **Object Oriented** – Java supports all the **Object Oriented features**.
- **Secured – No explicit pointer** support, run inside the java **virtual machine sandbox.** The **class loader, byte code verifier and security manager** component of JVM enable the secured environment. Java also supports application developer to use other security mechanism like **SSL, JAAS, Cryptography**, etc.

Features

- **Robust –** Java have strong memory management, lack of pointers that avoids security problems, automatic garbage collection, exception handling and the type checking.
- **Platform Independent** - Once compiled ,code will be run on any platform without recompiling or any kind of modification **-"Write Once Run Anywhere(WORA)".** This is made possible by making use of a **Java Virtual Machine(JVM).**

Note

- JVM, JRE and JDK are **ported to different platforms** to provide **hardware** and **operating system-independence.**

Features

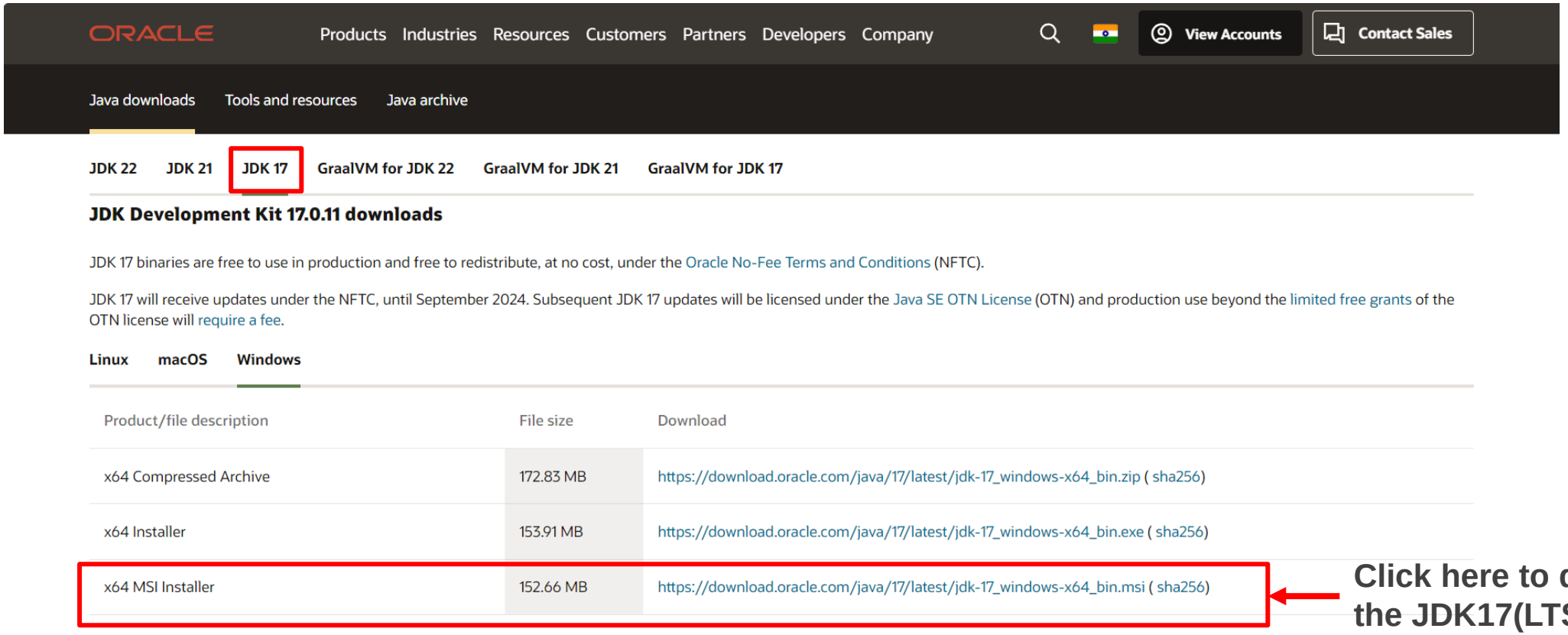
- **Architecture-neutral –** the size of **primitive types is fixed**
- **Distributed** – Java enable to create distributed application with the help of RMI and EJB
- **Multi-threaded –** Java run application concurrently.

Environmental Setup

- To **run Java applications**, we need to prepare the **environmental setup**. For installing Java we need the following:
  - The Java Runtime Environment **(JRE)** (Includes Java Virtual Machine (JVM))
  - The Java Developer Kit **(JDK)** (Includes the Java Compiler)
  - A text editor
  - If JDK is installed, you automatically get the JRE

    It can be **downloaded** here: Java Downloads | Oracle India

Environmental Setup



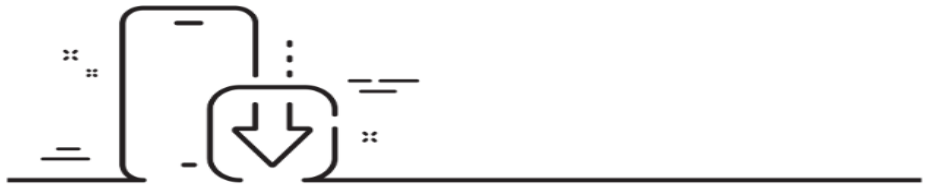
Click here to download the JDK17(LTS) version

Eclipse IDE

- Using a text editor is the basic way of creating a Java program, but there are tools available to make it easier to create Java programs called **IDEs** like **Eclipse**.
- **Eclipse** is a **third-party open-source IDE** that is very powerful and popular.
    - It can be downloaded here: http://www.eclipse.org/downloads/

Eclipse IDE

Eclipse download page



Download Eclipse Technology that is right for you

Building the World's Most Secure OpenJDK Distribution

Our new case study explores how the Eclipse Foundation and Adoptium Working Group are pioneering software supply chain security.

Download Today

Install your favorite desktop IDE packages

Learn More | Download x86_64

Download Packages | Need Help

Click Download packages

TEMURIN by ADOPTIUM

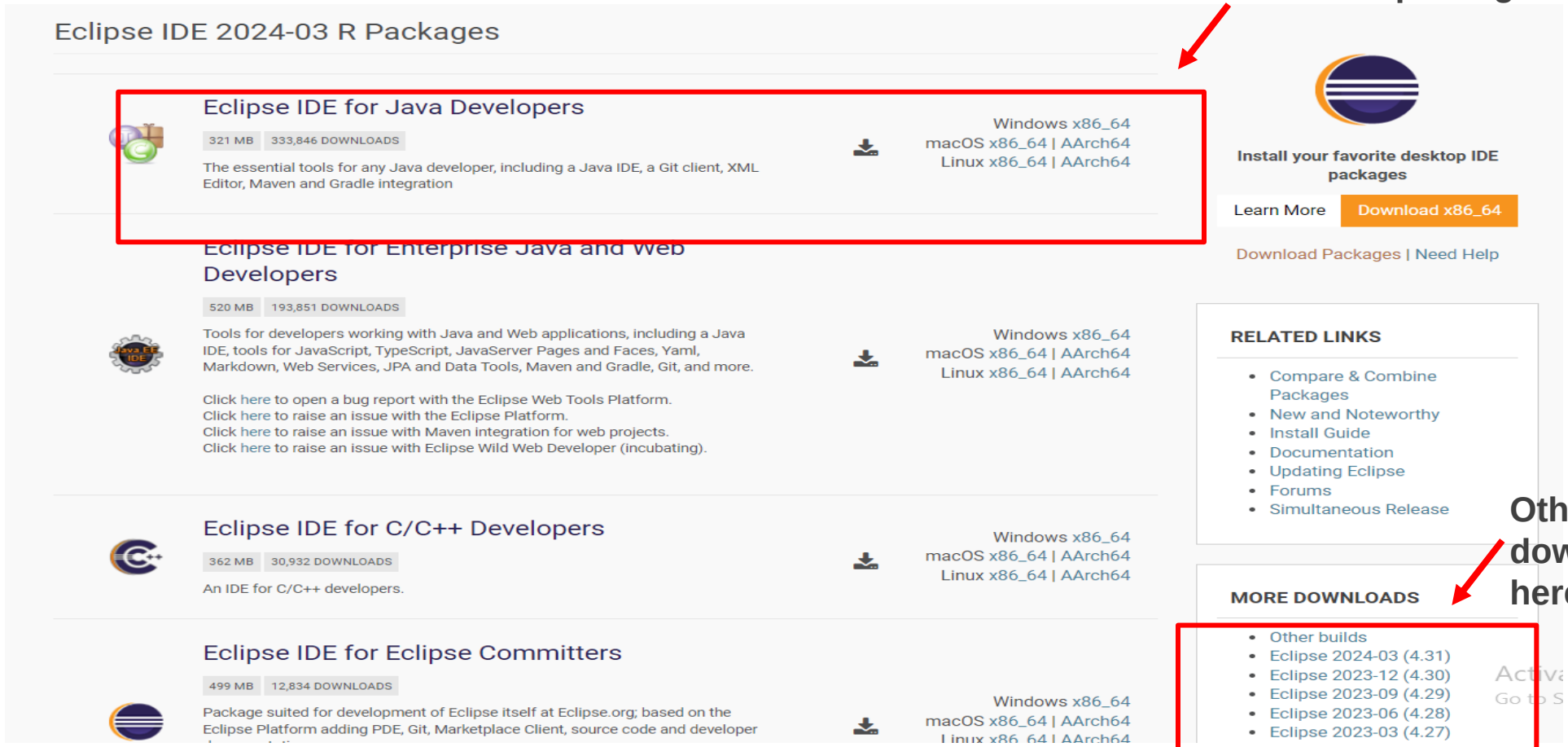
The Eclipse Temurin™ project provides high-quality, TCK certified OpenJDK runtimes and associated technology for use across the Java™ ecosystem.

Learn More | Download

Eclipse IDE

Choose an option from the list

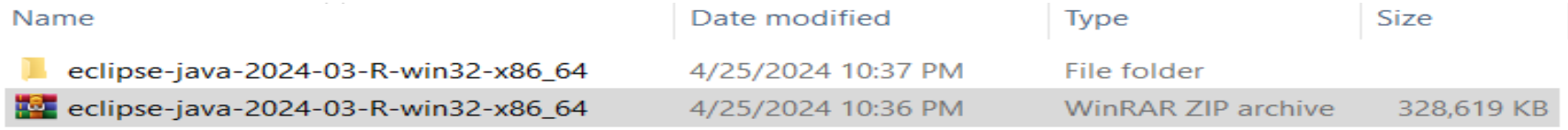
Download this package

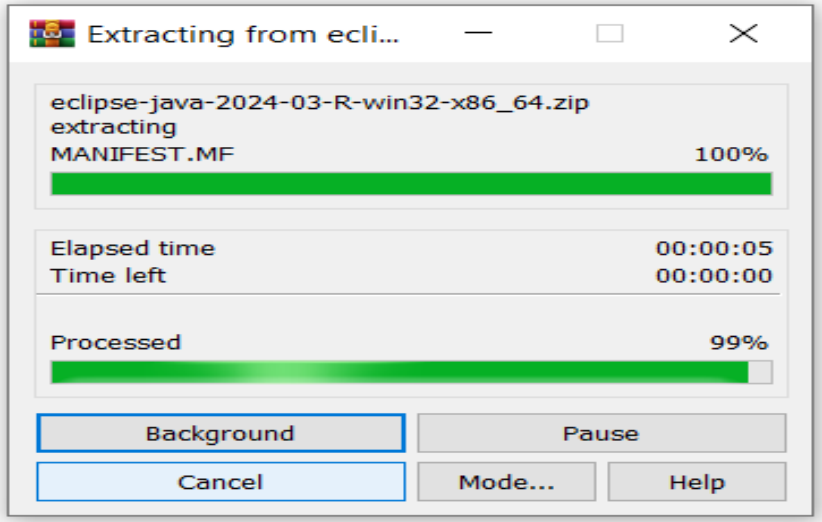


Other Versions to download from here

Eclipse IDE

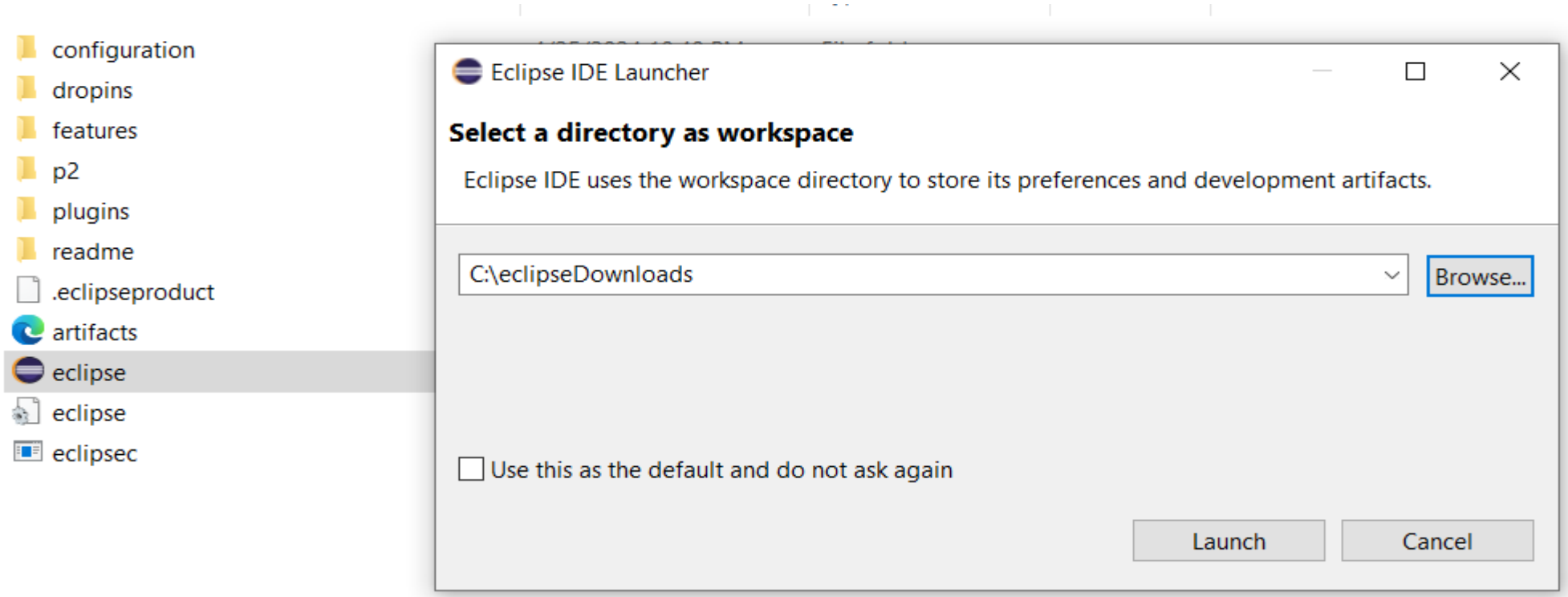
After download, right click and extract the eclipse file





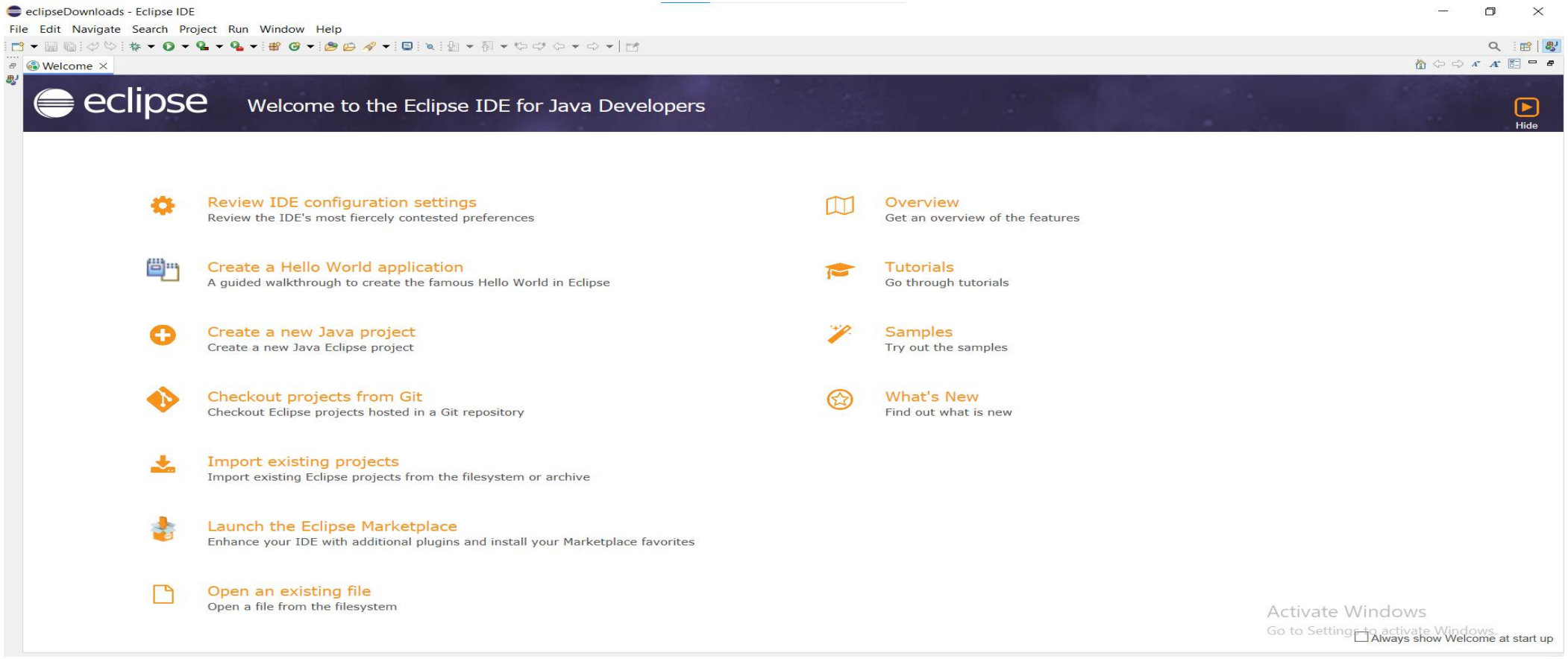
Eclipse IDE

Configuring Eclipse: Select a directory to create a workspace for projects



Eclipse IDE

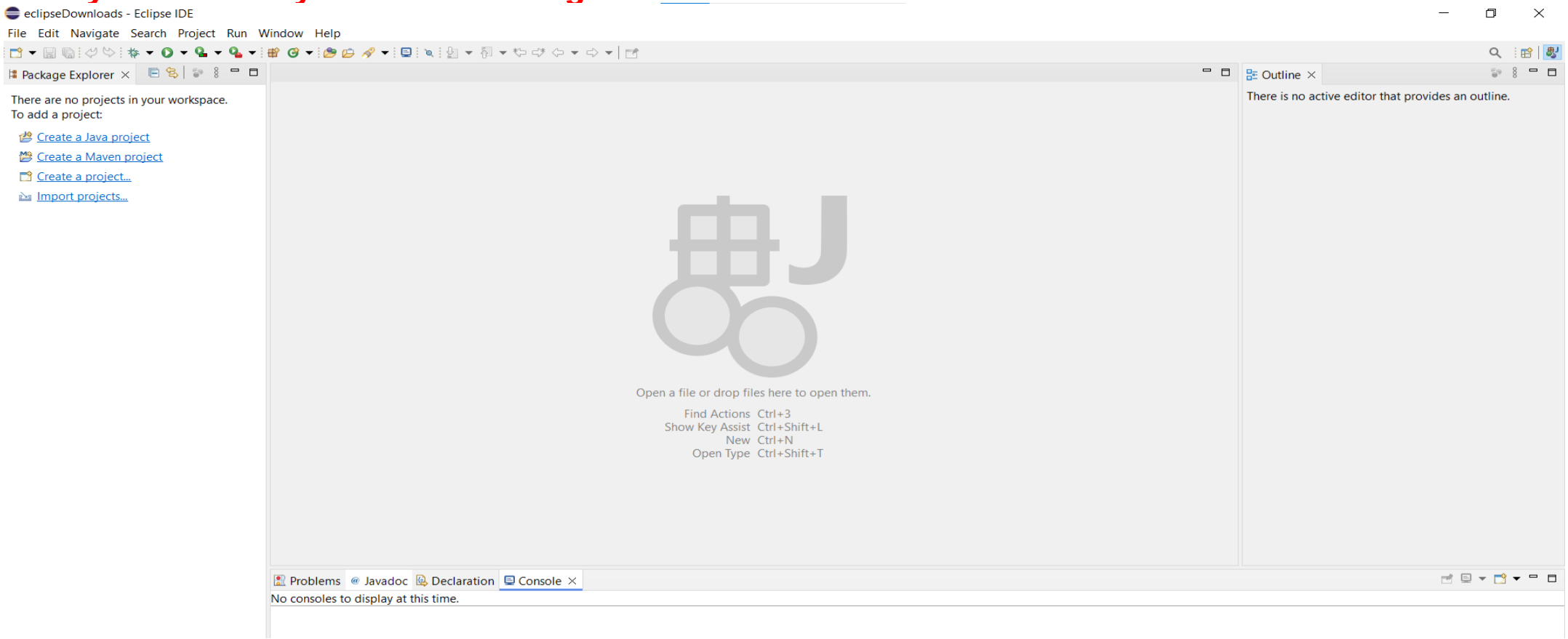
Once the workspace is chosen, you will get the welcome page



Eclipse IDE

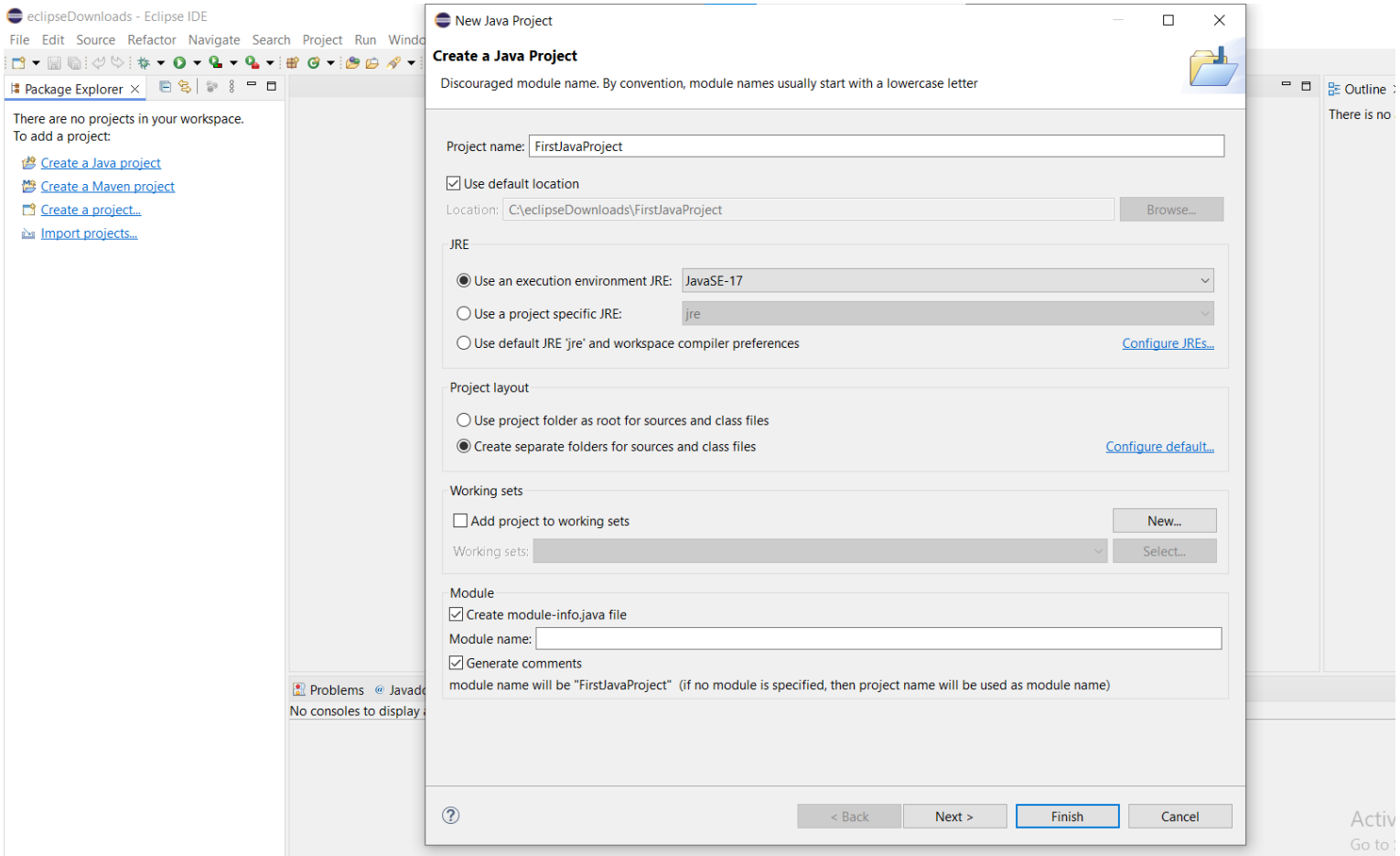
After closing the welcome screen ,the project explorer and other options will be shown up.

Now you are ready to start working.



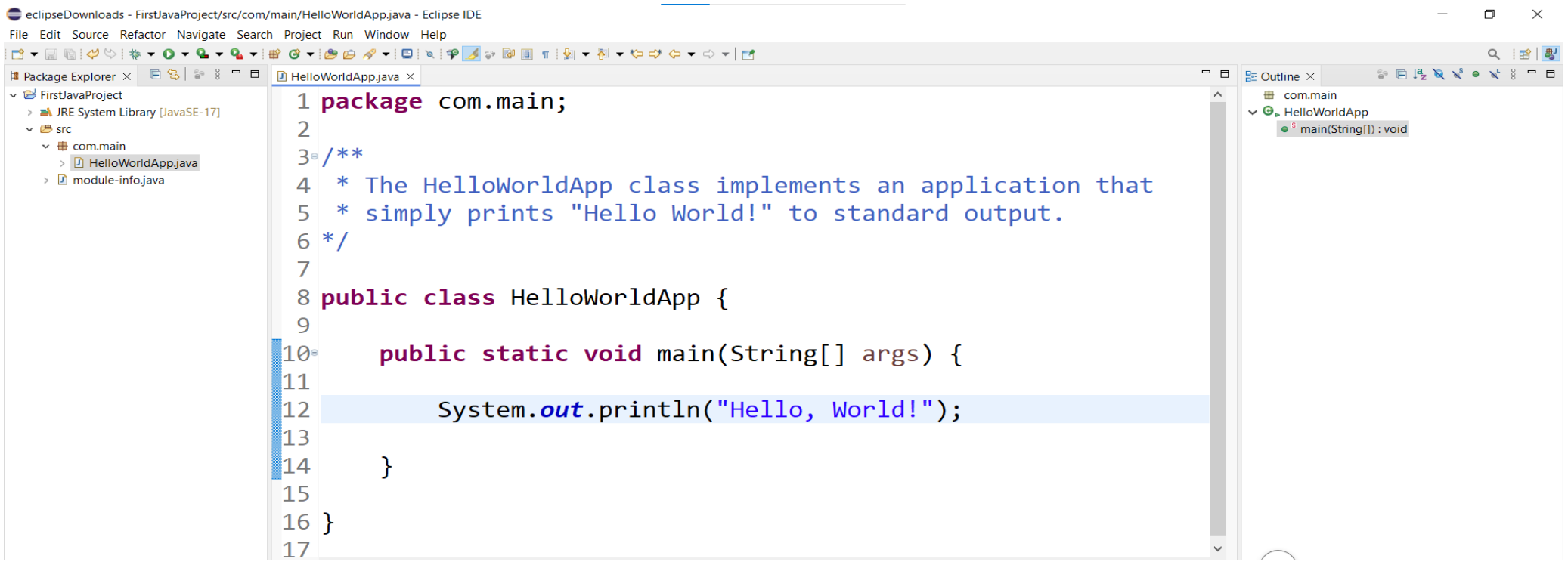
Eclipse IDE

Start working by creating the Java Project as FirstJavaProject



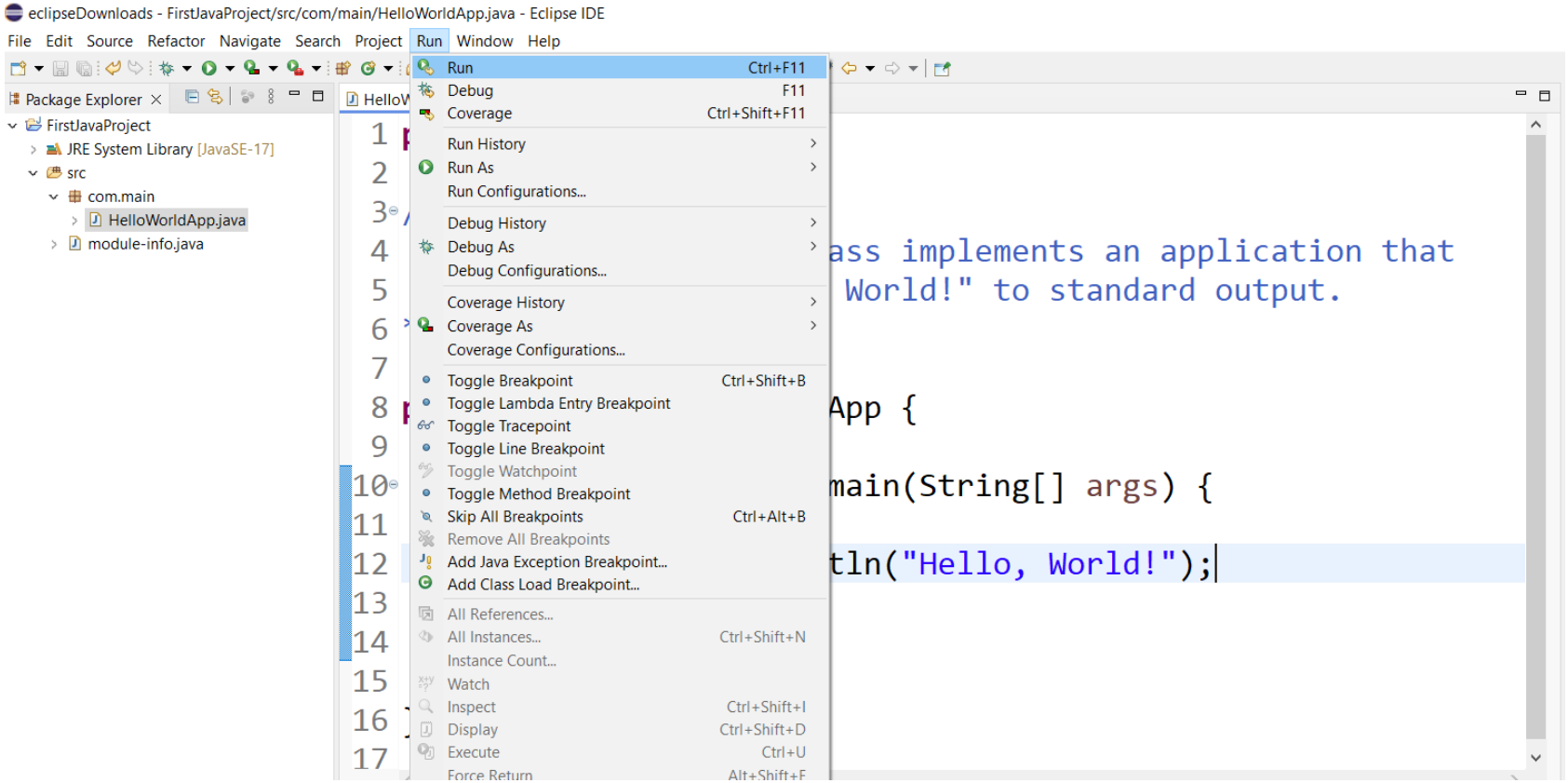
Eclipse IDE

Create First Java Program with file extension .java Example: HelloworldApp.java



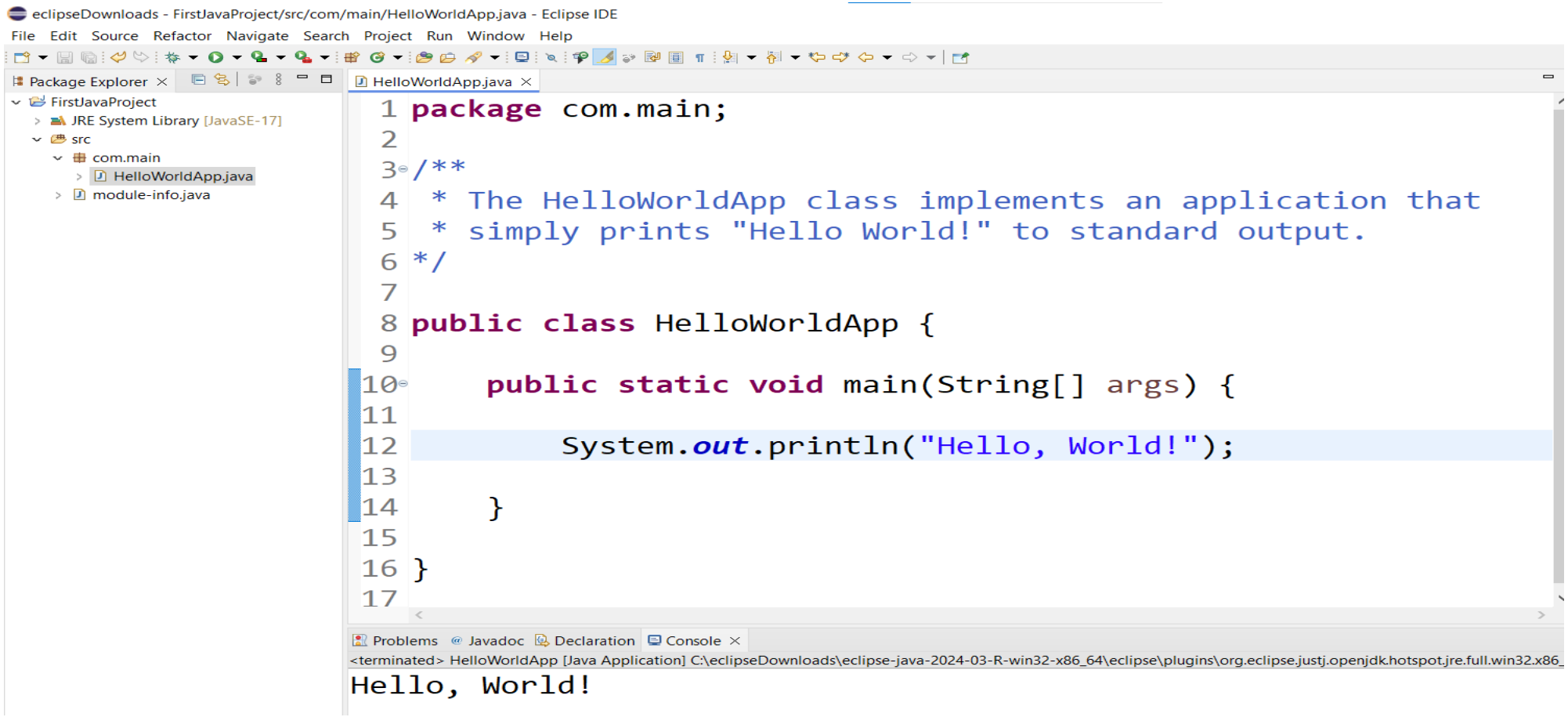
Eclipse IDE

After creating the application, run the application with a run command



Eclipse IDE

After execution, in the console, the output is displayed as Hello World!



First Java Program

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
*/
class HelloWorldApp {
    public static void main(String[] args){
        System.out.println("Hello World!"); // Display the Hello World!
    }
}
```

Need of Coding standards

- Code conventions are important to the programmer for the following reasons
    - To facilitate the **copying, changing, and maintenance** of the code.
    - To maintain the **readability** of the code.
    - Helps to **understand the code quickly** and thoroughly.
    - Helps to **well package the code** when deploying as the product.

General coding practice

- All code should have the following attribute:
  - **Simplicity:** Code should be easily understood and be concise.
  - **Readability:** Describe the code with comments and name the Identifiers relative to the objective of the code and ensure other people can understand the code.
  - **Modularity:** When there is a need to redo the small fraction of code then practice using the existing or reusing the code.
  - **Efficiency:** The code should be fast and economical. When using data files, read a value once and store it in a variable – don't go back and forward for the same value. Close connections if they are not required. Do not hold onto references to variables if not required, so as not to impose a memory leak.

Java source files

- Source file of java can be saved with the extension or suffixes **“Filename.java”.**
- **Java source file has the following order**:
    - Comments
    - Package and import statements
    - Class and interface declaration

Indentation

- In Java, the exact construction of the indentation is **unspecified.**
- so, set the tabs exactly for every **8 spaces**.
- The line length should **not exceed 80 characters**.
- When the statement is not fit in a single line, then wrap the lines based on the following principles.
    - Break the lines **after a comma**.

Example:

```
public void Cart(string product_name, double cost,
                 int quantity, string description)
```

Indentation

- Break the line **before an operator.**
- Prefer **higher-level breaks** to lower-level breaks.

Example: //Breaking an arithmetic expression before an operator and outside the parenthesized expression at a high level.

```
Calculation = a*(b + c - d)
                    + 4 * b
```

- Align the **new line** with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just **indent 8 spaces** instead.

Indentation

Example: //8 space indentation is used to breaking the if statements

```
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    Method();
}
```

- The **ternary expression** can be formatted as below:

```
Variable = Expression1 ? Expression 2 : Expression 3;
Variable = Expression1 ? Expression 2
                                : Expression 3;
Variable = Expression1
              ? Expression 2
              : Expression 3;
```

Indentation

Blank Lines:

- Blank lines improve **readability**.
- **Two blank lines** should always be used in the following circumstances:
    - Between sections of a source file.
    - Between class and interface definitions
- **One blank line** should always be used in the following circumstances:
    - Between methods
    - Between the local variables in a method and its first statement
    - Before a block or single-line comment.
    - Between logical sections inside a method.

Indentation

Blank Spaces:

- Black Spaces are used to improve the **reliability of the code** and can be used in the following circumstances
- A keyword followed by a **parenthesis should be separated by a space**.

Example:

```
while  (false){
  }
```

- It should **not** be used between a **method name and its opening parenthesis**. This helps to distinguish keywords from method calls.
- This can be included after **commas in an argument list**.

Indentation

- **Except '.' and unary operators** all the binary operators should be separated their **operands by spaces.**
- **Expression** in the **for statement** should be separated using **space**.

Comments

- Comments are the **overview** or **description** of the code.
- In Java, the comments can be implemented in **four styles** as follows
- **Block comments:**
    - Block comments **(/***/)** are used to provide **descriptions of files, methods, data structures, and algorithms.**
    - Block comments may be used at the **beginning** of each file and before each method and also within methods.
    - Block comments **inside** a function or method should be indented to the **same level** as the code they describe.

Example:

```
/*
*Here is a block comment
*/
```

Comments

- **Single-Line comments:**
  - **A short description of a block of code** can be given using single-line comments **(/*..*/).**
  - If a comment can't be written in a single line, it should follow the block comment format.

Example:

```
if(condition)  {
    /*   Description about bock of code in a single line */
                }
```

Comments

- **Trailing comments:**
  - Trailing comments **(/*..*/)** are **very short comments** that describe the statements.
  - These comments can be included in the **same line** of the statement but should be **separated from the statement using spaces**.
  - If more than **one short comment** appears in a code, they should all be **indented** to the **same tab.**

Example:

```
class HelloWorldApp     {
        public static void main(String[] args)   {
                System.out.println("Hello World!");      /* Display the Hello World!  */
        }
}
```

Comments

- **End - Of - Line comments:**
  - This comment line **(*//*)** can be used as a **single-line comment** or a short comment for a statement.
  - These comments can be included in the **same line** of the statement but should be **separated** from the statement using **spaces**.
  - This can be used in **consecutive multiple lines** for commenting out sections of code.

Example:

```
class HelloWorldApp     {
        public static void main(String[] args)   {
                System.out.println("Hello World!");      // Display the Hello World!
        }
 }
```

Documentation Comments

- Documentation comments are generally used when **writing code for a project/software package**.
- It helps to generate a **documentation page for reference**, which can be used for getting information about the present method, its parameters, class, variables, etc..,
- The **JavaDoc** tool is used to process the doc comments that come with JDK and it is used for generating Java code documentation in HTML(HyperText Markup Language) format from Java source code, which requires documentation in a predefined format.
- It is made up of **two parts**: **a description and Javadoc tags.**

Documentation Comments

Javadoc Tag:

- **Special tag** embedded within a Java doc comment.
- These doc tags enable you **to autogenerate** a complete, **well-formatted API from your source code.**
- The tags **start** with an "at" sign (@) and are **case-sensitive** (i.e).,they must be in upper and lowercase letters as shown in the below table.
- A tag must start at the beginning of a line (after any leading spaces and an optional asterisk) or it is treated as normal text.
- By convention, tags with the **same name are grouped together**.

Documentation Comments

Two Types of Javadoc tag :

- **Block tags -** Can be placed only in the **tag section** that follows the main description.

  - **Example:** @tag

- **Inline tags -** Can be placed anywhere in the main description or in the comments for block tags and Inline tags are denoted by curly braces

  - Example: {@tag}

Documentation Comments

Javadoc Tags:

- Below table describe the **Javadoc tag** that is used in documentation comments.

Tag	Description	Syntax
@author	Adds the author of a class.	@author name-text
{@code}	Displays text in code font without interpreting the text as HTML markup or nested javadoc tags.	{@code text}
{@docRoot}	Represents the relative path to the generated document's root directory from any generated page.	{@docRoot}
@deprecated	Adds a comment indicating that this API should no longer be used.	@deprecated deprecatedtext
@exception	Adds a **Throws** subheading to the generated documentation, with the classname and description text.	@exception class-name description

Documentation Comments

Tag	Description	Syntax
{@inheritDoc}	Inherits a comment from the **nearest** inheritable class or implementable interface.	Inherits a comment from the immediate surperclass.
{@link}	Inserts an in-line link with the visible text label that points to the documentation for the specified package, class, or member name of a referenced class.	{@link package.class#member label}
{@linkplain}	Identical to {@link}, except the link's label is displayed in plain text than code font.	{@linkplain package.class#member label}
@param	Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section.	@param parameter-name description
@return	Adds a "Returns" section with the description text.	@return description

Documentation Comments

Tag	Description	Syntax
@see	Adds a "See Also" heading with a link or text entry that points to reference.	@see reference
@serial	Used in the doc comment for a default serializable field.	@serial field-description | include | exclude
@serialData	Documents the data written by the writeObject( ) or writeExternal( ) methods.	@serialData data-description
@serialField	Documents an ObjectStreamField component.	@serialField field-name field-type field-description
@since	Adds a "Since" heading with the specified since-text to the generated documentation.	@since release
@throws	The @throws and @exception tags are synonyms.	@throws class-name description

Documentation Comments

Tag	Description	Syntax
{@value}	When {@value} is used in the doc comment of a static field, it displays the value of that constant.	{@value package.class#field}
@version	Adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used.	@version version-text

Documentation Comments

- **Format of Documentation comments.**
  - Each **line** above is **indented** to align with the **code below the comment**.
  - The **first line** contains the **begin-comment delimiter (/**)**.
  - Write the **first sentence** as a **short summary of the method**, as Javadoc automatically places it in the method summary table (and index).
  - The **inline tag** can be used **anywhere** that a comment can be written, such as in the text following block tags.

Documentation Comments

- If we have **more than one paragraph** in the doc comment, separate the paragraphs with a **<p> paragraph tag**.
- Insert a **blank comment line** between the **description and the list of tags**.
- The first line that begins with an **"@"** character **ends the description**.
- There is only **one description block** per doc comment; we **cannot continue** the **description following block tags**.
- The **last line** contains the **end-comment delimiter (*/).**

First Java Program using Documentation Comments

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 *
 * @author SmartCliff
*/
class HelloWorldApp    {
        public static void main(String[] args)    {
                System.out.println("Hello World!");      // Display the Hello World!
        }
}
```

Doc comment

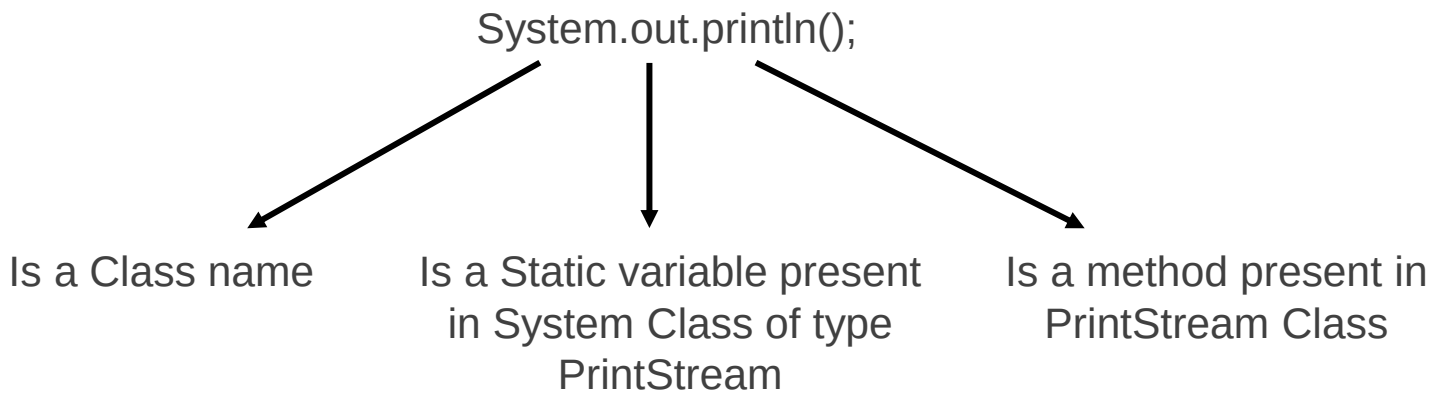
First Java Program in Detail

- **Class:** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method.
- The core **advantage of static method** is that there is **no need to create object** to invoke the static method.
- The **main method is executed by the JVM**, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** is the return type of the method, it means it doesn't return any value.

First Java Program in Detail

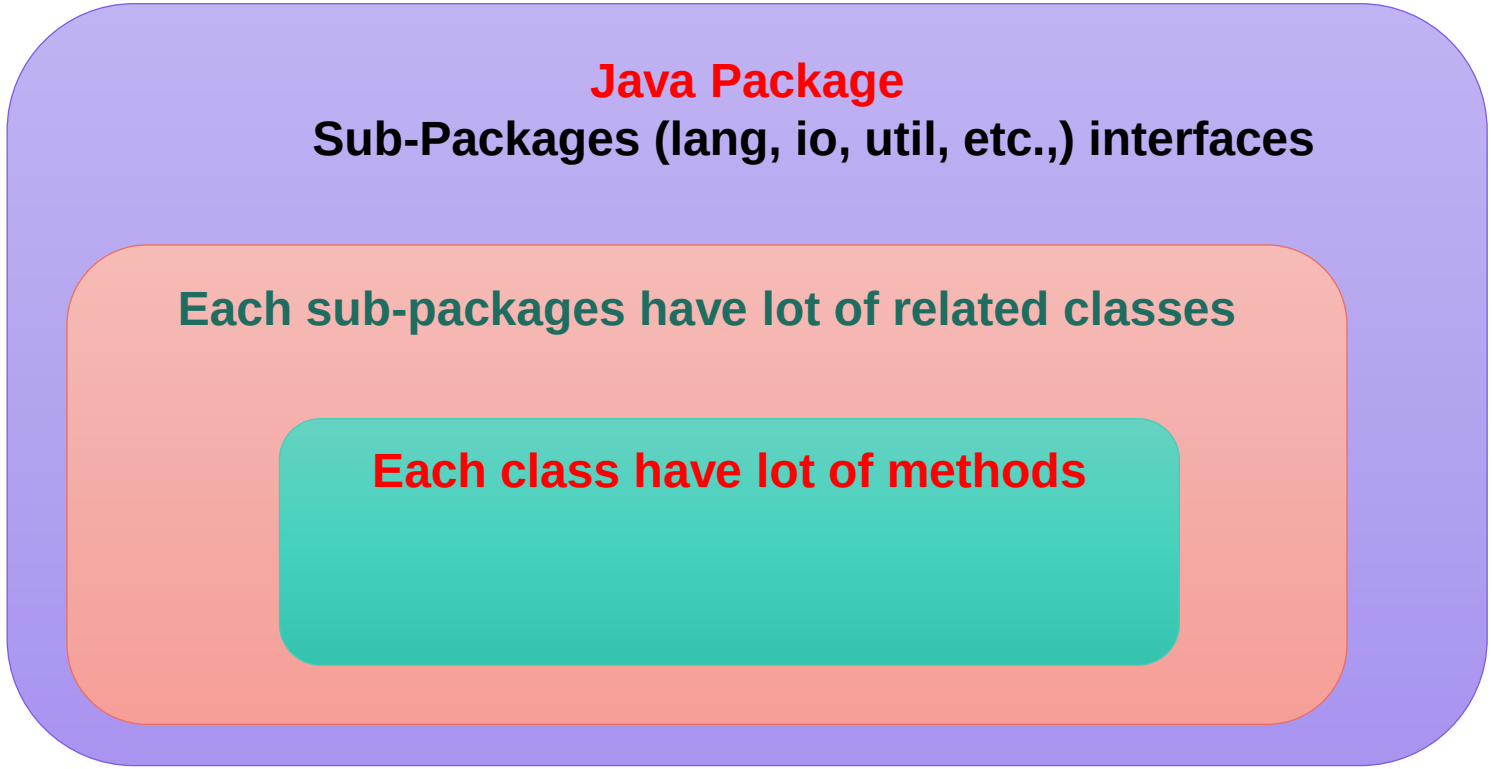
- **main** represents startup of the program.
- **String[] args** is used for command line argument.
- **System.out.println()** is used print statement.

  - System is a final class defined in the **java.lang package.**



Package: Introduction

- **Package:** It is a mechanism to **organize related classes, interfaces, and sub-packages** according to their functionality. It is like a folders in a file directory.



Package: Introduction

There are two types of Packages:

1. **Built-in Packages:** The already defined package in Java API like **java.io.*, java.lang.*** etc. are known as built-in packages. It is simply **import** based on your application needs.

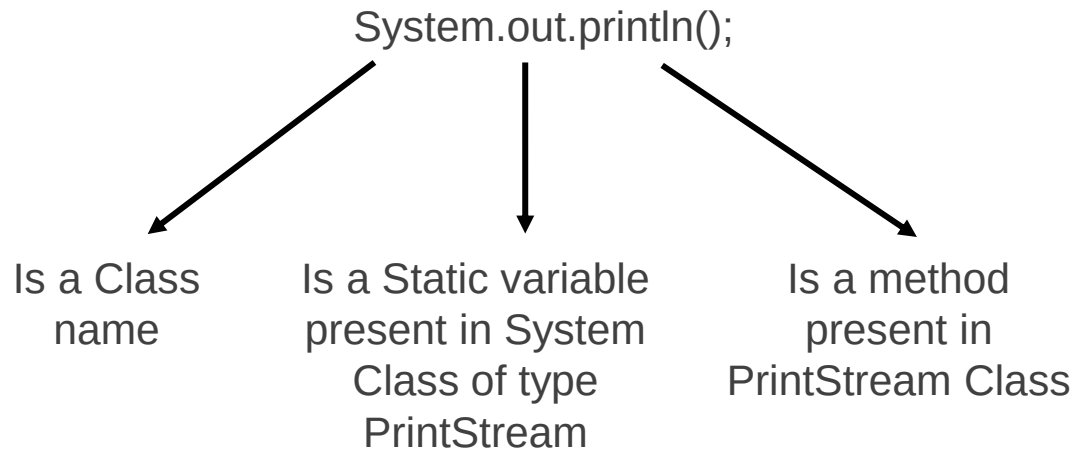
2. **User-defined Packages:** The package created by user and use based on application needs is called user-defined package.

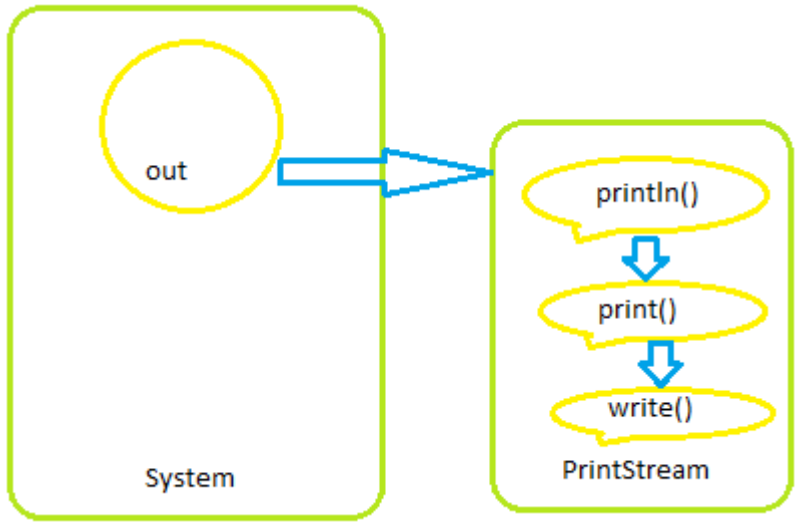
Note:

- Programmers typically **use packages** to **organize classes** belonging to the **same category** or providing **similar functionality**.

Package: Introduction

- **Example**: java.lang.System (System class is a one of the class of lang package)





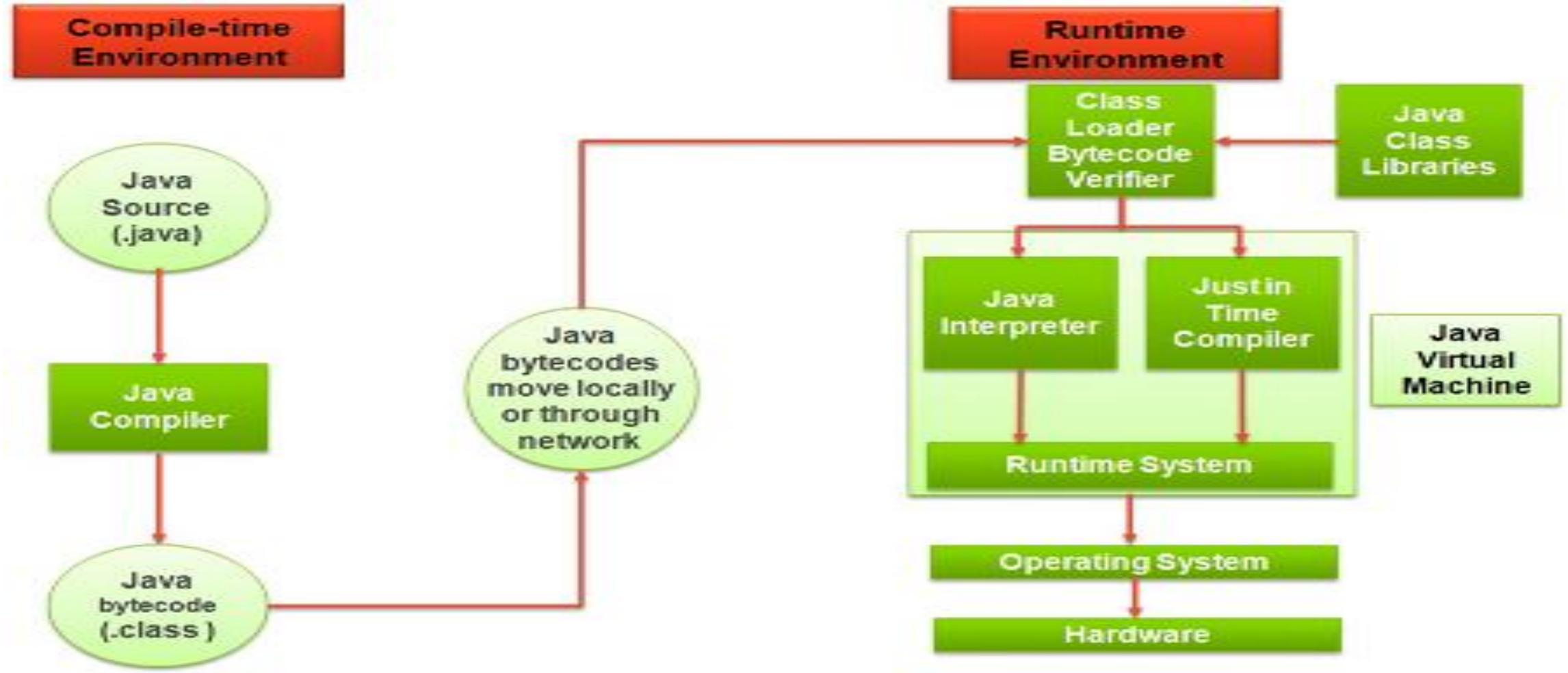
Note:

- As per Java 1.8 standard version, Java have **14 predefined packages**, **150 sub packages**, **7000 classes** and **7 lakh methods**.

Compiling and Executing

- To run java application both **compiler and interpreter** involves.
- The source code of Java will be created with file extension .java, **Example**: HelloworldApp.java
- The java Compiler compiles a java file and convert into **bytecode / classfile**.
- The byte code will be in a file with extension .class
- The .class file is **interpreted** by the JVM and morphed into **machine specific code**.

Compiling and Executing



Just-In-Time (JIT) Compiler

- The **Just-In-Time (JIT) compiler** is one of the integral parts of the Java Runtime Environment.
- It improves the **performance of Java applications** by compiling byte codes to native machine code at run time.
- The JIT compiler is **enabled by default**. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.
- Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow **the speed of the Java program** to approach that of a native application.

Java Tokens

Introduction

- The tokens are **the small building blocks** of a Java program that are meaningful to the Java compiler.
- The Java **compiler breaks the line** of code into text (words) is called **Java tokens**.
- The Java compiler identified these **words as tokens**.
- These tokens are **separated** by the **delimiters** and delimiters are not part of the Java tokens.
- It is useful for **compilers to detect errors.**

Example: In program, we will be using many statements and expressions to perform the operation. These statements and expressions are made up of tokens.

Introduction

- Java supports **5 types of tokens** as follows:

    1. Keywords
    2. Identifiers
    3. Literals
    4. Operators
    5. Special symbols

Keywords

- Keywords **are predefined or reserved words** that have special meaning to the Java compiler.
- Each keyword is assigned a **special task** or function and cannot be changed by the user.
- We **cannot** use keywords as **variables or identifiers** as they are a part of Java syntax itself.
- A keyword should always be written in **lowercase** as Java is a case-sensitive language.

Keywords

- They are **some available keywords** in Java

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

Identifiers

- An identifier is the **name given by the user** for the **various programming elements like variables, classes, methods, interface, etc.**
- **Rules for defining Identifiers**
  - Allowed characters for identifiers are all **alphanumeric characters**([A-Z],[a-z],[0-9]), '$'(dollar sign) and '_' (underscore).
    - **Example**: "blue@" is not a valid as it contain '@' special character.
  - Identifiers should not start with digits([0-9]).
    - **Example:** "123blue" is a not a valid
  - Java identifiers are **case-sensitive.**
    - **Example:** test and Test both are different

Identifiers

- There is no limit on the length of the identifier, but it is advisable to use an optimum length of **4 – 15 letters** only.
- Reserved Words can’t be used as an identifier.
  - **Example**: int while = 20; is an **invalid statement** as **while is a reserved word**.
- **Some Valid Identifiers**

  test, Test, _name, name100

  ABC, File_100, _abc_

Some Invalid Identifiers

int, 100name, test@123

Test.txt,

Naming Convention of Identifiers

- All the **identifiers** such as **classes, interfaces, packages, methods, and fields** of Java programming language are given according to the Java **naming convention.**
- By using this naming convention, we can achieve **readability** and can also easily **understand the code**.
- In programming, we often **remove the spaces** between words because programs of different sorts reserve the space (‘ ’) character for special purposes.
- Because the **space character is reserved**, we cannot use it to represent a concept that we express in our human language with multiple word

Naming Convention: Identifiers

Example

- That is in programming we cannot refer, *user login count=5.*
- We can represent as userLoginCount=5.
- Java follows the **CamelCase** for identifiers naming conventions.

CamelCase

- **Combines the compound words** and removes the space between the words.
- **Two types**: UpperCamelCase and lowerCamelCase.
- **UpperCamelCase:** The first letter of each word is capitalized.

  Example: Product, ProductDescription, CountValue.
- **LowerCamelCase:** The first letter of the compound words is lowercase.

  Example: product, iPad, countValue, productDescription

Naming Convention : Variables

- A variable's name can be **any legal identifier** i.e., begins with a letter, the dollar sign "$", or the underscore character “_“.
- The variable name should be **short** and **meaningful.**
- The choice of a variable name should be **mnemonic**- that is, designed to indicate to the casual observer the intent of its use.
- **One-character variable** names should be **avoided** except for temporary variables.
- The **temporary variables** can be **i,j,k,m, and n for integer** and **c,d,e for characters.**
- The variable name should be in **lowerCamelCase.**

```
Example:
String firstName;
int orderNumber ;
int count;
float price;
```

Naming Convention : Variables

- **Private class variables** should have an **underscore prefix**.
- Apart from its name and its type, the scope of a variable is its most important feature.
- Indicating class scope by using underscore makes it easy to **distinguish class variables** from local scratch variables.
- This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer

```
Example:
class Login{
private String _userName;
…
 }
```

Naming Convention : Constants and Methods

Constants:

- The names of variables declared class constants and should be in **uppercase**.
- If we specify the constant with **two words**, then separated by **underscores ("_").**

Example:

```
static final int MAX_HEIGHT = 70;
static final int MAX_WIDTH = 100;
static final int LENGTH = 20;
```

Naming Convention : Constants and Methods

Methods

- Method should be in the verb.
- Method name should be **lowerCamelCase.**

Example:

void calculateTax()

String getSurname()

void draw()

Naming Convention : Package and Import statements

- The first non-comment line of the java source file is a **package followed by import statements.**
- The prefix of a unique package name is always written in **all-lowercase ASCII letters**.

Example:

```
package java.util.*;
import java.util.Scanner;
```

Naming Convention: Class and interface

- **Class names** should be **nouns** and must be **simple** and **descriptive.**
- Use whole words-**avoid acronyms** and **abbreviations.**
- Class name should be in **UpperCamelCase.**

Example:

class Customer

class CustomerAccount

class Login

Naming Convention: Class and interface

- **Interface** tends to have a name that describes an operation that a class can do.
- Name of the interface should be **UpperCamelCase.**

Example:

interface Enumerable

interface CompEnumerable

interface Login

Data Types

- Data types defines the type data a variable can hold. It specify the different sizes and values that can be stored in the variable.
- Java is a **strongly typed language**
  - All variables must be declared before its use.
- **Two Types of Data types**

1. **Primitive data type**
   - byte, short, in, long, float, double, char, boolean
2. **Non Primitive data type**
   - Classes, interfaces, arrays, strings

Data Types

- **byte: (1 byte)**:
  - The byte data type is an 8-bit signed two's complement integer.
  - It has a minimum value of -128 and a maximum value of 127 (inclusive).
  - Useful for saving memory in large arrays.
  - **Default value** : 0
- **short: (2byte)**
  - The short data type is a 16-bit signed two's complement integer.
  - It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive).As byte ,can use a short to save memory in large arrays
  - **Default value** : 0

Data Types

- **int: (4 byte)**
  - By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of 2^{31} -1.
  - **Default value** : 0
- **long**: **(8 byte)**
  - The long data type is a 64-bit two's complement integer.
  - The signed long has a minimum value of -2^{63} and a maximum value of 2^{63}-1
  - **Default value** : 0L
- **float: (4 byte)**
  - The float data type is a single-precision 32-bit IEEE 754 floating point.
  - **Default value** : 0.0f

Data Types

- **double: (8 byte)**
  - The double data type is a double-precision 64-bit IEEE 754 floating point
  - **Default value** : 0.0d
- **boolean: (1 bit)**
  - The boolean data type has only two possible values: true and false
  - **Default value** : false
- **char**: **(2 byte)**
  - single 16-bit **Unicode** character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff‘
  - **Default value** : '\u0000'

Variables

- A **symbolic name associated with a value** and whose associated value may be changed.
- A variable is defined by the combination of an **identifier, a type and an optional initializer.**
- All variables have a **scope, which defines their visibility, and a lifetime**.
- There are **three types** of variables**:**

    1. Local Variables
    2. Instance Variables (Non-Static Fields)
    3. Class Variables (Static Fields)

Variables

Local Variables

- Variable that's declared within the body of a method.
- Will be used only within that method.
- Other methods in the class aren't even aware that the variable exists.
- A method will often store its state in local variables.
- A local variable **cannot** be defined with **"static"** keyword.

Variables

Instance Variables (Non-Static Variables)

- A variable declared **inside the class but outside the body of the method**, is called instance variable.
- It is **not declared as static**.
- Instance variables are created when the objects are instantiated and therefore they are **associated with the objects.**

Variables

Class Variables (Static Variables) :

- A variable which is **declared as static is called static variable**. It is also called as a **class variable**
- It **cannot be local**.
- You can create a single copy of static variable and share among all the instances of the class.
- Memory allocation for static variable **happens only once** when the class is loaded in the memory.

Variables

```
/**
  * The VariableApp class implements an application that
  * illustrate different Java variable

  * @author Smartcliff
*/
 class VariableApp {
            int mark = 95 ;//instance variable
            static char grade = 'S';     // static variable
            public static void main(String[] args) {
                        float average=95.0  // local variable
            }
 }
```

Literals

- A **fixed value** assigned to a variable

- Different **types of literals** are as follows:

Types	Examples

Integer Literals
1. Decimal
2. Hexa Decimal
3. Binary
	int decValue=56;
	int hexaValue=0x10;
	int binVal=0b11010 ;

Floating-Point Literals	double d1 = 123.4;
	double d2 = 1.234e2;
	float f1 = 123.4f;

Character Literals	char chrlit='0108';
String Literals	String check="Test" ;
Boolean Literals	boolean result=true;

Literals

- **Using Underscore Characters in Numeric Literals**
- In **Java SE 7 and later**, any number of **underscore characters** (_) can **appear anywhere** between digits in a numerical literal.
- Enables **to separate groups of digits in numeric literals**, which can improve the readability of your code.

Example:

```
long creditCardNumber = 1234_5678_9012_3456L;

long socialSecurityNumber = 999_99_9999L;

long bytes = 0b11010010_01101001_10010100_10010010;
```

Literals

- You can **place underscores** only between digits; you **cannot place underscores** in the following places:
  - At the beginning or end of a number
  - Adjacent to a decimal point in a floating point literal
  - Prior to an F or L suffix
  - In positions where a string of digits is expected

Unicode

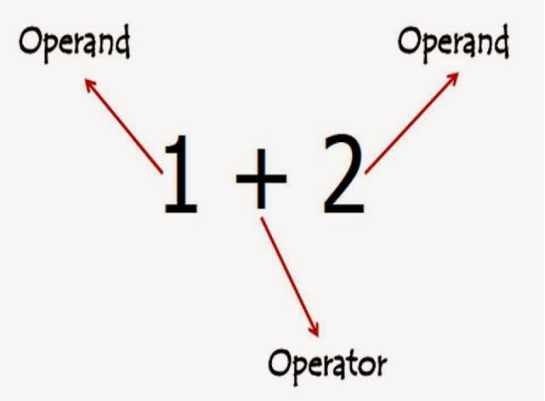
- It is Computing industry standard designed to **encode characters of the world's written languages**.
- **Unicode System?**
  - Before Unicode, there were many language standards:
  - ASCII (American Standard Code for Information Interchange) for the United States.
  - ISO 8859-1 for Western European Language.
  - KOI-8 for Russian.
  - GB18030 and BIG-5 for Chinese, and so on.

Unicode

- **Problems:**
  - A particular code value corresponds to different letters in the **various language standards**.
  - The encodings for languages with large character sets have variable length.
  - Some common characters are encoded as single bytes, other require two or more byte.
- **Solution:**
  - To solve these problems, a new language standard was developed i.e. Unicode System.
  - In **Unicode, character holds 2 byte**, so java also uses 2 byte for characters.
  - The **range of a char is 0 to 65,536**

Operators

- An operator in Java is a **special symbol** that signifies the **compiler** to perform **some specific mathematical or non-mathematical operations on one or more operands**.
- Value that the operator operates on is called operand.



- Java supports **8 types of operators**.

Expression

- An expression in Java is **any valid combination of tokens** like variables, constants and operators.
- An expression may consist of **one or more operands, and zero or more operators** to produce a value.

Examples:

- a + b * c
- (a * b) / (c + d)
- 10 – 4 * 5
- Etc.,

Operators & Expressions

- Below table we have listed down all the operators, along with their expressions:

Type	Operators	Expressions
Unary Operator	++,--,+(unary),-(unary), ~,!	a++,--a, -a, ~a, !a
Arithmetic Operator	+,-,%,*,/	a=b+c, a=b%d
Shift Operator	<<,>>,>>>	a=a>>2
Relational Operator	>,>=,<,<=,==, !=,instance of	a=10>5,5!=10,8==8
Bitwise Operator	&,^,|	a =2&3
Logical Operator	&&,||	(2>5)&&(10>8)
Ternary Operator	? :	a=(b>c)?1:0
Assignment Operator	= ,+=, -=, *= ,/= ,%= .&= ,^= ,|= <<= ,>>=,>>>	a=b a+=b a^=b

Precedence and Associativity

Precedence

- Operator precedence determines the **order** in which the operators in an expression are evaluated**.**

Example: int num = 6 -3 *8;

To evaluate the above expression Java, consider the precedence of the operator. Here, Multiplication (*) has the highest precedence over subtraction (-). So multiplication will be performed before the subtraction.

Associativity

- If an **expression has two operators** with **similar precedence,** the expression is evaluated according to its **associativity.**
- That is either **left to right** or **right to left.**

Example: X=Y=Z; Operator has same precedence. Here, the value of Z is assigned to variable Y. Then the value of Y is assigned of variable X because the associativity of = operator is from right to left

Precedence and Associativity

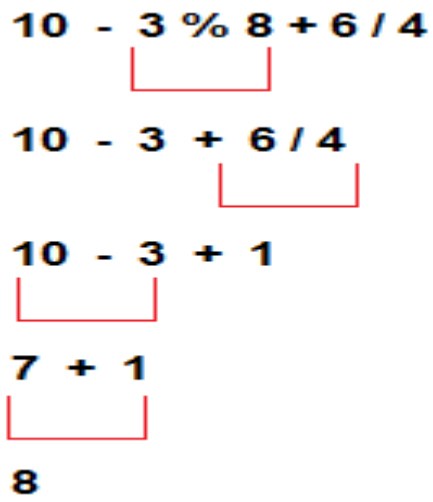
Operators	Precedence	Associativity
postfix increment and decrement	++ --	left to right
prefix increment and decrement, and unary	++ -- + - ~ !	right to left
multiplicative	* / %	left to right
additive	+ -	left to right
shift	<< >> >>>	left to right
relational	< > <= >= instanceof	left to right
equality	== !=	left to right

Precedence and Associativity

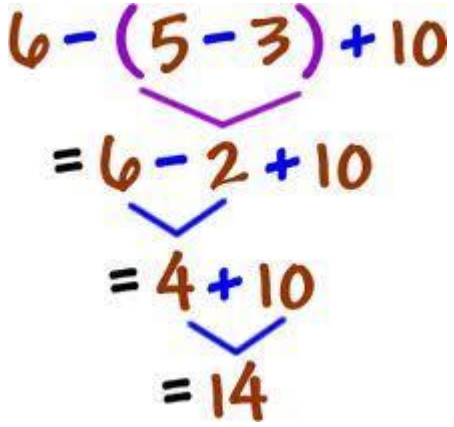
Operators	Precedence	Associativity
bitwise AND	&	left to right
bitwise exclusive OR	^	left to right
bitwise inclusive OR	|	left to right
logical AND	&&	left to right
logical OR	||	left to right
ternary	? :	right to left
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=	right to left

Expression Evaluation

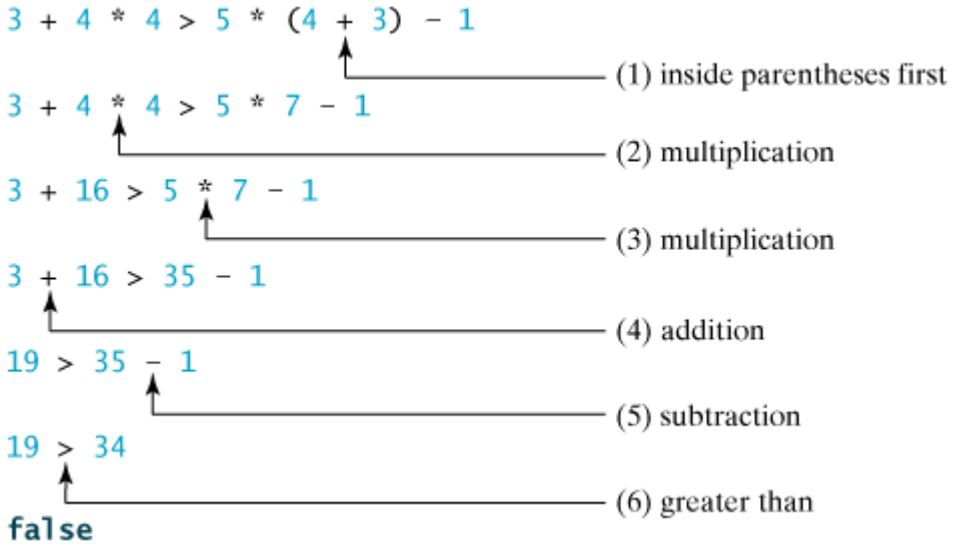
Example 1 : 10 – 3 % 8 + 6 / 4



Example 2: 6 - (5 – 3) + 10



Example 3 : 3+4*4>5*(4+3) -1



BITWISE OPERATOR

Introduction

- **Bitwise operators** are used to perform **operations** at the **bit level** and help to **manipulate data** at the **bit level** which we can call **bit-level programming**.
- Bit-level programming contains **0 and 1.**
- These can be done by first converting a **decimal value to its binary form**. This binary form is nothing but a **sequence of bits**. Bitwise operators perform operations on these bits.

Types

- There are **6 bitwise operators** in Java language. They are

Operator	Meaning
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR operator
~	Binary One’s Complement Operator
<<	Left shift operator
>>	Right shift operator

Types : AND(&)

- The **bitwise AND operator** is denoted using a **single ampersand symbol**,(&) and it needs **two operands** to work on.
- It takes the **binary values** of both the **left** and **right operands** and performs the **logical AND operation** over them on the **bit level**, i.e. if both the **operands** have **1** on the **specified position** then the **result** will also have **1** in **the corresponding position or else there will be 0**.

Num1	Num2	Result=Num1 & Num2
0	0	0
1	0	0
0	1	0
1	1	1

Truth table for Bitwise AND operator

Types : AND(&)

- **Example**: Let's take a look at the bitwise AND operation of two integers 12 and 25.
- Now **move from left to right**, and **perform logical AND** operations on the bits, and **store the result in the corresponding position.**

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

// Bitwise AND Operation of 12 and 25

  00001100

 & 00011001

_____________

  00001000 = 8 (In Decimal)
```

Types : AND(&)

```
// Bitwise AND
class Main {
  public static void main(String[] args) {

    int number1 = 12, number2 = 25, result;

    // bitwise AND between 12 and 25
    result = number1 & number2;
    System.out.println(result);    // prints 8
  }
}
```

Output: 8

Types : OR(|)

- The **bitwise OR operator** is much similar to the bitwise AND, i.e. if at **least any one of the operands** has **1**, then **the result** will also have **1** in the **corresponding position**, and **0** if they **both have 0** in the corresponding position.
- This is denoted using the **vertical bar** or **pipe symbol**, i.e. **|**.

Num1	Num2	Result=Num1 | Num2
0	0	0
1	0	1
0	1	1
1	1	1

Truth table for Bitwise OR operator

Types : OR(|)

- **Example**: Let's take a look at the bitwise OR operation of two integers 12 and 25.
- Now **move from left to right**, and **perform logical OR** operations on the bits, and **store the result in the corresponding position.**

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

  00001100

| 00011001

_____________

  00011101 = 29 (In Decimal)
```

Types : OR(|)

```
// Bitwise OR
class Main {
  public static void main(String[] args) {

    int number1 = 12, number2 = 25, result;

    // bitwise OR between 12 and 25
    result = number1 | number2;
    System.out.println(result);    // prints 29
  }
}
```

Output: 29

Types : XOR(^)

- This is similar to the other two, but the **only difference** is that they **perform logical XOR** on the **bit level**, i.e., if **exactly one of the operands** has **1** and the **other** has **0** then the **result** will have **1** in the corresponding position, and **0** if they have the **same bits** such as **both 0s or both 1s.**

Num1	Num2	Result=Num1 ^ Num2
0	0	0
1	0	1
0	1	1
1	1	0

Truth table for Bitwise XOR operator

Types : XOR(^)

- **Example**: Let's take a look at the bitwise OR operation of two integers 12 and 25.
- Now **move from left to right**, and **perform logical AND** operations on the bits, and **store the result in the corresponding position.**

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

// Bitwise XOR Operation of 12 and 25

  00001100

^ 00011001

_____________

  00010101 = 21 (In Decimal)
```

Types : XOR(^)

```
// Bitwise XOR
class Main {
  public static void main(String[] args) {

    int number1 = 12, number2 = 25, result;

    // bitwise XOR between 12 and 25
    result = number1 ^ number2;
    System.out.println(result);    // prints 21
  }
}
```

Output: 21

Types : Complement (~)

- We have seen three bitwise so far, if you have noticed, all of them were [binary i.e. they all require two operands to perform their functions.
- But **complement operator (~),** is the only bitwise operator that requires **only one operand**.
- The **bitwise complement** operator takes a **single value** and returns the **one's complement of the value**.
- The **one's complement** of a number is obtained by **changing all the 0's in its binary value to 1's** and by changing the **1's to 0's**.

Num1	Result=Num1 ^ Num2
0	1
1	0

Truth table for Bitwise Complement operator

Types : Complement (~)

- It is important to note that the bitwise complement of any integer **N** is equal to **- (N + 1)**.
- For example :Consider an integer **35**. As per the rule, the bitwise complement of **35** should be **-(35 + 1)** = **-36**. Now let's see if we get the correct answer or not.

```
35 = 00100011 (In Binary)

// using bitwise complement operator

~ 00100011

_____________

 11011100
```

Types : Complement (~)

- In the above example, we get that the bitwise complement of **00100011** (**35**) is **11011100**. Here, if we convert the result into decimal we get **220**.
- However, it is important to note that we cannot directly convert the result into decimal and get the desired output. This is because the binary result **11011100** is also equivalent to **-36**.
- To understand this we first need to calculate the binary output of **-36**.

2's Complement

- In binary arithmetic, we can calculate the binary negative of an integer using 2's complement.
- 1's complement changes 0 to 1 and 1 to 0. And, if we add 1 to the result of the 1's complement, we get the 2's complement of the original number. Example as Follows

Types : Complement (~)

```
// compute the 2's complement of 36

36 = 00100100 (In Binary)

1's complement = 11011011

2's complement:

 11011011

   +   1

_____________

11011100
```

- Here, we can see the 2's complement of **36** (i.e. **-36**) is **11011100**. This value is equivalent to the bitwise complement of **35**.Hence, we can say that the bitwise complement of **35** is **-(35 + 1) = -36**.

Types : Complement (~)

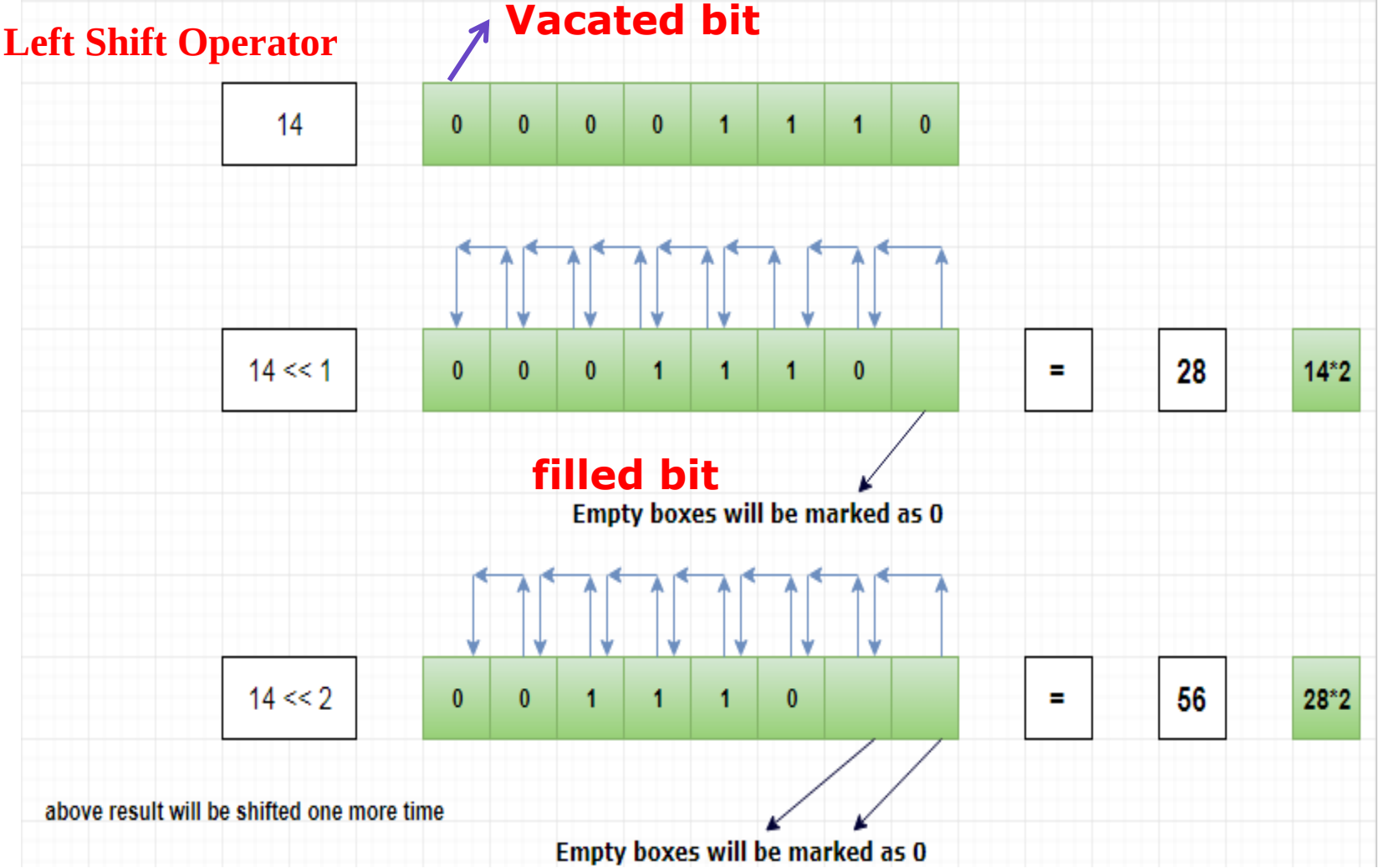
```
// Bitwise Complement
class Main {
  public static void main(String[] args) {
    int number = 35, result;
    // bitwise complement of 35
    result = ~number;
    System.out.println(result);    // prints -36
  }
}
```

Output: -36

Types : Shift Left (<<)

- The **left shift operator** (<<) is a **bitwise operator** that **shifts the bits** of **a binary number** to the **left** by a **specified number of positions**.
- In other words, it **multiplies the number by 2 raised to the power of the shift count**.
- Here's how the **left shift operation works**:
  - Each bit in the **binary representation** of the number is **shifted to the left** by the **specified number of positions.**
  - The **vacant positions** on the **right are filled with zeros**.
  - The **leftmost bits** that are **shifted out** (if any) are **discarded.**

Types : Shift Left (<<)



Types : Shift Left (<<)

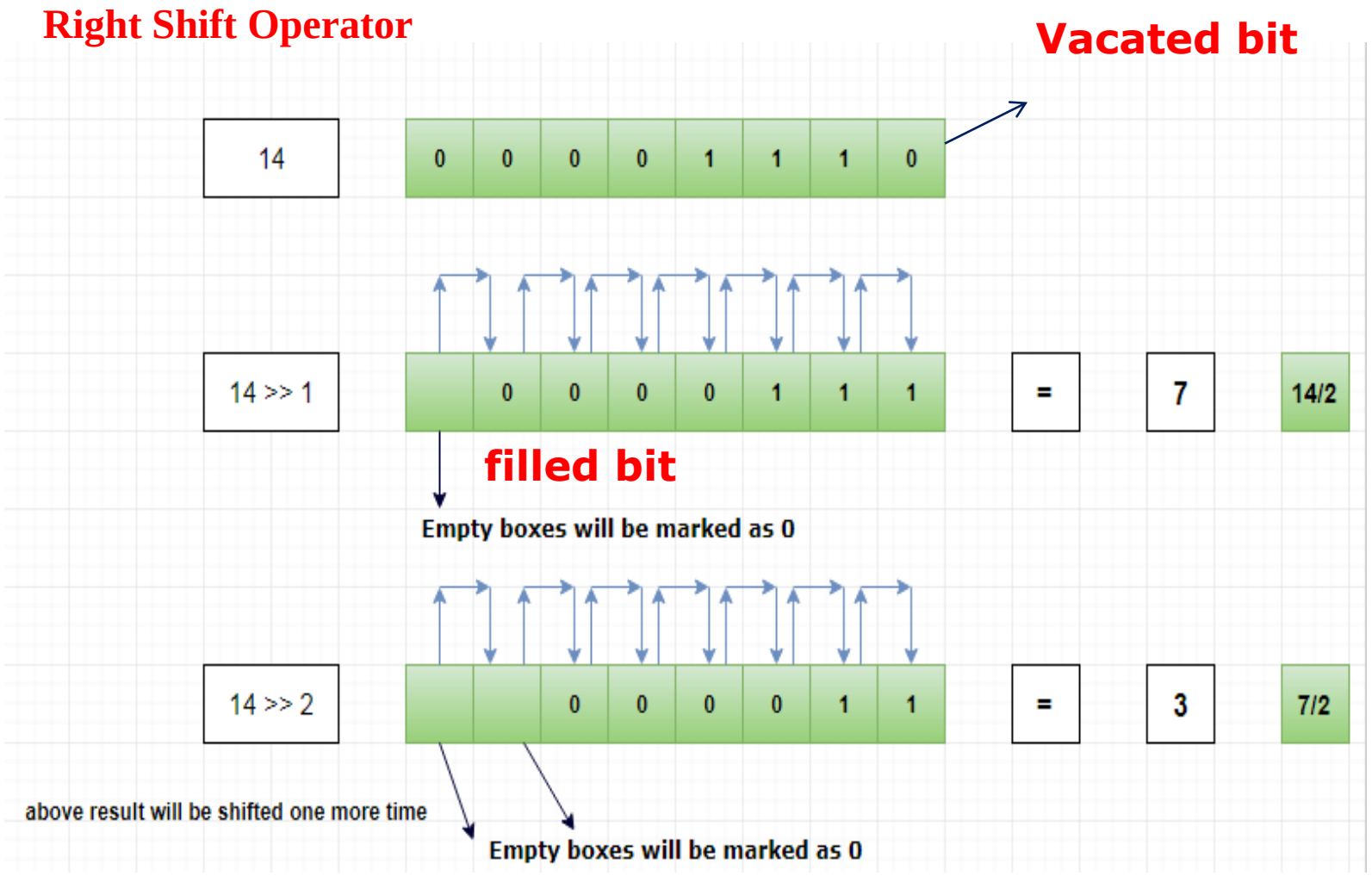
```
// Left shift Operators
class Main {
  public static void main(String[] args) {
    int number = 2;
    // 2 bit left shift operation
    int result = number << 2;
    System.out.println(result);    // prints 8
  }
}
```

Output: 8

Types : Shift Right (>>)

- The **right shift operator** (>>) is a **bitwise operator** that **shifts the bits** of a binary number to the **right** by a **specified number of positions**.
- In other words, it **divides the number by 2 raised to the power of the shift count**, **discarding the remainder.**
- Here's how the **right shift operation works**:
  - Each bit in the binary representation of the **number is shifted to the right** by the **specified number of positions**.
  - The **vacant positions** on the **left** are **filled with the sign bit** (for **signed integers**) or **with zeros** (for **unsigned integers**).
  - The **rightmost bits** that are **shifted out** (if any) are discarded.

Types : Shift Right (>>)



Types : Shift Right (>>)

```
// Right shift Operators
public class Main {
    public static void main(String[] args) {
        int num = 12;
        int result = num >> 2;
        System.out.printf("Right shift of %d by 2 is %d\n", num, result);
    }
}
```

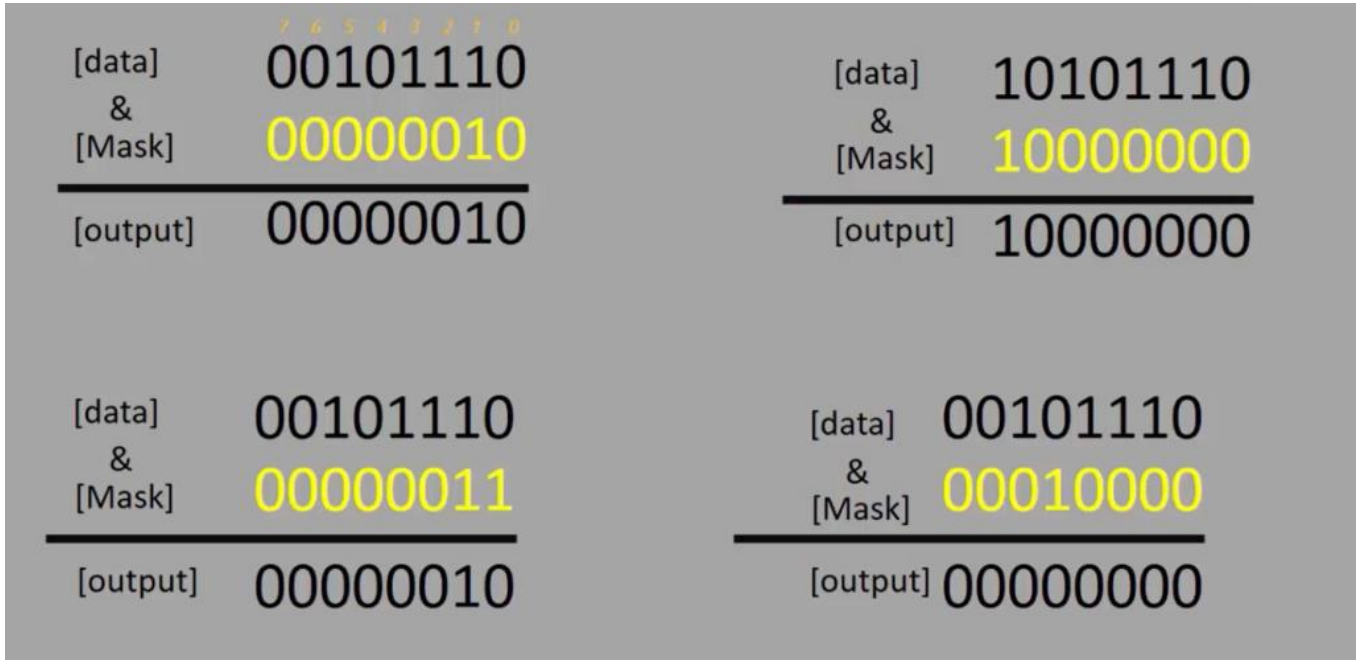
Output:

Right shift of 12 by 2 is 3

- In the example, the **value of num is 12**, and the **result of num >> 2 is 3.**
- The **binary representation of 12 is 0000 1100**, and after **right-shifting by 2 positions**, it becomes **0000 0011, which is 3 in decimal.**

Bit Masking

- **BIT MASKING:** Bit masking is a technique in programming used to test or modify the states of the bits of a given data.
- Here we have to decide what should be the **Mask_value**, and then take that **Mask_value and do a bitwisewise** operation with a **given data to get the output**.



Bit Masking

- Bit fields in programming are used to pack multiple data fields within a single word or byte, allowing for more efficient use of memory.
- Bitwise operators are particularly useful when working with bit fields because they enable manipulation and extraction of specific bits within a bit field.
- Here's how bitwise operators can be helpful for bit fields:

  1.Setting and Clearing Bits: Bitwise OR (|) is commonly used to set specific bits within a bit field. You can create a mask with the desired bit set to 1 and perform a bitwise OR with the existing value.

  Bitwise AND (&) with the complement of a mask can be used to clear specific bits within a bit field.

  2. Checking Bit Values: Bitwise AND (&) can be used to check the value of a specific bit within a bit field. If the result is non-zero, the bit is set; otherwise, it is clear.

Bit Masking

3.Toggle Bits: Bitwise XOR (^) can be used to toggle specific bits within a bit field. XORing with a mask toggles the corresponding bits.

4.Extracting Bit Fields: Right shifting (>>) can be used to extract a specific bit field from a larger bit field. By creating a mask with the relevant bits set to 1 and using it in a bitwise AND operation, you can obtain the desired bit field.

5.Combining Bit Fields: Bitwise OR (|) can be used to combine multiple bit fields into a single value. This is useful when you have separate values for different attributes packed into a single variable.

Bit Masking

TESTING OF BITS:

- **Testing a bit** refers to **checking the value** of a **specific bit** within a **binary representation** of a number or data.
- In the context of programming and bitwise operations, **testing a bit** typically involves checking whether a particular **bit is set (equal to 1)** or **clear (equal to 0) within a binary value**.
- Lets us understand with the below example:

> Write a program to find out whether a user entered a number is even or odd. Print an appropriate message on the console.Use testing of bits logic.

Bit Masking

LOGIC:

- How this works is, let's consider a **number an integer**, let's say 46.
- If you write **46 in binary form, you get 0010111<u>0</u>**, and here the **least significant bit is 0**.

For an EVEN number, the LSB is always 0.

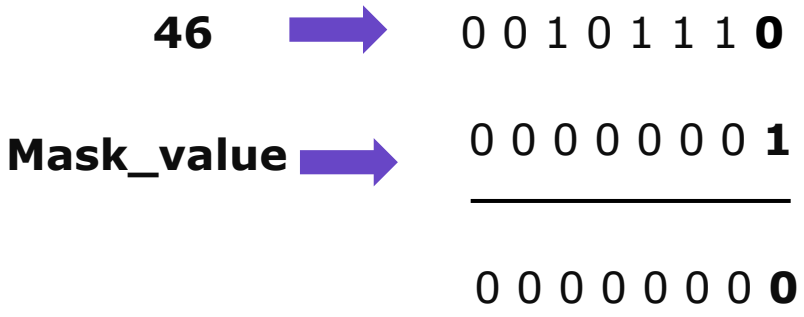
- Let's consider the **odd number 47**. **0010111<u>1</u>** is a **binary form of 47**. Here the **least significant bit is 1**.

For the ODD number, the LSB will always be 1.

- That's why by checking or by **testing the least significant bit(LSB) of a number**, we can find out whether a number is even or odd.

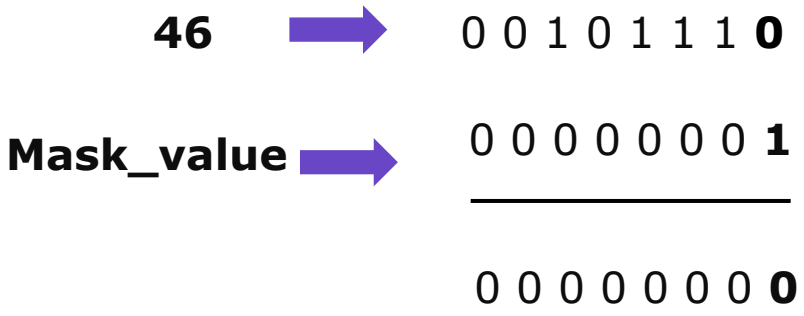
Bit Masking

- Testing of a bit can be achieved by the technique called **Bit-Masking**
- **Bit masking is a technique in programming used to test or modify the states of the bits of a given data.**
- Here we have to decide what should be the **Mask_value**, and then take that **Mask_value and do a bitwise AND(&)** operation with a **given data to get the output**.
- First we need to **create a Mask_value:** 00101110 is the input number written in a binary format. We divide this number into two areas: area1(0 0 1 0), and area2(1 1 1 0)



Bit Masking

- To find the even or odd of the number we **no need to concentrate** on the **area1** so we are **clearing as (0 0 0 0)** and in **area 2** we can determine with the **LSB bit** we set the **mask as (0 0 0 1) and performs bitwise AND.**
- As the result **if LSB is 1 denotes the number is ODD or 0 then the number is EVEN.**



Bit Masking

- If(number & 1) is a logic, Here the number is bitwise ANDed with a mask value which is 1.
- If(number & 1) this whole expression turns out to be a non-zero value, then the expression will be true. So, in that case, the print number is odd.
- If the output of if(number & 1) this expression is false, that is 0, then you can print the number as even.

```
// Even or ODD using Bitwise operation
if(number & 1){
        print(number odd);
}else{
        print(number even);
}
```

Bit Masking

SETTING OF BITS:

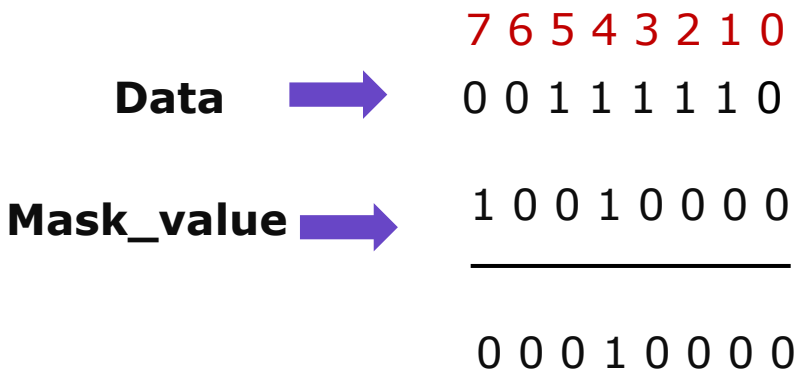
- Setting a bit means changing the value of a specific bit within a binary representation of a number or data to 1.
- Let us understand with the below example:

Write a program to set(make bit state to 1), 4th and 7th-bit position of a given number and print the result.

Bit Masking

- we have to **use a mask value**.,where the 4th and 7th bits are set(1).
- After that, we have to decide whether you want to use **bitwise & operator or bitwise | operator.**
- What happens if we use bitwise & operator?

  00010000 is the result you get.
- Here you can see that the output is not correct. We messed up a lot of bits here. So, we messed up with those bits, which we are not supposed to touch. It clearly shows that you cannot use bitwise & operator if you require to set the bits.



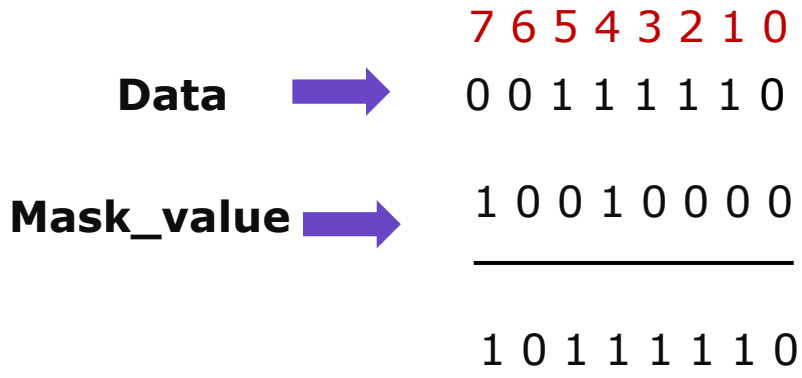
That's why the **'&' operator is used to 'TEST' not to 'SET'.**

Bit Masking

- What happens if we use bitwise | operator?

  10111110 is the result you get.

- Here you can see that the output is correct. you get this result 10111110. Here you can see that the un-affected data portion is not affected at the output. They are safe. So, only 4th and 7th-bit positions are set.



bitwise '&' is used to 'TEST' not to 'SET'; bitwise '|' is used to 'SET' not to 'TEST'.

Bit Masking

CLEARING OF BITS:

- **Clearing a bit** means **changing the value** of a **specific bit** within a **binary representation** of a number or **data to** 0.
- In the context of programming and bitwise operations, clearing a bit involves using bitwise operations to turn a particular bit off.
- Let us understand with the below example:

Write a program to clear(make bit state to 0) the 4th, 5th, 6th, bit positions of a given number and print the result.

Bit Masking

- First, which bitwise operation do you use to clear the given bit position of data? Do you use the **bitwise &** or **bitwise |** operator?

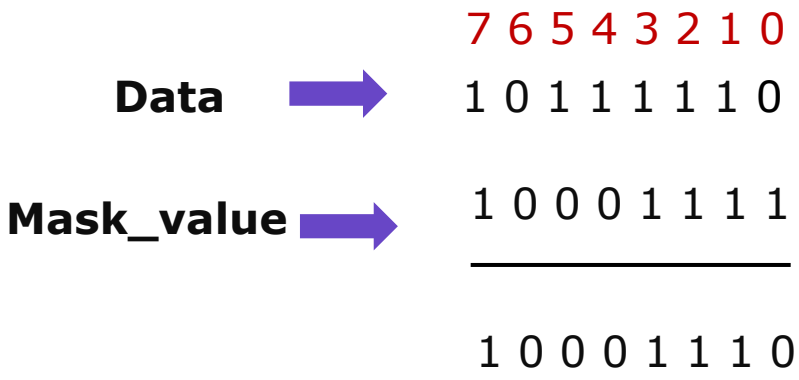
> ***&*** *‘ is used to ‘TEST and CLEAR’ not to ‘SET.’*
>
> ***|*** *‘ is used to ‘SET’ not to ‘TEST.’*

- In this case, certainly, we cannot use **bitwise OR** (**|**) because that is used to SET. So, you have to use **bitwise &** operator. So, bitwise ‘&’ is used to ‘TEST and CLEAR’.

Bit Masking

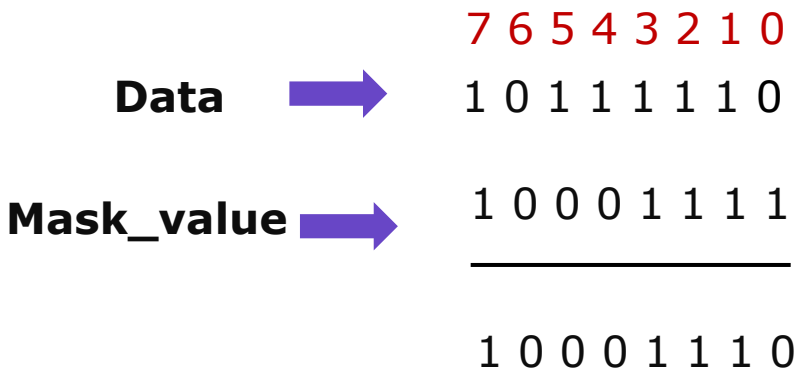
METHOD 1: *Here, the mask value should be 1 for the unaffected data portion of the data, and use zeroes in the mask value to zero out the required bit positions.*

- The goal is to reset the 4th, 5th, and 6th bit positions.
- The data is 10111110, and the 4th, 5th, and 6th portions must be zeroed out here.
- That's why we keep this portion of the mask value as zero's, and the remaining bit positions of the given data should not be affected.
- That's why let's mask those data portions with 1's. So, in this case, 10001110 (the mask value) turns out to be 0x8F.



Bit Masking

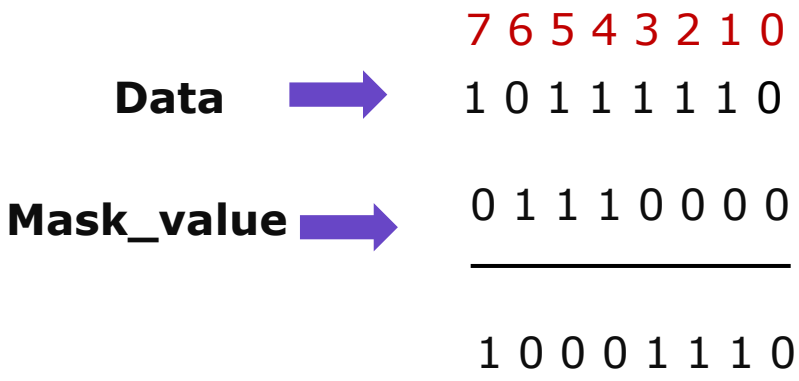
- Observe the output. Only the 4th, 5th and 6th portions are zeroed out,
- so the 7th portion and 0 to 3 portion are not affected, as shown.
- So, the takeaway from this post is **bitwise &** is used to both 'TEST' and 'CLEAR'.



Bit Masking

METHOD 2: The combination of the **bitwise &** and the **negation(~),** that is **bitwise NOT**.

- In this case, negate the mask value first and then perform the **bitwise &**.
- Here, 01110000 is the mask value, and you negate that. When you negate that, the final output is 10001110.



Bit Masking

BITWISE operator Vs Logical Operator:

- The **bitwise operators** work at the **bit level**, **manipulating binary representations of numbers**, while **logical operators** work with **boolean values**, **evaluating conditions** and **controlling program flow.**
- The **bitwise AND, and OR** use **‘&’ and ‘|’** as their operators, while **the logical AND**, and **OR** use **‘&&’ and ‘||’ as their operators.**

Type Conversion

Type Conversions

- **Widening or Automatic Type Conversion –** lower data types are automatically convert into higher data types. It is also known as **implicit conversion**.
- Automatic type conversion will take place if the following **two conditions** are met:
  - The two types are compatible.
  - The destination type is larger than the source type

Type Conversions

- Widening primitive conversions:

  byte → short, int, long, float, double

  short → int, long, float, double

  char → int, long, float, double

  int → long, float, double

  long → float, double

  float → double

- **Generally safe** because they tend to go from a small data type to a larger one
- No automatic conversion is supported from **numeric type to char or boolean**

Type Conversions

```
/**
  * The ConversionAutomatic class implements an application that
  * Illustrate the automatic type conversion
*/
 class ConversionAutomatic {
        public static void main(String[] args) {
                        int i = 100;
                        long l = i; // automatic type conversion
                        float f = l; // automatic type conversion
                        System.out.println("Int value "+i);
                        System.out.println("Long value "+l);
                        System.out.println("Float value "+f);
        }
 }
```

Type Conversions

• **Narrowing or Explicit Conversion -** If we want to assign a value of **larger data type to a smaller data type** we perform explicit type casting or narrowing.

• Useful for incompatible data types **where automatic conversion cannot be done**.

• Here, target-type specifies the desired type to convert the specified value to.

• **Narrowing primitive conversions**:

byte → char
short → byte, char
char → byte, short
int → byte, short, char
long → byte, short, char, int
float → byte, short, char, int, long
double→ byte, short, char, int, long, float

Type Conversions

```
/**
  * The ConversionExplicit class implements an application that
  * Illustrate the explicit type conversion
*/
 class ConversionExplicit {
        public static void main(String[] args)  {
                        double d = 100.04;
                        long l = (long)d;   //convert double into long
                        int i = (int)l;  // long convert into int
                        System.out.println("Double value "+d);
                        System.out.println("Long value "+l);
                        System.out.println("Int value "+i);
        }
 }
```

Type Conversions

- **Type promotion in Expressions -** While evaluating expressions, the intermediate value may **exceed the range of operands** and hence the expression value will be promoted.
- **Some conditions for type promotion are:**
    1. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
    2. If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

Type Conversions

```
/**
 * The TypePromotion class implements an application that
 * Illustrate the type promotion

 * @author Smartcliff
 */

class TypePromotion{
        public static void main(String[] args){
                        byte b = 50;
                        b = (byte)(b * 2);   //promote into int
                        System.out.println(b);
        }
}
```

Type Conversions

```
/**
 * The TypePromotion1 class implements an application that
 * Illustrate the type promotion
 */

class TypePromotion1{
	public static void main(String[] args){
		byte b = 42;
		char c = 'a';
		short s = 1024;
		int i = 50000;
		float f = 5.67f;
		double d = .1234;
		double result = (f * b) + (i / c) - (d * s); //promote into double
		System.out.println("result = " + result);
	}
}
```

Special Symbols

- Special symbols in Java are a **few characters which have special meaning** known to Java compiler and cannot be used for any other purpose.
- In the below table we have listed down the special symbols supported in Java along with their description.

Special Symbols

Symbols	Description
brackets []	These are used as an array element reference and also indicates single and multidimensional subscripts
Parentheses()	These indicate a function call along with function parameters
Braces{}	The opening and ending curly braces indicate the beginning and end of a block of code having more than one statement
Comma (,)	This helps in separating more than one statement in an expression
Semi-Colon (;)	This is used to invoke an initialization list

Read User Input

Introduction

- There are **three different ways** to read input from the user:
  1. Buffered Reader Class
  2. Scanner Class
- **Scanner Class**: It is a class in **java.util package** used for obtaining the user input of the primitive types like int and double. It is the **easiest way to read input** in a Java program.
- **Scanner object** is constructed from **Scanner Class** and **System.in (input stream)** object is passed as a parameter while creating a scanner object.
- **Constructing a Scanner object to read console input:**

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

Methods

- After creating the scanner object, we can use below **Scanner class methods** for reading the **respective primitive data types from the console.**

Method	Description
boolean nextBoolean()	This method reads the boolean value from the user.
byte nextByte()	It reads the byte value from the user.
double nextDouble()	It accepts the input in double datatype from the user.
float nextFloat()	It takes the float value from the user.
int nextInt()	It reads the integer value from the user.

Methods

Method	Description
String nextLine()	This method reads the String value from the user.
long nextLong()	This method reads the long type of value from the user.
short nextShort()	It reads the short type of value from the user.
String next()	It reads the character input from the user.

Note:

- Scanner class **no specific method** to read character type.
- To read a single character, we use **next().charAt(0). next() function** returns the next token/word in the **input as a string** and **charAt(0) function** returns the **first character** in that string.

Example: #1

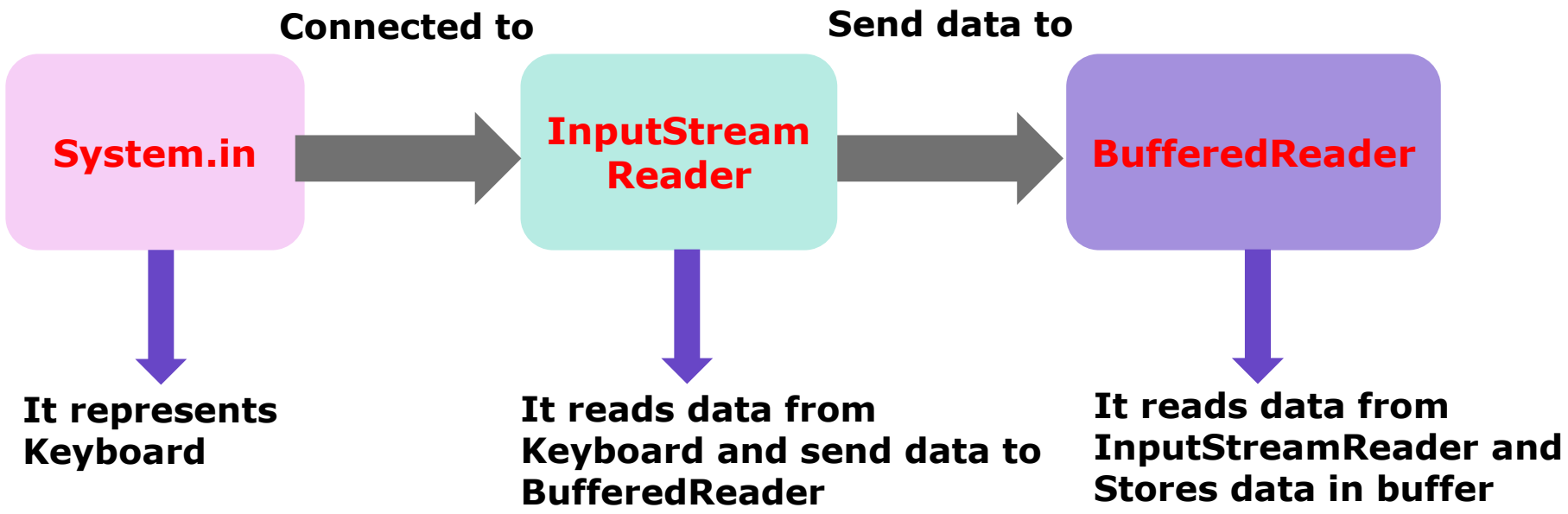
```
/**
  * The ReadSomeInput class implements an application that
  * Illustrate reading a console input  */
import java. util.Scanner; //import Scanner class from util package
public class ReadSomeInput {
        public static void main(String[] args) {
                Scanner console = new Scanner(System.in);
                System.out.print("Enter your Name : ");
                String name = console.next();
                System.out.println("Hi, " +name+" . Welcome to the Training Program ");
                console.close();
    }}
```

Output:

```
Enter your Name : Arun
Hi, Arun . Welcome to the Training Program
```

Introduction

Read User Input from console



Example #2

```
/**
 *This example demonstrate read user input from console using InputStreamReader and BufferedReader class
*/
class BufferedStreamDemo{
        public static void main(String[] args) {
                //Accepting Different type of Input(integer,float,double,short,long, byte, char, string, boolean) values from Keyboard
               Boolean bul=false;
               BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
               System.out.println("Enter a string:");String str=br.readLine();
               System.out.println("Enter a integer:");int n=Integer.parseInt(br.readLine());
               System.out.println("Enter float value:");Float ft=Float.parseFloat(br.readLine());
               System.out.println("Enter short value:");Short sht=Short.parseShort(br.readLine());
               System.out.println("Enter a Double value:");Double dub=Double.parseDouble(br.readLine());
              System.out.println("Enter a long value:");long log=Long.parseLong(br.readLine());
               System.out.println("Enter a Byte value:");Byte bit=Byte.parseByte(br.readLine());
               System.out.println("Enter a character:");char ch=(char)br.read();
```

Example #20

```
        //Displaying values on console
        System.out.println("Entered Values are :");
        System.out.println("Integer value is: "+n);
        System.out.println("Float value is: "+ft);
        System.out.println("Short value is: "+sht);
        System.out.println("Double value is: "+dub);
        System.out.println("Long value is: "+log);
        System.out.println("Byte value is: "+bit);
        System.out.println("Character value is: "+ch);
        System.out.println("String value is: "+str);
        System.out.println("Boolean value is: "+bul);
    }
}
```

Quiz

1) Which of the following are primitive data types?

a) int

b) float

c) double

d) boolean

e) All the above

e)All the Above

Quiz

2) A local variable stores temporary state; it is declared inside a

a) Class

b) Method

c) Block

d) Object

b) & c)

Quiz

3) A _____ is a value that should not be altered by the program during normal execution.

a) Variable

b) int

c)Identifiers

d) Constant

d) Constant

Quiz

4) Which of these can not be used for a variable name in Java?

a) Identifier

b) Keyword

c) Both a and b

d) None of the above

b) Keyword

Quiz

5) Which conversion also called Automatic Type Conversion?

a) Widening

b) Narrowing

c) Both A and B

d) All the above

a) Widening

Quiz

6) Long Literals in java must be appended by which of these?

a) L

b) l

c) D

d) 0x

a) & b)

Quiz

7) Which variables are created when an object is created and destroyed when the object is destroyed?

a) Local variables

b) Instance variables

c) Class Variables

d) Static variables

b) Instance variables

Quiz

8) Which of the following are not Java keyword ?

a) double

b) switch

c) instanceof

d) then

d) then

Quiz

9) Which of these is not a bitwise operator?

a) &

b) |

c) ^

d) <=

d) <=

Quiz

10) Which operator is used to invert all the digits in binary representation of a number?

a) ~

b) <<

c) ^

d) >>>

a) ~

Quiz

11) It is time to buy a new phone when at least one of the following situations occurs:

- **the phone breaks**
- **the phone is at least 3 years old**

```
int phoneAge;        // in years
boolean isBroken;
……..                 //code initializes variables
boolean needPhone =_____________
```

```
(isBroken == true) || (phoneAge >= 3);
```

Quiz

12) Evaluate the following expression:

(17 < 4*3+5) || (8*2 == 4*4) && !(3+3 == 6)

a) true

b) false

b)false

Quiz

13) Assume num1=10. Which of the following is not a valid statement?

a) num1>>2

b) num1<<<2

c) num1%=2;

d) num1<<2;

b) num1<<<2

Quiz

14) In Java, after executing the following code what are the values of x, y, and z?

```
int x,y=10; z=12;

x = y++ + z++;
```

a) x=22, y=10, z=12

b) x=24, y=10, z=12

c) x=24, y=11, z=13

d) x=22, y=11, z=13

d) x=22, y=11, z=13

Quiz

15) Which Scanner class method is used to read integer value from the user?

a)next()

b) nextInt()

c) nextInteger()

d) readInt()

b) nextInt()

Quiz

16) Which is the correct syntax to declare Scanner class object?

a) Scanner obj=Scanner();

b) Scanner obj=new Scanner();

c) Scanner obj= Scanner (System.in);

d) Scanner obj=new Scanner (System.in);

d) Scanner obj=new Scanner(System.in);

Quiz

17) Consider the following object declaration statement

Scanner obj= new Scanner(System.in).

What is System.in in this declaration?

a)Class which point input device

b) Reference to Input stream

c) Reference to Computer System

d)None of these

b) Reference to Input stream

Success is the sum of all efforts, repeated day-in and day-out

- R. Collier


THANK YOU
SmartCliff
Career Mobility Solutions